

Mapping Adaptive Hardware Systems With Partial Reconfiguration Using CoPR for Zynq

Kizheppatt Vipin
School of Engineering Sciences
Mahindra École Central, Hyderabad, India
Email: vipin.kizheppatt@mechyd.edu.in

Suhaib A. Fahmy
School of Computer Engineering
Nanyang Technological University, Singapore
Email: sfahmy@ntu.edu.sg

Abstract—Dynamically adaptive systems respond to environmental conditions by modifying their processing at runtime, selecting alternative configurations of computation. While FPGAs with partial reconfiguration (PR) seem to offer an ideal platform for flexible hardware, designing such systems is difficult, and no standardised model and methodology exists. We present CoPR, a fully automated framework for implementing PR based adaptive hardware systems on the Zynq family of hybrid FPGAs. The designer specifies a set of valid configurations comprising hardware modules. CoPR automates partitioning of modules into regions, floorplanning on the FPGA fabric, and generation of partial bitstreams. The runtime framework offers an abstracted view of system configuration through an API that allows the designer to focus on adaptation software without considering details of the underlying hardware. We present a case-study on the design of a multi-standard adaptive radio system.

I. INTRODUCTION

Dynamically adaptive systems (DASs) continuously monitor their environment and adapt their behaviour in response to changes in environmental conditions [1]. They are able to deal with uncertainty in system environment through their adaptability. A DAS can be considered as a collection of different system operating modes, called *configurations*, of which only one is active at any given point in time [2]. A *configuration* is a valid combination of *primitives* which map to operations implemented in software or hardware, and is suited to certain operating conditions. At runtime, changes in the operating environment cause a DAS to switch its *configuration* to adapt to new conditions. The configuration switching operation, *reconfiguration*, is controlled and managed by a *configuration manager* (CM) that monitors parameters and applies *adaptation algorithms*. DASs have demonstrated applications in home automation [1], communication systems [3], and military systems [4].

Although extensive research has been done on software DASs, there has been limited work on mapping to hardware. As autonomous and adaptive systems rely on an ever-increasing number of sensors, with increasing fidelity, the computational processing required to implement advanced applications is exceeding what can be done in software, especially in embedded, portable, and mobile scenarios. Field Programmable Gate Arrays (FPGAs) offer hardware flexibility

by allowing a circuit to be changed by loading a new bitstream (configuration information). Partial Reconfiguration (PR) is even more appropriate as it allows part of the FPGA configuration memory can be overwritten at runtime, thus modifying the behaviour of only parts of a circuit, while the remainder continues to function. This capability, formerly only offered for high-end FPGAs, is now supported across most FPGAs from Xilinx [5], and recently, Altera [6]. PR enables systems that swap different *configurations* into the FPGA fabric at different times. Harnessing this capability in a systematic manner can enable the design of hardware DASs that offer the parallel, high-performance processing necessary for modern applications with the flexibility needed for adaptation.

While there is a healthy body of work on designing partially reconfigurable systems, focus has primarily been on time multiplexing a small FPGA by breaking a large application into a set of smaller tasks that are then swapped in and out of the FPGA according to the order and dependency among these tasks [7]. This allows use of a smaller FPGA than would otherwise be required for a fully spatial implementation, reducing cost. In such cases, the reconfiguration order, and hence adaptability, is restricted to a sequence determined at design time from the application description. As a result, the control mechanism is simple.

With DASs, we know in advance what configurations are possible, and the adaptation mechanism is more complex, relying on real-time processing of environmental information for decision making. Such adaptation is more suited to software implementation where advanced learning and recognition algorithms can be easily applied. This is where application experts innovate to improve robustness, generality, and application performance. The challenge then is how to integrate this adaptation layer with the underlying hardware to offer the benefits of hardware performance and software flexibility.

Hybrid FPGA platforms such as the Xilinx Zynq offer new opportunities for DAS implementation by tightly integrating processors with a reconfigurable fabric. The compute-intensive *configurations* can be implemented on the reconfigurable fabric while complex adaptation algorithms can be implemented in software, making them easily programmable. Such software/hardware systems are crucial to implementing new generations of cyber-physical systems with complex processing [8],

many of which have adaptive requirements.

DAS implementation on partially reconfigurable FPGAs comprises three important aspects. The first is a definition of regions on the device called partially reconfigurable regions (PRRs) that will house the hardware modules to be reconfigured at runtime. For each of these, we must generate a set of valid bitstreams to be loaded at runtime. The second is the underlying mechanism by which such bitstreams are stored and loaded at runtime. The final aspect is the high-level application code that manages reconfiguration at runtime in response to environmental changes.

PR research has focussed primarily on overcoming the limitations of vendor tools, typically through augmentation in the middle of the flow, and custom bitstream generation approaches. Although models have been proposed for mapping adaptive system descriptions to FPGAs, actual implementation of the resulting systems remains challenging [4], [9]. A fully automated flow that allows designers to map DAS applications to a PR implementation on a hybrid FPGA without the need for FPGA design expertise has so far failed to materialise. We believe this is an essential step in PR achieving more widespread adoption, as it is the application experts who can apply PR to realistic and emerging applications. A high-level flow, and clear interface to runtime adaptation software are crucial for such a framework.

Our main contributions in this paper are as follows:

- A proposed approach for modelling adaptive hardware systems for implementation on hybrid FPGAs.
- An automated end-to-end tool flow, suitable for non experts, that maps high-level DAS descriptions to a real implementations on hybrid FPGAs.
- A runtime configuration manager that provides an API for describing adaptation through an abstraction with automated seamless management of the PR process.

The rest of this paper is organised as follows: Section II discusses related work, Section III presents adaptive systems mapping, Section IV introduces the proposed tool flow, Section V presents a case study using the proposed method, and Section VI concludes the paper.

II. RELATED WORK

Partial reconfiguration has been the subject of significant research effort in recent years, however, much of the work has focussed on low-level techniques for overcoming the many limitations of vendor-provided tool flows. These include ways of using the same bitstream for modules in different locations [10] and run-time relocation of modules to maximise free capacity [11]. Such low-level techniques are highly-architecture dependent, resulting in complications when new architectures emerge. Furthermore, despite a significant body of contributions in such areas, this has not resulted in a more widespread adoption of PR. We argue that this is because the primary obstacle to adoption is not technical limitations, but rather design complexity.

Some research efforts recognised this challenge, and proposed operating systems for managing FPGAs [12]. Many of

these were concerned only with a fixed hardware configuration. Others supported loading of modules at runtime [13], but did not offer any abstraction of the management process. Other frameworks focussed solely on the use of PR to time multiplex the tasks of a large application [14].

Some more recent tool flows include OpenPR [15], which provides more placement flexibility than the vendor flow, and GoAhead [16] which improves on the way PR designs are routed. These tools can help overcome some limitations of the official flows, but do not address the high-level/abstract design issues. The use of an embedded processor to manage PR in the case of a time-multiplexed application has also been previously proposed [17]. An integrated high level synthesis framework for PR was proposed in [18], but the high level model is translated into a system model for simulation and a physically-aware architecture description that targets a virtual architecture, and hence cannot be mapped to real devices. The object-oriented framework in [19] abstracts runtime management through a Java-like language, but does not deal with the hardware aspects of PR design. A layered approach to PR systems is presented in [20]. It enables user-level applications to manage tasks running on the FPGA with reconfigurations translated into the appropriate low-level operations, but minimal detail on the design specification is provided. The framework in [21] proposes a separate control and data plane with an abstracted interface between them, but PR is still managed using low level functions in software.

Our main objective is to provide a high-level abstract view of PR for hardware DAS implementation with minimal need for FPGA architecture knowledge. This means a designer should be able to assemble hardware modules into a set of configurations, label them with relevant labels, then write adaptation software that references these labels, with all underlying operations managed automatically. These include design time tasks: partitioning the FPGA into regions, assigning modules to regions, creating wrappers for the region configurations, floorplanning PRRs on the FPGA, and exposing the configuration interface; and runtime tasks: interfacing the designer's adaptation code with the framework, automatic storing and loading of bitstreams as needed, and applying reconfiguration. Rather than circumvent the restrictions imposed by vendor flows, our approach optimises and automates through integration with the tools. This makes the framework portable as the vendor tools and architectures evolve. Our focus on modelling is the software-hardware system, and not the adaptation approach itself, since such techniques have already been proposed [22], and we aim to provide flexibility in this regard.

III. MAPPING ADAPTIVE SYSTEMS

This section describes different aspects of a DAS and its mapping on hybrid FPGAs.

A. System Decomposition

The system level architecture we propose is depicted in Fig. 1. The overall system is divided into two logical planes,

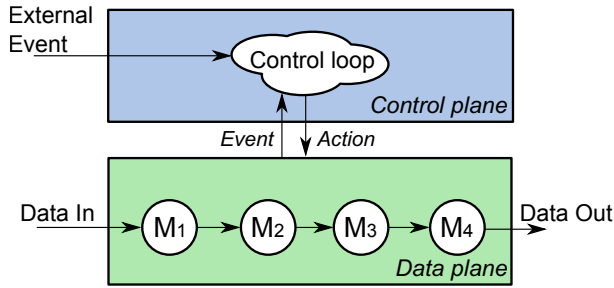


Fig. 1. The control and data planes of a DAS.

namely the *control plane* and the *data plane*, adapted from the model in [23]. The *configurations* that complete data processing comprise the data plane, while the control plane monitors and regulates system state, managing *reconfiguration*. The data plane can be made to support intensive computation by mapping parts of it to hardware. Meanwhile, the control plane typically functions at a much lower data rate, but might use complex sequential algorithms, and is hence more suitable for software implementation.

The data plane is composed of several functional units or *primitives*, such as M_1 , M_2 , M_3 and M_4 , interfaced with each other, as shown in Fig. 1. We define the atomic functional unit as a *module*, such as an edge detector in an image processing system, or a modulator in a radio system. Each module may have a set of parameters which determine its operating characteristics, such as the cut-off frequency of a filter module. These parameters can be modified at runtime to control the modules and thus data plane behaviour.

The control plane implements the DAS *configuration manager* (CM). The CM monitors and regulates system state by implementing the *observe, decide, act* loop [3]. This constantly monitors the system environment to detect changes in operating conditions called *events*. These events are analysed to decide whether changes in system state are required and how to reach the intended state through *actions*. Control plane actions usually involve modification of the data plane (*reconfiguration*) to support operation in the new environment.

B. Models of Computation

We model the data plane using Kahn Process Networks (KPNs), where a number of concurrent processes interact with each other through communication links [24]. Processes are functions executing asynchronously, which map input data elements or *tokens* to output tokens. Processes can interact with each other only through the communication *channels*, which are modelled as First-in First-Out (FIFO) queues with unbounded capacity. Each channel can possibly contain an infinite number of tokens, each of which can be produced and consumed only once. Writes to channels are non-blocking (write operations succeed immediately) but read operations are blocking. In other words, a process is stalled until it receives sufficient data from the input channels to satisfy the operation. Non-blocking writes mean each channel should have infinite capacity. KPNs are highly suitable for modelling

streaming applications such as video and audio processing, signal processing, and 3D multimedia applications [25], which are classical targets for FPGA implementation.

One difficulty with implementing KPNs in hardware is the requirement for unbounded channel FIFOs. To map KPNs to hardware, some restrictions and assumptions must be made. The FIFOs between the processes (modules) must be bounded in size and writes to them are blocked until there is sufficient space. If the output of one channel is shared by multiple processes (modules), read operations are blocked until all the consumer processes are ready to accept data. To avoid deadlocks, applications are restricted to unidirectional dataflow. In most streaming applications we are concerned with, this restriction is not problematic as dataflow is inherently unidirectional. We adopt the AXI4-Stream interface to implement inter-module communication due to its high throughput, and since Xilinx has adopted it as the standard for interfacing IP cores since the 6-series FPGAs. The first or last module in a chain is implemented in software, allowing the user application to source/sink data to/from the hardware chain. In the case of external interfaces like an RF interface, these interact directly with the hardware data plane modules using the same AXI4-Stream signalling.

C. Modelling Adaptation

A set of modules in the data plane which implements a mode of functionality is called a *configuration*. For example in Fig. 1, $\{M_1, M_2, M_3, M_4\}$ comprise a configuration. In a video processing system, these could be a $\{\text{filter, edge_detector, hough_transform}\}$ processing chain. The configuration gives a static snapshot of dynamic system operation. When the system adapts to a new configuration, one or more modules are replaced with new ones. This form of configuration switching is called a *structural reconfiguration*.

Alternatively, one or more parameters of the existing modules can be modified without replacing the modules themselves. This could be for actions like updating the coefficients of a digital filter. We call this form of reconfiguration a *parametric reconfiguration*, which is usually achieved through modifying a module's internal registers. Ideally a system designer should be able to model both these types of reconfiguration in a way that suits the application without worrying about how they are actually implemented.

D. Zynq Architecture

New hybrid reconfigurable devices, such as the Xilinx Zynq are an ideal choice for implementing DASs as they include a powerful processor, standard communication infrastructure, and an integrated reconfigurable fabric [26]. The Zynq tightly couples a dual-core ARM Cortex A9 processor with a reconfigurable fabric as shown in Fig. 2. The ARM processor communicates with on-chip memory, memory controllers, and peripheral blocks through AXI interconnect. Together, these hardened blocks constitute the Processor System (PS). The PS is attached to the Programmable Logic (PL) through multiple AXI ports, offering high bandwidth between the

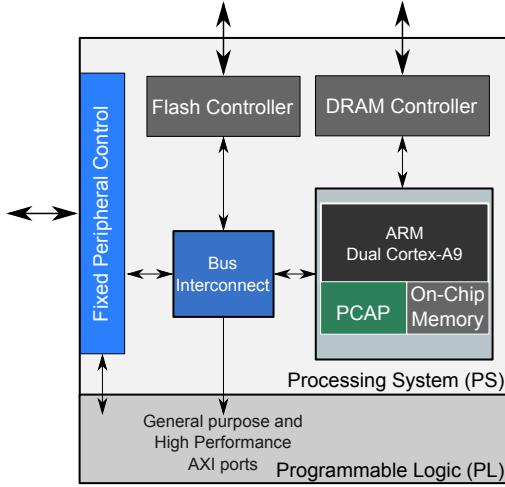


Fig. 2. Zynq SoC architecture showing the processor system and the programmable logic.

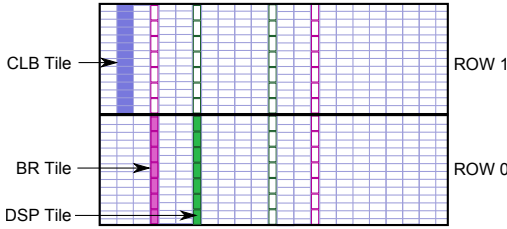


Fig. 3. Two rows of the Zynq PL showing different tile types.

two key components of the architecture. The PS processor configuration access port (PCAP) supports full and partial configuration of the PL.

Optimising PR based designs requires a solid understanding of the underlying configurable fabric architecture. Like other Xilinx FPGAs, the Zynq configurable fabric (PL) is divided into a number of rows and columns. Resources such as CLBs, Block RAMs, etc. are arranged in a columnar fashion extending the full height of the PL. A *tile* is one row high and one column wide, and contains a single type of resource, as shown in Fig. 3. The basic unit for defining a partially reconfigurable region (PRR) is a tile and a tile cannot be shared between multiple regions. For the Zynq PL, one CLB tile contains 50 CLBs, one DSP tile contains 20 DSP Slices, and one BRAM tile contains 20 Block RAMs arranged vertically. Present partial reconfiguration tool flows require the designer to know these low level architecture details for system implementation and optimisation. In our tool-flow, these details are abstracted away from the designer’s point of view but are used internally by the tool to optimise implementation.

E. Architecture Mapping

The DAS data plane is implemented on the Zynq PL with the software modules in the PS communicating through DMA over an AXI4-Stream interface. For maximum performance, it is preferable for a software module to only be used at the beginning or end of a hardware processing chain. Structural

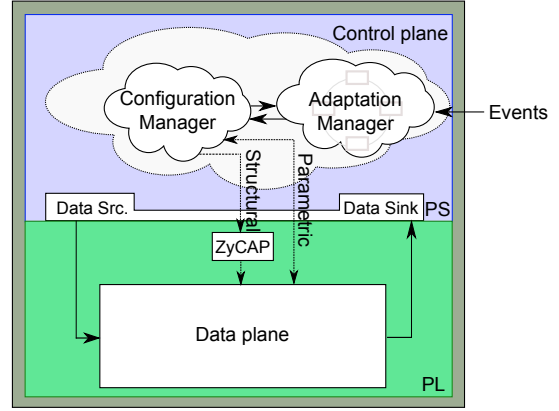


Fig. 4. Mapping of the proposed model to the Zynq architecture.

reconfiguration of the data plane is achieved by configuring the PRRs with appropriate *partial bitstreams*. The control plane is implemented as two logically separate software components called the *adaptation manager* and the *configuration manager* running on the Zynq ARM processor as shown in Fig. 4. The *adaptation manager* is software written by the system designer that implements the *observe, decide, act* loop discussed in Section III-A in an implementation independent format. It communicates with the *configuration manager* through the API provided by our framework. The *configuration manager* performs the architecture dependent structural and parametric reconfigurations by loading specific partial bitstreams or varying module register contents.

The *adaptation manager* can implement simple state machines or advanced techniques for adaptive system management [22]. Since the *adaptation manager* is written at a higher level and abstracted from the details of PR implementation by the *configuration manager*, this allows adaptation techniques to be explored independently of actual implementation. A lightweight *AXI4-Lite* interface is used between the control and data planes for managing parametric reconfiguration and monitoring events in the data plane.

IV. TOOL FLOW

Fig. 5 shows the proposed DAS design tool flow called CoPR for Zynq. The flow includes both software and hardware, accepts user specifications, applies optimisation algorithms, and interfaces with vendor tools through a set of custom scripts. The designer describes the overall system as a set of different configurations, each composed of modules from a library of parametrised modules, or custom modules designed to the required interface specification. They also describe how the system should adapt between different valid configurations in software. CoPR takes these descriptions and creates a working partially reconfigurable system without the designer needing to work at the detailed hardware level.

A. Specification

The primary designer inputs to CoPR are the *configuration* and *adaptation* specifications. The configuration specification

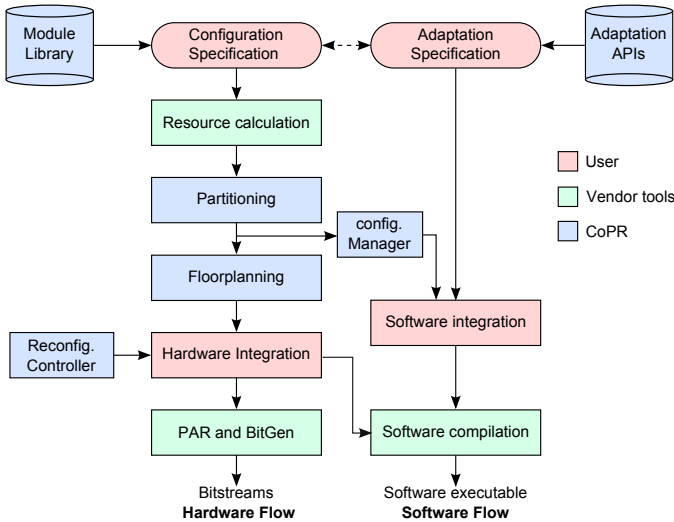


Fig. 5. CoPR for Zynq tool flow showing steps performed by the user, vendor tools, and the CoPR framework.

```

1 <configurations>
2 <config name="tx_chain">
3 <module name="encoder", source="encoder.v", input="DMA">
4 <parameter standard="enc1"/>
5 </module>
6 <module name="modulator", source="modulator.v", input="encoder">
7 <parameter standard="mod1"/>
8 </module>
9 </config>
10 <config name="rx_chain">
11 <module name="demodulator", source="demodulator.v" input="DMA">
12 <parameter standard="dmod1"/>
13 </module>
14 <module name="decoder", source="decoder.v", input="demodulator">
15 <parameter standard="dcod1"/>
16 </module>
17 </config>
18 ...
19 </configurations>

```

Fig. 6. Configuration specification in XML format. Each configuration is specified by its name and the list of modules it connects.

details the different valid system configurations and the corresponding library modules present in each configuration. It is entered in XML format as shown in Fig. 6. Each configuration has an associated *name* and the associated modules in the processing chain. For each module, the HDL *source file* (in Verilog) and its data source in the processing chain are also specified. A special setting (*input = DMA*) indicates the start of the chain with a software source supplying data over DMA. While selecting modules, the user can specify which parameters they require access to at run-time and constrain their possible values. These parameter modifications may lead to parametric or structural reconfiguration, which the runtime takes care of.

Module parameters can be changed at runtime, and as discussed in Section III-C, typically result in setting internal register values. CoPR automatically analyses parameter definitions and elaborates the configuration specification to include additional configurations resulting from parametric reconfiguration of the system. The distinction between parametric and structural reconfiguration is thus abstracted from the designer’s point of view, so that they have a unified model

for reconfiguration.

The *adaptation specification* contains the software code for the *adaptation manager* described in Section III-D. Since the low-level configuration management details are transparent to the adaptation manager, it is implemented based on the configuration names specified in the configuration specification using general API functions.

B. Partitioning and Interface Generation

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. This step can significantly impact the resource utilisation and reconfiguration time of the final implementation. We use our automated approach in [27] which uses configuration information, and low-level architecture details to arrive at an efficient partitioning. This step produces a list of PRRs and the corresponding module allocation to them. The Configuration Manager (CM) is also generated in this phase. It offers the generic interface to the designer’s adaptation specification, translating all reconfiguration requests at runtime into low-level operations as required based on the resulting partition.

Wrapper modules that instantiate the required modules are then generated for each PRR, ensuring a unified interface across different configurations. A *pr_system_top* wrapper is also generated which instantiates and connects all the PRRs as *black boxes*. The generated wrapper module is in IP core format, which can be directly imported to Xilinx’s XPS tool flow. As detailed in Section III-D, using a consistent interface (AXI4-Stream and AXI4-Lite) across all the modules enables automatic wrapper generation and automatic region instantiation.

C. Floorplanning

The next step is to determine the physical locations of the PRRs on the PL fabric. Regions must be rectangular in shape and should not overlap. As discussed in Section III-D, the basic unit for floorplanning is a *tile*. The kernel tessellation approach in [28] is used to generate an efficient floorplan, resulting in a *user constraints file* (UCF) that specifies the coordinates of all PRRs.

D. Hardware Integration

At this stage the designer can add the outputs of partitioning (*pr_system_top* wrapper) and floorplanning (UCF file) to a Zynq embedded project using the Xilinx XPS software. The AXI4-Lite interfaces of the wrapper module coming from the reconfigurable modules are connected to a processor AXI master interface. This step can also be automated, but designer input offers the flexibility to choose additional system peripherals and to connect the input/output data stream from the PR system either to the system memory or to external peripherals. The designer is offered the choice between using the PCAP controller in the PS, or the faster ZyCAP controller [29] we provide as part of CoPR.

E. Placement and Routing and Bitstream Generation

The designer now runs the CoPR automation scripts which direct the vendor placement and routing tools, then the bitstream generation tool to generate all the necessary partial bitstreams and the full system bitstreams. The generated partial bitstreams must then be copied to an SD card which is inserted in the board. At system start-up, these partial bitstreams are copied to DRAM automatically by the reconfiguration manager, for optimal reconfiguration performance. CoPR is written in Python and integrates with Xilinx command-line tools and EDK 14.6.

F. Software Implementation

As described in Section IV-A, the adaptation specification is programmed by the user, referring to the configuration names defined in the configuration specification. Presently it is to be written in *C* compatible with the Zynq ARM compiler. CoPR automatically generates the configuration manager (CM) based on the *configuration specification* and the output of the partitioning step.

API functions are provided for determining the present configuration (`get_config()`) and for changing configuration (`set_config(configuration_name)`). The designer does not need to know which partial bitstream corresponds to which configuration or where they are stored. The *configuration_name* used in the function is the same as the one specified by the user in the configuration specification file.

Additional `get_param(module, parameter)` and `set_param(module, parameter, value)` functions are provided for accessing hardware module parameters and modifying them. Changing parameters sometimes leads to the reconfiguration of modules, which is automatically handled by the CM. Unlike other PR management flows, the user does not have to explicitly call the partial bitstreams corresponding to a specific configuration. This means that the software developer can think in terms of modules and configurations, rather than bitstreams or regions. All required PRR reconfigurations and parameter changes are handled automatically. Fig. 7 shows a sequence diagram corresponding to a `set_config()` function call from the adaptation manager which results in two PRR reconfigurations. The CM seamlessly manages the process in a way that is transparent to the designer.

The CM also contains the ZyCAP driver which takes care of bitstream caching and is non-blocking in nature, meaning the low-level reconfiguration tasks do not adversely impact other running software. The software modules are compiled to generate the final software executable.

V. CASE STUDY

To demonstrate the effectiveness of CoPR, we implement a multi-standard cognitive radio transmitter, comprising the blocks shown in Fig. 8. The baseband transmitter can be configured with different OFDM symbol lengths and frame formats to support three standards: IEEE 802.11, IEEE 802.16, IEEE 802.22,

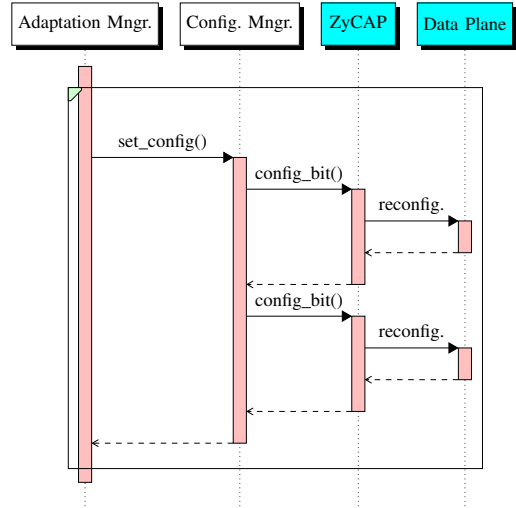


Fig. 7. Sequence diagram showing a `set_config()` function call causing reconfiguration of two regions.

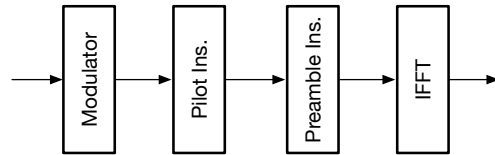


Fig. 8. An active processing chain for the radio transmitter case study.

and IEEE 802.22. An active transmission chain has the structure shown in Fig. 8. The main specifications of the transmitter blocks are summarised in Table I.

TABLE I
PROCESSING SPECIFICATIONS FOR THE CASE STUDY.

Specifications	IEEE 802.11	IEEE 802.16	IEEE 802.22
FFT size (N_{FFT})	64	256	2048
CP Length	16	32	512
Number of data carriers	48	192	1440
Number of pilots	4	8	240
Modulation schemes	QPSK, 16-QAM, 64-QAM		

The *Modulator* supports QPSK, 16-QAM, and 64-QAM modulation schemes. The *Pilot Ins.* block forms the OFDM symbol according to the specification of the different IEEE standards. The preamble (used for active gain control, frame detection, synchronisation and channel estimation at the receiver) is inserted by the *Preamble Ins.* block. The *IFFT* block performs the inverse-fast Fourier transform (IFFT) to modulate the sub-carriers in the frequency domain. The parameters of the *pilot* (NUM_PILOT), *preamble* (CP_LEN) and *IFFT* (LEN) modules are standard-dependent and their modification leads to structural reconfiguration. Overall, combining the different OFDM standards and modulation schemes, there are 9 valid *configurations*, as shown in Table II.

TABLE II
RADIO TRANSMITTER CONFIGURATIONS.

Configuration	Modulator	Pilot	Preamble	IFFT
Tx_11_Q	QPSK	PI ₁₁	PR ₁₁	IFFT ₁₁
Tx_11_16	QAM16	PI ₁₁	PR ₁₁	IFFT ₁₁
Tx_11_64	QAM64	PI ₁₁	PR ₁₁	IFFT ₁₁
Tx_16_Q	QPSK	PI ₁₆	PR ₁₆	IFFT ₁₆
Tx_16_16	QAM16	PI ₁₆	PR ₁₆	IFFT ₁₆
Tx_16_64	QAM64	PI ₁₆	PR ₁₆	IFFT ₁₆
Tx_22_Q	QPSK	PI ₂₂	PR ₂₂	IFFT ₂₂
Tx_22_16	QAM16	PI ₂₂	PR ₂₂	IFFT ₂₂
Tx_22_64	QAM64	PI ₂₂	PR ₂₂	IFFT ₂₂

```

1 <configurations>
2   <config name="conf_802_11">
3     <module name="modulator", source="modulator.v", input="DMA">
4       <parameter SCM="QPSK,QAM16,QAM64"/>
5     </module>
6     <module name="pilot", source="modulator.v", input="modulator">
7       <parameter NUM_PILOT="4"/>
8       <parameter TYPE="0"/>
9     </module>
10    <module name="preamble", source="pilot.v", input="pilot">
11      <parameter CP_LEN="16"/>
12    </module>
13    <module name="ifft", source="ifft.v", input="preamble">
14      <parameter LEN="64"/>
15    </module>
16  </config>
17  ...
18 </configurations>

```

Fig. 9. Configuration specification in XML format for 802.11 standard.

In the *configuration specification*, only three are listed (conf_802_11, conf_802_16 and conf_802_22) with a modulator parameter (*SCM*) that has three possible values, resulting in the 9 configurations. Fig. 9 shows the configuration specification for the IEEE 802.11 configurations. Note that from the designer’s perspective, there are 3 radio chains with 3 possible modulation schemes each, set by the *SCM* parameter, which can take any of the three values (QPSK, QAM16 and QAM64). A single configuration is expanded into three separate configurations by CoPR during the configuration specification analysis. The parameter specifying the pilot *TYPE* can also accept different runtime values, which determines the sub-carrier type (such as null, data, positive pilot, negative pilot). But modifying this parameter causes no structural reconfiguration since this parameter only sets an internal register through the AXI4-Lite interface.

```

1   ...
2   switch(configuration) {
3
4   case conf_802_11:
5     if (gpio == 2) {
6       mod = get_param(modulator, SCM);
7       if (mod == "QPSK")
8         set_config(conf_802_16);
9       else if (mod == "QAM16")
10        set_config(conf_802_22);
11    }
12    else if (gpio == 3)
13      set_param(modulator, SCM, "QAM64")
14    break;
15    ...

```

Fig. 10. Adaptation specification code excerpt.

The adaptation manager is written in C as a state machine where each state represents a configuration listed in the *configuration specification* file as shown in Fig. 10. External events to trigger changes in configuration were emulated using physical input pins that initiate a request to the reconfiguration manager.

The baseband transmitter is implemented on a Xilinx ZC702 evaluation board hosting a Zynq XC7Z020. Running the proposed partitioning flow on the configuration specification generates a design with two PRRs, one containing only the Modulator and the other containing the Pilot, Preamble, and IFFT blocks. This reflects the expectation we might have, given the different configurations, but importantly, the designer need not determine or even be aware of this.

In Table III, we compare the resource requirements for the design generated by CoPR, and using standard module-based partitioning. A static fully-multiplexed implementation is included for reference.

TABLE III
RESOURCE UTILISATION FOR DIFFERENT IMPLEMENTATIONS.

Implementation	Registers	LUTs	BRAMs	DSPs
Static	23223	17701	18	30
Single Region	15094	11089	11	15
Four Region	15364	11851	11	15
Proposed framework	15114	11204	11	15

It is clear that the PR approaches offer a significant saving over a multiplexed static implementation in terms of area. The scheme proposed by CoPR is also more efficient than a standard one-region-per-module scheme, while being within 1% of the resource usage of the most resource efficient single-region scheme.

TABLE IV
RECONFIGURATION TIME FOR PR AND NON-PR DESIGNS.

Scheme	Bitstream (Bytes)	Reconf. time (ms)
Full reconfiguration	4045564	31.12
Single Region	706192	5.18
Four Regions		
Region-1	14544	0.11
Region-2	14544	0.11
Region-3	677104	5.12
Region-4	14544	0.11
Total	720736	5.45
Proposed PR method		
Region-1	14544	0.11
Region-2	691648	5.07
Total	706192	5.18

Table IV shows the reconfiguration times for different schemes using the Zynq PCAP (to allow for comparison with a full reconfiguration). Using the ZyCAP controller included in CoPR would reduce reconfiguration time by a factor of 3. A single region scheme requires the whole region to be

reconfigured any time one module is reconfigured. Since the total resource requirement for the four region scheme is higher, the total reconfiguration time is higher than the arrangement determined by CoPR. CoPR places the modulator in its own region, minimising the time taken to switch modulation schemes too.

We have seen that CoPR generates an efficient PR design in terms of both reconfiguration time and area, and does so without the designer working at the FPGA architecture level. Details of the physical implementation are adapted to allow the adaptation specification to be programmed at a higher level, greatly easing the design of PR systems.

VI. CONCLUSIONS AND FUTURE WORK

We have introduced CoPR, a framework for the design of dynamically adaptive hardware systems using partial reconfiguration on Xilinx Zynq hybrid FPGAs. It takes a high-level description of an adaptive system, automatically partitions the design into reconfigurable regions and determines a floorplan. Runtime adaptation control is also automatically generated, isolating the designer from the low-level aspects of the implementation. We have shown an example cognitive radio application to demonstrate the effectiveness of CoPR.

We are working on porting the reconfiguration manager to Linux and integrating CoPR with the new Xilinx Vivado tool flow. CoPR is available as open source software to allow adaptive systems designers to more easily implement adaptive hardware systems using PR on hybrid FPGAs [30].

REFERENCES

- [1] B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2009, pp. 468–483.
- [2] H. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes, "Goal-based modeling of dynamically adaptive system requirements," in *Proceedings of International Conference and Workshop on the Engineering of Computer Based Systems*, 2008, pp. 36–45.
- [3] J. Mitola and G. Q. Maguire, "Cognitive radio: making software radios more personal," *IEEE Personal Communications*, vol. 6, no. 4, pp. 13–18, 1999.
- [4] T. Bapty, S. Neema, J. Scott, J. Sztipanovits, and S. Asaad, "Model-integrated tools for the design of dynamically reconfigurable systems," *VLSI Design*, vol. 10, no. 3, pp. 281–306, 2000.
- [5] *UG702: Partial Reconfiguration User Guide*, Xilinx, 2010.
- [6] M. Bourgeault, "Altera's partial reconfiguration flow," Altera, Tech. Rep., 2011.
- [7] H. Walder, C. Steiger, and M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-hashing," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.
- [8] K. Vipin, S. Shreejith, S. A. Fahmy, and A. Easwaran, "Mapping time-critical safety-critical systems to hybrid FPGAs," in *Proceedings of the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2014, pp. 31–36.
- [9] J. Vidal, F. De Lamotte, G. Gogniat, J.-P. Diguët, and S. Guillet, "Dynamic applications on reconfigurable systems: from UML model design to FPGAs implementation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2011.
- [10] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "REPLICA: A bitstream manipulation filter for module relocation in partial reconfigurable systems," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.
- [11] S. P. Fekete, J. C. van der Veen, A. Ahmadinia, D. Ghringer, M. Majer, and J. Teich, "Offline and online aspects of defragmenting the module layout of a partially reconfigurable device," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 9, pp. 1210–1219, Sept. 2008.
- [12] H. K. H. So and R. W. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2006.
- [13] M. Santambrogio, V. Rana, and D. Sciuto, "Operating system support for online partial dynamic reconfiguration management," in *Proc. of International Conf. on Field Programmable Logic and Applications (FPL)*, 2008, pp. 455–458.
- [14] E. Carvalho, N. Calazans, E. Briao, and F. Moraes, "PADReH - a framework for the design and implementation of dynamically and partially reconfigurable systems," in *Proceedings of Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2004, pp. 10–15.
- [15] A. Sohanchpurwala, P. Athanas, T. Frangieh, and A. Wood, "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)*, 2011, pp. 228–235.
- [16] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A partial reconfiguration framework," in *Proceeding of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 37–44.
- [17] A. Donato, F. Ferrandi, M. Redaelli, M. Santambrogio, and D. Sciuto, "Caronte: A methodology for the implementation of partially dynamically self-reconfiguring systems on FPGA platforms," in *Proceedings of International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC)*, 2007, vol. 87, pp. 87–109.
- [18] M. Boden, T. Fiebig, M. Reiband, and P. Reichel, "GePaRD - a high-level generation flow for partially reconfigurable designs," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2008.
- [19] N. Abel, "Design and implementation of an object-oriented framework for dynamic partial reconfiguration," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2010.
- [20] H. Tan and R. DeMara, "A multilayer framework supporting autonomous run-time partial reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 5, pp. 504–516, May 2008.
- [21] S. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser, "Generic software framework for adaptive applications on FPGAs," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009, pp. 55–62.
- [22] C. Bolchini, M. Carminati, A. Miele, and E. Quintarelli, "A framework to model self-adaptive computing systems," in *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2013, pp. 71–78.
- [23] J. Lotze, S. Fahmy, J. Noguera, and L. Doyle, "A model-based approach to cognitive radio design," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 29, no. 2, pp. 455–468, Feb. 2011.
- [24] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, 1974, pp. 471–475.
- [25] M. Geilen and T. Basten, "Requirements on the execution of Kahn Process Networks," in *Proceeding of European Symposium on Programming*, 2003, pp. 319–334.
- [26] *UG585: Zynq-7000 All Programmable SoC Technical Reference Manual*, Xilinx, Mar. 2013.
- [27] K. Vipin and S. A. Fahmy, "Automated partitioning for partial reconfiguration design of adaptive systems," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshops – Reconfigurable Architectures Workshop*, 2013, pp. 172–181.
- [28] K. Vipin and S. A. Fahmy, "Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration," in *Reconfigurable Computing: Architectures, Tools and Applications – Proceedings of Int. Symp. on Applied Reconfigurable Computing (ARC)*, 2012, pp. 13–25.
- [29] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded System Letters (ESL)*, vol. 6, no. 3, pp. 41–44, 2014.
- [30] CoPR GitHub public repository. [Online]. Available: <https://github.com/archntu/copr/>