FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications

KIZHEPPATT VIPIN, Nazarbayev University, Kazakhstan
SUHAIB A. FAHMY, University of Warwick, United Kingdom

Dynamic and partial reconfiguration are key differentiating capabilities of field programmable gate arrays (FPGAs). While they have been studied extensively in academic literature, they find limited use in deployed systems. We review FPGA reconfiguration, looking at architectures built for the purpose, and the properties of modern commercial architectures. We then investigate design flows, and identify the key challenges in making reconfigurable FPGA systems easier to design. Finally, we look at applications where reconfiguration has found use, as well as proposing new areas where this capability places FPGAs in a unique position for adoption.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; *Embedded systems*; • **Hardware → Reconfigurable logic and FPGAs**; *Reconfigurable logic applications*; *Methodologies for EDA*;

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

ACM Reference Format:
KIZHEPPATT VIPIN and SUHAIB A. FAHMY. 2017. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Comput. Surv.* 1, 1, Article 1 (January 2017), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Field programmable gate arrays (FPGAs) have gone from being chips for implementing glue-logic to platforms for implementing advanced mixed software-hardware systems-on-chip (SOCs). As their capabilities and sizes have increased, FPGAs have found use in a wide range of domains, where their reprogrammability offers a distinct advantage over fixed application specific integrated circuit (ASIC) implementations. This capability allows hardware designs to be upgraded or re-purposed after deployment. An even more differentiating feature of FPGAs is their dynamic programmability, whereby their function is changed at runtime in response to application requirements. FPGAs have also supported *partial* reconfiguration, where only parts of the hardware are modified at runtime, for over a decade. However, though the reconfigurable computing community has demonstrated the effectiveness of these features, they have failed to find favour with a more general audience, due to a combination of architectural, design, and implementation challenges.

While there have been a number of surveys on reconfigurable computing generally [Compton and Hauck 2002; Todman et al. 2005], dynamic and partial reconfiguration are only touched upon briefly. Some previous work discuss tools for partial reconfiguration developed by specific research

Authors' addresses: KIZHEPPATT VIPIN, School of Engineering, Nazarbayev University, Astana, Kazakhstan, vipin.kizheppatt@nu.edu.kz; SUHAIB A. FAHMY, School of Engineering, University of Warwick, United Kingdom, s.fahmy@warwick.ac.uk.

0360-0300/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

groups [Koch et al. 2012; Platzner et al. 2010], but does not survey the large body of work in this area. This survey is an attempt to bring together the wide body of work in the specific area of dynamic and partial reconfiguration from the perspectives of architectures, tools, and applications, with a detailed discussion of efforts to date, and key research challenges standing in the way of widespread adoption. A detailed survey on these topics serves as a valuable foundation for further research in this area, as the first examples of general use in the design approach taken by modern accelerator platforms emerge.

1.1 Background and Motivation

Conceptually all FPGA devices can be considered as being composed of two distinct layers: the configuration memory layer and the hardware logic layer [Becker et al. 2007], as shown in Fig. 1(a). FPGAs achieve their unique re-programmability and flexibility due to this composition. The hardware logic layer contains the computational hardware resources, including lookup tables (LUTs), flip-flops, digital signal processing (DSP) blocks, memory blocks, transceivers, and others. This layer also contains the routing resources and switch boxes that allow components to be connected to form a circuit.

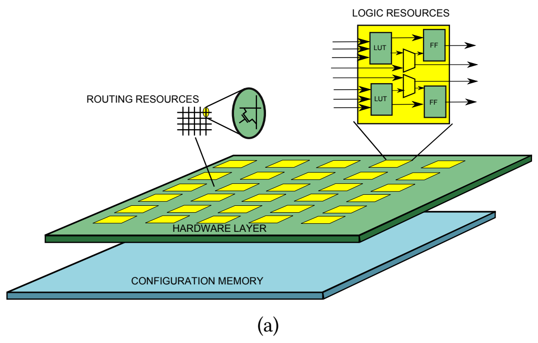
The configuration memory layer stores the FPGA configuration information through a binary file called a *configuration file* or *bitstream*. This binary file contains all the information that determines the implemented circuit, such as the values stored in the LUTs, initial set and reset status of flip-flops, initialisation values for memories, voltage standards of the input and output pins, and routing information for the programmable interconnect to enable the resources to form the described circuit. The function implemented by the hardware logic layer is thus wholly determined by the values stored in the configuration memory.

Most modern devices have SRAM based configuration memory and are hence volatile. To change the circuit implemented in the FPGA, a user modifies the contents of the configuration memory by loading a new bitstream. This operation is called FPGA *configuration/reconfiguration* and is generally performed through external FPGA interfaces such as JTAG, or SelectMap (on Xilinx devices) [Peattie 2009]. The entire configuration memory is reloaded and the FPGA remains inactive/inaccessible during this period. FPGAs built using non volatile technologies are not designed to support such dynamic loading of the configuration memory.

Partial reconfiguration (PR) refers to the modification of one or more *portions* of the FPGA logic while the remaining portions are not altered. Although the terms dynamic reconfiguration and partial reconfiguration have been frequently used interchangeably in the literature, they can be different. The PR operation can be static or dynamic, meaning that the reconfiguration operation can occur while the FPGA logic is in a reset state (static) or running (dynamic). It is also not necessary that all dynamic reconfigurations are partial in nature. For example in *context switching FPGAs*, the whole configuration is changed during reconfiguration, but the operation is dynamic. PR is supported through external FPGA interfaces as well as special internal interfaces such as the Internal Configuration Access Port (ICAP) on Xilinx devices [Xilinx Inc. 2010].

1.2 Advantages of Partial Reconfiguration

PR can bring several advantages to FPGA designs. First, the effective logic density of the chip can be increased by time-multiplexing hardware resources between mutually exclusive computations, thereby allowing a larger application to be contained on a smaller chip. PR also has the benefit of reduced reconfiguration time compared to a full reconfiguration, since this time is directly proportional to the size of the configuration file which in turn is proportional to the area of the chip being reconfigured. This means reconfiguration can be applied in systems with time-critical



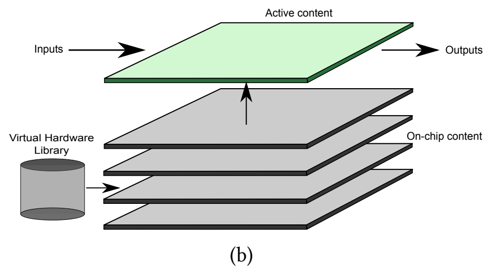


Fig. 1. (a) Typical FPGA architecture composed of configuration memory and hardware logic layer. (b) Multi-Context FPGAs increase effective logic capacity by using more than one configuration memory plane.

requirements. PR is beneficial in adaptive hardware systems, as they can adapt computation to a changing environment while continuing to process data.

PR is also useful in scenarios where an interface is required to persist while functionality changes. Consider an FPGA system interfaced with a host computer via PCI Express (PCIe). A full reconfiguration of the FPGA breaks the communication link, which may even require a host reboot to re-establish. PR allows the link to be maintained by keeping the interface circuitry active while the accelerator portion undergoes reconfiguration. PR also has the benefit of reduced external memory footprint for configuration files since partial configuration files are smaller than full configuration files. This can be especially beneficial for embedded systems with constraints on size, cost, and power consumption.

1.3 Desired Features of a PR Platform

In considering the state of research in dynamic and partial reconfiguration, we set out a set of desired features that would make the adoption of PR more widespread. These include aspects of device architecture, design tool support, and run-time system management. An important architectural property is the supported **granularity** for reconfiguration. Designers may want to dynamically modify portions from as small as a single LUT up to the entire chip. A large granularity increases the overhead of the partially reconfigurable area on the chip, while a fine granularity supports flexbility, but may entail significant architectural cost. Another beneficial feature is support for **run-time relocation**. This allows the same bitstream to be used to configure a circuit in different locations on the FPGA, much like the virtual to physical memory mapping in software environments. **Reconfiguration time** should be negligible for an ideal PR system. Long reconfiguration times can outweigh the other advantages provided by PR, since even if the rest of the system is functioning, waiting for an accelerator to load limits the performance benefit of the accelerator. Faster reconfiguration would allow accelerators to be loaded and unloaded as fast as task switching in multi-core processors. The reconfiguration operation should be **transparent** to the application, such that the system continues with useful work, while reconfiguration occurs, and that code managing the reconfiguration is not concerned with implementation details. It is desirable to have a **high-level design** tool that automates the mapping of an adaptive application description at the system level to a specific PR implementation, without the need for low level architecture understanding.

In the subsequent sections we analyse the extent to which FPGA platforms and development tools support these features as well as discuss applications that exploit the advantages of PR.

2 ARCHITECTURES

Dynamic reconfiguration was initially proposed to increase effective logic capacity and reduce reconfiguration time. Early on, the limited resource availability in FPGAs was a major constraint when implementing large designs. Fetching configuration files from external memory to reconfigure over the (external) configuration ports also resulted in slow reconfiguration. Early dynamically reconfigurable architectures overcame these issues by increasing the number of configuration contexts, allowing much faster reconfiguration, and effectively increasing logic capacity, as shown in Fig. 1(b). These devices were referred to as Context-Switching FPGAs or Multi-Context FPGAs (MC-FPGAs) [Chong et al. 2005].

For modern FPGAs with multi-million gate logic capacity, lack of resources is no longer the primary motivation for PR. New driving factors include sharing a single physical device among multiple users, keeping communication links alive during system reconfiguration, and adaptive applications with varying computational requirements.

2.1 Academic and Non-Commercial Architectures

The development of dynamically reconfigurable architectures dates back to 1995, when Xilinx filed a patent for an FPGA which can store multiple configurations simultaneously [Ong 1995]. In the initial design, there were two configuration memory arrays available in the FPGA which could store different configuration data. During alternate halves of a clock cycle, switches at the output of the configuration memory cells would select the configuration data stored in the first or second half of the configuration memory array and intermediate results would be stored in data latches. At the end of every other cycle, the FPGA would output the results of its function.

This idea was further extended in 1997, with a time multiplexed FPGA based on the Xilinx XC4000E product family [Trimberger et al. 1997]. Although combinational logic could be multiplexed through several configuration contexts, state storage could not. This work used micro registers to store the outputs of LUTs and flip-flops, with eight configurations supported. Reconfiguration could be performed in a single clock cycle, taking about 5 ns. An inactive configuration plane could be modified at runtime by loading configuration data from off-chip storage. Through a special “RAM” mode user designs could directly access configuration memory, allowing self-modifying hardware. Hence, MC-FPGAs supported dynamic reconfiguration, but the granularity of reconfiguration was the entire device.

The main drawback of MC-FPGA architectures was their high power consumption. Due to the large number of configuration bits and high switching activity, the power consumption of these devices was in the tens of Watts, making them unsuitable for many applications. Chong et al. proposed the reconfigurable context memory (RCM) to tackle this issue [Chong et al. 2005]. RCM exploits the redundancy and regularity in configuration bits between different contexts, based on a previous study that showed that during context switching, less than 3% of the configuration data was modified [Kennedy 2003]. Additionally, ferroelectric functional pass-gates are used in RCM to achieve compactness and lower power. This design claimed to reduce FPGA area to 37% of other MC-FPGAs and consume much less power.

Another major hurdle to the adoption of MC-FPGAs was the lack of design tools that could efficiently map to these platforms. Designs had to be manually partitioned into multiple segments and mapped to different contexts. Advances in EDA have renewed interest in MC-FPGAs recently as discussed in Section 2.2.

Another early architecture proposed to support dynamic reconfiguration was the *Dynamically Programmable Gate Array* (DPGA) [Tau et al. 1995]. Its architecture closely resembles that of an MC-FPGA but in this case, each LUT and interconnect cell had an associated 4-context memory

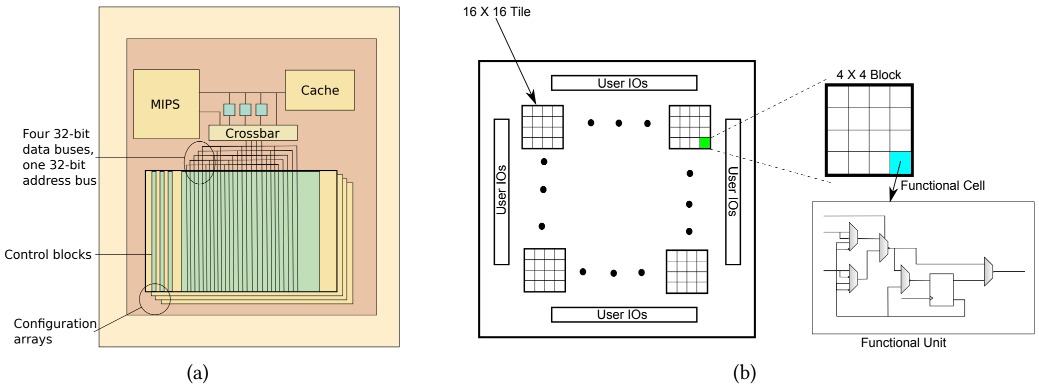


Fig. 2. (a) GARP architecture with processor and reconfigurable fabric (b) Xilinx XC6200 architecture.

implemented using DRAM. The motivation was to overcome slow off-chip configuration loading which would take several milliseconds to complete. DPGAs supported different usage models with multiple independent functions in different configurations [DeHon 1996]. The prototypes developed had limited logic capacity, low operating frequency, and a lack of tools. Using DRAM for the configuration memory also enforced a minimum operating frequency of 5 MHz due to DRAM refresh requirements.

More recently researchers have proposed a new architecture for FPGAs with a single configuration plane that can support run-time relocation through PR [Huriaux et al. 2014]. Due to their heterogeneous architecture, run-time circuit relocation is difficult in modern FPGAs as discussed in Section 2.2. In this proposed architecture, the hardware layer is logically partitioned into two (but with a single configuration plane). The first layer, called the homogeneous plane, contains traditional logic elements such as LUTs and flip-flops and associated routing resources, while the second layer, called the heterogeneous layer, contains heterogeneous resources such as memory blocks and signal processing blocks and associated routing. Heterogeneous-only long lines are connected to the homogeneous routing network through switch boxes at horizontal and vertical channel intersections, allowing a circuit to be moved horizontally at run-time. The drawback of this architecture is that it leads to an increase in horizontal delay for routing in the homogeneous plane, and since the positioning of heterogeneous function blocks is not known at place and route time, it is assumed that such a block is physically located in a position that can be routed to.

GARP was a dynamically reconfigurable architecture, that combined reconfigurable hardware with a standard MIPS processor [Hauser and Wawrzynek 1997]. The reconfigurable fabric was a slave compute unit located on the same die as the processor as shown in Fig. 2(a). Loading and execution on the reconfigurable array was controlled by a programme running on the processor, and the standard memory hierarchy of the processor was also accessible to the reconfigurable fabric. Each reconfigurable array was divided into blocks with one block used as control block, and the others as logic blocks. GARP allowed partial array configuration down to individual blocks. A physical implementation of GARP was never made available for practical use. In Section 2.2 we discuss more recent devices that closely resemble GARP.

Recently, work on FPGA overlay architectures has gained some attention in the academic community. This consists of building a coarse grained architecture on top of an FPGA and targeting that through design tools [Capalija and Abdelrahman 2013; Jain et al. 2016b]. The architecture is designed to support a particular domain, and the interconnect can be tailored to the domain to make it more efficient [Jain et al. 2016a]. A key benefit cited by researchers working on overlays is significantly reduced reconfiguration time [Stitt and Coole 2011], since a small number of registers controlling the coarse grained functional units and routing need to be set to modify the computed function, rather than a PR bitstream that is setting configurations at the bit-level. The main architectural limitation of overlays is that they can entail significant area and timing overheads and are not as flexible as using the fine-grained FPGA architecture, though it has been suggested that PR can be used to switch between a variety of different overlays to overcome this [Coole and Stitt 2015].

2.2 Commercial Devices Supporting PR

Presently the only two FPGA vendors commercially supporting PR are Xilinx and Altera (now part of Intel).

2.2.1 **Xilinx.** Among the major vendors, Xilinx's FPGAs have supported PR for two decades, and are hence the most popular devices for these applications. The first Xilinx FPGA to support dynamic partial reconfiguration was the XC6200 series [Xilinx Inc. 1996]. This device contained only a single configurable memory plane and had a tiled architecture with each tile divided into a number of cells containing *functional cells*. The functional cells were composed of 2:1 multiplexers for combinational logic, a flip-flop, and routing resources. Using a special interface, an external processor could access any specific functional cell in the FPGA (Fig. 2(b)), and modify its configuration, with the configuration SRAM mapped to the processor address space. Due to a regular structure with every cell and its associated routing being similar, reconfiguration was simpler with these devices than for modern ones. Run-time circuit relocation was also possible with such architectures.

PR became more popular with the introduction of the Virtex-II [Xilinx Inc. 2003] and Virtex-II Pro [Xilinx Inc. 2011a] series of FPGAs from Xilinx. In these devices FPGA primitives are arranged in a columnar fashion. These primitives include configurable logic blocks (CLBs), Block RAMs, and multipliers. CLBs are the basic logic elements in Xilinx FPGAs, composed of LUTs and flip-flops in two *slices*. The number of LUTs and flip-flops in a slice is device family dependent. A configuration binary file (partial bitstream in Xilinx terminology) can be loaded externally using the SelectMap or JTAG interfaces. In Virtex devices, Xilinx introduced a new configuration interface called the Internal Configuration Access Port (ICAP). This made it possible to load bitstreams from within the FPGA fabric. A soft-processor or a custom state machine could fetch configuration information from external memory and write to the configuration memory through the ICAP, thereby allowing a circuit implemented on the FPGA to modify itself autonomously.

In these devices, the configuration memory is organised in *frames* which are 1-bit wide and extend the whole height of the device – hence the size of a frame is device dependent [Xilinx Inc. 2004a] . A frame does not map to any single hardware resource, but it configures a narrow vertical slice of many physical resources. Configuration frames are grouped into six different configuration columns depending upon their hardware-mapping, called IOB, IOI, CLB, GCLK, BlockRAM, and BlockRAM Interconnect. IOB columns configure the voltage standard, internal pull-up, and other options for the I/O interfaces. CLB columns program the configurable logic blocks, routing, and most interconnect. BlockRAM columns program the small internal memory blocks.

For Virtex devices, it is necessary to designate at design time the portions of the FPGA that will undergo PR. These regions are called partially reconfigurable regions (PRRs) and are composed of several frames. There are several restrictions on the size and shape of PRRs: they must extend the full height of the device and must align horizontally with a multiple of four slices. These restrictions can make a design inefficient in terms of hardware utilisation, but floorplanning is relatively simple. Since PRRs extend the full device height, floorplanning is only concerned with the width of these

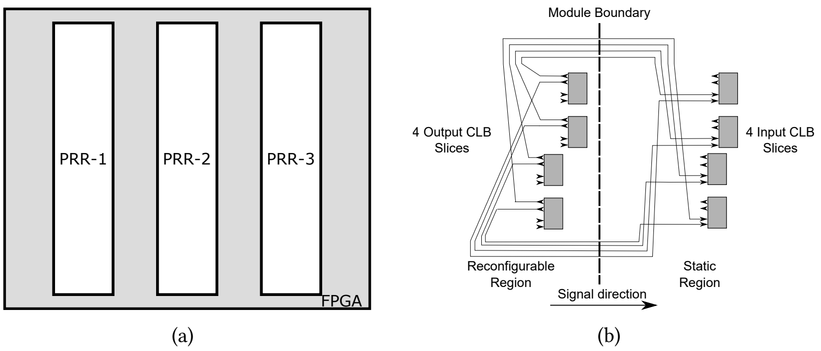


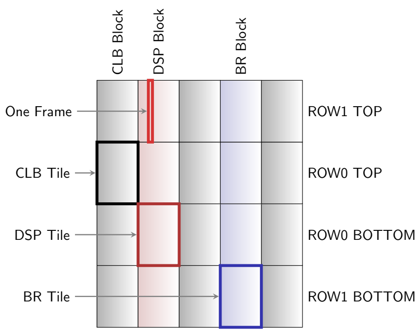
Fig. 3. (a) Floorplanning of Virtex-II device showing PR regions. (b) A bus macro showing the connectivity between the static region and a reconfigurable region. The CLB slices to the left of the module boundary are implemented in the reconfigurable region and those to the right of are implemented in the static region.

regions and so they are more like vertical *slots* as shown in Fig. 3(a). Runtime circuit relocation is still relatively easy for such architectures as shown by the *Erlangen Slot Machine* [Majer et al. 2007].

Since a single PRR hosts multiple circuits at run-time in a time multiplexed manner, every circuit targeted for the same PRR must have a similar interface to the static (non-PR) region. In order to fix the routing between the static and PR regions, special anchoring logic is necessary. Virtex-II and Virtex-II Pro devices use internal tri-state buffers (TBUFs) to manage this connectivity. To support run-time circuit relocation, the relative positions of these TBUFs must also match for different PRRs. The number of TBUFs available on these devices is restricted and their positions fixed, leading to further restrictions on the size and positions of PRRs.

The Virtex-4 family of FPGAs [Xilinx Inc. 2008] incorporated architectural improvements over the Virtex-II with better support for PR. TBUFs were replaced by bus macros [Lysaght et al. 2006], which are constructed out of LUTs as shown in Fig. 3(b). Using LUTs instead of TBUFs for fixing routing between regions was initially demonstrated by researchers on Virtex-II FPGAs [Huebner et al. 2004]. Since these could be placed anywhere, as opposed to the fixed locations of TBUFs in the Virtex-II, this allowed for a more flexible arrangement of connectivity. The size of frames was also reduced in the Virtex-4 [Xilinx Inc. 2008]. Unlike the Virtex-II, where frame size is dependent on device size, it is constant for all Virtex-4 devices. Each frame is 1 bit wide and 16 CLBs high and contains forty-one 32-bit words (1312 bits). The reconfigurable region also no longer needs to span the full height of the device, but rather must be a height that is a multiple of 16 CLBs. Because of this modified architecture, the floorplanning problem is no longer one dimensional but two dimensional. This has made run-time relocation more difficult since relocation is possible only between two PRRs with exactly the same dimensions and resource arrangement. The ICAP interface width was also increased from 8 to 32 bits, considerably improving reconfiguration throughput.

In the Virtex-5 architecture, the entire device is divided into several rows and columns as shown in Fig. 4(a). A row essentially represents a clock region and device size determines how many there are. The columns, called *blocks*, span the entire device height. Each block contains a single type of FPGA primitive arranged in a columnar fashion. The FPGA is composed of several *tiles* where a block and a row intersect: CLB tiles, DSP tiles, and BRAM tiles. Xilinx uses the term *reconfigurable frame* to denote these tiles and these are the basic unit for PR. One CLB tile contains 20 CLBs, one DSP tile contains 8 DSP slices, and one BRAM tile contains 4 Block RAMs. Virtex-6 FPGAs follow the basic architecture of Virtex-5 FPGAs with a CLB tile containing 40 CLBs, a DSP tile containing 8 DSP slices, and a BRAM tile containing 8 18Kbit Block RAMs. Xilinx 7-series FPGAs (Artix, Kintex



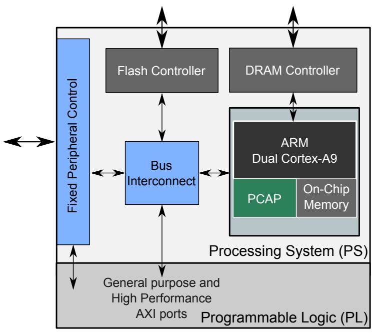


Fig. 4. (a) Xilinx Virtex FPGA architecture (b) Zynq SoC Architecture

and Virtex-7) also have a similar tile architecture with one CLB tile containing 50 CLBs, and DSP and BRAM tiles containing 10 DSP slices and 10 18Kbits Block RAMs respectively.

These improved architecture features enable FPGAs to implement more complex circuits as well as to reduce resource wastage. Designers are now able to define multiple PRRs with varying sizes with different kinds of resources. On the other hand, all these architectural improvements make run-time circuit (bitstream) relocation almost impossible using vendor provided tools.

Xilinx supports PR on newer hybrid reconfigurable devices, such as the Xilinx Zynq-7000 SoC too. The Zynq architecture [Xilinx Inc. 2013a] couples a powerful ARM Cortex A9 processor, standard communication infrastructure, and an integrated reconfigurable fabric, as shown in Fig. 4(b). The ARM processor communicates with on-chip memory, memory controllers, and peripheral blocks through Advanced eXtensible Interface(AXI) interconnect. Together, these hardened blocks constitute the Processor System (PS). The on-chip PS is attached to the Programmable Logic (PL) through multiple AXI ports, offering high bandwidth between the two key components of the architecture. The PS processor configuration access port (PCAP) supports full and partial (re)configuration of the PL from the PS. The reconfigurable fabric of the Zynq uses the 7-series FPGA architecture which can also be partially reconfigured through an ICAP interface within the PL. Thus the Zynq architecture bears some similarity to the GARP architecture discussed earlier.

The latest Xilinx Ultrascale and Ultrascale+ families of FPGAs also support PR. The major improvements in these devices are the capability to partially reconfigure resources such as PLLs, input/output buffers, and high-speed transceivers, which was not possible previously [Xilinx Inc. 2016]. These devices introduce a new configuration access port called the media configuration access port (MCAP), which is connected to one of the PCIe hardmacros [Xilinx Inc. 2015]. These improvements come at the cost of additional reconfiguration time overhead. For Ultrascale devices, the reconfiguration process is now composed of two stages. Before loading a new partial bitstream, the corresponding PR region must be *cleaned* using a small bitstream. Each PRR requires a separate *cleaning partial bitstream*, but the size of this is only about 10% of a normal partial bitstream.

2.2.2 **Altera.** Altera (now part of Intel) recently began supporting PR on their Stratix-V, Cyclone-V and Arria-10 series FPGAs. Adaptive logic modules (ALMs) are the basic building blocks in Altera FPGAs, containing a fracturable LUT with 8 inputs, 4 flip-flops (on the Stratix-V), and auxiliary circuits such as adders and multiplexers. Multiple ALMs are combined to form logic array blocks (LABs), which are arranged in a columnar fashion in the device. Columns of memory blocks and variable precision DSP blocks are also present for efficient circuit implementation. Partial reconfiguration is supported for logic elements, DSP slices, memory blocks, and routing resources.

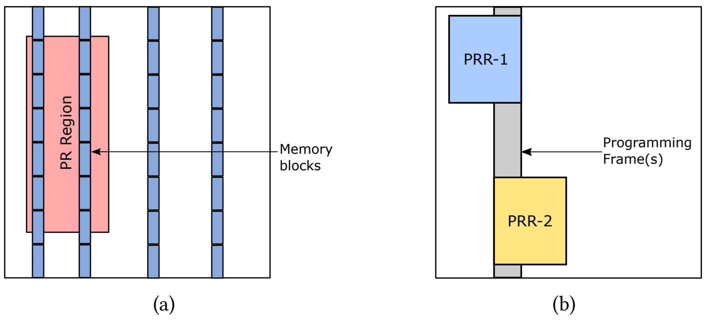


Fig. 5. (a) A reconfigurable region in a Stratix-V FPGA with PR region not extending the full FPGA height (b) Two PR regions sharing the same programming frames

Other primitives such as PLLs and transceivers support only dynamic configuration (not using reconfiguration frames) through a special reconfiguration port tied to these primitives.

The Stratix-V architecture is similar to that of the Xilinx Virtex FPGAs, with *programming frames* being the unit of reconfiguration [Altera 2013a]. Similar to the Xilinx Virtex-II, the FPGA is divided into multiple columns but only a single row. This results in additional restrictions when a PR region does not span the full device height and contains memory blocks as shown in Fig. 5(a). During a partial reconfiguration operation, the contents of memory blocks outside the PR region but in the same columns are also reconfigured. To avoid this issue, PR regions should span the entire device height or memory blocks above and below the PR regions should not be used by static logic or other PR regions. Altera also uses LUT-based proxy logic for preserving routing between PR regions and the static region.

Two different PR implementation schemes are possible, depending on the arrangement of reconfigurable regions. The *SCRUB* mode is used when programming frames are not shared between PR regions. In this mode, the unchanged configuration bits of the static region are *scrubbed* back to their present values. All configuration bits corresponding to PRRs are overwritten with new data irrespective of what was previously contained in the region(s).

The two-pass AND/OR reconfiguration scheme is used when configuration frames are shared among multiple PRRs as shown in Fig. 5(b). In the first pass, all the bits in the programming frame for a column passing through a PRR are ANDed with 0s while those outside the region are ANDed with 1s. In the second pass, for each frame, new data is ORed with the current value of 0 in the PR region, and in the static region, bits are ORed with 0s. The main drawback is that the bitstream size of a PR region using the AND/OR scheme can be twice the size of one using SCRUB mode since each frame is written twice. Furthermore, to individually configure PRRs when regions share programming frames, multiple variations of bitstreams equal to the Cartesian product of variants of PR logic (called personas) are required. Since in Xilinx FPGAs, configuration frames do not extend the full device height, this limitation exists to a more limited extent as PR boundaries are drawn along device rows. Since the Altera PR flow is still new, we may see similar improvements to those seen in the Xilinx flow in the coming years.

The new Arria 10 and Arria 10 SoC devices also support PR. The Arria 10 SoC has a similar architecture to the Xilinx Zynq SoC with an ARM processor system integrated with an FPGA fabric. For these devices, the FPGA fabric architecture and implementation schemes supported are same as for Stratix V FPGAs. Altera uses a special IP block called the Partial Reconfiguration IP (PR-IP) to send partial bitstream data into the configuration memory from external hosts as well as from the internal PR controller [Altera 2017]. This block supports data widths from 1 bit to 32 bits. It is also possible to partially reconfigure these devices through a PCIe interface by interfacing a PR-IP to the PCIe hard macro [Altera 2016a].

The recently announced Altera Stratix 10 architecture includes further architectural changes that benefit PR. The overall FPGA architecture is divided into multiple sectors, with each sector having its own configuration memory and reconfiguration infrastructure [How and Atsatt 2016]. Reconfiguration is managed through small processors in each sector called secure digital managers (SDMs), and the bitstream format for each sector is identical. This presents an exciting development as it opens the door to making bitstreams relocatable within an FPGA more easily, and also allows for higher reconfiguration bandwidth with the prospect of configuration data being broadcast across multiple sectors. However, these devices are yet to reach market, and the supported flows do not currently exploit these possibilities.

2.2.3 **Other Vendors.** Other FPGA vendors such as National Semiconductor, Lattice Semiconductor, and Atmel previously supported PR on their FPGA devices. However, this is no longer the case, partly due to the limited adoption of this technique for practical applications and partly due to the challenge in providing a robust supported toolflow. We will however explore how these alternative architectures integrated PR.

Lattice Semiconductor produced the ORCA series of FPGAs to support PR [Lattice Corp. 2003]. These were coarse-grained FPGA with a grid of programmable logic cells (PLCs), programmable I/Os, and embedded RAMs (EBRs). Each PLC consisted of a programmable functional unit (PFU), system level interconnect (SLIC), and routing resources. PR was done by setting a bitstream option in the previous configuration sequence that would tell the FPGA not to reset all of the configuration RAM during a reconfiguration. Then only the configuration frames to be modified would be rewritten. Here the reconfiguration was partial but static in nature.

The AT40K series of FPGAs from Atmel supported both partial as well as dynamic reconfiguration [Atmel 2013] . The AT40K architecture was a symmetrical array of identical cells except for bus repeaters spaced between every four cells. At the intersection of each repeater row and column there was a 32×4 RAM block accessible by adjacent buses. The FPGA configuration memory was viewed as a simple memory-mapped address-space and the user had full read/write access to it. These FPGAs were ideal for building adaptive filters, variable coefficient multipliers, etc., but unsuitable for logic intensive applications due to limited logic capacity.

The National Semiconductor CLAy FPGA family contained a 56×56 array of fine grained logic cells [National 1993]. The CLAy logic cell (up to 3 inputs, 2 outputs) implemented a set of simple logic functions like NOR, AND, NAND, OR, XOR, INV, MUX, Flip Flop and complex functions using combinations of these [Gokhale and Gomersall 1997]. Each cell had 2 direct connections to each of the four nearest neighbours, and connections to horizontal and vertical local buses, and each row could be partially reconfigured using a host processor.

Tabula also produced programmable logic devices that used a technique they called Spacetime technology [Tabula 2010], similar to the multi-context FPGAs discussed earlier. Logic, memory, and interconnect resources were dynamically reconfigured up to eight times in each user cycle. The Spacetime compiler automatically mapped, placed, and routed a user design into the device using standard VHDL/Verilog inputs and flows. A major limiting factor of previous context-switching FPGAs was their power consumption. Tabula claimed to have overcome this through new manufacturing techniques, however, exact power consumption measurements for these devices were never published. Tabula ceased operation in March 2015 [Lipsky 2015].

Table 1. Architectural features of commercial FPGAs supporting PR.

Architecture	PR Granularity	Circuit Relocation	PR Primitive
Xilinx Virtex-5/6/7	One clock region high	Very difficult	ICAP
Xilinx Zynq	One clock region high	Very difficult	ICAP/PCAP
Xilinx Ultrascale	One CLB	Very difficult	ICAP/MCAP
Altera Stratix-V/Arria-10	One ALM	Difficult	PR-IP
Altera Stratix-10	One sector	Easy	SDM

2.3 Summary

Table 1 compares the architectural features of some of the commercially available FPGAs supporting PR. There remain several research opportunities related to FPGA architecture design for PR. In modern commercial FPGAs, PR is an auxiliary feature rather than something around which the architecture is designed. This means many aspects of PR design are tied to low-level architecture details requiring significant expertise. Limitations on how the configuration memory is accessed (e.g. only a single active port) limit reconfiguration throughput and prevent parallel PR. Alternative approaches to organising the reconfiguration memory and how it is configured could make PR more effective and easier to design for. A fine configuration granularity could lead to higher reconfiguration time and too coarse a granularity results in resource wastage. More advanced architectural PR support could also offer significant power benefits through run time adaptation of resource usage. The new Altera sector-based architecture could address a number of these issues if access to its features is provided. Currently no commercial coarse grained FPGA architectures are available which support PR. Commercial architectures have improved in their support of PR, and coupled with more advanced tools, this can lead to wider adoption in the long term.

3 DESIGN, IMPLEMENTATION, AND SIMULATION TOOLS

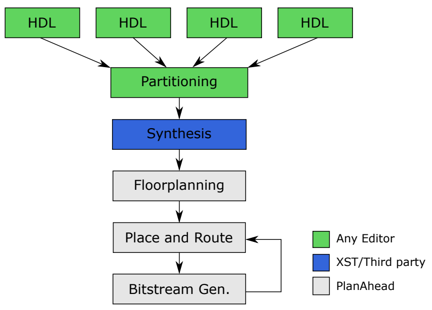
In this section we review design, implementation, and simulation tools for PR systems. We discuss the steps involved in converting a design specification into a working hardware implementation. Widespread adoption of PR will depend upon the existence of effective tool chains that offer a high level view to the designer, while incorporating low-level architecture understanding. We review approaches from both industry and the research community.

3.1 Vendor PR Design Flows

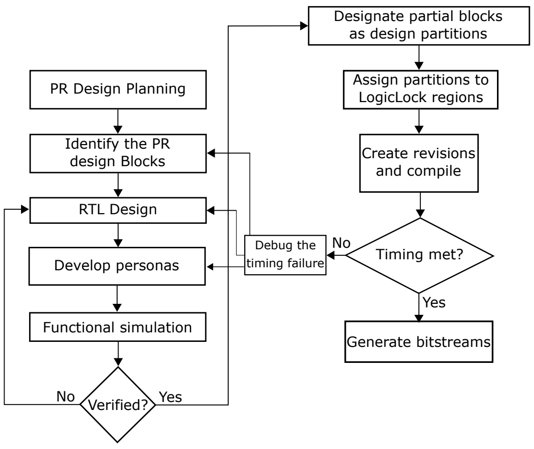
The tool flows offered by both vendors, Xilinx and Altera, are similar with slight differences due to architectural variations; both require a designer who is proficient in low-level FPGA architecture.

3.1.1 **Xilinx PlanAhead PR Flow.** Xilinx initially offered a difference-based partial reconfiguration flow [Eto 2007]. This allowed minor changes, by editing an already placed and routed design using the FPGA Editor software available as part of the Xilinx ISE software suite. Implementation tools would then generate a partial bitstream containing only the difference between the new and old designs. This flow was not scalable to large circuit changes and is no longer supported.

Xilinx later supported PR through a hierarchical module-based design tool called PlanAhead [Xilinx Inc. 2013b], with the main steps required shown in Fig. 6(a). Each PR design is composed of a number of *modules*, or functional units. All modules are described using a hardware description language (HDL) or can be pre-synthesised netlists. The hardware design is composed of two parts, the *static region* and one or more *reconfigurable regions* (PRRs). PRRs may contain LUTs, BRAMs,



(a)


PR Design Planning
Identify the PR design Blocks
RTL Design
Develop personas
Functional simulation
Verified?
No
Yes
Designate partial blocks as design partitions
Assign partitions to LogicLock regions
Create revisions and compile
Timing met?
No
Yes
Debug the timing failure
Generate bitstreams

(b)

Fig. 6. (a) Xilinx PlanAhead Partial Reconfiguration Flow, (b) Altera Quartus Partial Reconfiguration Flow

and DSP slices but cannot contain clock modifying logic such as PLLs and clock buffers. The static region is the portion of the design, which does not change its functionality during system operation. This usually contains a processor running the reconfiguration management software, internal configuration interface, and memory interface modules. PRRs implement the reconfigurable modules, and can be reconfigured at runtime. A single reconfigurable region can implement many modules in a time multiplexed fashion; all reconfigurable modules implemented in the same PRR constitute a *reconfigurable partition*.

The first design step is to decide on the number of reconfigurable regions and corresponding module allocation to them (partitioning). Each individual module is synthesised to generate a corresponding netlist. Floorplanning must then be performed manually to specify the locations and bounding boxes of PRRs in the FPGA fabric. These regions must be rectangular in shape and should be aligned to clock region boundaries (tiles). Floorplanning details are stored in the *user constraints file* (UCF) for incorporation in the implementation stage. The designer must have expertise in low-level architecture details to efficiently implement the system.

The designer must then determine the valid combinations of modules assigned to the PRRs, to make up the overall modes of the system; each valid combination is called a *configuration*. During implementation, the static region is implemented only once, with the first configuration used as a placeholder, and the final placement and routing of the static region are preserved for all other configurations. Logic implemented in the static region can use the routing resources (but not LUTs or flip-flops) available in the PRRs but not vice versa. If a reconfigurable module were to use routing resources in the static region, that would cause glitches during reconfiguration. Bus macros play an important role here in interfacing the static region and PRRs as discussed in Section 2.2. The tool automatically inserts them and designers have no control over their location.

Finally, the tool generates a full reconfiguration bitstream (configuration file) as well as partial bitstreams for each PRR, for each configuration. This results in full bitstreams for each configuration and partial bitstreams corresponding to the Cartesian product of modules assigned to each region. At run-time, the FPGA is initially configured using one of the full bitstreams and later any single PRR can be reconfigured using a partial bitstream.

3.1.2 **Xilinx Vivado PR Flow.** From 7-series FPGAs onwards, Xilinx supports PR through the Vivado Design Suite [Xilinx Inc. 2014]. This flow is very similar to PlanAhead, but not yet fully integrated with the GUI-based project flow. The designs are implemented using the Vivado Tcl based command flow or using a combination of Tcl commands and the GUI.

The first step is again to synthesis the static and reconfigurable modules separately using Xilinx tools or third-party synthesis tools. A reconfiguration controller (ICAP controller) should be included in the design if the target FPGA is not a hybrid device such as Zynq SoC. For SoC devices instantiating the ICAP is optional since PR is supported through the PCAP interface in the processor system. Floorplanning restrictions are the same as for PlanAhead but the 7-series FPGAs enforce an additional restriction that partition boundaries should not cross *interconnect tiles*. Interconnect tiles are special resources that manage routing between different resource columns. One major improvement in Vivado is in the implementation of anchor logic. Unlike previous tools, Vivado does not use LUT-based bus macros but rather directly uses interconnect tiles, which are dedicated routing resources [Xilinx Inc. 2017c]. This helps improve routing efficiency and thus timing performance, but makes run-time bitstream relocation even more difficult.

Since the Xilinx flows allow static region wires to pass through PRRs, any minor modification of the static logic requires complete reimplementation of the static region and all PRRs.Using routing resources in the PRRs can also cause routing congestion for subsequent configuration implementation. Meanwhile, restricting the static region from using PRR routing resources could adversely affect overall system timing performance.

3.1.3 **Altera PR Flow.**

The Altera PR flow is supported through Quartus-II and the newer Quartus Prime design software [Altera 2013b, 2016b]. This flow is similar to the PlanAhead flow with different nomenclature. Altera refers to configuration frames as *programming frames* and calls configurations *revisions*. Module variations implemented in the same PR region are called *personas*.

Altera partial reconfiguration is based on the *revision* feature in the Quartus software. The initial revision is the base revision, where the boundaries of the static region and PRRs are defined. From the base revision, multiple revisions can be created. The PRR boundaries are fixed using *LogicLock* assignments in Quartus. LUT-based anchor logic is automatically inserted by the tool to fix the routing between the static and PR regions. Later the incremental compilation flow technique is used to preserve the static region across different revisions. Unlike the Xilinx tool flow, Quartus creates a full configuration file only for the base revision, with only partial configuration files corresponding to each PRR generated for other revisions. Hence, the FPGA can be initially configured only in the base revision.

Altera FPGAs have a set of restrictions when the height of the region is less than the full FPGA as discussed in Section 2.2. There are further restrictions when using LUTs in the PRRs to build memory elements (called LUT-RAMs). LUT-RAMs inside PRRs cannot have an initialisation value when used in AND/OR configuration mode. When AND/OR mode is used for designs without initialised LUT-RAMs, a logic 1 has to be written to all memory locations before reconfiguring the region. Otherwise it causes a configuration error [Altera 2016b]. Altera also allows static routing through PRRs , so any modification to the static region requires complete reimplementation of the static as well as all reconfigurable modules.

3.1.4 **Partial Reconfiguration Support in Vendor OpenCL Software Tool-Chains.**

Newer FPGA platforms from Intel and Xilinx support implementation of higher level designs using OpenCL through their *SDAccel* and *FPGA SDK for OpenCL* tools [Intel 2017b; Xilinx Inc. 2017a]. Xilinx uses PR as a way of implementing OpenCL kernels in the FPGA. The kernels use a predefined interface for data communication, such as AXI4. The compiler compiles these kernels for implementation on a pre-partitioned, pre-floorplanned FPGA. At run-time, the kernels are loaded by programming the FPGA over the PCIe interface (which is in the static region). The advantage of these tool flows are that they abstract the integration of user-designed accelerators with the host system. A single

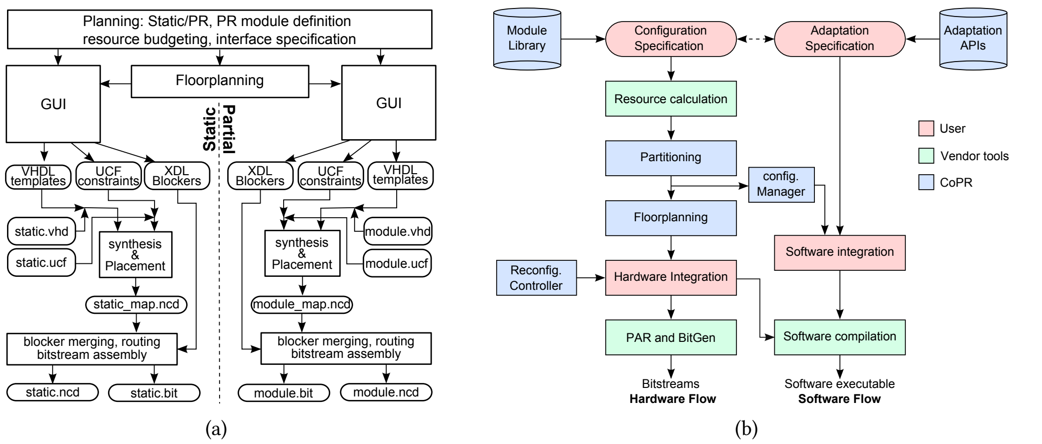


Fig. 7. (a) GoAhead PR tool flow [Beckhoff et al. 2012]. (b) CoPR for Zynq tool flow showing steps performed by the user, vendor tools, and the CoPR framework.

PRR is used for kernel implementation, removing the complexity of floorplanning multiple PRRs. However, this restricts the granularity to a single monolithic "mode" for the system, limiting the benefits of using PR in an adaptive system.

3.2 Academic PR Development Tools

In this section we discuss some academic tools developed to support PR. Most of these tools target Xilinx FPGAs and many use vendor tools for low-level device dependent operations such as placement and routing and configuration file generation.

3.2.1 **OpenPR Tool Flow.** OpenPR is functionally close to the Xilinx PR design flow [Sohanghpurwala et al. 2011]. It relies upon the logic and wiring database and bitstream manipulation capabilities provided by an open-source FPGA development tool called *Torc* [Steiner et al. 2011]. The designer initially creates an XML project file, specifying the design name, static design file system path, path to the constraints file (UCF), target device name, etc. The Xilinx PlanAhead tool is then used to manually floorplan the reconfigurable regions. OpenPR then generates the static design by generating placement constraints, generating blocker routes to prevent the static region from using routing resources in the PR regions, and merges the blockers with the static design. Later, the clock tree routing information from the static design is inserted into the reconfigurable modules. This is done by manipulating the intermediate files generated by the Xilinx implementation tools. Finally, the partial bitstreams are generated with the help of the Xilinx bitstream generation tools.

The major attraction of OpenPR is its availability as an open source development environment. Since it blocks the static region from using routing resources in the PRRs, it allows them to be implemented separately, with changes in the static region not necessitating reimplementation of all PR modules.

3.2.2 **GoAhead Tool Flow.** GoAhead [Beckhoff et al. 2012] also attempts to overcome some of the limitations of the Xilinx incremental PR design flow. It also prevents routing resources in PRRs from being used by the static region, with the aim of supporting module relocation between PRRs.

The overall GoAhead tool flow is shown in Fig. 7(a). The static and reconfigurable modules are implemented through independent design flows. The designer makes an initial plan defining the static parts of the design and the modules that will be reconfigured. Then the design is manually floorplanned using a GUI tool and bounding boxes are drawn around PR regions. GoAhead implements the static portion of the design, while masking the PR regions with *blocker macros* that occupy all wires inside the PR regions, thereby preventing static nets from crossing PR regions. The reconfigurable modules are implemented in a similar fashion, where the blocker macros prevent wires crossing from PR regions into the static region. Finally vendor tools are used to generate partial and full bitstreams from the routed design.

The primary difference between GoAhead and OpenPR is that GoAhead uses *blocker macros* to control clock signals in the PR regions and uses vendor tools to generate the final clock tree. In OpenPR, the tool adds the clock tree routing without using vendor tools. OpenPR and GoAhead can help overcome some of the limitations of the vendor flows, but do not address the high-level/abstract design issues, and hence require FPGA design expertise. GoAhead has recently been supplemented with a feature for automatic floorplanning [Beckhoff et al. 2013]. Both these tools manipulate Xilinx Design Language (XDL) files to manipulate the placement of blocker macros. Dependence on XDL is a problem as it has been discontinued in the Vivado design flow.

3.2.3 **CoPR Tool Flow.** CoPR is an automated PR toolflow specifically targeting the Zynq architecture [Vipin and Fahmy 2015], focused on raising the abstraction level for describing partially reconfigurable applications. Many of the manual operations required in the vendor flow are automated and low-level FPGA architecture dependent details are abstracted from the designer. It also abstracts the runtime management of the reconfiguration process so that the system designer need not be aware of the details of the hardware PR implementation. The overall flow is shown in Fig. 7(b).

The primary designer inputs to CoPR are the *configuration* and *adaptation* specifications. The configuration specification details the different valid system configurations and the corresponding library modules present in each configuration in XML format. The *adaptation specification* contains software code for changing configurations at runtime. Neither of these references any low-level PR features, making CoPR accessible to non-experts.

CoPR first uses the vendor synthesis tool (XST) to synthesise all modules for the target FPGA to determine resource requirements. The partitioning step involves determining the number of reconfigurable regions (PRRs) and allocating modules to them. Later a kernel tessellation approach is used to generate a floorplan, resulting in a *user constraints file* (UCF) that specifies the coordinates of all PRRs. The PR design is then integrated with the ARM processor system in the Zynq with the help of Xilinx XPS software. The low-level implementation and bitstream generation operations are performed using the Xilinx command line tools. The software for managing low-level reconfiguration operations is automatically generated by the tool in C programming language, and later integrated with the high-level adaptation specification using the Xilinx SDK tool-chain. The ARM processor runs Xilinx's Standalone operating system and manages reconfiguration through a custom reconfiguration controller and an associated driver. The CoPR tool flow integrates with Xilinx ISE, XPS, and SDK tools for backend implementation, but is not supported with Vivado.

3.2.4 **PaRAT Tool Flow.** The Partial Reconfiguration Amenability Test (PaRAT) flow [Kumar and Gordon-Ross 2015] attempts to bridge between high-level synthesis (HLS) descriptions and PR implementation. The tool initially analyses Xilinx Vivado HLS code and extracts control and data dependency information to generate a high-level model of the PR system using its PR modelling language (PRML) [Kumar and Gordon-Ross 2013]. This is a directed acyclic graph representation of the system, where nodes model algorithmic constructs and control while edges model control and data dependency behaviour. The graphs are automatically partitioned to determine the number of PRRs needed and the module assignment to them in the form of an XML file. This information can be then used with the Vivado PR toolflow to implement the complete system.

3.2.5 **OSSS+R Framework.** OSSS+R is a SystemC based design methodology enabling algorithmic specification in C/C++, functional simulation, and automated synthesis [Schallenberg et al. 2010, 2009]. The approach uses object-oriented techniques as an abstraction mechanism for PR. Reconfigurable components are modelled as polymorphic objects. A group of objects where each member is rarely accessed at the same time as other members of the same group is considered a good candidate for reconfiguration. The designer identifies potential candidates for dynamic reconfiguration, marks them, and observes the effects of combining them in a PRR through simulation. Reconfiguration and context switch times are supported through annotations provided by the designer. Once satisfied with the simulation, the model can be fed into the *Fossy* synthesis tool to generate VHDL for the PRRs. The designer is still required to creating wrapper modules for each PRR and floorplan the system manually. The output RTL code is then processed through the vendor PR implementation tools to place and route and create the final bitstreams.

3.2.6 **Other PR Supporting Frameworks.** There have been other models, tools, and methodologies focused on specific aspects of PR system design. [Harkin et al. 2004; Luk et al. 1996]. Many of these have not been publicly released, or rely on hypothetical architectures, and hence have not gained widespread adoption.

Researchers have proposed the use of general purpose modelling languages such as Unified Modeling Language (UML) for high-level specification of PR systems [Fuente et al. 2015]. In this work, RTL specifications of hardware modules, testbenches, and implementation constraints (such as floorplanning constraints) can be directly interfaced with the model. It allows better design space exploration, and supports the choice of an optimal partitioning of PR modules. However, the wrapping of modules in each partition, and floorplanning must still be done manually. Similar to CoPR, this modelling supports easy integration of the PR infrastructure with a processor for run-time management of the system.

The *Caronte methodology* [Donato et al. 2007] takes a fixed task-graph as input and determines how to allocate tasks to the regions specified by the designer in order to complete execution of the application with dynamic loading of tasks. The designer is assumed to have determined how many regions to use and to have floorplanned them. Runtime management is done using an embedded processor.

The GePaRD flow [Boden et al. 2008] tries to enhance the Xilinx PR flow with a high-level synthesis framework. The flow uses a high-level specification of the PR system as input and generates both a system model for simulation and a physically-aware architecture description as input for implementation on the target device using the Xilinx PR design flow. The design flow includes template abstraction, high-level synthesis, and temporal modularisation. The authors do not specify how the output of the proposed framework can be integrated with the vendor toolflow to implement real systems. It targets a *virtual architecture* that adapts to the reconfiguration mechanisms of a dedicated target device, but this mapping is not explained.

The design framework in [Fahmy et al. 2009] defines an adaptive system with two planes. The data plane implements the data processing, such as the signal processing in a radio, and can be composed using a high-level tool that stitches together blocks from an IP library. The control plane implements the management and control functionalities in software. The control plane reconfigures the data plane as needed, from software code written by an adaptive system designer. This framework only supports a single reconfigurable region and suffers from moderate data throughput due to the low-bandwidth interface between software and hardware.

In [Navas et al. 2013] the authors suggest a design approach where an IP block integrates a reconfigurable partition along with the required communication and reconfiguration infrastructure. Using predefined communication interfaces enables PR regions to host any module whose resource

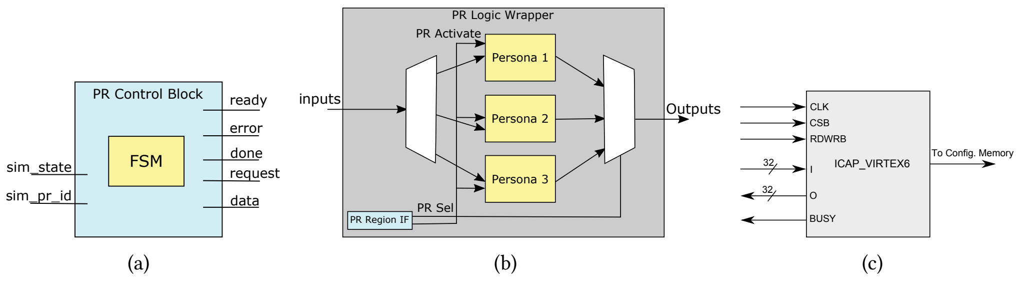


Fig. 8. (a) Altera Arria-10 PR control block [Intel 2017a] (b) Altera PR simulation through spatial multiplexing of different personas (c) Xilinx ICAP controller.

requirements are satisfied. This idea is very similar to the Erlangen Slot Machine with the additional suggestion of a unified software and reconfiguration interface to simplify design.

3.3 PR Simulation

Simulating PR systems is challenging. All vendor-supported simulators are capable of functional and timing verification of the designs for a particular configuration (in Xilinx terminology) or revision (in Altera terminology). But simulating the reconfiguration operation itself is not possible as this is a low-level device operation. Workarounds have been suggested by the vendors to overcome this. One approach is to create a system that contains all the required modules and simulate different configurations by selecting between them. However, this does not offer an accurate representation of the reconfiguration process.

3.3.1 **Vendor PR Simulation Support.** Among the vendors, Altera has better PR simulation support. The simulation model of the hardware primitive (PR-IP) that loads configurations into the FPGA is shown in Fig. 8(a). Each configuration file embeds a unique 32-bit identifier that is used to indicate the loaded bitstream on the *sim_pr_id* port during simulation, while *sim_state* indicated whether the operation has completed. The designer creates wrappers for each PR region by multiplexing different modules (personas) implemented in that region as shown in Fig. 8(b). The *sim_state* and *sim_pr_id* outputs from the PR-IP primitive control these multiplexers. During simulation, a configuration file is injected into the PR-IP and while configuration is occuring, the testbanch sets all outputs from the PRR into an unknown state. When configuration is complete, the multiplexer control signals are driven by the testbench based on the *sim_pr_id* which selects one of the modules specified by the id number.

The hard-macro in traditional Xilinx FPGAs that serves the purpose of writing to the configuration memory is the ICAP. It is possible to send actual configuration files into the ICAP simulation model and obtain configuration status, but this does not simulate actual module switching. The Xilinx tools do not offer further support, though it is possible to multiplex modules assigned to the same region, as in the Altera flow, but this must all be managed manually.

3.3.2 **Academic PR Simulation Efforts.** There has been some limited work in the academic community on simulating PR systems. Since PR is closely associated with the targeted FPGA architecture, fully modelling it requires modelling of low-level architectural details, which would be too slow. Another issue is with using the real configuration files for simulation. Configuration files are generated as the final development step but functional simulation must be completed before they are generated. The earlier multiplexing approach was proposed in [Luk et al. 1997].

Table 2. Comparison of features supported by different PR tools. ○ : Operation is fully manual or not supported. ◐ : Partial automation or support provided by the tool. Designer input is required to complete the step, ● : The step is fully automated by the tool requiring no designer intervention.

Tool	High-level Spec.	Partitioning	Floorplanning	Physical implementation	Circuit relocation	Run-time Mgmt.
Xilinx PlanAhead	○	○	◐	●	○	○
Xilinx Vivado	○	○	◐	●	◐	○
Altera Quartus	○	○	◐	●	○	○
Xilinx SDAccel	●	○	N/A	●	N/A	●
Altera OpenCL	●	○	N/A	●	N/A	●
OpenPR	○	○	○	◐	●	○
CoPR	◐	●	◐	○	○	◐
GoAhead	○	○	◐	◐	●	○
Caronte	◐	◐	○	○	◐	○
GePaRD	●	○	○	○	○	○
PaRAT	●	●	○	○	○	○
OSSS+R	●	◐	○	○	○	◐

An improved technique called dynamic circuit switching (DCS) was presented in [Lysaght and Stockwood 1996]. In that work, a reconfiguration scheduler oversees the PR operation. The tool automatically inserts multiplexers at the outputs of mutually exclusive modules by modifying synthesised netlists. Isolation logic is also inserted, which simulates the behaviour or signals from a PR region as it undergoes reconfiguration. During simulation, the reconfiguration scheduler monitors signals from the modules and activates the required multiplexer controls to simulate the PR operation.

In the above-mentioned approaches, the process of configuration files being transferred from external memory never undergoes functional simulation. The isolation logic, which is inserted between PRRs and the static region to isolate glitches between them during PR operation is also not simulated. A more comprehensive approach is the ReSim library [Gong and Diessel 2011], that proposes a simulation only model of the reconfiguration controller (ICAP for Xilinx FPGAs) and simulation only configuration file (SIMB). Like Altera's work, the SIMB file contains a unique identifier which indicates the circuit being reconfigured. The framework automatically inserts multiplexers and controls them at run-time based on the outputs of the reconfiguration controller. It is also possible to inject errors during reconfiguration to analyse system behaviour. The framework is written in SystemVerilog, which makes it highly portable across different simulators.

3.4 Summary

Table 2 summarises the features supported by the different PR development and implementation tools. It is clear that research has sought to address a range of challenges in PR system design, however, there is no complete framework or toolflow that addresses the high-level design of PR systems abstracted away from low level details, and that can be mapped onto real commercial architectures. In Section 5 we discuss some of the approaches taken to manage the PR process, some of which interact with specific tools in this section. The dependency of academic tools on vendor-specific data files remains a challenge as the tools become obsolete as vendors stop supporting those files. Most research work has focused on Xilinx FPGAs to date, though we expect Altera's recent support of PR to provide further opportunities. From the discussion in the previous two sections, it is clear that advances in both FPGA architecture as well as design methodologies/modelling are key to increasing design productivity for PR-based systems.

4 OVERHEAD REDUCTION TECHNIQUES

The two major overheads associated with PR are resource wastage and reconfiguration time. FPGA resources are wasted, especially in vendor supported tool-flows, due to the constraints on the shape and location of PR regions. Reconfiguration time overhead corresponds to the time required to programme the FPGA with one or more partial configuration files. Unlike multi-context FPGAs where reconfiguration can happen as fast as in a single clock cycle, modern FPGAs take several milliseconds or even seconds depending on the configuration interface, the size of the bitstreams, and how these are stored and transferred [Xilinx Inc. 2017c]. A detailed analysis of factors influencing reconfiguration time and a corresponding cost model is presented in [Papadimitriou et al. 2011]. A major factor restricting the use of PR in online hardware adaptive systems is the runtime of the vendor implementation tools. It may take hours or even days to implement a PR system from synthesis through implementation and configuration generation for the static region and all PRRs. In addition the need to generate multiple configuration files per module for each different possible region compounds this issue in vendor flows. In this section we review techniques proposed for reducing these overheads.

4.1 Partitioning

Determining the number of PRRs to use in a design and how to allocate specific modules to them constitutes the design partitioning phase. Choices made during partitioning can significantly impact both resource usage and reconfiguration time. In vendor PR design flows, the designer must manually determine the number of PRRs and corresponding module allocation to them and hence the granularity of reconfiguration. A fundamental approach is that modules that are mutually exclusive during system execution can be implemented in the same PRR since only one of them needs to be active at any given time. Conversely, it must be possible to simultaneously configure modules which need to be executed concurrently. It is to be noted whenever a module is reconfigured, the entire region to which it is assigned must be reconfigured. Hence, while combining modules into fewer regions can allow the tools to optimize resource usage, it is clear that reconfiguration time can increase dramatically as illustrated in Fig. 9. Furthermore, having more modules in a region means that region is likely to be configured more often.

Much of the work on automated partitioning tries to schedule a graph of dependent tasks onto a fixed number of regions, minimising runtime [Ayadi et al. 2014; Charitopoulos et al. 2015; Purgato et al. 2016]. They assume that multiple FPGA *regions* are used similar to a multi-processor system with each region processing an independent task. Such assumptions completely ignore the communication between PR regions which simplifies implementation but with limited practical

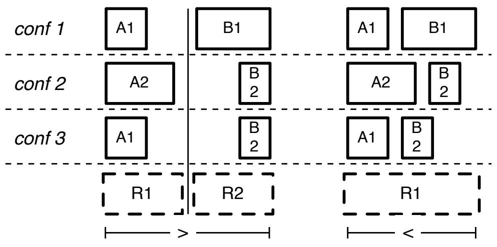


Fig. 9. When assigning modules to separate regions, if some combinations (*configurations* in Xilinx terminology or *revisions* in Altera terminology) do not exist, combining modules into a single region can save area. But when modules are reconfigured, the entire region to which they are assigned has to be reconfigured. For example, changing from *conf 2* to *conf 3* when using two regions requires reconfiguring a small region (R1), but using a single region requires reconfiguring a much larger area.

applications. The work in [Ganesan and Vemuri 2000] describes a reconfigurable processor system with two reconfigurable regions for execution speed up, achieved by overlapping the task execution in one region with the reconfiguration of the other. The task graph is partitioned in such a way that reconfiguration and execution can be carried out concurrently without mutual dependency.

In [Rana et al. 2009], the authors present a method for minimising reconfiguration time based on analysing communication graphs. The algorithm tries to group modules which require simultaneous reconfiguration into the same PRR. However, the number of PRRs must be determined by the designer. In [Jara-Berrocal and Gordon-Ross 2009], the authors assume the number of reconfigurable regions is fixed and resources are considered to be homogeneous. The number and size of the regions must be determined by the designer. Simulated annealing is used to assign hardware modules to the regions while minimising reconfiguration time. The number of modules required to execute a task is assumed to be equal to the number of regions and if any region is unoccupied, an *empty* module is assigned to it. Modern FPGAs have a heterogeneous architecture with distributed DSP and memory blocks, which breaks the homogeneous resource assumption.

The work in [Montone et al. 2010] explores partitioning in more detail. The authors describe a simulated annealing based algorithm for determining the allocation of modules to regions based on minimisation of area variance at different time instances. This work considers the latest FPGA architectures as well as PR requirements. However, it also makes use of fixed task graphs for the optimisation. Furthermore, the impact on reconfiguration time is not accounted for in their method.

In [Vipin and Fahmy 2011], the authors use integer linear programming (ILP) to find the optimal number of PRRs and corresponding module allocation to them. The formulation can be tuned to either minimise reconfiguration time or resource utilisation. Although it provides an optimal solution, as the number of modules to be allocated increases, run-time and complexity becomes excessive. A heuristic algorithm based on connectivity graphs, where modules with the highest probability of concurrent execution are grouped into the same PRRs, was presented in [Vipin and Fahmy 2013].

4.2 Floorplanning

Floorplanning involves physical partitioning of the FPGA fabric for the optimal placement of PRRs in order to improve routability, timing, or density. For standard non-PR based FPGA designs, floorplanning is generally of less interest and is only used by expert designers to achieve high area optimisation or timing performance. For static FPGA designs, vendor tools perform timing driven placement and routing, while fitting the design within the available resources. Further manual tweaking can help improve performance to meet particularly stringent timing constraints.

Vendor PR tools do not support automatic floorplanning, and require manual input from the designer. This requires the designer to have knowledge about low-level physical architecture as well as the run-time costs associated with PR. Manual floorplanning based on these factors consumes a large amount of design time and is cumbersome, often leading to sub-optimal results. This has contributed to making PR less attractive to system designers, since most FPGA designers never deal with floorplans for static designs. An intelligent arrangement and allocation of PR regions can result in reduced area and hence allow designs to fit on smaller devices. It is also important to note that the implementation tools cannot perform logic optimisation across PRR boundaries, and hence their locations are important in achieving timing closure.

Although a number of approaches to FPGA floorplanning have been published, work related to floorplanning for PR is less abundant. Traditionally, FPGA floorplanning is considered as a fixed-outline floorplanning problem, as introduced in [Adya and Markov 2001] and further extended in [Feng and Mehta 2006]. The authors present a resource-aware fixed-outline simulated-annealing and constrained floorplanning technique, but the resulting floorplans may contain irregular shapes, which are not allowed in supported PR flows. A study in [Yuan et al. 2005] presents an algorithm called “Less Flexible First (LFF)". In order to perform placement, the authors define the flexibility of the placement space as in a cost function and a greedy algorithm is used to place modules. The generated floorplan has only rectangular shapes, but the approach is unsuitable for recent FPGAs due to their heterogeneous resource layout.

The approach in [Banerjee et al. 2011] is based on slicing trees, and can ensure that a floorplan contains only rectangular shapes. Here, the authors assume that the entire FPGA fabric is composed of a repeating basic tile, which contains all types of Xilinx FPGA resources including CLBs, Block RAMs and DSP slices. Again, this assumption does not hold for modern FPGAs.

In [Montone et al. 2010], the authors present a reconfiguration-aware “floorplacer". Their algorithm is based on the more recent Virtex-5 FPGA architecture. The algorithm initially divides a design into reconfiguration regions based on the minimisation of temporal variance of resource requirements. The floorplacer tries to minimise area slack using simulated-annealing. In [Singhal and Bozorgzadeh 2007], a floorplanning method based on sequence pairs is presented. In this work, the authors showed how sequence pairs can be used to represent multiple designs together. An objective function tries to maximise the common areas between designs and simulated-annealing is used for optimisation.

In [Vipin and Fahmy 2012a], a greedy algorithm called *columnar kernel tessellation*was presented. This technique defines *kernels*, which are basic units for floorplanning composed of different resource tiles discussed in section 2.2. Kernels containing different resources types(CLB, CLB-DSP, CLB-BRAM, etc.) are predefined and stored in a library for different FPGA families. These kernels are replicated vertically to create the required PRRs. This works since the Xilinx FPGA architecture is uniform in the vertical direction. However, with a greedy algorithm, the quality of the final floorplan depends upon the initial placements of PRRs.

More recently an optimal floorplanner based on mixed-integer linear programming (MILP) was proposed to solve the PR floorplanning problem [Rabozzi et al. 2014]. Although this technique can provide improved results, a solution takes several hours for reasonably sized problems and the search space increases exponentially with the number of regions. The authors propose that the designer provide an initial solution, which can then be refined using heuristics.

4.3 Runtime Placement and Configuration File Manipulations

Researchers have tried to overcome long PR tool chain runtime by enabling runtime placement and routing and bitstream manipulation techniques to make the implemented logic relocatable. Practically these techniques have had limited success due to the heterogeneous architecture of modern FPGAs and limited processing power of embedded processors, which are generally employed for these online manipulations.

The work in [Bazargan et al. 2000] considered online relocation as an on-line bin-packing problem . Later, [Lu et al. 2008] introduced an algorithm for online task placement. Both these approaches assume FPGAs to have a homogeneous architecture, allowing modules to be freely placed in any location. Practically this is not true and connectivity between the modules must somehow be preserved while relocating them. Due to the complex routing architecture of FPGAs, this is infeasible.

Another method for online placement and removal of modules on Virtex-II FPGAs was presented in [Raaijmakers and Wong 2007]. The approach performs the necessary routing to disconnect and connect modules to others already present in the fabric. Before assigning a new module to a region, the interface of the previous module is unrouted to prevent any damage. However, this work only considered designs exclusively using CLBs.

In [Koester et al. 2009], a method is proposed for increasing the placeability of reconfigurable modules. The authors consider regions consisting of reconfigurable tiles, supporting heterogeneous resources such as BRAMs and DSP blocks. The algorithm defines the set of feasible positions for PR modules and optimises the regions to minimise the degree of overlap with other regions. Another method for improving placeability is described in [Becker et al. 2007], targeting Virtex-4 FPGAs. The technique utilises a compatible subset of resources in non-identical regions, making it possible to place modules in non-identical regions.

Several tools have been developed for online module placement targeting different FPGAs. PARBIT (PARtial BItfile Transformer) was a widely used tool targeting Virtex-E FPGAs [Horta and Lockwood 2001]. Modules could be relocated by manipulating the contents of a partial configuration file. To generate a new placement, PARBIT read the configuration frames from the original file and copied to the new file only the configuration bits related to the new area. It then generated new values for the configuration address registers. REPLICA (RElocation Per onLIne Configuration Alteration) [Kalte et al. 2005] was another tool targeting Virtex-E FPGAs. It was implemented on the FPGA itself and performed address manipulation for relocation at run-time. Replica2Pro [Kalte and Porrmann 2006] was an advanced version supporting Virtex-II and Virtex-II Pro FPGAs. It also supported relocation of BRAMs and multiplier blocks.

The major disadvantage of online place-and-route tools is their lack of portability. Due to architectural variations, the tools must be modified for each device, even for different FPGAs in the same device family. The released low-level details of configuration frame contents available from Xilinx has also considerably decreased since the Virtex-5, meaning significant reverse engineering would be required. Even for earlier FPGAs, researchers used trial and error to find the detailed mapping of individual configuration bits. Hence, most of these tools support very few FPGAs belonging to the same family. Support tools such as JBits [Guccione et al. 2004], which provided JAVA based APIs for bitstream manipulations, are no longer endorsed by Xilinx. When relocating modules, it is difficult to ensure the communication infrastructure remains intact. Vendor tools do not directly support explicit positioning of the bus macros used to fix the communication between PR and static regions, so even two PRRs with identical sizes and resources may have different routing to the static region depending on their relative positions. Academic tools such as GoAhead can be helpful in this regard.

Another possible way to support run-time relocation is to consider it as a requirement at design time. If multiple regions with the exact same shape and resources are identified at design time, run time relocation involves only manipulating the frame address fields in the partial bitstreams. In [Backasch et al. 2014] an algorithm to identify multiple identical regions in an FPGA with a given mix of resource requirements is proposed. The ILP-based floorplanner discussed in Section 4.2 ([Rabozzi et al. 2014]) was later extended to provide similar support [Rabozzi et al. 2015]. Here the floorplanner reserves multiple PRRs for the same set of PR modules grouped as a single partition.

4.4 Configuration File Compression

Configuration file (bitstream) compression is a widely used technique for reducing reconfiguration time. In [Pan et al. 2004], the authors exploit redundancies both within a configuration bitstream as well as bitstreams of different configurations. Their analysis shows that frames configuring CLBs have a high degree of mutual similarity. Huffman encoding is also used to compress the bitstreams. [Hauck et al. 1998] and [Hauck et al. 1999] present an algorithm to compress bitstreams for Xilinx XC6200 FPGAs, reducing configuration time by a factor of 4. The algorithm generates a new configuration file from the original, with fewer configuration writes by using the *wildcard* registers present in FPGAs. These enable configuration of multiple frames with a single write by only modifying the frame addresses. [Li and Hauck 2001] and [Haiyun and Shurong 2008] present algorithms for bitstream compression for Virtex FPGAs using different compression techniques such as Huffman coding, Arithmetic coding, and LZ coding, among others.

Bitstream compression is useful in reducing configuration time when bitstream transfer time from external memory to the FPGA is considerably higher than the time taken to send the bitstream to the configuration memory. Otherwise, since the compressed bitstream must be decompressed before final reconfiguration, the effective reconfiguration time may increase. Presently, bitstreams are typically stored in high-speed external memory such as DRAM which offers higher throughput than the maximum reconfiguration throughput (400MB/s), and hence, bitstream compression has limited practical application. A better solution for this problem is to increase the speed at which data is written to the configuration memory. It is worth noting that FPGA vendors support custom bitstream compression techniques, which do not require separate decompression before reconfiguration [Xilinx Inc. 2017b]. For example, Xilinx tools use a special register in the ICAP called the multiple frame write register (MFWR) to configure repeating frame data in the bitstream to different configuration memory locations. To enable this a special flag is set during bitstream generation.

A compression technique specifically targeting run-time module relocation is presented in [Beckhoff et al. 2014]. Recall that circuits targeted for different PRRs with the same shape and resources may vary only in frame addresses (see Section 4.3). In this case for each module, only the bitstream targeted for the first PRR is stored in external memory in an uncompressed format. Bitstreams targeted for other PRRs contain only the difference data with reference to the reference bitstream, and a special reference command indicates where the data is the same. At runtime, the configuration controller analyses the bitstream and fetches the configuration data accordingly. This is beneficial when bitstreams are stored in slow external memory such as flash memories.

4.5 High-Speed Reconfiguration Controllers

One way to reduce reconfiguration time is to improve the speed of reconfiguration itself. Most efforts in this direction have targeted Xilinx FPGAs. The hard-macro in traditional Xilinx FPGAs that serves the purpose of writing to the configuration memory is the Internal Configuration Access Port (ICAP) as depicted in Fig. 8(c). The ICAP works the same way as the SelectMAP external configuration interface but has separate read/write buses [Xilinx Inc. 2011b]. The ICAP data interface can be set to one of three data widths: 8, 16, or 32 bits. The maximum recommended frequency of operation for the ICAP is 100 MHz.

The low-level hardware module which is responsible for delivering bitstreams to the ICAP macro in the required format is called a *reconfiguration controller*. Maximising ICAP throughput has a significant effect on minimising configuration time. Traditionally, the reconfiguration operation is controlled by a processor, through a vendor-provided reconfiguration controller such as the OPBHWICAP or XPSHWICAP, connected as a slave device to the processor bus [Xilinx Inc. 2006, 2010]. Using these vendor-provided controllers gives low throughput in the region of 4.6-10.1 MB/s [Claus et al. 2007a; Liu et al. 2009a]. The ICAP hard macro itself, however, supports speeds of up to 400 MB/s (32 bits at 100 MHz).

In [Gohringer et al. 2010], the authors propose connecting the ICAP controller to the fast simplex link (FSL) bus of a Microblaze soft processor. The drawback is that the processor becomes consumed with the task of requesting configuration data from external memory and sending it over the FSL bus. The resulting throughput of under 30 MB/s remains well below the theoretical limit of the ICAP.

Using DMA to transfer partial bitstreams from external memory to ICAP has been shown to be effective in increasing throughput [Liu et al. 2009b; Vipin and Fahmy 2012b, 2014b]. Elsewhere, some have tried to achieve better performance by over-clocking the ICAP primitive [Hansen et al. 2011]. Since the maximum frequency at which the controller can operate depends upon manufacturing variability and specific placement and routing, this would need to be determined on a device-by-device basis, which is cumbersome.

Some work on optimised ICAP controllers has often made unrealistic assumptions, such as the complete configuration bitstream being stored in FPGA Block RAMs [Liu et al. 2009a]. This is not practical, as FPGAs have limited memory that is often insufficient for even a small number of bitstreams, and these memories are often required for system implementation. Researchers have also proposed directly streaming partial bitstreams from a host computer through high-speed communication channels such as PCIe [Vipin and Fahmy 2014a]. This technique is capable of achieving near theoretical maximum performance and practically unlimited memory for storing partial bitstreams, since they are not stored in limited on-board memory. The drawback is dedicating a PCIe controller for just reconfiguration if the system has no other need for it, or eating into valuable PCIe bandwidth if it is used for data transfer aside from PR.

Currently the only custom reconfiguration controller for Altera (Stratix V) FPGAs is discussed in [Xiao et al. 2016]. This controller also contains the logic for decompressing a pre-compressed bitstream, which further helps to improve reconfiguration throughput.

Table 3 summarises the resource consumption and performance of different proposed reconfiguration controllers. In all cases, the maximum theoretical reconfiguration speed is 400 MB/s except for [Xiao et al. 2016] where it is 200 MB/s.

4.6 Summary

Partitioning and floorplanning for PR remain open to further research. Most existing work does not perform partitioning in a manner that considers the runtime aspects of PR and does not consider the latest FPGA architectures. They generally assume a scheduled graph as the input where each task independently executes in a region. This may not be true for systems where the order in which module execution happens is known only at run-time. Similarly automatic floorplanning has yet to be full tackled. The addition of new hardware macros such as hardened PCIe cores, Ethernet controllers, and memory controllers, along with further restrictions on routing resources makes this task more challenging on modern devices.

Table 3. Performance comparison of configuration controllers.

Implementation	Resource Utilisation			Throughput
	FFs	LUTs	BRAMs	(MB/s)
[Liu et al. 2009a]	1083	918	2	235.20
[Claus et al. 2008]	NA	NA	NA	295.40
[Manet et al. 2008]	NA	NA	NA	353.20
[Liu et al. 2009b]	367	336	0	392.74
Xilinx (PLB) [Xilinx Inc. 2010]	746	799	1	8.48
Xilinx (AXI) [Xilinx Inc. 2011a]	477	502	1	9.10
DyRACT [Vipin and Fahmy 2014a]	672	586	8	399.80
PCAP [Vipin and Fahmy 2014b]	0	0	0	128
Xilinx ICAP for Zynq(non-DMA) [Vipin and Fahmy 2014b]	443	296	0	19
Xilinx ICAP for Zynq(with DMA) [Vipin and Fahmy 2014b]	443	296	0	67
ZyCAP [Vipin and Fahmy 2014b]	806	620	0	382
[Xiao et al. 2016]	6804	1701	0	200

For lower-level architecture-dependent operations such as placement, and module relocation, it is more productive to use vendor-provided tools and find ways to minimise the impact of their limitations. Otherwise, the device specificity of such work limits its appeal and longevity. From Vivado 2016.1, Xilinx supports a hierarchical design flow for PR regions, using which a PR module can be placed and routed independent of the static region. The "stitching" between the PR regions and the static region can be done at a later stage. Although this flow does not support module relocation, this could be an initial step towards it.

The overhead reduction techniques discussed require further investigation for porting to newer Xilinx and Altera (Intel) FPGAs. The relatively simpler architecture of Altera Stratix 10 and Arria 10 FPGAs may allow more effective automatic partitioning and floorplanning.

5 RUN-TIME MANAGEMENT OF PR SYSTEMS

Another important aspect of PR-based systems is runtime management. This includes deciding when reconfiguration should happen, which regions should be reconfigured, how reconfiguration is achieved, and so on. This can be controlled entirely in software control, using a mix of software and hardware, or entirely in hardware. The specific techniques used depend upon factors such as required reconfiguration performance, the presence or absence of a processor in the overall system design, and a-priori knowledge of reconfiguration sequence.

The vendor toolflows expect the presence of a processor to manage the reconfiguration operation. They expect the software developer to be aware of the reconfiguration process, and only provide a low-level driver API for the reconfiguration controller (such as ICAP, PCAP, etc.) [Xilinx Inc. 2004b]. These APIs are available in the standard header files of the Vendor software development suites such as the DevC header files and associated API for Zynq SoCs). The software responsible for runtime management (deciding when to reconfigure and how) should send the corresponding *partial bitstream* data to the reconfiguration controller, usually one word at a time. This process is not only inefficient, but also makes software development highly dependent on the hardware details.

Both Xilinx and Altera offer low-level configuration controller macros (ICAP and PR-IP) that can be interfaced with other hardware. This has allowed the development of custom reconfiguration controllers as discussed in Section 4.5. These controllers can be hardware-only implementations or with associated software drivers. These custom drivers provide one level of abstraction to the software developers, allowing them to specify the names of partial bitstreams required for reconfiguration and not their physical memory addresses or sizes. However, the developer must still know which bitstreams correspond to each region and the combination of region configurations required to achieve a specific system functionality (configuration).

5.1 Management of Reconfigurable Tasks

In a large body of work, FPGAs are considered as general compute resources where *hardware tasks* can be dynamically loaded and unloaded in a similar way to software tasks being scheduled on a multi-processor system. A hardware task is a synthesized digital circuit that has been compiled into a partial bitstream. Most early work in this area assumes FPGAs to be composed of several homogeneous compute units which can be seamlessly combined together to implement tasks of varying compute complexity [Lu et al. 2009]. This holds for coarse grained PR-supporting FPGAs such as the Xilinx XC6200 and dynamic task scheduling has been successfully demonstrated on them [Brebner 1996]. However, modern fine grained FPGAs do not directly support relocation of hardware due to their heterogeneous nature. Since tasks might have to be scheduled to run in different locations on the FPGA fabric, bitstream relocation capability discussed in Section 4.3 becomes a primary requirement for such systems. Another more practical solution is to floorplan the FPGA and generate bitstreams for all tasks at all possible PRRs and store them in a database. The scheduler can then load the correct bitstream when a task is scheuled to a particular PRR [Charitopoulos et al. 2015]. This scheduling can be either online or offline and the scheduler can abide by hard real-time requirements where present.

In [Steiger et al. 2004] a different approach is taken. Here, all the PRRs extend the entire height of the FPGA and have equal width. They are also arranged so they contain exactly the same number and kind of resources and follow the same communication architecture. Each hardware task is implemented using an integer multiple of these slots. Run-time bitstream manipulation is then used to modify the addresses written to when a task is loaded. The online scheduler uses standard scheduling algorithms such as *first fit* or *best fit* to schedule the task into available resource.

Configuration caching is another run-time management technique suggested for reducing reconfiguration time. The technique, described in [Li et al. 2000], tries to minimise reconfiguration time in the case of a task sequence that must be executed in a fixed number of PRRs. Simulated annealing is used to determine the allocation that minimises reconfiguration time, leading to reductions by a factor of 5. Such techniques only apply in the case of using PR to switch tasks in fixed-sequence applications. For dynamically adaptive systems, we do not know the transitions or reconfiguration sequence up front.

5.2 Software/Processor-Based Runtime Management

In these systems software running on a processor (either within the FPGA or in a host machine) manages the reconfiguration operation. At the highest abstraction, the run-time reconfiguration operation is completely transparent to the user. This is the technique used in OpenCL frameworks where the software developer is completely unaware of the reconfiguration operation. At the back-end the software on the host system automatically loads a partial bitstream when a new kernel is configured on the target FPGA. A similar approach is adopted in high-performance FPGA platforms such as Maxeler Dataflow Engines [Ciobanu et al. 2013]. Switching between configurations is implemented using conditional statements in software and loading new hardware is done through an API call. Similar methods that rely on the software designer knowing which bitstream to load are the most prevalent in the literature.

The CoPR flow [Vipin and Fahmy 2015] discussed in Section 3.2.3 uses a two layer architecture for runtime system management. The control plane is implemented in software and refers only to the set of valid configuration labels that are defined in the system specification. Information about how configuration changes map to physical reconfigurations is automated and managed by the configuration manager seamlessly. By simply passing the required configuration name to the configuration manager through the API, whatever PR operations are necessary are carried out automatically. This abstraction works well for adaptive systems where the designer is more concerned about defining adaptive behaviour than the low-level details of how this is achieved.

There has been work on integrating run-time reconfiguration into operating systems (OSs). In [Santambrogio et al. 2008], GNU/Linux is extended to support run-time PR. A number of new system calls, such as *module_request*, and *module_release*, are implemented to enable the OS to manage hardware modules similar to software processes. They also propose different caching and allocation policies to decide how a PR region should be handled once the allocated module finishes execution and to map new module requests to available regions. For low-level reconfiguration operations, dedicated device drivers are integrated with these system calls and user libraries. OS support provides better software abstraction and code reuse, but may cause significant overhead due to the multiple software layers involved during reconfiguration.

In [Reis and Fröhlich 2009], the authors target OS support for systems implementing difference based PR (discussed in Section 3.1.1). Here the FPGA implements a soft processor (such as a MIPS processor) and a number of IP cores. The IP cores are controlled by the software running on the soft processor. The entire FPGA acts as a co-processor to the main processor running in the EPOS framework [Fohlich and Wanner 2008]. To change the co-processor, the difference-based partial bitstream is sent to the FPGA and the corresponding IP core drivers are loaded. The reconfiguration process is fast since the size of the partial bitstream is relatively small. The challenge is that such reconfiguration changes the position of the soft-processor in the FPGA hence software status is lost. To overcome this, before reconfiguration the MIPS status is saved to the main processor memory and after reconfiguration, the software state is restored by sending this saved information back to the FPGA. This system is an example of static partial reconfiguration, since the FPGA cannot perform any processing during the reconfiguration operation.

A custom FPGA architecture and a custom OS supporting PR is presented in [Wang and Jean 2012]. The OS has standard features such as a scheduler, placer, and deadlock detector. The proposed FPGA architecture supports dynamic module placement and routing and appears similar to an *overlay* on an existing FPGA architecture. In this case the reconfiguration is more like a *virtual reconfiguration*, which involves controlling MUXes for enabling dynamic routing. However, the proposed system is not mapped to a real architecture.

Another popular operating system developed for PR is ReconOS [Agne et al. 2014], which offers unified operating system services for functions executing in software and hardware and a standardized interface for integrating custom hardware accelerators. In ReconOS, the target application is partitioned into threads, which can be either blocks of sequential software or parallel hardware modules (hardware thread). Each hardware thread is a PR module currently configured in a PRR. Threads can communicate and synchronize using one or more of the established OS techniques such as message queues or mailboxes, barriers or semaphores, or through mutually exclusive locks (mutexes). ReconOS thus extends a host operating system with support for hardware threads.

5.3 Custom Hardware Based Runtime Management

In these systems the reconfiguration control is completely implemented in hardware through custom state-machines and reconfiguration controllers. Most controllers discussed in Section 4.5

follow this approach. The reconfiguration schedule is either pre-stored in internal memory or dynamically decided by the state machines by observing the surroundings through sensors. The main advantage of these systems are they can achieve high reconfiguration throughput since many of these controllers support DMA transfer of bitstreams from external memory to the configuration interface. But they do not offer much flexibility as they generally use simple adaptation algorithms due to the difficulty and resource requirements of implementing complex algorithms in hardware. Hence, the more promising approach has been to interface these low-level hardware management blocks for low level reconfiguration management with higher layer software to abstract these operations.

5.4 Summary

Further opportunities exist in the area of run-time abstraction for PR systems, including improved abstraction at the application level, and OS policies and mechanisms for improved efficiency of the reconfiguration process. The OS frameworks suffer because they are so general, and the overheads can be significant. A more application centric approach, such as being tied to the requirements of adaptive systems can allow a lean management approach that still retains a high level of abstraction. Presently, the research community is exploring how to manage the reconfiguration process in the context of virtualised cloud accelerators, and this is likely to borrow some ideas from existing OS approaches and integrate these with the specific aspects of general cloud frameworks.

6 APPLICATIONS OF PARTIAL RECONFIGURATION

Some applications fit the concept of partial reconfiguration well, while others benefit from improved efficiency through the use of PR. A wide range of applications exploiting PR have been discussed in the literature. These can be classified based on the specific features of PR being exploited such as adaptability, overhead reduction, reliability improvement, and hardware computing.

6.1 Dynamic System Adaptation

PR enables implementation of adaptive hardware systems that can modify their behaviour dynamically at the hardware level to adapt to their surroundings (operating conditions). This is especially important in applications where the high computational requirements exceed what software can provide, but custom hardware would be too rigid. A popular application with such adaptability is software defined radio (SDR) [Delahaye et al. 2007], where combining flexibility with hardware performance makes PR attractive. Flexible implementations of specific radio blocks using PR, such as adaptive filters, have also been demonstrated [Choi and Lee 2006; Pham et al. 2017]. Cognitive radios are more advanced SDRs that modify their own functionality at runtime in order to operate more effectively in unknown environments [Delorme et al. 2008]. Adaptation of the modulation scheme, coding, filters, and other baseband features at runtime necessitates low power hardware implementations that are also flexible. PR allows these to be adapted individually rather than have separate basebands. A generic development frame-work for implementing PR-based cognitive radios was presented in [Lotze et al. 2009], where the cognitive radio is decomposed into two parts. The static region comprises the control plane, integrating a processor running Linux, while the data plane implements baseband components with high computational requirements in a PR region. This two layer architecture maps well to modern hybrid FPGAs like the Xilinx Zynq, the ARM processor implements the control plane and the FPGA fabric implements the baseband [Shreejith et al. 2015]. A multi-standard OFDM transceiver architecture is presented in [Pham et al. 2017] where a mix of PR modules and parametrised modules is shown to offer a significant improvement in reconfiguration time compared to all-PR modules in a single PRR or multiple PRRs.

Another example is applications with adaptive data clustering (K-means clustering, support vector machines (SVMs), etc.) where kernels are selectively modified with multiple kernels hosted in the same FPGA [Hussain et al. 2012, 2014]. Concurrent implementation of multiple classifiers improves overall system performance. PR allows individual classifiers to be adapted, overcoming the need for a large number of multiplexed classifiers.

Researchers have shown the potential of PR in automotive applications, especially in driver assistance systems [Claus et al. 2007b]. Since vehicles have a very long life, and frequent upgrades are not possible, and given the rapid development of approaches for driver assistance, PR on FPGAs offers the benefits of real-time video processing with the flexibility to upgrade in future. In [Claus et al. 2007c], the authors present a system that uses a PowerPC processor for control and management, with different image processing functional units implemented as co-processors, loaded dynamically as needed.

A packet processing system called Field Programmable Port Extender (FPX) also uses PR [Lockwood et al. 2001] to dynamically reprogram hardware modules and route individual traffic flows in network applications. The reconfigurable virtual network presented in [Yin et al. 2011] combines software virtual routers with several partially-reconfigurable hardware virtual routers, that are configured using dynamic reconfiguration. Functions such as header verification, checksum verification, IP lookup, ARP lookup, and time to live updates, are implemented in PR regions and loaded as needed. The forwarding table for the virtual router can also be updated via the PCI bus. Using PR was shown to offer better flexibility and forwarding performance compared to a fixed hardware implementation.

Within space applications, [Osterloh et al. 2009] describes the implementation of the System-on-Chip Wire (SoCWire) architecture on a partially reconfigurable Virtex-4 FPGA. SOCWire is a well established network-on-Chip protocol in the space community, supporting link initialisation, credit-based flow control, detection of link errors, link error recovery, hot-plug ability, and more. In this work, the SoCWire routing architecture is implemented in a static region and the processing elements (PEs) are implemented in the PRRs, enabling dynamic loading and unloading of PEs based on processing requirements.

PR has also been used extensively in high energy physics experiments. It was used in the Compressed Baryonic Matter experiment conducted at the Facility for Antiproton and Ion Research in Darmstadt, Germany [Gao et al. 2009]. This experiment used an Active Buffer Board (ABB) for receiving, buffering, and forwarding hit data. In a high energy physics experiment, since the surrounding conditions can change, it is required that the ABB functionality change post-installation. PR was also used in the ALICE (A Large Ion Collider Experiment) experiment conducted in the CERN Large Hadron Collidor (LHC) [Papadimitriou et al. 2010]. Special photo-detectors were used to monitor particles generated by the collisions in the LHC. A collection of 120 Xilinx Virtex-4 FX FPGAs with PR were used for first level processing and data reduction on the photo-detector outputs.

Applications that deal with changing environments are ideal candidates for PR systems, as the varying compute modules can be loaded as needed at runtime. Most of these applications have been designed in an ad-hoc manner, rather than using a specific high-level flow, but they demonstrate the applicability of PR in a range of domains.

6.2 System Cost Reduction

PR can help reduce overall system cost by enabling time multiplexing of functionality on a smaller chip instead of a larger FPGA. Since the energy consumption of smaller chips is generally lower, this also helps reduce overall cost. PR has been demonstrated to be useful in audio and video processing applications, such as MP3 decoding [Taghipour et al. 2008] and JPEG encoding [Bouchoux et al. 2004].

As the logic availability in older generation FPGAs was limited, these functions would be temporally partitioned into smaller tasks to be performed sequentially using subsequent configurations of the same PR region. In [Khraisha and Lee 2010], a PR based scalable H.264/AVC deblocking filter architecture is described. The filter adapts to different user requirements at runtime. A real-time video processing system using PR is described in [Bhandari et al. 2009], where different image processing filters are implemented in the same reconfigurable region to reduce resource requirements and power consumption. In [Birla and Vikram 2008], the AdaBoost algorithm for human detection is implemented on a Virtex-4 FPGA using PR. Two computationally intensive tasks, integral image computation and feature extraction/decision, are alternately implemented in a single PRR, saving significant area.

Other such applications include using the same PR region to implement different stages of hardware cryptographic functions [Patterson 2000] and time multiplexing different stages of image/video processing [Bhandari et al. 2009; Krill et al. 2010]. In [Noguera and Kennedy 2007], the authors propose a method for power saving in networks by changing the implementation of the same function under different conditions. By closely monitoring environmental changes (number of users, time of day, distance from the central node, etc.) and adapting the implementation accordingly, network power consumption was reduced, potentially also improving reliability due to the lower thermal footprint.

6.3 Reliability

A hurdle in the use FPGAs in space applications is the effect of Single Event Upsets (SEUs) [Ceschia et al. 2003], which are changes of state caused by ions or electro-magnetic radiation striking a sensitive node in a micro-electronic device such as semiconductor memory. SRAM based FPGAs are highly vulnerable to SEUs, which can lead to corruption in the configuration memory and serious system damage. PR has been proposed as a method for mitigating SEU effects on SRAM based FPGAs since it provides an auxiliary path to the configuration memory. In [Bolchini et al. 2007a,b], the authors partition the FPGA into a number of regions in order to isolate SEU errors, then apply duplication with comparison to ensure correct computation. Once an error is detected, that region is reconfigured. Another simple method to overcome SEUs using PR is configuration scrubbing [Heiner et al. 2009]. Here, the configuration data is stored in a radiation hardened memory and the configuration controller reconfigures portions of the FPGA using this memory periodically, called blind scrubbing. Since the configuration operation is glitchless, this does not impact continuing operation. In a more advanced method, the configuration controller reads data from the FPGA and detects the presence of an error and writes back configuration data only if an error is present. Advanced SEU mitigation using both PR as well as traditional triple modular redundancy (TMR) methods have also been suggested [Carmichael 2000, 2006].

Researchers have also proposed enabling redundancy in automotive electronics through PR to improve reliability [Shreejith et al. 2013]. Here redundant electronic control units (ECUs) are implemented in PR regions, and whenever an error condition is detected, the corresponding region is reconfigured to recover from the error, while a redundant ECU with reduced performance acts as a backup. PR has also been proposed for improving the security of automotive systems at the network controller level [Shreejith and Fahmy 2015]. In this work, the network controller is not loaded onto a network node until the hardware and software checksums are confirmed as being valid, thereby ensuring that tampered with nodes cannot access the network.

6.4 Computing Systems

Perhaps the most generalised use of PR is as a mechanism for integrating accelerator hardware within general purpose computing systems. PR here serves the purpose of integrating adaptable hardware with fixed compute interfaces. The dynamic instruction set computer (DISC) [Wirthlin and Hutchings 1995] supports demand-driven modification of its instruction set. Each instruction is implemented as an independent circuit module, and these are paged into hardware in real-time as dictated by the application. Hardware limitations are eliminated by replacing unused instruction modules with usable instructions at run-time. The concept of high-performance reconfigurable computing has also been proposed [El-Araby et al. 2007]. Here, the FPGA takes on a significant portion of a large scientific application, with PR allowing the fabric to be used by different computational steps at runtime, as in the case of system cost reduction just discussed, but in this case, the applications are too large to fit on a single FPGA.

In [Steiner 2008], autonomous computing systems were discussed, with placement and routing implemented on the FPGA fabric itself, allowing the FPGA to create new circuit bitstreams, for self-modifying hardware. The main challenge is the logic overhead of implementing these tools and the slow speed of creating new bitstreams.

An emerging application of PR is in accelerated cloud computing [Byma et al. 2014]. Microsoft has already presented a comprehensive demonstration of the benefits of FPGAs in the datacenter applied to Bing search [Putnam et al. 2014], although published implementations do not use PR. PR can extend this approach to allow integration of custom hardware accelerators that can be dynamically changed at runtime. PR allows virtualisation of a single FPGA device into multiple *virtual FPGAs* by hosting multiple accelerators concurrently in different PR regions [Fahmy et al. 2015]. Here each PR region acts as a virtual FPGA (vFPGA) on a commercial FPGA development board. The required drivers and the virtualisation environment (hypervisor) run on server machines hosting the FPGA boards. Work in this area is gaining significant attention. [Kachris and Soudris 2016] provides a comprehensive survey on FPGA based hardware accelerators for cloud computing.

FPGAs in general offer high performance in neural network implementations as demonstrated by Chinese search engine company, Baidu [Wirbel 2014]. But many of these applications require adaptation of the inference computation to the task at hand. [Torresen et al. 2008] presents an online evolvable pattern recognition system, where the classification module is dynamically evolved using PR. Here a processor configures a PR region with different *classification modules* to evaluate the input pattern.

As discussed in Section 3.1.4, the OpenCL compute framework has been ported to FPGA platforms. PR allows dynamic loading of compute kernels as needed into a single PR region at runtime. Virtual accelerators for cloud computing and OpenCL integration are perhaps the most promising applications for PR in the near future.

6.5 Summary

As evident from the discussion, PR has demonstrated its applicability across a range of application domains. Many of these have been demonstrated in a research environment or only as prototype models. As discussed in Sections 2 and 3, hardware expertise requirements, constraints due to device architectures, and limited tool support have limited more widespread adoption in the past.

However, we are now seeing a renewed interest in hardware virtualisation, with the vendors playing an important part in facilitating this with better architectures and tools [Intel 2017b; Xilinx Inc. 2017a]. It remains the case, however, that these tools address the computing systems integration aspect of PR, rather than the more general adaptive systems idea. Recent improvements such as the direct interfacing between PCIe and reconfiguration controller in Xilinx MCAP and the sector-based architecture of the Altera Stratix 10 promise to improve support further. Further architectural improvements will be necessary. These include more built-in hard macros (including memory controllers), improving the relative positions of these hard macros (PCIe, Ethernet, etc.) to maximise the area available for PRRs, and increasing the number of reconfiguration controllers and their reconfiguration speed.

7 CONCLUSIONS AND FUTURE DIRECTIONS

PR has evolved significantly over recent years, and found use in a diverse range of applications. The design of PR systems remains difficult, and hence, only accessible to FPGA experts. Many published techniques for overcoming the limitations of vendor tools have slowly become obsolete, as a result of the increasing heterogeneity of modern devices and less open access provided by vendors. Since many techniques are also heavily tied to specific architectures, with their evolution, these tools can become unusable. As a result of these difficulties, most systems that use PR at present must be designed at a low level with detailed hardware design expertise required.

The emerging interest in using FPGAs in the datacenter represents the first widespread use of PR in deployed systems, and there remain numerous challenges to fully virtualise FPGA resources using PR. The trends towards more autonomous systems in areas such as automotive, communication, and aerospace applications also presents an opportunity well-suited to PR system design. To truly bring PR system design into the mainstream, we believe there are a number of research challenges in need of attention:

- At the architecture level, how to better support the idea of multiple loadable accelerators with easy relocation and reconfiguration, particularly on commercial devices.
- In methods, how to bring together the strong body of research done to date to overcome the limitations of existing flows, and abstracting away the hardware aspects through automation from high-level descriptions.
- In frameworks and applications, finding better application-oriented ways of describing adaptive systems that can be automatically mapped to PR implementations.
- At the management level, improved abstraction to allow loading and unloading of new configurations similarly to dynamic loading and unloading of software modules.
- Exploring how autonomously self-adaptive systems can be built that combine reconfiguration capability with intelligence and the ability to adapt bitsream capabilities.

This article has thoroughly reviewed all aspects of dynamic and partial reconfiguration in the literature to present the reader with a structured overview of the research to date and pose a number of challenges we believe stand in the way of more widespread adoption of PR. We are confident that with renewed interest in this area, these challenges will be addressed by the community in a way that finally brings PR to the mainstream.

REFERENCES

S.N. Adya and I.L. Markov. 2001. Fixed-outline floorplanning through better local search. In *Proceedings of ACM/IEEE International Conference on Computer Design.* 328 – 334.

A. Agne, M. Happe, A. Keller, E. LÃijbbers, B. Plattner, M. Platzner, and C. Plessl. 2014. ReconOS: An Operating System Approach for Reconfigurable Computing. *IEEE Micro* 34, 1 (Jan 2014), 60–71.

Altera. 2013a. *Design Planning for Partial Reconfiguration.* Altera.

Altera. 2013b. *Quartus II Handbook Version 13.1.* Altera.

Altera. 2016a. *Arria 10 CvP Initialization and Partial Reconfiguration over PCI Express User Guide.*

Altera. 2016b. *Quartus Prime Standard Edition Handbook.* Altera.

Altera. 2017. *ug-partrecon : Partial Reconfiguration IP Core.*

Atmel. 2013. *AT40K05, AT40K10, AT40K20, AT40K40 Datasheet.*

Ramzi Ayadi, Bouraoui Ouni, and Abdellatif Mtibaa. 2014. Integrated temporal partitioning and partial reconfiguration techniques for design latency improvement. *Evolving Systems* 5, 2 (01 Jun 2014), 133–141.

R. Backasch, G. Hempel, S. Werner, S. Groppe, and T. Pionteck. 2014. Identifying homogenous reconfigurable regions in heterogeneous FPGAs for module relocation. In *Proceedings of International Conference on ReConFigurable Computing and FPGAs (ReConFig14).* 1–6.

P. Banerjee, M. Sangtani, and S. Sur-Kolay. 2011. Floorplanning for partially reconfigurable FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 30, 1 (Jan. 2011), 8–17.

K. Bazargan, R. Kastner, and M. Sarrafzadeh. 2000. Fast Template Placement for Reconfigurable Computing Systems. *IEEE Design and Test of Computers* 17, 1 (Jan 2000), 68–83.

T. Becker, W. Luk, and P.Y.K. Cheung. 2007. Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM).*

C. Beckhoff, D. Koch, and J. Torresen. 2012. GoAhead: A Partial Reconfiguration Framework. In *Proceeding of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 37–44.

C. Beckhoff, D. Koch, and J. Torresen. 2014. Portable module relocation and bitstream compression for Xilinx FPGAs. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL).*

C. Beckhoff, D. Koch, and J. Torreson. 2013. Automatic floorplanning and interface synthesis of island style reconfigurable systems with GOAHEAD. In *Proceedings of International Conference on Architecture of Computing Systems(ARCS).* Springer Berlin Heidelberg, 303–316.

S. U. Bhandari, S. Subbaraman, S. Pujari, and R. Mahajan. 2009. Real Time Video Processing on FPGA Using on the Fly Partial Reconfiguration. In *Proceedings of International Conference on Signal Processing Systems (ICSPS).* 244–247.

M. Birla and K.N. Vikram. 2008. Partial Run-time Reconfiguration of FPGA for Computer Vision Applications. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing (IPDPS).*

M. Boden, T. Fiebig, M. Reiband, and P. Reichel. 2008. GePaRD - A High-Level Generation Flow for Partially Reconfigurable Designs. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI).*

C. Bolchini, A. Miele, and M. D. Santambrogio. 2007a. TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT).*

C. Bolchini, D. Quarta, and M. D. Santambrogio. 2007b. SEU Mitigation for SRAM-Based FPGAs through Dynamic Partial Reconfiguration. In *Proceedings of ACM Great Lakes symposium on VLSI.*

S. Bouchoux, E. Bourennane, and M. Paindavoine. 2004. Implementation of JPEG2000 arithmetic decoder using dynamic reconfiguration of FPGA . In *Proceedings of International Conference on Image Processing (ICIP).*

G. Brebner. 1996. *A virtual hardware operating system for the Xilinx XC6200.* Springer Berlin Heidelberg, Berlin, Heidelberg, 327–336.

S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow. 2014. FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 110–116.

D. Capalija and T. S. Abdelrahman. 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).*

C. Carmichael. 2000. *XAPP216: Correcting Single-Event Upsets Through Virtex Partial Configuration.* Xilinx Inc.

C. Carmichael. 2006. *XAPP197: Triple Module Redundancy Design Techniques for Virtex FPGAs.* Xilinx Inc.

M. Ceschia, M. Violante, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori. 2003. Identification and Classification of Single-Event Upsets in the Configuration Memory of SRAM-Based FPGAs. *IEEE Transactions on Nuclear Science* 50, 6 (Dec. 2003), 2088–2094.

George Charitopoulos, Iosif Koidis, Kyprianos Papadimitriou, and Dionisios Pnevmatikatos. 2015. *Hardware Task Scheduling for Partially Reconfigurable FPGAs.* Springer International Publishing, Cham, 487–498.

C.S. Choi and H. Lee. 2006. An Reconfigurable FIR Filter Design on a Partial Reconfiguration Platform. In *Proceedings of Communications and Electronics (ICCE).*

W. Chong, S. Ogata, M. Hariyama, and M. Kameyama. 2005. Architecture of a Multi-Context FPGA Using Reconfigurable Context Memory. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

C. B. Ciobanu, D. N. Pnevmatikatos, K. D. Papadimitriou, and G. N. Gaydadjiev. 2013. FASTER Run-time Reconfiguration Management. In *Proceedings of ACM International Conference on Supercomputing (ICS '13).* ACM, 463–464.

C. Claus, F. H. Muller, J. Zeppenfeld, and W. Stechele. 2007a. A new framework to accelerate Virtex-II Pro dynamic partial self reconfiguration. In *Proceedings of IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW).*

C. Claus, W. Stechele, and A. Herkersdorf. 2007b. Autovision - A Run-time Reconfigurable MPSoC Architecture for Future Driver Assistance Systems. *Information Technology* 49 (2007), 181–186.

C. Claus, J. Zeppenfeld, F. Muller, and W. Stechele. 2007c. Using Partial-Run-Time Reconfigurable Hardware to accelerate Video Processing in Driver Assistance System. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE).*

C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker. 2008. A multi-platform controller allowing for maximum Dynamic Partial Reconfiguration throughput. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL).* 535 – 538.

K. Compton and S. Hauck. 2002. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*

(CSUR) 34, 2 (June 2002), 171–210.

J. Coole and G. Stitt. 2015. Adjustable-cost overlays for runtime compilation. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 21–24.

A. DeHon. 1996. DPGA Utilization and Application. In *Proceedings of ACM/SIGDA International Symposium on FPGAs*.

J.P. Delahaye, J. Palicot, C. Moy, and P. Leray. 2007. Partial Reconfiguration of FPGAs for Dynamical Reconfiguration of a Software Radio Platform. In *Proceedings of IST Mobile and Wireless Comms. Summit*.

J. Delorme, J. Martin, A. Nafkha, C. Moy, F. Clermidy, P. Leray, and J. Palicot. 2008. A FPGA partial reconfiguration design approach for cognitive radio based on NoC architecture. In *Proceedings of International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*. 355–358.

Alberto Donato, Fabrizio Ferrandi, Massimo Redaelli, MarcoDomenico Santambrogio, and Donatella Sciuto. 2007. Caronte: A methodology for the Implementation of Partially dynamically Self-Reconfiguring Systems on FPGA Platforms. In *VLSI-Soc: From Systems To Silicon*. Vol. 240. Springer US, 87–109.

E. El-Araby, I. Gonzale, and T. El-Ghazawi. 2007. Performance bounds of partial run-time reconfiguration in high-Performance reconfigurable computing. In *Proceedings of International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*.

E. Eto. 2007. *XAPP290: Difference-Based Partial Reconfiguration*. Technical Report. Xilinx Inc.

S.A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser. 2009. Generic Software Framework for Adaptive Applications on FPGAs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 55–62.

S.A. Fahmy, K. Vipin, and S. Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *Proceedings of IEEE International Conference on Cloud Computing Technology and Science, , Vancouver, Canada*. 430–35.

Y. Feng and D.P. Mehta. 2006. Heterogeneous floorplanning for FPGAs. In *Proceedings of International Conference on VLSI Design*.

A.A. Fohlich and L. F. Wanner. 2008. Operating System Support for Wireless Sensor Networks. *Journal of Computer Science* 4, 4 (2008), 272–281.

D. De La Fuente, J. Barba, X. Pena, J. C. Lopez, P. Penil, and P. P. Sanchez. 2015. Building a dynamically reconfigurable system through a high development flow. In *Proceedings of Forum on Specification and Design Languages (FDL)*.

S. Ganesan and R. Vemuri. 2000. An integrated temporal partioning and partial reconfiguration technique for design latency improvement. In *Proceedings of Design, Automation and Test in Europe (DATE)*. 320–325.

W. Gao, K. Kugel, R. Manner, N. Abel, N. Meier, and U. Kebschull. 2009. DPR in high energy physics. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

D. Gohringer, J. Noguera, and J. Becker. 2010. Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs. In *Proceedings of IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*.

M. Gokhale and D. Gomersall. 1997. High level compilation for fine grained FPGAs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 165 – 173.

L. Gong and O. Diessel. 2011. ReSim: A reusable library for RTL simulation of dynamic partial reconfiguration. In *Proceeding of International Conference on Field-Programmable Technology*. 1–8.

S. Guccione, D. Levi, and P. Sundararajan. 2004. *JBits: Java based interface for reconfigurable computing*. Technical Report. Xilinx Inc.

G. Haiyun and C. Shurong. 2008. Partial Reconfiguration Bitstream Compression for Virtex FPGAs. In *Proceedings of Congress on Image and Signal Processing (CISP)*.

S. Gimle Hansen, D. Koch, and J. Torresen. 2011. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*.

J. Harkin, T.M. Mcginnity, and L.P. Maguire. 2004. Modeling and optimizing run-time reconfiguration using evolutionary computation. *ACM Transactions on Embedded Computing Systems (TECS)* 3, 4 (Nov. 2004), 661–685.

S. Hauck, Z. Li, and E. Schwabe. 1998. Configuration Compression for the Xilinx XC6200 FPGA. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*.

S. Hauck, Z. Li, and E. Schwabe. 1999. Configuration compression for the Xilinx XC6200 FPGA. In *Proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

J.R. Hauser and J. Wawrzynek. 1997. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb. 2009. FPGA Partial Reconfiguration via Configuration Scrubbing. In *Proceedings of International Conference on Field Programmable Logic and Applications ((FPL)*.

E.L. Horta and J.W. Lockwood. 2001. *PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of FIeld Programmable Gate Arrays (FPGA)*. Washington University.

D. How and S. Atsatt. 2016. Sectors: Divide & Conquer and Softwarization in the Design and Validation of the Stratix-10 FPGA. In *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

M. Huebner, T. Becker, and J. Becker. 2004. Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In *Proceeding of Symposium on Integrated Circuits and Systems Design*. 28–32.

C. Huriaux, O. Sentieys, and R. Tessier. 2014. FPGA Architecture Support for Heterogeneous, Relocatable Partial Bitstreams. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*.

H.M. Hussain, K. Benkrid, A. Ebrahim, A.T. Erdogan, and H. Seker. 2012. Novel Dynamic Partial Reconfiguration Implementation of K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs. *International Journal of Reconfigurable Computing (IJRC)*, Article 1 (Jan. 2012), 15 pages.

H. Hussain, K. Benkrid, and H.Seker. 2014. Novel dynamic partial reconfiguration implementations of the support vector machine classifier on FPGA . *Turkish Journal of Electrical Engineering & Computer Sciences* (2014), 3371–3387. Issue 24.

Intel. 2017a. *UG-20066 : Partial Reconfiguration Solutions IP User Guide.*

Intel. 2017b. *UG-OCL002 Intel FPGA SDK for OpenCL: Programing Guide.*

A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. 2016a. DeCO: A DSP block based FPGA accelerator overlay with low overhead interconnect. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–8.

A. K. Jain, D. L. Maskell, and S. A. Fahmy. 2016b. Throughput oriented FPGA overlays using DSP blocks. In *Proceedings of the Design, Automation and Test in Europe Conference(DATE)*. 1628–1633.

A. Jara-Berrocal and A. Gordon-Ross. 2009. Runtime Temporal Partitioning Assembly to Reduce FPGA Reconfiguration Time. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*.

C. Kachris and D. Soudris. 2016. A survey on reconfigurable accelerators for cloud computing. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*. 1–10.

H. Kalte, G. Lee, M. Porrmann, and U. Rückert. 2005. REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

H. Kalte and M. Porrmann. 2006. REPLICA2Pro: Task Relocation by Bitstream Manipulation in Virtex-II/Pro FPGAs. In *Proceedings of conference on Computing frontiers*.

I. Kennedy. 2003. Exploiting redundancy to speedup reconfiguration of an FPGA. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*. 262–271.

R. Khraisha and J. Lee. 2010. A scalable H.264/AVC Deblocking filter architecture using dynamic partial reconfiguration. In *Proceedings of IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*.

D. Koch, J. Torresen, C. Beckhoff, D. Ziener, C. Dennl, V. Breuer, J. Teich, M. Feilen, and W. Stechele. 2012. Partial reconfiguration on FPGAs in practice; Tools and applications. In *Proceedings of ARCS Workshops (ARCS)*. 1–12.

M. Koester, W. Luk, J. Hagemeyer, and M. Porrmann. 2009. Design optimizations to improve placeability of partial reconfiguration modules. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

B. Krill, A. Amira, A. Ahmad, and H. Rabah. 2010. A new FPGA-based dynamic partial reconfiguration design flow and environment for image processing applications. In *Proceedings of European Workshop on Visual Information Processing (EUVIP)*. 226–231.

R. Kumar and A. Gordon-Ross. 2013. PRML: A Modeling Language for Rapid Design Exploration of Partially Reconfigurable FPGAs. In *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 117–120.

R. Kumar and A. Gordon-Ross. 2015. An Automated High-Level Design Framework for Partially Reconfigurable FPGAs. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*.

Lattice Corp. 2003. *ORCA Series 4 FPGAs.* Lattice Semiconductor Corporation.

Z. Li, K. Compton, and S. Hauck. 2000. Configuration caching management techniques for reconfigurable computing. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*.

Z. Li and S. Hauck. 2001. Configuration Compression for Virtex FPGAs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

J. Lipsky. 2015. (Jan. 2015). http://www.eetimes.com/document.asp?doc_id=1325499

M. Liu, W. Kuehn, Z. Lu, and A. Jantsch. 2009a. Run-Time partial reconfiguration speed investigation and architectural design space exploration. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*.

S. Liu, R. N. Pittman, and A. Forin. 2009b. *Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller.* Technical Report MSR-TR-2009- 150. Microsoft Research.

J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. 2001. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

J. Lotze, S.A. Fahmy, J. Noguera, B. Ozgul, L. Doyle, and R. Esser. 2009. Development framework for implementing FPGA-based cognitive network nodes. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM)*.

Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev. 2009. *Online Task Scheduling for the FPGA-Based Partially Reconfigurable*

Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, 216–230.

Y. Lu, T. Marconi, G.N. Gaydadjiev, K. Bertels, and R.J. Meeuws. 2008. A Self-adaptive on-line Task Placement Algorithm for Partially Reconfigurable Systems. In *Proceedings of Parallel and Distributed Processing Symposium (IPDPS)*.

W. Luk, N. Shirazi, and P.Y.K. Cheung. 1996. Modelling and Optimising Run-time Reconfigurable Systems. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*.

W. Luk, N. Shirazi, and P. Y. K. Cheung. 1997. Compilation tools for run-time reconfigurable designs. In *Proceeding of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 56–65.

P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. 2006. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*.

P. Lysaght and J. Stockwood. 1996. A simulation tool for dynamically reconfigurable field programmable gate arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4, 3 (Sept 1996), 381–390.

M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. 2007. The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 47, 1 (01 Apr 2007), 15–31.

P. Manet, D. Maufroid, L. Tosi, G. Gailliard, O. Mulertt, M. D. Ciano, J. D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, V. L. Barba, P. Cuvelier, B. Rousseau, and P. Gelineau. 2008. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP Journal on Embedded Systems* 2008 (2008), 1–11.

A. Montone, M.D. Santambrogio, D. Sciuto, and S.O. Memik. 2010. Placement and floorplanning in dynamically reconfigurable FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3, 4 (Nov. 2010), 24:11–24:34.

National. 1993. *Configurable Logic Array (CLAy) Data Sheet*. National Semiconductor.

B. Navas, I. Sander, and J. Oberg. 2013. The RecoBlock SoC platform: A flexible array of reusable Run-Time-Reconfigurable IP-blocks. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*. 833–838.

J. Noguera and I. O. Kennedy. 2007. Power Reduction in Network Equipment Through Adaptive Partial Reconfiguration. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*. 240–245.

R.T. Ong. 1995. Programmable Logic Device which stores more than one configuration and means for switching configurations. (1995).

B. Osterloh, H. Michalik, S. A. Habinc, and B. Fiethe. 2009. Dynamic Partial Reconfiguration in Space Applications. In *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems*.

J. H. Pan, T. Mitra, and W. Wong. 2004. Configuration bitstream compression for dynamically reconfigurable FPGAs . In *Proceedings of IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

K. Papadimitriou, A. Anyfantis, and A. Dollas. 2010. An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems. *IEEE Transactions on Instrumentation and Measurement* 59, 6 (June 2010), 1642–1651.

K. Papadimitriou, A. Dollas, and S. Hauck. 2011. Performance of Partial Reconfiguration in FPGA Systems: A Survey and Cost Model. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4, 4 (Dec. 2011), 36:1–36:24.

C. Patterson. 2000. High performance DES encryption in Virtex FPGAs using JBits. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 113–121.

M. Peattie. 2009. *Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode*. Technical Report. Xilinx Inc.

T. H. Pham, S. A. Fahmy, and I. V. McLoughlin. 2017. An End-to-End Multi-Standard OFDM Transceiver Architecture Using FPGA Partial Reconfiguration. *IEEE Access* 5 (2017), 21002–21015.

M. Platzner, J. Teich, and N. Wehn. 2010. *Dynamically Reconfigurable Systems*. Springer Netherlands.

A. Purgato, D. Tantillo, M. Rabozzi, D. Sciuto, and M. D. Santambrogio. 2016. Resource-Efficient Scheduling for Partially-Reconfigurable FPGA-Based Systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 189–197.

A. Putnam, A. M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the International Symposium on Computer Architecture*. 13–24.

S. Raaijmakers and S. Wong. 2007. Run-Time Partial Reconfiguration for Removal, Placement and Routing on the Virtex-II Pro. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*.

M. Rabozzi, R. Cattaneo, T. Becker, W. Luk, and M. D. Santambrogio. 2015. Relocation-Aware Floorplanning for Partially-Reconfigurable FPGA-Based Systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. 97–104.

M. Rabozzi, J. Lillis, and M.D. Santambrogio. 2014. Floorplanning for Partially-Reconfigurable FPGA Systems via Mixed-Integer Linear Programming. In *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

V. Rana, S. Murali, D. Atienza, M. D. Santambrogio, L. Benini, and D. Sciuto. 2009. Minimization of the reconfiguration latency for the mapping of applications on FPGA-based systems. In *Proceedings of IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS).*

T. A. Reis and A.A. Fröhlich. 2009. Operating System Support for Difference-Based Partial Hardware Reconfiguration. In *Proceedings of IEEE/IFIP International Symposium on Rapid System Prototyping (RSP).* 75–80.

M.D. Santambrogio, V. Rana, and D. Sciuto. 2008. Operating system support for online partial dynamic reconfiguration management. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL).* 455–458.

A. Schallenberg, W. Nebel, A. Herrholz, P.A. Hartmann, K. Grüttner, and F. Oppenheimer. 2010. *Dynamically Reconfigurable Systems* (1 ed.). Springer, Chapter POLYDYN-Object-Oriented Modelling and Synthesis Targeting Dynamically Reconfigurable FPGAs, 139–158.

A. Schallenberg, W. Nebel, A. Herrholz, P. A. Hartmann, and F. Oppenheimer. 2009. OSSS+R: A framework for application level modelling and synthesis of reconfigurable systems. In *Proceedings of the Design, Automation and Test in Europe Conference(DATE).* 970–975.

S. Shreejith, B Banarjee, K Vipin, and S. A. Fahmy. 2015. Dynamic Cognitive Radio on the Xilinx Zynq Hybrid FPGA. In *Proceedings of International Conference on Cognitive Radio Oriented Wireless Networks (CROWNCOM).*

S. Shreejith and S. A. Fahmy. 2015. Security Aware Network Controller for Next Generation Automotive Embedded Systems. In *Proceedings of Design Automation Conference (DAC).*

S. Shreejith, K. Vipin, S. A. Fahmy, and M. Lukasiewycz. 2013. An approach for redundancy in FlexRay networks using FPGA partial reconfiguration. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE).*

L. Singhal and E. Bozorgzadeh. 2007. Multi-layer floorplanning for reconfigurable designs. *IET Computers & Digital Techniques* 1, 4 (July 2007), 276–294.

A.A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood. 2011. OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW).* 228–235.

C. Steiger, H. Walder, and M. Platzner. 2004. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Comput.* 53, 11 (Nov 2004).

N.J. Steiner. 2008. *Autonomous Computing Systems.* Ph.D. Dissertation. Virginia Polytechnic Institute and State University.

N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French. 2011. Torc : Towards an Open-Source Tool Flow. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).*

G. Stitt and J. Coole. 2011. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters* 3, 3 (2011), 81–84.

Tabula. 2010. *ABAX Product Brief.* Technical Report. Tabula.

H. Taghipour, J. Frounchi, and M. H. Zarifi. 2008. Design and Implementation of MP3 Decoder using Partial Dynamic Reconfiguration on Virtex-4 FPGAs. In *Proceedings of International Conference on Computer and Communication Engineering.*

E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. 1995. A First Generation DPGA Implementation. In *Proceedings of the Canadian Workshop on Field-Programmable Devices (FPD).* 138–143.

T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. 2005. Reconfigurable computing: architectures and design methods. *IEE Proceedings - Computers and Digital Techniques* 152, 2 (Mar. 2005), 193–207.

J. Torresen, G.A. Senland, and K. Glette. 2008. Partial Reconfiguration Applied in an On-line Evolvable Pattern Recognition System. In *Proceedings of The Nordic Microelectronics event (NORCHIP).*

S. Trimberger, D. Carberry, A. Johnson, and J. Wong. 1997. A time-multiplexed FPGA. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM).* 22–28.

K. Vipin and S. A. Fahmy. 2011. Efficient Region Allocation for Adaptive Partial Reconfiguration. In *Proceedings of the International Conference on Field Programmable Technology (FPT).* 1–6.

K. Vipin and S. A. Fahmy. 2012a. Architecture-Aware Reconfiguration-Centric Floorplanning for Partial Reconfiguration. In *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC).* 13–25.

K. Vipin and S. A. Fahmy. 2012b. A high speed open source controller for FPGA partial reconfiguration. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT).*

K. Vipin and S. A. Fahmy. 2013. An Automated Partitioning Scheme for Partial Reconfiguration based Adaptive Systems. In *Proceedings of Reconfigurable Architecture Workshop (RAW).*

K. Vipin and S. A. Fahmy. 2014a. DyRACT: A Partial Reconfiguration Enabled Accelerator and Test Platform. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL).*

K Vipin and S A. Fahmy. 2014b. ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq. *IEEE Embedded Systems Letters* 6, 3 (Sept. 2014), 41–44.

K. Vipin and S. A. Fahmy. 2015. Mapping Adaptive Hardware Systems with Partial Reconfiguration Using CoPR for Zynq. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS).*

F. Wang and J.J. Jean. 2012. Architecture and operating system support for two-dimensional runtime partial reconfiguration.

The Journal of Supercomputing 59, 2 (2012), 610–635.

L. Wirbel. 2014. *Xilinx SDAccel : A Unified Development Environment for Tomorrow's Data Center.* Technical Report. Xilinx Inc.

M.J. Wirthlin and B.L. Hutchings. 1995. A Dynamic Insruction Set Computer. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines.*

Z. Xiao, D. Koch, and M. Lujan. 2016. A partial reconfiguration controller for Altera Stratix V FPGAs. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL).*

Xilinx Inc. 1996. *Programmable Logic Data Book.*

Xilinx Inc. 2003. *DS031:Virtex-II Platform FPGAs.*

Xilinx Inc. 2004a. *XAPP151: Virtex Series Configuration Architecture User Guide.*

Xilinx Inc. 2004b. *Xilinx Device Drivers Documentation.* Xilinx Inc.

Xilinx Inc. 2006. *DS280: OPB HWICAP.* Xilinx Inc.

Xilinx Inc. 2008. *UG070: Virtex-4 FPGA User Guide.* Xilinx Inc.

Xilinx Inc. 2010. *DS586: XPS HWICAP.* Xilinx Inc.

Xilinx Inc. 2011a. *DS083: Virtex-II Pro and Virtex-II Pro-X Platform FPGAs.* Xilinx Inc.

Xilinx Inc. 2011b. *UG360 : Virtex 6 FPGA Configuration User Guide.* Xilinx Inc.

Xilinx Inc. 2013a. *UG585: Zynq-7000 All Programmable SoC Technical Reference Manual.* Xilinx Inc.

Xilinx Inc. 2013b. *UG682: PlanAhead User Guide.* Xilinx Inc.

Xilinx Inc. 2014. *UG910: Vivado Design Suite User Guide.* Xilinx Inc.

Xilinx Inc. 2015. *UG570: UltraScale Architecture Configuration.* Xilinx Inc.

Xilinx Inc. 2016. *UltraScale Architecture and Product Overview.* Xilinx Inc.

Xilinx Inc. 2017a. *UG1023: SDAccel Environment User Guide.*

Xilinx Inc. 2017b. *UG893: Vivado Design Suite User Guide.* Xilinx Inc.

Xilinx Inc. 2017c. *UG909: Vivado Design SuiteUser Guide Partial Reconfiguration.* Xilinx Inc.

D. Yin, D. Unnikrishnan, Y. Liao, L. Gao, and R. Tessier. 2011. Customizing Virtual Networks with Partial FPGA Reconfiguration. *ACM SIGCOMM Computer Communication Review* 41, 1 (Jan. 2011), 57–64.

J. Yuan, S. Dong, X. Hong, and Y. Wu. 2005. LFF algorithm for heterogeneous FPGA floorplanning. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC).* 1123–1126.

APPENDIX-I: LIST OF ABBREVIATIONS/ACRONYMS

ALM	Adaptive Logic Module
ASIC	Application Specific Integrated Circuits
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
DPGA	Dynamically Programmable Gate Arrays
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Arrays
HLS	High-Level Synthesis
ICAP	Internal Configuration Access Port
ILP	Integer Linear Programming
JTAG	Joint Test Action Group
LAB	Logic Array Block
LUT	Look Up Table
MCAP	Media Configuration Access Port
MC-FPGA	Multi-Context Field Programmable Gate Arrays
OpenCL	Open Computing Language
PCAP	Processor Configuration Access Port
PCIe	Peripheral Component Interconnect express
PL	Programmable Logic
PLC	Programmable Logic Cell
PLL	Phase Locked Loop
PR	Partial Reconfiguration
PRR	Partially Reconfigurable Region
PS	Processing System
RCM	Reconfigurable Context Memory
RTL	Register Transfer Level
SoC	System on a Chip
SRAM	Synchronous Random Access Memory
SDM	Secure Digital Managers
SDR	Software Defined Radio
SEU	Single Event Upset
TBUF	Tri-state Buffer
UCF	User Constraints File
UML	Unified Modeling Language
VHDL	Very High-speed integrated circuit Hardware Description Language
XML	eXtensible Markup Language
XDL	Xilinx Description Language
XST	Xilinx Synthesis Technology