FPGA VIRTUALISATION ON HETEROGENEOUS COMPUTING SYSTEMS

—

MODEL, TOOLS, AND SYSTEMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2020

Khoa Dang Pham

School of Computer Science

Contents

List of Tables

List of Figures

Abstract

Field Programmable Gate Arrays (FPGAs) can be utilised to speed up applications by two orders of magnitude as compared to running on conventional CPUs. However, designing FPGA accelerators still remains challenging for most users. Furthermore, despite the trend of integrating FPGA resources into high-performance and cloud computing systems, FPGA management is still immature commonly following a run-to-completion execution model and with providing no abstraction layers to underlying hardware. Thus, FPGA virtualisation is highly desired to provide an abstraction level for FPGA development and deployment in complex heterogeneous CPU+FPGA computing systems which provide us an opportunity to adapt the workload on either software (running on CPUs) or hardware (running on FPGAs). This PhD thesis aims at promoting FPGA virtualisation for such heterogeneous computing systems ranging from embedded, to edge, to high-performance and cloud nodes. Consequently, this thesis proposes a fully FPGA-virtualised computing model to tackle these obstacles on complex heterogeneous CPU+FPGA systems. Moreover, partial reconfiguration is one of the key techniques to implement the proposed model, yet has still significant limitations which prevented us to implement such model on real hardware. These limitations motivated this PhD project and resulted in the development and implementation of several solutions to overcome these limitations, and hence, advance the partial reconfiguration technique towards the propsed fully FPGA-virtualised computing model. Combining this new capability of partial reconfiguration with other academic design tools, a novel design methodology has been developed to realise the proposed model on real hardware. Finally, resulting systems of the proposed design methodology on various heterogeneous computing platforms have shown significant improvements in compute performance thanks to the here implemented FPGA-virtualised techniques.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisor, Dr. Dirk Koch, for his patient guidance, enthusiastic support, and strong encouragement. His vision, experience and expertise helped to clarify my doubts and overcome the hurdles. I also thank Dr. James Garside, my co-supervisor, for his valuable feedback and advice.

I appreciate my friends Anuj Vaishnav, Edson Horta, Malte Vesper, Jose Raul Garcia, Kristiyan Manev and Athanasios Stratikopoulos for their help, support, encouragement, deep insights, thorough discussions, and fruitful collaborations during my research of this PhD project. People from APT group are also thankful for the positive and professional research environment.

I also thank to K. Georgopoulos, A. Ioannou, I. Mavroidis, and P. Malakonakis from the ECOSCALE project for the friendly support and effective collaborations.

Finally, I would like to thank my parents, my loving family, and my dear Vietnamese friends in Manchester whose support and understanding helped me to achieve this milestone.

This work is supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632).

Chapter 1

Introduction

People who are really serious about software should make their own hardware.

Alan Kay

Field Programmable Gate Array (FPGA) devices can be used not only to substantially accelerate applications, but they are also able to achieve these performance advantages with fairly low power overhead [CSPJ03]. Therefore, more and more software applications such as deep learning [AOC+17], computer vision [JCP+10], telecommunication [BCB18], and networking [LMW+07] are being implemented as FPGA-based accelerators to exploit high performance and energy efficiency. Moreover, with the capability of reconfiguration and short time-to-market, the demand for FPGAs has increased in many industrial domains, as summarised in Figure 1.1.

However, FPGA accelerators are difficult to develop for most of software developers as they have to deal with new programming languages/paradigms, and/or coding styles, as well as several low-level system development including IP cores, system integration, bootloaders, and drivers which can be error-prone and time-consuming. In addition, despite the availability of large capacity devices and demand for more complex applications, FPGA management is still rudimentary without real abstraction layers to underlying resources. This renders FPGA acceleration similar to the traditional bare-metal embedded application use case. In those systems, applications are able to access the underlying hardware freely but are not able to switch to another application arbitrarily or to start new processes at run-time.

There are two major types of FPGA systems: 1) heterogeneous (SoC-based) computing systems, such as Intel Xeon-Arria hybrid chips [Huf], Xilinx Adaptive Compute Acceleration Platform [Xil19a], and Xilinx Zynq UltraScale+ MPSoC devices [Xil17b], in which CPUs and FPGAs are tightly coupled on the same die; and 2) PCIe-based systems, which are targeting high performance computing (HPC) and data centre markets. Hence, it is worth to highlight the differences of architecture and organisation between the SoC-based and PCIe-based systems.

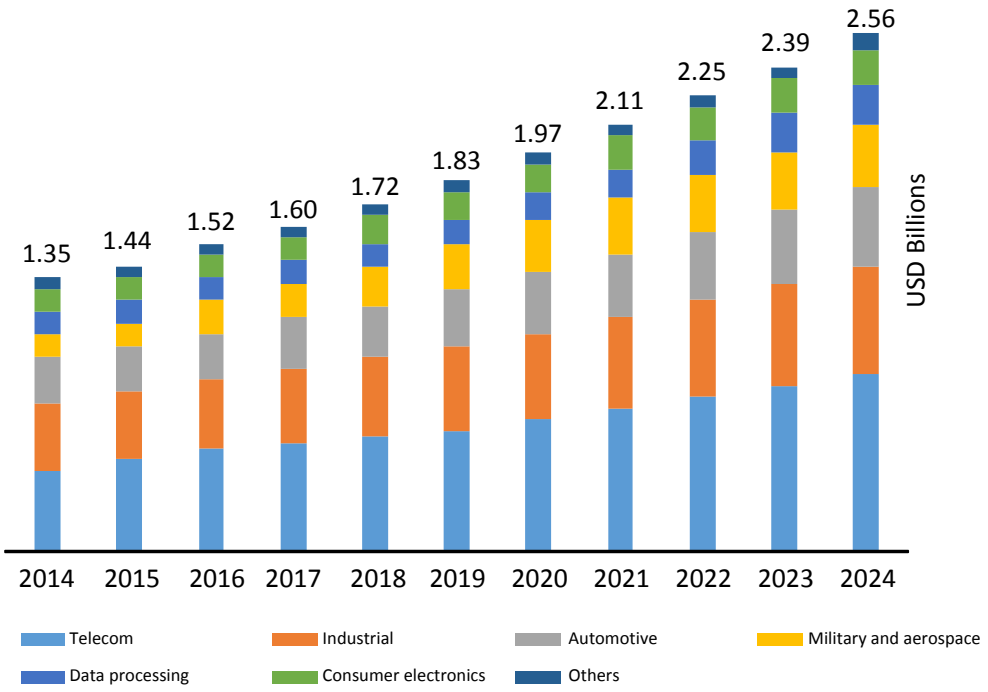


Figure 1.1: The U.S. FPGA Market by applications, 2014 – 2024 (USD Billion) [Gra18]

A heterogeneous computing system can utilise the built-in system bus interface (such as ARM AMBA [ARMa]) to communicate with main memory and the host CPU. Therefore, no reconfigurable logic resources have to be used to implement a PCIe core and DDR controllers, which can contribute as much as 70% of the resource utilisation of the PCIe-based static system, as reported in [VKP17]. Furthermore, the bus interconnect that is required to connect the different modules with memory and resources needed to cross clock domains between kernel and DDR memory is hardened on heterogeneous devices. The hardened bus interconnect allows for higher bus clock frequencies than in PCIe-based systems. An SoC-based system allows accelerators working directly on the main memory used by the CPU, saving the transfers to and from dedicated FPGA memory required by PCIe-based counterparts. Therefore, an SoC-based system is often able to deliver higher clock frequency, more efficient memory access, and lower energy requirements, while a PCIe-based system provides space for larger kernels and allows choosing the host CPU alongside more various host memory configurations.

Since this PhD project aims at deploying reconfigurable resources in high performance computing systems as well as data centres [MPL+16, SVC+16], there is a need to adapt swiftly to changing workloads and to a different number of tasks to execute [MPL+16]. This swift adaptability does not fit the traditional methods of operating FPGAs. Moreover, to achieve high utilisation, individual systems often have a number of active tasks, each operating without any impact or knowledge of each other. For such systems, it would be highly desirable to have a middleware that can automatically adapt for any workload scenario that may arise at run-time

in the best possible way.

With the rise of heterogeneous computing systems, we have now an unprecedented opportunity to adapt the workload on either software (running on CPUs) or hardware (running on FPGAs), as shown in Figure 1.2. Therefore, FPGA virtualisation is required to provide an abstraction level for both FPGA development and deployment in an analogue fashion as known from traditional software operating systems (OSs).

Overall objectives of such FPGA virtualisation can be summarised in [VPK18a, VPK18b] as following:

- **Multi-tenancy**: The ability to serve multiple different users simultaneously by sharing the same FPGA fabric.
- **Resource Management**: Providing an abstraction/driver layer to the FPGA fabric and means of scheduling tasks to the CPU and the FPGA as well as monitoring the CPU+FPGA resource usage.
- **Flexibility**: Ability to support a wide range of acceleration workload, i.e. from custom accelerators handcrafted in RTL to accelerators designed in a High-Level Language (HLL) or a Domain Specific Language (DSL).
- **Isolation**: Providing the illusion of being a sole user of the FPGA resources for better security, fewer dependencies and correctness of the task execution.
- **Scalability**: A system/application that can scale to multiple different FPGAs or can support multiple different users at relatively low overhead.
- **Performance**: The impact of virtualisation should be minimal on the performance achievable and therefore, should maximise FPGA resources usable by the user application.
- **Security**: Ensuring that information of other tenants is not leaked and for safekeeping the infrastructure from malicious users. This also contributes to overall robustness of a system.
- **Resilience**: Ability to keep the system/service running despite failures.
- **Maintenance**: Ability to update hardware/software in a node without affecting the executing applications.
- **Design Productivity**: Improving the time-to-market and reducing the complexity of deploying a design to an FPGA from its software description.

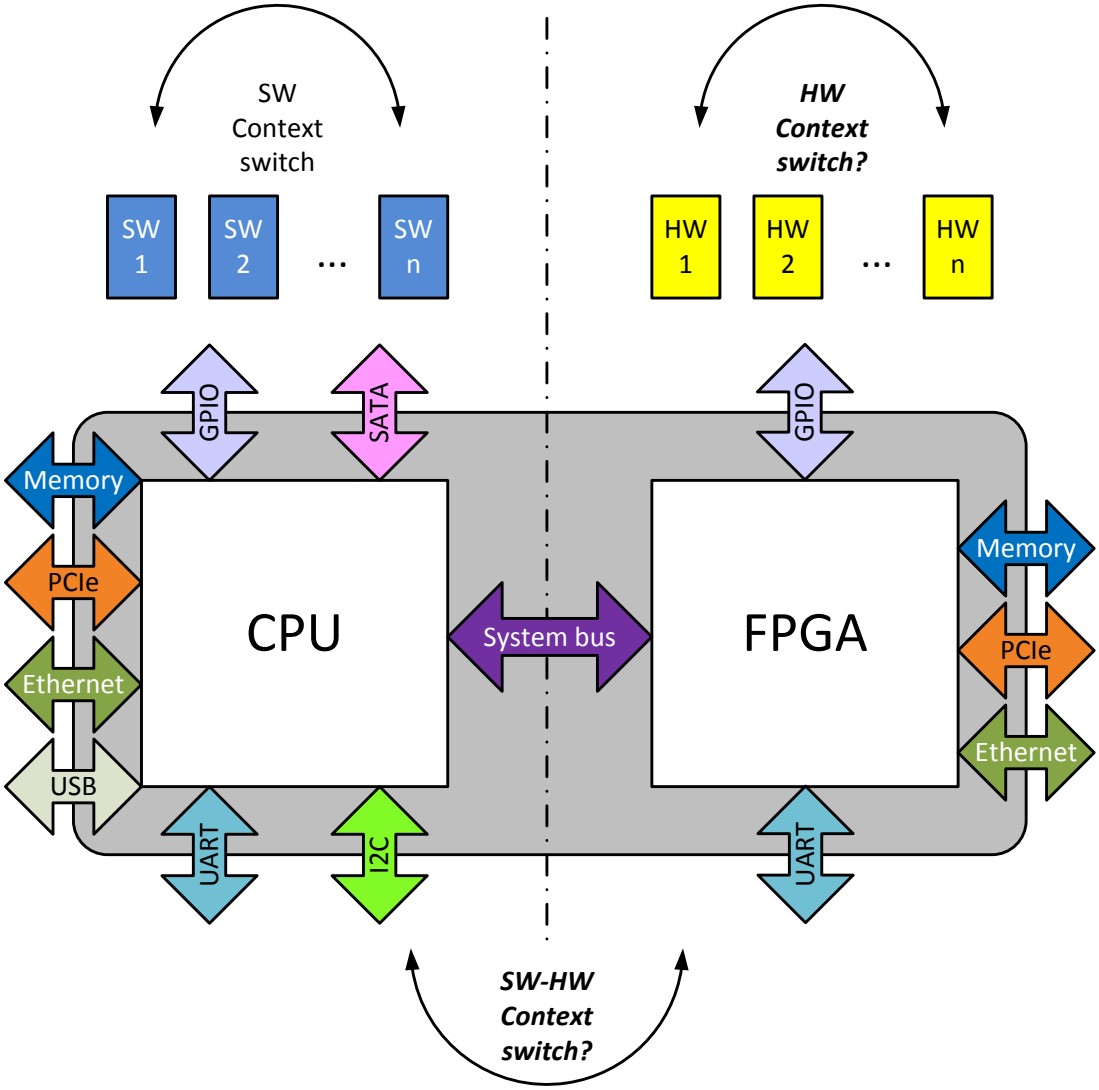


Figure 1.2: The organisation of a heterogeneous CPU+FPGA computing system. Depending on the application, peripherals and memory may be connected to either the CPU, the FPGA fabric, or arbitrary to both. This implies some challenges, in particular if both parts the CPU and the FPGA need access to the peripherals or a shared memory. Moreover, although context switching on the CPU side (SW context switching) is a mature topic in computer science, performing context switching on the FPGA side (HW context switching) or between both sides (HW-SW context switching) is another challenge and open question.

The FPGA ecosystem is steadily improving and IP integration has become much easier thanks to a widespread adoption of the ARM Advanced Micro-controller Bus Architecture (AMBA) specification [ARMa]. Similarly, application development is now well-supported through standardised High-level synthesis (HLS) approaches, such as OpenCL [KHR]. Moreover, partial reconfiguration (PR) [Koc12], a noticeable feature allowing FPGAs to change a portion of reconfigurable hardware circuitry while the rest of the FPGA is still functioning, can be utilised to decouple accelerator design from the underlying platform and to switch between various hardware applications arbitrarily. A PR system includes a *static part* which does not change the functionality during system operation, and the *reconfigurable part* with one or more *PR regions*. Therefore, combining these technologies would allow software developers building their accelerators once but instantiating them multiple times in the provided regions on the platform in a hot plug-and-play manner. Such kind of feature is essential towards building FPGA-virtualised systems. Despite this progress, there are still few examples of methodologies, toolflows, and systems available that fulfil the aforementioned objectives of FPGA virtualisation.

1.1 Motivation and Aims

The main motivation of this research is to advance the existing knowledge of FPGA virtualisation ranging from embedded/edge to high-performance/cloud computing systems. Thus, key research questions tackled in this thesis include:

- **Models and Methodologies**: How to efficiently virtualise and abstract resources in heterogeneous computing systems? What should the model of an FPGA-virtualised heterogeneous computing system be? How to speed up the design, implementation, and deployment of hardware applications?

- **Systems**: How to launch and manage multiple hardware tasks transparently? How to dynamically orchestrate software and hardware context switching in heterogeneous computing systems for better performance and better utilisation of resources? And how to protect data and execution of a specific user from another on a multi-tenanted platform?

- **Tools and Flows**: How to physically implement FPGA-virtualised heterogeneous computing systems at a high level of abstraction with the latest FPGA technology?

Therefore, this PhD project is going to 1) investigate state-of-the-art FPGA virtualisation techniques; 2) propose a novel approach/model to virtualise FPGA resources in heterogeneous computing systems; 3) explore novel design methodology, tools, and frameworks to implement the proposed approach; 4) propose a compilation path for productive application development;

5) provide an abstraction layer between accelerators and the underlying platform as a ready-to-use hardware operating system (OS) infrastructure; 6) enable run-time support for running, sharing and swapping multiple accelerators; and 7) investigate mechanisms to make the system safer, more secure, and more resilient.

1.2 Contributions

To tackle the aforementioned research questions, this PhD thesis proposes a 1) novel model of FPGA virtualisation including three level of resource abstractions (i.e. infrastructure abstraction, development abstraction, and deployment abstraction) as well as 2) introducing a tool for bitstream abstraction supporting modern Xilinx FPGAs, 3) a novel design methodology targeting heterogeneous computing systems, and 4) FPGA-virtualised system prototypes, as discussed in the following paragraphs.

1.2.1 FPGA Virtualisation Model for Heterogeneous Computing Systems

Based on the deep understanding and thorough analysis of state-of-the-art trends and techniques on FPGA virtualisation [VPK18a] as well as due to the increasing popularity of heterogeneous computing systems (see Figure 1.2), this thesis proposes that **FPGA resources in heterogeneous systems should be abstracted, managed, and virtualised in an analogous way as the way that is known from software operating systems (OSs)**. With this model, we can contrast and compare OS services known from software system with services needed for FPGA management. The equivalent concept of the software OS kernel is named the *FPGA shell*, or shell for short and it provides all essential infrastructure, whereas the user applications (often called *roles*) do perform the actual work. A brief comparison between a software OS and the proposed model of FPGA virtualisation is shown in Table 1.1. Ultimately, the proposed model is a great fit to partial reconfiguration (PR) of FPGAs, in which a *shell* is located in the fixed part of a PR system, while *roles* are assigned to reconfigurable parts, called *slots* (i.e. PR regions), and deployed at run-time in a hot plug-and-play manner, as illustrated in Figure 1.3.

To implement the proposed model for achieving the aforementioned objectives of FPGA virtualisation, this thesis proposes the following abstractions:

- **Infrastructure abstraction:** the underlying heterogeneous resources including their various layouts and available primitives should be designed, abstracted and managed as shells to provide common essential services, such as system bus communication, I/O access, and memory access to the deploying roles.
- **Development abstraction:** the requirements for this level of abstraction are:

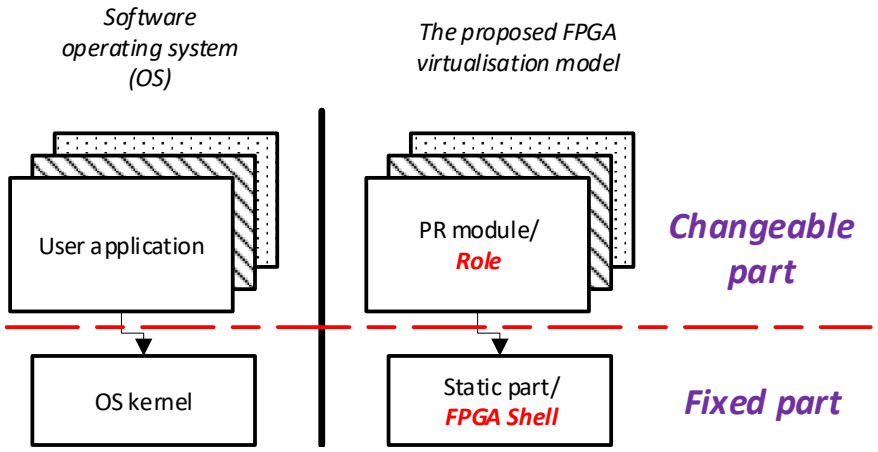


Figure 1.3: Analogies between the software operating system concepts and the proposed FPGA virtualisation model for heterogeneous computing systems.

1) In this project, shell and roles are able to be designed independently, and hence, to be updated individually as long as the pre-defined system communication and the pre-allocated resources of a slot remain the same (i.e. decoupling the development of shell and roles). 2) A role should be compiled once and reused as many times as possible regardless of its physical mapping to the shell.

Requirement 1 of decoupled compilation may lead to *internal resource fragmentation* [Koc12] as a role can occupy more or less resources than what are available in a slot. We have further targeted this issue by utilising design space exploration (DSE) techniques [MMRL17], providing the flexibility of 2D module-stitching at run-time [PHK+18] (for roles occupying much less resources than what a slot can provide), or merging multiple adjacent slots to host a big role [PVVK18] (for roles occupying more resources than what a slot can provide), as explained further in Chapter 4. Moreover, Requirement 2 is not supported by state-of-the-art vendor PR flows because a role needs to be compiled many times to *all possible mappings* to the shell, as discussed in Section 2.2.

- **Deployment abstraction:** at this level of abstraction, there are four requirements:

  1) Roles can be deployed on top of the underlying shell in a hot plug-and-play manner. 2) A user may run multiple roles, or multiple users may run one or multiple roles on the same shell to utilise the underlying resource, to reduce each user's waiting time and to enhance the overall system throughput. Moreover, 3) temporal and spatial resource sharing between roles via hardware/hardware-software context switching, and cooperative scheduling must be performed transparently from the user's point of view. Finally, 4) role deployment must not imply any security threat to the whole system, including the underlying hardware and other users [MLG+20].

Table 1.1: Comparison between a software OS and the proposed model of FPGA virtualisation.

	Software OS	**Proposed model**
Organisation	* Fixed part: *OS kernel* * Changeable part: *user applications*	* Fixed part: *FPGA shell* (i.e. static infrastructure) * Changeable part: *roles* (i.e. acceleration modules)
Objects of hardware abstractions	CPU, memory, and I/Os	Physical FPGA primitives (may include memory and FPGA I/Os)
Resource sharing/-management	Time domain	Space and time domains

The positions and functional roles of these proposed abstractions in real-world are illustrated in Figure 1.4. The *infrastructure abstraction* is provided by shells according to the physical FPGA layouts as well as the essential infrastructure, as introduced in [PCC+14, KBT08], while the author of this thesis has implemented shells for heterogeneous platforms ranging from Xilinx Zynq-7000 to Xilinx Zynq UltraScale+ devices. Moreover, the *development abstraction* and the *deployment abstraction* are original contributions of this PhD project as they are direct results from an enhanced partial reconfiguration approach (via bitstream abstraction), as discussed in Section 1.2.2 and Chapter 3, and a novel compilation flow, as discussed in Section 1.2.3 and Chapter 4. Ultimately, operational system prototypes for CPU+FPGA heterogeneous computing systems including the proposed abstractions are presented and evaluated against the objectives of FPGA virtualisation, as described in Section 1.2.4 and Chapter 6.

1.2.2 Bitstream Abstraction

In the proposed model for FPGA virtualisation, partial reconfiguration [Koc12] is playing a key role to improve design productivity, to perform hardware context switching, to enhance the resilience as well as the maintainability of the system. However, despite of current progresses, state-of-the-art approaches of partial reconfiguration have significant limitations which lessen its potential contribution to FPGA virtualisation dramatically, as pointed out in a recent survey [VF18]. Hence, this thesis has explored an alternative approach to leverage the principle of partial reconfiguration (PR) for FPGA virtualisation. PR is innovated with design productivity, system efficiency, and physically isolated modularity [PHK+18] in mind.

At the heart of the problem, the ability to adapt FPGA designs at bitstream level for latest architectures is the crucial missing key for any PR flow towards FPGA virtualisation (see Section 2.2). This thesis, therefore, has provided a generic tool and API called BitMan [PHK17]

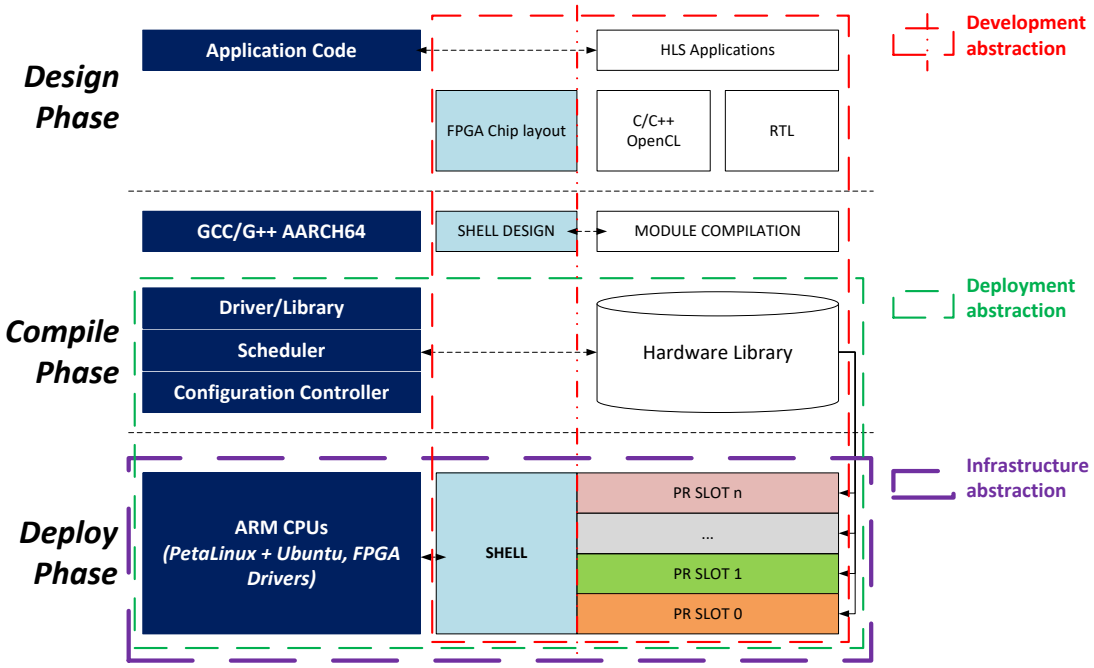


Figure 1.4: The proposed abstractions and their functional roles in the FPGA development/deployment process. Note that the work of scheduler and driver/kernel library is done by A. Vaishnav, while the remaining parts are core contributions of this PhD thesis.

for this FPGA bitstream adaptation purpose. At the time of this PhD thesis written, BitMan is the only tool which can support all recent Xilinx FPGAs such as Spartan-6, Virtex-6, 7-Series, UltraScale and UltraScale+ devices, as discussed detail in Chapter 3.

BitMan, thus, complements the PR flow in [Koc12] and contributes two pivotal attributes: 1) to enable the *decoupled compilation flow* for development abstraction; and 2) to dynamically switch PR modules (i.e. roles) at run-time via the *Configuration Controller* (See Figure 1.6) for deployment abstraction to implement the proposed model of FPGA virtualisation in heterogeneous computing systems.

1.2.3 Decoupled Compilation Flow for FPGA Virtualisation

Although many progresses have been made to improve the design productivity and the ease of FPGA application deployment [Ama] such as High-Level Synthesis [CLN+11, CCA+13], IP integration [Xild, Xilc], and partial reconfiguration [VF18], there are very few notable solutions that allow direct application development such that software engineers can develop accelerators for hugely dynamic FPGA-based systems. Designing such a partially reconfigurable system still requires very specific knowledge in the FPGA domain and expertise in the vendors'

partial reconfiguration toolchains, which is a challenge even for many experienced FPGA developers. Furthermore, user applications (roles) must be designed and implemented with the surrounding infrastructure (shell) as a monolithic system. This approach commonly causes unnecessarily lengthy design phase and even worse, a single change in either the application or the surrounding infrastructure will likely request the whole system to be rebuilt. Therefore, it is impossible to maintain or upgrade either shell or role independently. This limitation still remains in most of the state-of-the-art partial reconfiguration design flows [Xil18c, Alt17], as discussed in Chapter 2.

Thus we have introduced a novel design methodology to design FPGA shells and to compile hardware modules (roles) from user applications. A resulting shell provides the infrastructure abstraction on the targeting platform featuring multiple adjacent partial reconfiguration (PR) regions, called *slots*, having an identical resource layout. Module compilation for application development takes HLS/RTL/netlist descriptions to produce *relocatable accelerator partial bitstreams* in a fully software-centric design process [PVVK18], which essentially presents the development abstraction. Note that the BitMan tool, which were mentioned in Section 1.2.2, is an essential part of this compilation flow, as explained in Chapter 4. This module compilation also delivers prerequisites for the deployment abstraction by allowing users building their hardware modules once but instantiating them multiple times in the provided regions in a hot plug-and-play manner as well as implementing different implemented module variants that can share the reconfigurable resources at run-time. The development and deployment scenarios are similar to compiling a software binary from its source code and then running it on top of a software OS. The proposed compilation is summarised and compared to the conventional software counterpart in Figure 1.5.

To demonstrate this automatically implemented process, we have compiled applications written in RTL, OpenCL [GAPK16], and in C language as partially reconfigurable modules without manual intervention, as shown in Chapter 4. Although the Xilinx Vitis development flow [Xil20], released when this thesis was already submitted, can offer similar automatic compilation experience for heterogeneous computing systems, this decoupled compilation flow offers more attributes than the Vitis flow. For instance, this module compilation can be done independently from shell development process as long as they keep in pre-defined resource footprints for roles and the interface between shell and roles. In one showcase, we deliberately designed a shell and several roles in different Xilinx Vivado versions, and then launched roles on top of shell at run-time (see Section 6.1). Moreover, Vitis has not yet reached the level of flexibility introduced by this proposed compilation flow for compiling *relocatable PR modules*, as discussed further in Section 2.4. Note that the work in this thesis has been carried out, demonstrated, and published in [PVVK18, PPV+19, VPMK19] before the introduction of the Xilinx Vitis flow.

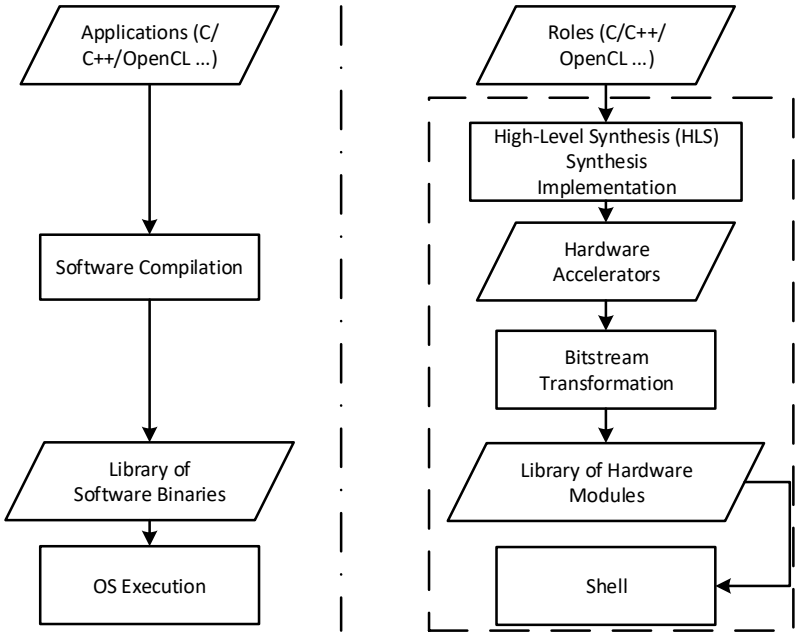


Figure 1.5: Left: conventional software compilation. Right: the proposed module compilation flow [PVVK18].

1.2.4 System Prototypes

Four system prototypes [HSPK17, PHK+18, PVVK18, PPV+19] using different FPGA families, including Xilinx Zynq-7000 [Xil18e] and Zynq UltraScale+ [Xil17b] heterogeneous devices, have been built on top of the proposed methodology, tools, and framework. The systems are composed of three essential parts contributing to their distinct features: shell, the module compilation, and the run-time management. These systems, as shown in Figure 1.6, realised the proposed model of FPGA virtualisation by providing the following features:

- *Decoupled Implementation*: static systems (*shells*) and reconfigurable modules (*roles*) are implemented independently; supports most application design techniques such as HLS (OpenCL, C/C++), RTL (Verilog/VHDL), and netlist.
- *Bus Virtualisation*: supporting different AXI interfaces (Master/Slave/Stream as well as 32/64/128-bit data width) transparently.
- *I/O FPGA Virtualisation*: allowing the same reconfigurable module bitstream to be used multiple times in systems with different I/O layouts.
- *Run-time Flexibility*: supporting different partial configuration styles: 1) island-style (where a module is placed exclusively in a reconfigurable region); 2) slot-style (where

multiple modules can be daisy-chained within one shared region); and 3) 2-dimensional module placement as modules may occupy multiple adjacent regions.

- *Cooperative Scheduling*: allowing hardware context switching to adjust resource allocation dynamically and transparently through a user-friendly API. The research and implementation of run-time heterogeneous scheduling algorithms has been conducted by A. Vaishnav and interacts with the low-level FPGA API developed in this thesis project.
- *Memory Isolation*: through memory management enabling isolation of simultaneously running hardware tasks in multi-tenanted environments. The work of Linux kernel driver for this memory management is contributed by K. Paraskevas, while the author of this thesis is in charge of system integration and user interface.
- *Physical Module Isolation*: fulfilling the requirements for single chip cryptography (SCC) [NIS01] as well as building robust triple modular redundancy (TMR) systems. The here presented design methodology has been extended to guarantee module isolation requirements [PHK+18].

1.3 Thesis Structure

This thesis is organised in seven chapters. In addition, an appendix provides another showcase in applying the proposed design methodology to build SCC and robust TMR systems. Brief descriptions of the content of the remaining six chapters are described in the following paragraphs.

1.3.1 Chapter 2 – Background and Related Works

Since the primary idea of this PhD project is to leverage partial reconfiguration for FPGA virtualisation, this chapter discusses state-of-the-art partial reconfiguration toolflows and obstacles have been examined. Additionally, related works of bitstream manipulation and HLS framework for designing partially reconfigurable systems are being reviewed. Finally, shell-based systems are classified and examined to highlight the contributions of this PhD thesis amongst recent works.

1.3.2 Chapter 3 – Bitstream Manipulation Tool and API

As the ability to adapt FPGA designs at binary (bitstream) level for modern FPGAs is highly desired for the proposed FPGA-virtualised model yet missing in state-of-the-art design methodology and deploying systems, this chapter describes our effort to fulfil this requirement. A tool

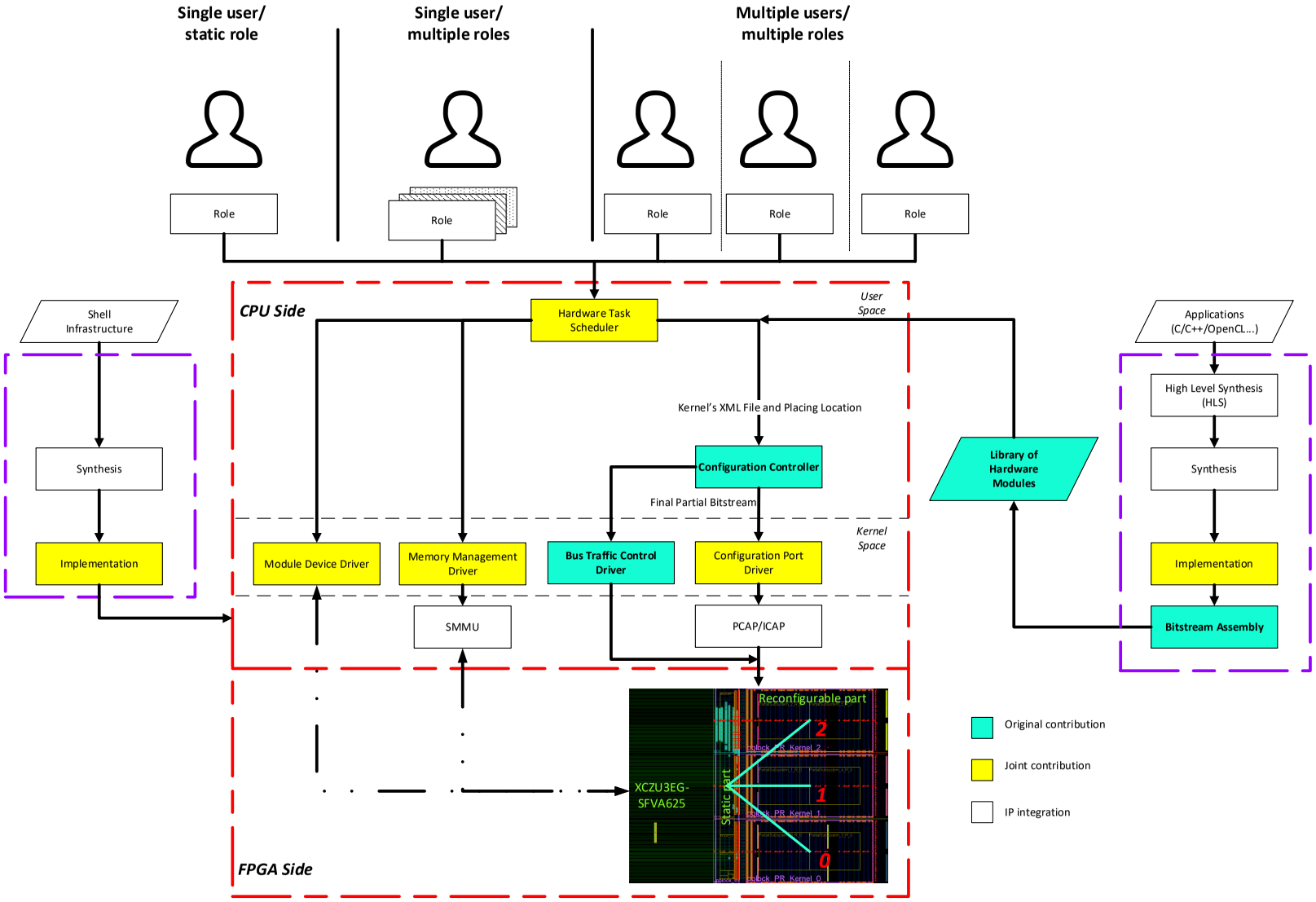


Figure 1.6: The partially reconfigurable system framework built in this PhD project consisting of design elements for building the static shell as well as the reconfigurable roles and run-time services for configuring and running roles on the system.

and API, called BitMan, has been developed and utilised to provide the development and deployment abstractions.

1.3.3 Chapter 4 – Decoupled Compilation Flow for FPGA Virtualisation

With the availability of academic design tools such as GoAhead/TedTCL and BitMan, a novel design methodology is introduced in this chapter to incorporate these academic tools and the Xilinx Vivado toolflow into a single decoupled compilation flow to design systems for the proposed FPGA-virtualised heterogeneous computing model. This design methodology essentially provides the development abstraction which decouples the development of the static shell and the reconfigurable roles. Moreover, static shells developed by this design methodology crucially provide the infrastructure abstraction to the underlying FPGA platforms. A number of FPGA-virtualised systems have been implemented and evaluated on top of various development platforms featuring different FPGA devices as well as families by utilising the proposed methodology.

1.3.4 Chapter 5 – Run-time Management

The run-time management which ultimately provides the deployment abstraction is described in this chapter. Components of the run-time management including the configuration controller, the hardware task scheduler, the module device driver, and the memory isolation framework are presented.

1.3.5 Chapter 6 – System Evaluation

Five case studies of the resulting systems are presented and evaluated against the aforementioned objectives of FPGA virtualisation in order to demonstrate the versatility of the here introduced FPGA virtualusation techniques in this Chapter.

1.3.6 Chapter 7 – Conclusion

Finally, this chapter summarises this thesis and discusses future work, which has been enabled by the research carried out through this project.

1.4 Publications

The research conducted throughout this project has been produced through the 16 research articles listed below.

1. **K. D. Pham**, E. Horta, and D. Koch, ***BITMAN: a Tool and API for FPGA Bitstream Manipulations***, in *Design Automation and Test in Europe (DATE)*, 2017.

2. E. Horta, X. Shen, **K. D. Pham**, and D. Koch, ***Accelerating Linux Bash Commands on FPGAs Using Partial Reconfiguration***, in *FPGAs for Software Programmers (FSP)*, 2017.

3. M. Vesper, D. Koch, and **K. D. Pham**, ***PCIeHLS: an OpenCL HLS framework***, in *FPGAs for Software Programmers (FSP)*, 2017.

4. A. Vaishnav, **K. D. Pham**, D. Koch, and J. Garside, ***Resource Elastic Virtualization for FPGAs using OpenCL***, in *Field-Programmable Logic and Applications (FPL)*, 2018.

5. A. Vaishnav, **K. D. Pham**, and D. Koch, ***A Survey on FPGA Virtualization***, in *Field-Programmable Logic and Applications (FPL)*, 2018.

6. **K. D. Pham**, A. Vaishnav, M. Vesper, and D. Koch, ***ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications***, in *FPGAs for Software Programmers (FSP)*, 2018.

7. **K. D. Pham**, E. Horta, D. Koch, A. Vaishnav, and T. Kuhn, ***IPRDF: an Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs***, in *Multicore/Many-core Systems-on-Chip (MCSoC)*, 2018.

8. A. Vaishnav, **K. D. Pham**, and D. Koch, ***Live Migration for OpenCL FPGA Accelerators***, in *Field-Programmable Technology (FPT)*, 2018.

9. **K. D. Pham**, M. Vesper, D. Koch, and E. Hung, ***EFCAD – an Embedded FPGA CAD Tool Flow for Enabling On-Chip Self-Compilation***, in *Field-Programmable Custom Computing Machines (FCCM)*, 2019.

10. A. Vaishnav, **K. D. Pham**, and D. Koch, ***Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures***, in *Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2019.

11. A. Vaishnav, **K. D. Pham**, Kristiyan Manev, and D. Koch, ***The FOS (FPGA Operating System) Demo***, in *Field Programmable Logic and Applications (FPL)*, 2019.

12. **K. D. Pham**, Kyriakos Paraskevas, Anuj Vaishnav, Andrew Attwood, Malte Vesper, and D. Koch, ***ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs***, in *FPGAs for Software Programmers (FSP)*, 2019.

13. K. Georgopoulos, K. Bakanov, I. Mavroidis, I. Papaefstathiou, A. Ioan-nou, P. Malakonakis, **K. Pham**, D. Koch, and L. Lavagno, ***A Novel Framework for Utilising Multi-FPGAs in HPC Systems***, in *Heterogeneous Computing Architectures: Challenges and Vision*, Taylor and Franchis Group, 2019.

14. A. Vaishnav, **K. D. Pham**, J. Powell, and D. Koch, ***FOS: A Modular FPGA Operating System for Dynamic Workloads***, in *arXiv*, 2020.

15. K. Matas, T. La, N. Grunchevski, **K. D. Pham**, and D. Koch, ***Invited Tutorial: FPGA Hardware Security for Datacenters and Beyond***, in *Field-Programmable Gate Arrays (FPGA)*, 2020.

16. K. Matas, T. La, **K. D. Pham**, and D. Koch, ***Power-hammering through Glitch Amplification – Attacks and Mitigation***, in *Field-Programmable Custom Computing Machines (FCCM)*, 2020.

Papers are currently under review process:

- A. Vaishnav, **K. D. Pham**, J. Powell and D. Koch, ***FOS: A Modular FPGA Operating System for Dynamic Workloads***, in *ACM Transaction Reconfigurable Technology (TRETS)*.

- T. M. La, K. Matas, N. Grunchevskip, **K. D. Pham**, and D. Koch, ***FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs***, in *ACM Transaction Reconfigurable Technology (TRETS)*.

1.5 Open Source Releases

This PhD thesis has also contributed the following open-source projects:

1. BitMan: a Tool and API for FPGA Bitstream Manipulations.

   Available at `https://github.com/khoapham/bitman.git`

2. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications.

   Available at `https://github.com/zuclfpl/zucl_fsp.git`

3. EFCAD — an Embedded FPGA CAD Tool Flow for Enabling On-Chip Self-Compilation.

   Available at `https://github.com/khoapham/efcad.git`

4. FOS — FPGA Operating System Demo

   Available at `https://github.com/khoapham/fos.git`

Chapter 2

Background and Related Works

This too shall pass.

Rumi

2.1 Overview

To provide layers of abstraction and virtualisation for FPGA development and deployment, this PhD project is going to leverage partial reconfiguration (PR) of FPGAs. Therefore, basic concepts of PR and a brief overview of state-of-the-art PR flows, including industrial and academic flows, are provided in Section 2.2. Moreover, remaining problems of current PR approaches are further discussed to emphasise the need of conducting the work on bitstream abstraction/manipulation. Next, Section 2.3 provides background and related works for bitstream manipulation. Further, state-of-the-art compilation flows High-level Synthesis (HLS) applications are reviewed in Section 2.4. Finally, shell-based systems are analysed to identify our work amongst state-of-the-art ones in Section 2.5.

2.2 Partial Reconfiguration (PR)

Partial reconfiguration is a distinguished feature of FPGA technology in which a portion of the FPGA fabric can be reconfigured to change the functional behaviour and/or structure of the system whereas the remaining of the fabric can keep functioning. Hence, PR has been being an active research topic in the FPGA community. Ultimately, this PhD project is going to exploit partial reconfiguration to construct a novel compilation flow, which is used to implement such FPGA-virtualised systems as described in Chapter 1. Hence, this section is going to briefly review the topic focusing on design methodologies and toolchains as well as highlighting current

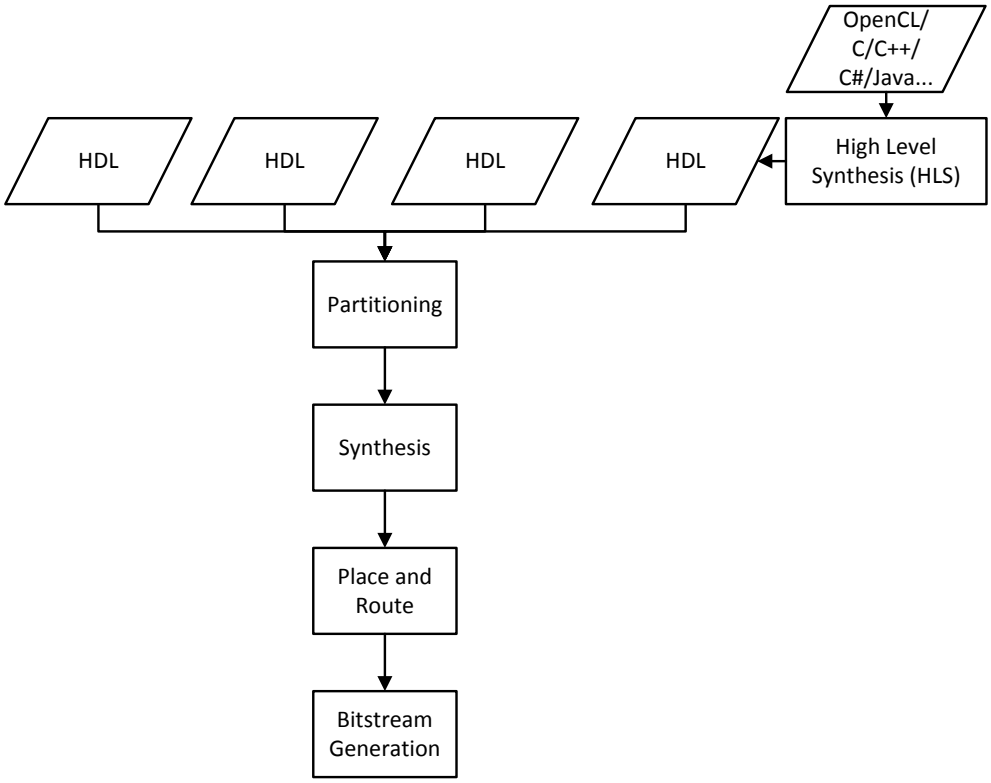


Figure 2.1: A typical FPGA vendor's PR design flow.

limitations which are preventing us to have a system which could fulfil the objectives of FPGA virtualisation.

2.2.1 FPGA Vendors' PR Design Flows

Xilinx Vivado PR Flow: Vivado PR flow [Xil18c] is an integral part of the Vivado Design Suite to implement partially reconfigurable designs for the latest 7-Series, UltraScale, and UltraScale+ FPGAs. A PR design consists of the *static part* which does not change its functionality during system operation and the *reconfigurable part* including one or more *reconfigurable regions*. In 7-Series and older Xilinx FPGAs, reconfiguration regions may contain only LUTs, BRAMs, and DSP slices but cannot contain clocks and clock modifying logic, I/O and I/O related components, serial transceivers and related components, and individual architecture feature components. However, these restrictions have been lessened for UltraScale/UltraScale+ devices so that only components needed for the actual partial reconfiguration process must remain in the static part of the design [Xil18c]. The static part commonly contains a processor (running the run-time management), a configuration interface as well as memory and network interface modules (i.e. the essential infrastructure to keep the whole system functioning). Reconfigurable regions *instantiate* the reconfigurable modules to perform the real computing work

and can be reconfigured to change the behaviour at run-time. In the Xilinx PR flow, a PR design is implemented by using TCL commands.

First of all, partitioning is conducted to decide how many reconfigurable regions the system should provide and their capacity in terms of resources. Modules are synthesised to produce corresponding netlists. In the default PR flow, floorplanning is then performed manually to specify the locations and bounding boxes of reconfigurable regions in the FPGA fabric, although it could be assisted by academic tools such as [GK15, NK16] . A characteristic of the vendor PR flow is that reconfigurable modules are implemented as an increment to the static system which prevents using modules at different locations inside one system or using a reconfigurable module across multiple systems. These regions may not need to be rectangular in shape or be aligned to clock region boundaries in the latest UltraScale/UltraScale+ FPGAs. Floorplanning details are stored in the *Xilinx Design Constrains* (XDC) file for incorporation in the implementation stage. The Xilinx PR flow is illustrated in Figure 2.1. While the PR flow introduced in the Vivado toolchain is easier to use than earlier versions of the Xilinx vendor tools [Xila], building sophisticated reconfigurable systems is still a discipline requiring very specific FPGA domain knowledge.

In this Vivado PR flow, static signals are still permitted to use routing resources in reconfigurable regions which help to improve the routability as well as timing performance [Xil18c]. Therefore, this characteristic arguably results in faster PR designs but obviously is not able to implement relocatable modules. Moreover, a single modification of the static logic may require the whole PR design including all modules to be re-implemented through this already lengthy process. Further, the Vivado PR flow is using interconnect tiles directly for implementation of anchor logic instead of LUT-based bus macros, which were introduced in [LBM+06], to enhance routing efficiency and timing performance. However, this practice makes run-time bitstream relocation even more difficult because Vivado has no means to specify the exact physical interface between the static system and the reconfigurable modules. Consequently, multiple reconfigurable regions could not be implemented with the same physical interface.

Intel/Altera PR Flow: Intel/Altera PR flow is supported in the latest Quartus Prime design software [Alt17] and fundamentally similar to the Xilinx PR flow as illustrated in Figure 2.1. Ultimately, this flow permits static routing using reconfigurable regions' routing resources, so building relocatable modules is also impossible.

2.2.2 Academic PR Development Tools

As run-time module relocation for deployment abstraction as well as decoupling of static and module designs for development abstraction are very useful and highly desired features,

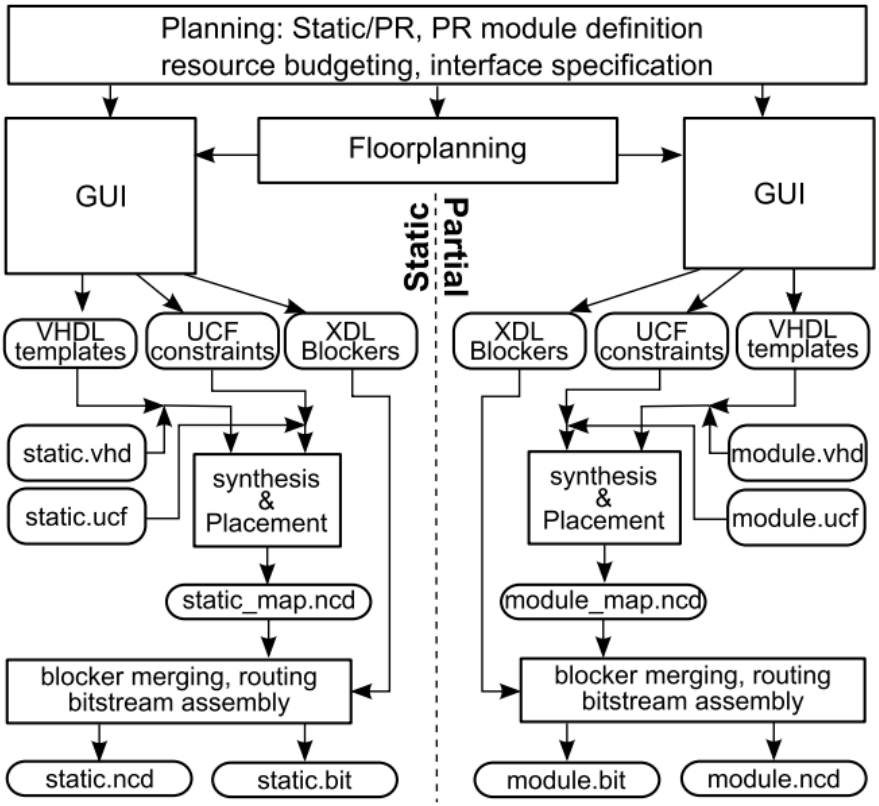


Figure 2.2: Overview of the GoAhead PR toolflow [BKT12]. GoAhead and BitMan [PHK17], a tool and API for bitstream manipulation developed in this PhD project, are major building blocks of the novel decoupled compilation flow (see Chapter 4).

several academic PR tools such as ReCoBus-Builder [KBT08], OpenPR [SAFW11], GoAhead [BKT12], TPaR [VFBS14], RapidSmith 2.0 [HNH15], and RapidWright [LK18] have been proposed to provide these features. Although the changes in Xilinx architectures, toolchains as well as low-level API support such as the abandonment of XDL interface [BKT11] for TCL-like interface made most of the academic tools obsolete, GoAhead, RapidSmith 2.0 and RapidWright were able to catch up these changes, and hence, to work on the Vivado Design Suite to support the latest FPGA devices.

Figure 2.2 shows the GoAhead toolflow as an example of academic PR development tools, in which the static and reconfigurable parts are implemented independently in different designs. Moreover, *blocker macros* are used to prevent the static nets from crossing PR regions to support module relocation between resource-identical reconfigurable regions. However, by constraining a system in such a way that static routing follows a regular pattern, GoAhead also allows static routing crossing reconfigurable regions without sacrificing the availability of module relocation.

2.2.3 Discussion

Although there have been significant progresses in the partial reconfiguration research, many obstacles still exist:

1. The ability to adapt bitstreams at run-time for latest FPGA devices is crucial for module relocation yet not a trivial task, as discussed in Section 2.3.

2. Despite the availability of academic PR development tools to support development and deployment abstractions, a ready-to-use solution, including the integration of those tools with High-Level Synthesis (HLS) tools and, especially, with the underlying infrastructure, is still absent. Such kind of solution is extremely useful for users who do not posses the expertise of low-level FPGA details yet want to accelerate their applications on FPGA devices. Related works of this research direction is discussed in Section 2.4.

3. None of the existing academic PR tools provides an integrated or complementary run-time management for improved abstraction to allow loading and unloading of new configurations similarly to dynamic loading and unloading of software modules [VF18]. Moreover, scheduling policies to orchestrate the modules with the available heterogeneous computing resources are other missing pieces.

These obstacles are the major motivator for this PhD project, and solutions to these three issues will be provided in Chapter 3, Chapter 4, and Chapter 5, respectively.

2.3 Bitstream Manipulation

A bitstream contains all information of a design which is mapped, placed, and routed on a dedicated FPGA chip. Therefore, bitstream manipulation needs to be done with care since a corrupted bitstream may damage an FPGA device physically and permanently [BKT10]. However, bitstream manipulation also enables powerful features such as updating designs at run-time, fully flexible module replacements, or even composing overlay architectures on-the-fly [YKL15]. To perform this bitstream manipulation, we need detailed information about the bitstream format.

Early efforts, such as JBit [GLS99], JBG [RS02] and ParBit [HLK02], provided means to dynamically link and assemble partial hardware modules into FPGA fabric. However, these approaches do not support latest devices and cannot easily reroute connections to modules and maintain clock resources.

Previously, Note et al. [NR08] suggested to use the Xilinx Description Language [BKT11] and cross-correlation algorithm to analyse the Xilinx bitstream and reconstruct the netlist. In

this PhD project, we did not use the cross-correlation algorithm since all bitstream information can be derived precisely from Xilinx FPGA bitstreams for CLB, DSP, BRAM, and the interconnection fabric.

RapidSmith [LPL+14] released by the Brigham Young University can parse, manipulate, and export bitstreams for Xilinx Virtex-4, Virtex-5 and Virtex-6. Moreover, in their latest attempts, Kulkarni et al. provided a similar API for bitstream manipulation to change the LUT contents and switch blocks configuration in Virtex 5 and 7-Series devices as part of their Dynamic Circuit Specialisation system [KS16, KVS+16]. Their works were significant, and hence, we have aimed at generalising it for later devices since they do not support any newer FPGAs than the 7-Series devices. Additionally, [LPL+14, KS16, KVS+16] were based on the old Xilinx ISE design suite which is obsolete for latest devices and represents a hurdle for porting these tools to recent FPGAs. Note that since the UltraScale family, all recent FPGA families of the vendor Xilinx are only supported on the Vivado design tool. Instead of using only Xilinx ISE, we can support both ISE and Vivado design suites [Xil14].

Another significant effort was Project X-Ray [Sym], which documented all bitstreams of Xilinx 7-Series FPGAs. In addition, Project X-Ray was integrated to an academic CAD flow, named Maverick [GGNW19], which can take Verilog source codes, synthesise, implement the designs, and finally generate bitstreams without any Xilinx tool in the process. This open-source Maverick project has used the TCL interface on Vivado toolchain to extract the bitstream encoding information for 7-Series devices.

Throughout this PhD project, a bitstream manipulation tool and API, called BitMan, has been developed for targeting the Vivado toolchain as well as latest Xilinx FPGAs including 7-Series, UltraScale, and UltraScale+ families [PHK17], as described in Chapter 3. The author of this PhD thesis developed BitMan and extracted the required bitstream information. BitMan 1) enabled the novel decoupled compilation flow for FPGA virtualisation, as discussed in Chapter 4, and 2) formed up the configuration controller to deploy PR modules at run-time in a hot plug-and-play manner, as described in Chapter 5. Furthermore, BitMan was also integrated to another academic vendor-independent CAD flow to generate partial bitstreams from Verilog source codes for Xilinx UltraScale+ FPGA devices [PVKH19].

2.4 Compilation Flows for HLS Applications

FPGA applications have conventionally been written in Hardware Description Language (HDL) such as Verilog and VHDL. To ease software users to develop FPGA applications, High-Level Synthesis tools [CLN+11, CCA+13] have been introduced to compile applications from high-level languages such as C/C++, OpenCL to their HDL formats. Then, the HDL

source codes go through steps of a normal FPGA development process, including synthesis, placement, and routing, to produce the FPGA configuration binary of resulting FPGA designs (i.e. FPGA bitstreams). Note that the HLS applications are required to be integrated to the surrounding system infrastructure for operation as well as to the low-level driver/kernel API for user-space software interface. Therefore, compilation flows are usually integral parts of the corresponding HLS development platforms/frameworks, which provide the essential hardware-software services.

To support software developers for using the heterogeneous Zynq UltraScale+ FPGAs, Xilinx has formerly provided the SDSoC development environment [Xild]. Xilinx SDSoC was a framework, built on top of the Vivado HLS toolchain [Xil19b], for compiling C/C++/OpenCL applications into hardware systems and running them on FPGAs. Further, this SDSoC framework provided profiling user applications, integrating user applications to required system infrastructure such as the system bus and the memory controller, optimising full systems for performance as well as providing various memory access patterns. However, Xilinx SDSoC did not support partial reconfiguration. This means, when a user wants to change their system functionality, it is required to shutdown the whole system, reconfigure the FPGA fabric and start the system again. This limitation introduced a performance overhead in systems that require frequent reconfiguration. Further, if a design would require more resources than what are available on the device, that design cannot be implemented at all. Ultimately, as partial reconfiguration was not available together with SDSoC, using PR for sharing resources is impossible.

On the other hand, Xilinx SDAccel [Xilc] has been introduced to compile OpenCL, C/C++ applications and to run them on PCIe-based FPGA platforms. This SDAccel framework enabled partial reconfiguration (PR) on these devices since the communication between the host CPU and OpenCL kernels was performed via PCIe connectivity, which must be active when swapping kernels. However, this framework did not allow users to build modules independently from the static design. A single change in the static design required the whole system to be re-implemented. This meant that design decoupling of static system and modules is infeasible. Moreover, users could only swap kernels which are built for a pre-defined PR region. This limited users to adapt their applications to various run-time scenarios.

Xilinx has released Vitis development framework [Xil20], which is supporting both SoC-based and PCIe-based FPGAs, when this PhD thesis was submitted. In addition, Vitis enables PR for SoC-based systems and provides a run-time management layer, called Xilinx RunTime (XRT) library [Xilf], to share FPGA resources between PR modules. Despite these progresses, Vitis does not fully support the development and deployment abstractions as proposed and implemented in this PhD work. For instance, the same HLS application *needs to be compiled many times to be deployed on different PR regions*. Ultimately, changes in the static design will *force the whole system to be rebuilt*.

There are academic approaches [GBLV12, RFG16, GPK18] to implement systems supporting module relocation with the help of the Xilinx Isolation Design Flow (IDF) [Xil16c]. Unfortunately, they do not work reliably with wide module interfaces and systems using multiple clock domains, which is commonly the case.

Other academic and commercial frameworks compiling HLS applications to FPGA platforms are RIFFA [JRHK15], hCODE 2.0 [ZHA+17], DyRACT [VF14a], RC2F [KGS17], and Dyplo [vZ13]. However, none of them supports relocatable hardware modules, design decoupling of static and PR modules or flexibility to instantiate different sized PR modules, as summarised in Table 2.1 and discussed later in Chapter 6.

Instead, this PhD project aimed at developing and deploying real-world applications from HLS source codes with complicated control flows, AXI4 system bus communication, multiple clock domains, various memory access patterns as well as decoupled compilation flow and hot-plug-and-play module deployment, as described in Chapter 4 and demonstrated in Chapter 6. The work in this PhD thesis has been integrated to a novel FPGA Operating System (FOS) [VPPK20] to fully virtualise FPGA resources in heterogeneous (SoC-based) systems, as shown in Figure 1.4.

2.5 Shell-based Systems

Building reconfigurable systems on FPGAs has a long history, and most related systems partition the FPGA resources into the *static part* providing the infrastructure and the *reconfigurable part* for hosting the actual accelerator modules [DPP02, MTAB07, Koc12]. In recent years, however, the term *shell* emerged for the static part of the system, as discussed in [PCC+14]. This was done to express the encapsulation and I/O virtualisation that these infrastructures provide. Ultimately, the shell approach provides the infrastructure abstraction to deploy user applications on top with ease.

The following paragraphs had been adopted from our previous publication [VPK18a] for completeness. Moreover, Eckert et al. shed another light on the topic by their survey [EMHK16].

For this shell approach, the execution models are the prime drivers for deciding which functionality is required to be a part of a shell and these shell-based systems can be categorised into four types:

1. Single FPGA Single Application (SFSA): This is an ideal model for relatively simple systems with a few accelerators that are used entirely exclusive to each other, as proposed in [PCC+14, PA12, FVS15, TLF+17, VF14a, ZHA+17, Xilc, JRHK15, VKVF16];

2. Single FPGA Multiple Applications (SFMA): One or more applications may share the

Table 2.1: HLS Development Frameworks.

Features	Xilinx SDSoC [Xild]	Xilinx SDAccel [Xilc]	Xilinx Vitis [Xil20]	RIFFA [JRHK15]	hCODE 2.0 [ZHA+17]	DyRACT [VF14a]	PCIeHLS[a] [VKP17]	RC2F [KGS17]	Dyplo [vZ13]	FOS[b] [VPPK20]
FPGA families	ZYNQ US+	Virtex US+	ZYNQ US+ & Virtex US+	Virtex-7	Virtex-7	Virtex-7	Virtex-7	Virtex-7	ZYNQ 7000	ZYNQ US+
Partial reconfiguration (PR) support	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Design decoupling of static and PR modules	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Relocatable PR modules	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Flexibility to instantiate different sized PR modules	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Independent maintenance of static and reconfigurable parts	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Open-source access	✗	✗	✗	✓	✓	✓	✗	✓	✓	✓

[a]The author of this PhD thesis has contributed by providing and integrating the BitMan tool to this framework for Virtex-7 data centre FPGAs.
[b]This framework targeting CPU+FPGA heterogeneous computing systems has been developed throughout this PhD project.

same FPGA fabric at run-time to achieve higher system throughput and shorter waiting time for each application such as systems in [BSB+14, KGS17, CSZ+14, FVS15, MTAB07, Xilc, VKP17, PVVK18, ZHA+17, AGV+17, VPPK20];

3. Multiple FPGAs Single Application (MFSA): For instance, in the Microsoft Catapult project [PCC+14], a large number of FPGAs are connected to accelerate the Bing Search engine. Other examples can be found in [PA12, WPAH16, TLF+17, VKVF16, GBM+19];

4. Multiple FPGAs Multiple Applications (MFMA): This model provides a certain level of flexibility to the users for deploying their applications and therefore, are ideal for large data centres and cloud computing systems, as proposed in [KGS17, WAHH15, ZXX+17, BSB+14, VKP17, PVVK18, VPPK20].

It would be desirable to have an efficient (light-weight) shell for MFMA as that provides most flexibility and could also support all other modes.

Multiple applications on a single FPGA are achieved using multiple partial regions. A partial region is constituted by a set of adjacent functional primitives such as look-up tables (LUTs), Block RAMs (BRAMs), and DSPs, and associated routing resources. *Instantiating* a reconfigurable module inside a partial region to launch a user application on top of a shell means *configuring* the functional primitives and routing resources such that the partial region provides the functionality of the reconfigure module.

These regions can be asymmetric as well as symmetric in shape. Asymmetric regions, usually called *islands*, support multiple different sizes of modules without having to reconfigure the entire FPGA [ZHA+17]. The number of reconfigurable resources inside a partial region has to satisfy the footprint of the *largest* reconfigurable module (i.e. the module which has the biggest resource budget that might be instantiated in it). Please note that a partial region has to satisfy all kind of resources, such as LUTs, BRAMs, and DSPs together. For FPGA applications in Register-Transfer Level (RTL), this asymmetric approach offers various options to map a specific reconfigurable module to its *best fit* partial region as *the number of resources per reconfigurable module cannot be changed easily*. However, there are cases when there are substantial amount of unused resources in the targeting partial regions, as illustrated in Figure 2.3. This under-utilisation leads to the *internal fragmentation*, which is the major drawback of this asymmetric approach [Koc12]. Nevertheless, the asymmetric solution is the only approach that is officially supported by partial reconfiguration design flows of the FPGA vendor tools [Xil18c, Alt17].

The symmetric approach uses similar or identically sized partial regions with the same resource footprint, called *tiled regions* or *resources slots* or *1D slots*, as classified in [KBT08]. With the symmetric solution, a module can occupy one or more adjacent slots which provides more flexibility for the resource allocation to trade-off resource for system throughput

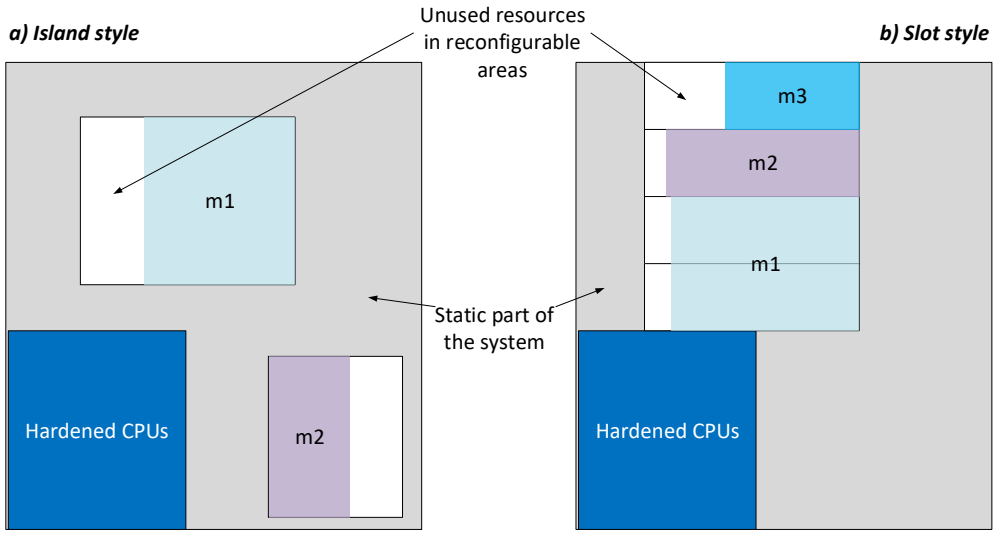


Figure 2.3: Island style placement *vs* slot style placement. In all cases, the total reconfigurable area and the module sizes are the same. With more fine-grained slots, *internal fragmentation* can be minimised, and the number of simultaneously placed modules can be maximised. The figure is adopted from [Koc12] to incorporate the hardened CPUs found in modern CPU+FPGA heterogeneous computing systems.

at run-time, as shown in [VPKG18, PVVK18, VPPK20]. However, according to the run-time module allocation, a waiting multi-slot module cannot be deployed although there are enough free resources at that time as the free slots are not in a contiguous region. This issue is called *external fragmentation*, which can be resolved by the system scheduler as explained in Chapter 5. Ultimately, this symmetric approach needs a custom design flow to be implemented [KBT08, BKT12].

Each execution model requires one of the following classes of connectivity:

- Host connectivity: this includes connectivity to a CPU for controlling register files or for shared memory access,
- Exclusive connectivity: this typically includes connectivity to some I/O primitives such as I/O pins or Gigabit transceivers,
- Hybrid connectivity: a combination of host connectivity and exclusive connectivity.

Because the host connectivity approach has the host CPU handling the resource management of the FPGA, its shells could reserve as many resources as possible for the applications as shown in [FVS15, ZXX+17]. Another large-scale example is the Amazon EC2 F1 FPGA Cloud Service [Ama] where its instances are built from FPGA accelerators each plugged into PCIe slots and where all network connectivity is provided through the host CPU.

On the other hand, the exclusive connectivity approach allows sharing of FPGA resources

among multiple different CPUs [WAHH15, WPAH16] or even standalone FPGAs for the application [PCC+14, WAHH15, BSB+14, TLF+17]. Certainly, this class of exclusive connectivity introduces significantly more resource overhead in the shells than the host connectivity counterparts since FPGA support for the network layer and other I/O resources (e.g. Ethernet and memory controllers) are required, as reported in Table 2.2.

The hybrid approach aims at supporting both forms of connectivity to benefit from offloading control intensive or complex resource management tasks to CPUs, but also to use the special hardware on an FPGA to accelerate I/O accesses if required. For example, in Microsoft's Catapultv2 shell [CCP+16], the FPGAs are located between the network and the host machine. This approach is also known as *bump-in-the-wire* integration.

Although the aforementioned shells have provided certain levels of infrastructure abstraction, due to obstacles of state-of-the-art partial reconfiguration design methodologies as examined in Section 2.2, none of them is able to support development abstraction (i.e. design decoupling of static and PR modules) and deployment abstraction, as discussed in Chapter 4 and demonstrated in Chapter 6. This PhD work, instead, contributes to all three level of abstractions to a novel full-stack FPGA Operating System [VPPK20] thanks to the enhanced PR approach, as illustrated in Figure 1.4.

2.6 Chapter Summary

As mentioned before, partial reconfiguration can be utilised to build up a system that provides abstraction layers and the sense of virtualisation for FPGA development and deployment, hence achieves the overall goals for FPGA virtualisation. Although there are a lot of solutions available, no fully integrated framework considering the whole FPGA ecosystem to support the proposed FPGA-virtualised model has ever been built. This is due to the obstacles which are discussed in Section 2.2. Therefore, current design methodology as well as design tools need to be revised to support our ultimate FPGA virtualisation targets. The next two chapters will explain our innovation in design tool and methodology in detail.

Table 2.2: Comparison of shells. *Scalability* considers the support of connectivity in the shell (Low: host connection only, Medium: network connection, High: both host and network connection), *Flexibility* considers the support for range of accelerator design is considered i.e. custom accelerator, framework, and bump-in-the-wire (Low: 1 model, Medium: 2 model, High: all 3 models), *Design Productivity* considers if shell supports RTL, HLS/DSL and frameworks (Low: only RTL, Medium: HLS / DSL + RTL, High: HLS/DSL + RTL + frameworks), *Isolation* considers how advanced support is provided by component managers and finally (Low: 1, Medium: 2, High: > 2 resource multiplexing), *Utilisation* considers the amount of region which would be in use at run-time this includes support for multiple accelerators on chip as well as level of scheduling techniques employed (Low: 1 region, Medium: > 1 region, High: > 1 region + spatial scheduling). Initial table provided in [VPK18a].

Shell	Acceleration Logic	Multi-tenancy	Scalability	Flexibility	Design Productivity	Isolation	Utilisation
Byma et al. [BSB+14]	26%	✓	Medium	Medium	Medium	Low	Medium
Putnam et al. [PCC+14]	24%	✗	High	High	Low	-	Low
Chen et al. [CSZ+14]	59%	✓	Medium	Medium	High	Medium	Medium
Weerasinghe et al. [WAHH15]	67%	✓	High	Medium	Medium	High	Medium
Fahmy et al. [FVS15]	93%	✗	Low	Low	Medium	-	Low
Weerasinghe et al. [WPAH16]	68%	✗	Medium	Medium	Medium	-	Low
Tarafdar et al. [TLF+17]	80%	✗	High	Medium	High	-	Low
Zhang et al. [ZXX+17]	87%	✓	High	High	High	High	Medium
Knodel et al. [KGS17]	58%	✓	High	High	Medium	High	Medium
Asiatic et al. [AGV+17]	-	✓	Medium	Low	Medium	Medium	High
Khawaja et al. [KLP+18]	-	✓	High	High	Low	High	High
Amazon EC2 F1 shell [Ama]	66%	✗	High	High	Low	High	Low
ECOSCALE shell* [GBM+19]	53%	✓	High	High	Medium	High	High
FOS shell* [VPPK20]	83%	✓	Low	Medium	Medium	Medium	High

* These shells have been developed throughout this PhD Project.

Chapter 3

Bitstream Manipulation Tool and API

Where there's a will, there's a way.

The deployment abstraction as mentioned in Chapter 1 requires run-time relocation of hardware accelerators on FPGA resources. Many high performance reconfigurable systems, such as proposed in the projects EXTRA [SVC+16], ECOSCALE [MPL+16], OpenStack-enabled virtualised FPGA platform [BSB+14], and embedded systems [vZ13], recommend to use partial reconfiguration to achieve this flexible module relocation. However, fully flexible relocation is hard to achieve since a relocatable module requires inter-communications to other modules as well as fitting clock resources in order to work accurately [Koc12]. This level of automation does not exist in current vendor design flows, as pointed out in Chapter 2. Therefore, to overcome this obstacle, we may need either to build our own FPGAs and the corresponding design toolchains or to perform bitstream manipulation. Although do-it-yourself FPGAs and toolchains [Uni19] would allow full control of the architecture and the application development process, bitstream manipulation of modern FPGAs provides the ability of flexible module relocation on top of the existing FPGA ecosystem. In the scope of this PhD project, bitstream manipulation has been chosen as we are targeting the modern heterogeneous CPU+FPGA computing systems.

With a deep understanding of the bitstream format, we are able to:

- modify the configuration of FPGA primitives (e.g., LUT values [KS16] or memory (BRAM) contents [SFK+17]),
- reroute wires,
- reconfigure clock buffers,
- flexibly relocate and duplicate hardware modules,

- route through physical LUTs or FPGA resources [KBL13, HW13],
- perform hardware linking at binary (bitstream) level for overlay architectures [YKL15],
- generate full/partial bitstreams which are compatible with the Xilinx FPGA bitstream format [PVKH19].

In this chapter, we introduce a generic methodology to analyse and manipulate the Xilinx FPGA bitstreams with the goal to build a generic tool that may even support device families that may be introduced after UltraScale and UltraScale+ ones. We provide a low-level API providing access to FPGA fabric resources such as LUT/BRAM contents, routing and clock resources. Furthermore, high-level functions such as module placement and relocation are fully supported.

Besides a generic *X*-*Y* coordinate system abstraction for defining geometrical parameters, BitMan supports a coarse abstraction in resource column of a definable height. In the case of Xilinx FPGAs, the height of a clock region is the smallest vertically atomically reconfigurable unit of these FPGAs. Any module relocation has to consider the primitive layout of the fabric, and we adopted a string model approach as presented in [GK15].

A part of this work has been published in [PHK17] and is reproduced here with additional material. The author of this PhD thesis developed BitMan as well as extracted necessary bitstream information to implement the aforementioned features.

3.1 Bitstream Investigation and BitMan Implementation

In this section, we are taking a closer look into a bitstream's structure, frame address, resource description, and how they are being used in the BitMan tool.

3.1.1 Bitstream Format

The FPGA bitstream consists of configuration commands and configuration data. A configuration bitstream for Xilinx FPGAs has a header (including a bus width detection pattern, a SYNC word, and some configuration commands) and the actual configuration data (for all primitives and the routing), which is followed by a footer. We refer readers to configuration user guides from Xilinx vendor such as [Xil15a] and [Xil15b] for further information about header, bus width pattern and SYNC word. A footer may have CRC values, if any, and a DESYNC word to indicate the end of configuration data. In this work, we focus on the configuration frames, the device description and on how an FPGA device is reconfigured to help understanding how the bitstream manipulation tool works.

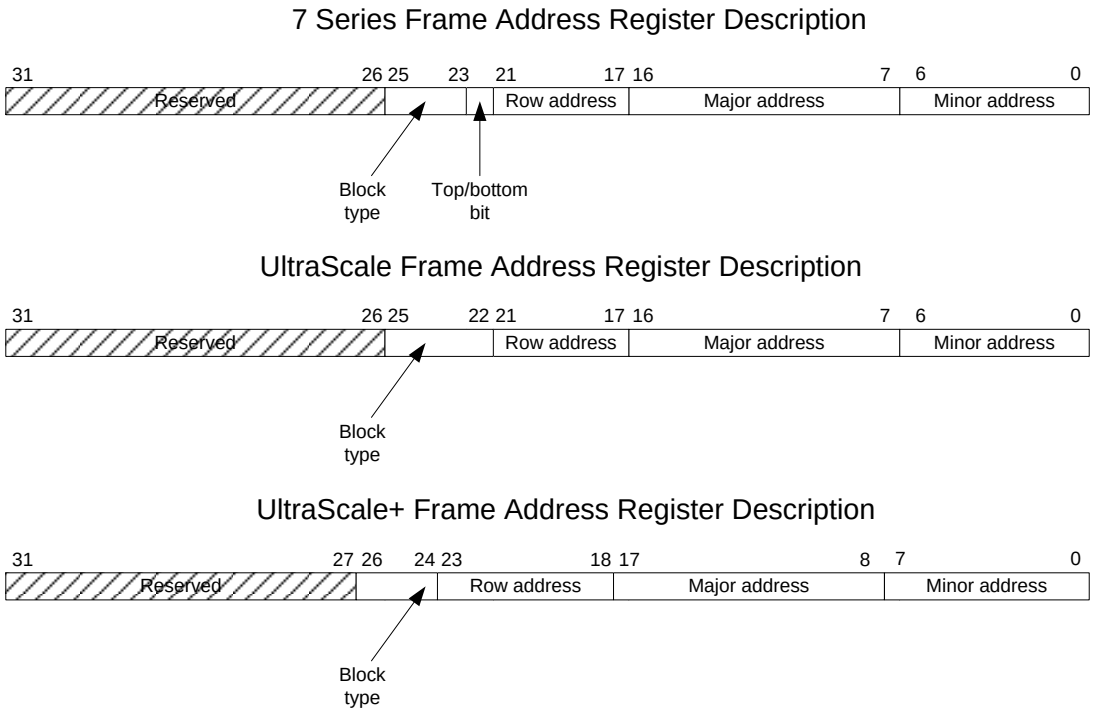


Figure 3.1: 7-Series, UltraScale and UltraScale+'s Frame Address Register Descriptions.

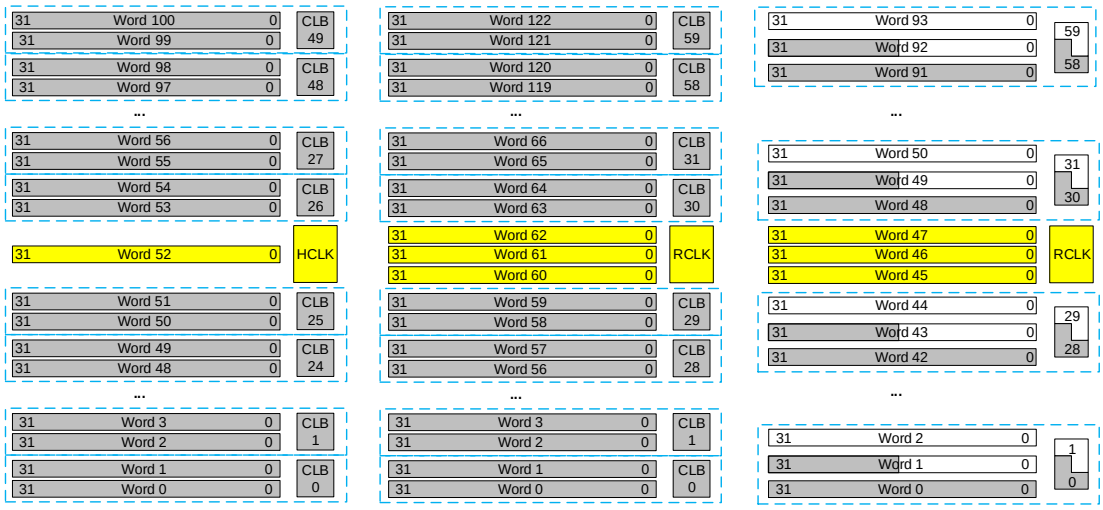


Figure 3.2: Frame configurations for a CLB column in the 7-Series (left), UltraScale (middle), and UltraScale+ (right) devices.

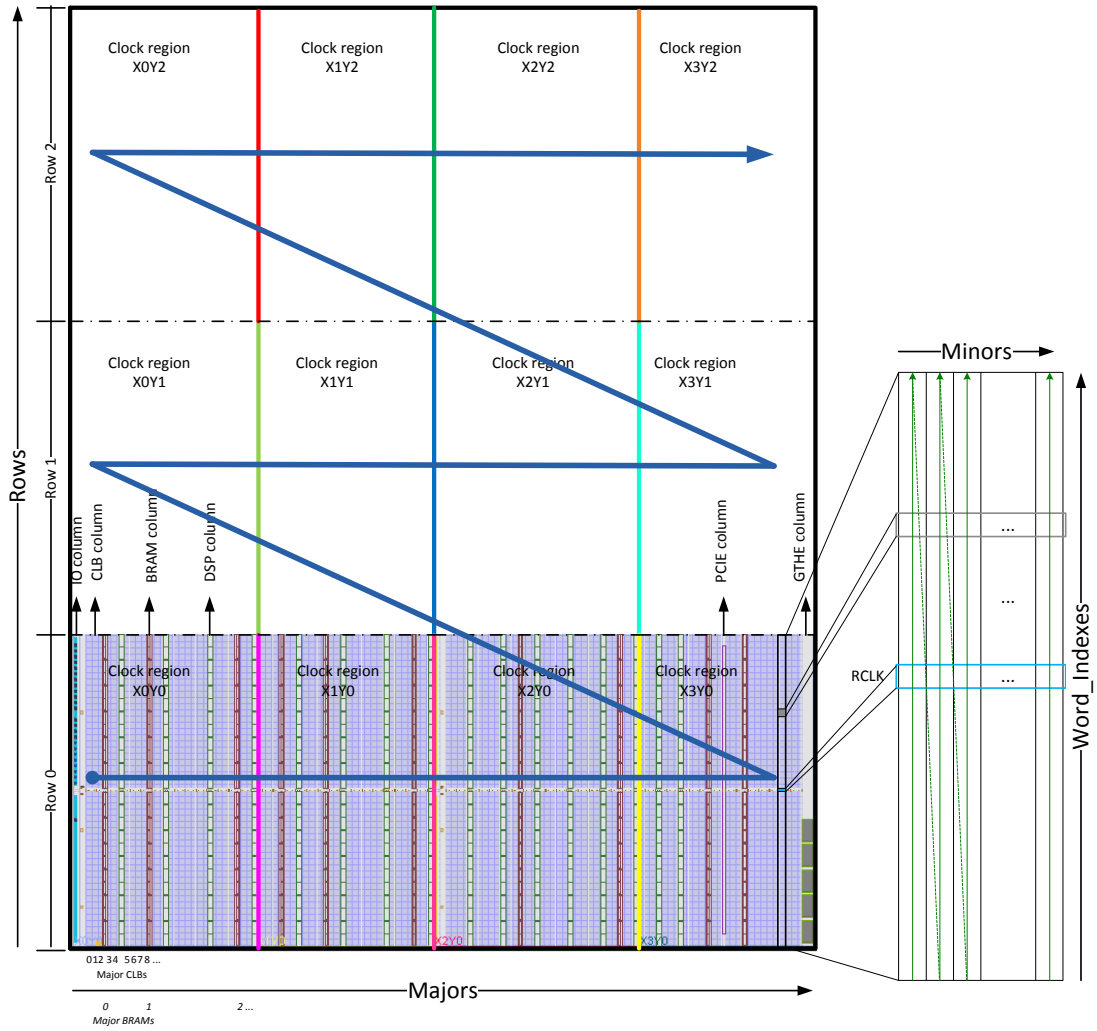


Figure 3.3: Overview of Kintex UltraScale XCKU025's device layout. This device has in total 3 repetitive resource rows and 12 clock regions. The figure is adopted from the Vivado design tool [Xil14].

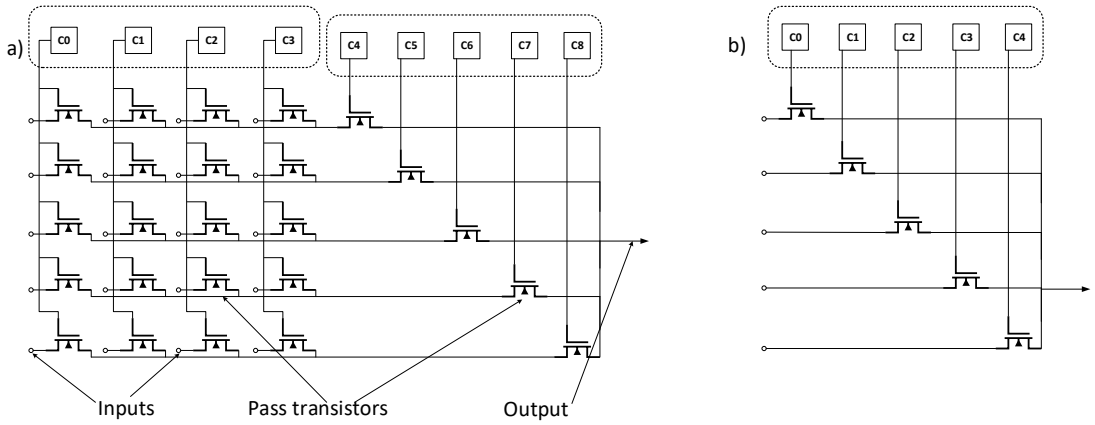


Figure 3.4: (a) Switch matrix multiplexer implementation on Xilinx 7-series FPGAs; (b) Switch matrix multiplexer implementation on UltraScale+ FPGAs.

Configuration memory frames: are *atomic, indivisible* elements in FPGA configuration data. Each frame has its own address, which consists of a minor, major, and row address field as well as the block type of the resource (e.g., the routing of BRAMs and the actual BRAM content are stored in different sections of the bitstream, each belonging to a different block type). Consequently, the *block type* identifies if a resource is CLB (Configurable Logic Block), BRAM content or CFG_CLB [Xil09]. Please note that the allocation of the FPGA resources into block type may vary across different device families of the vendor Xilinx and BitMan is designed generically to take such family specific properties into account.

The *row address* shows which row of clock regions the resources belong to, while the *major address* specifies the resource column. The *minor address*, in turn, defines a specific configuration frame within a specific column of resources. Figure 3.1 depicts the descriptions of frame address in 7-Series, UltraScale, and UltraScale+ devices. Note that the 7-Series has a bit to select a *top-half* or *bottom-half* row (bit **22**), while UltraScale/UltraScale+ FPGAs dismissed this bit.

Depending on the FPGA family, a frame has a *specific number of data words*. For example, in 7-Series family a configuration frame has **101** 32-bit words, while it has **123** and **93** words in UltraScale and UltraScale+ families, respectively. Figure 3.2 shows configuration frames for a CLB column in 7-Series, UltraScale, and UltraScale+ devices. We can identify that the number of words per frame relates to the number of CLBs and how many configuration bits are used per CLB. 7-Series devices have **50** CLBs per column and a reconfiguration frame has 101 words, whereof **1** word is used for configuring some clock resources. Respectively, UltraScale and UltraScale+ devices provide **60** CLBs per column with 123 and 93 words per frame and **3** words for the CLKs.

Table 3.1: Resource information in the Xilinx Virtex-6, and 7-Series families.

Device family	Virtex-6		
Resource column	CLB	DSP	BRAM
Block Type	0x000	0x000	0x001
# of frames for interconnect	28	28	28
# of frames for content	8	0	128
Device family	7-Series		
Resource column	CLB	DSP	BRAM
Block Type	0x000	0x000	0x001
# of frames for interconnect	28	28	28
# of frames for content	8	0	128

Table 3.2: Resource information in the UltraScale and UltraScale+ families.

Device family	UltraScale			
Resource column	CLB	DSP	Switch Matrix	BRAM
Block Type	0x000	0x000	0x000	0x001
# of frames for interconnect	0	4	58	4
# of frames for content	12	0	0	128
Device family	UltraScale+			
Resource column	CLB	DSP	Switch Matrix	BRAM
Block Type	0x000	0x000	0x000	0x001
# of frames for interconnect	0	8	76	6
# of frames for content	16	0	0	256

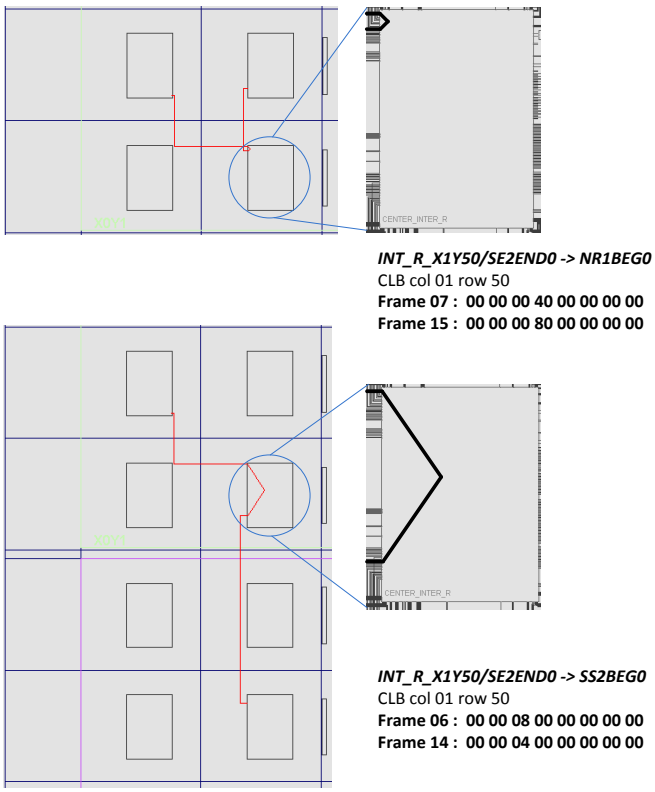


Figure 3.5: Example of bitstream encodings for routing in the 7-Series family.

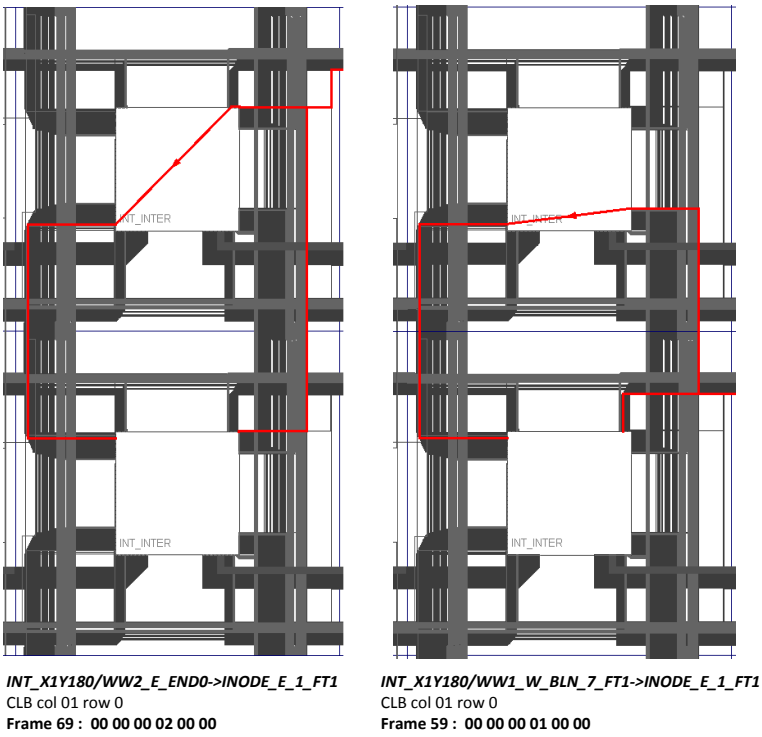
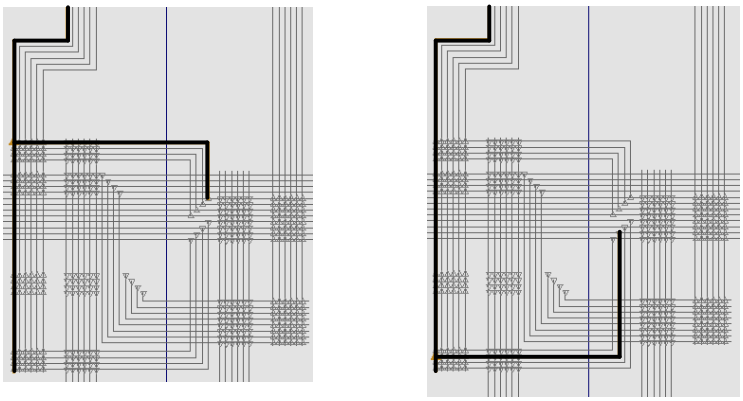


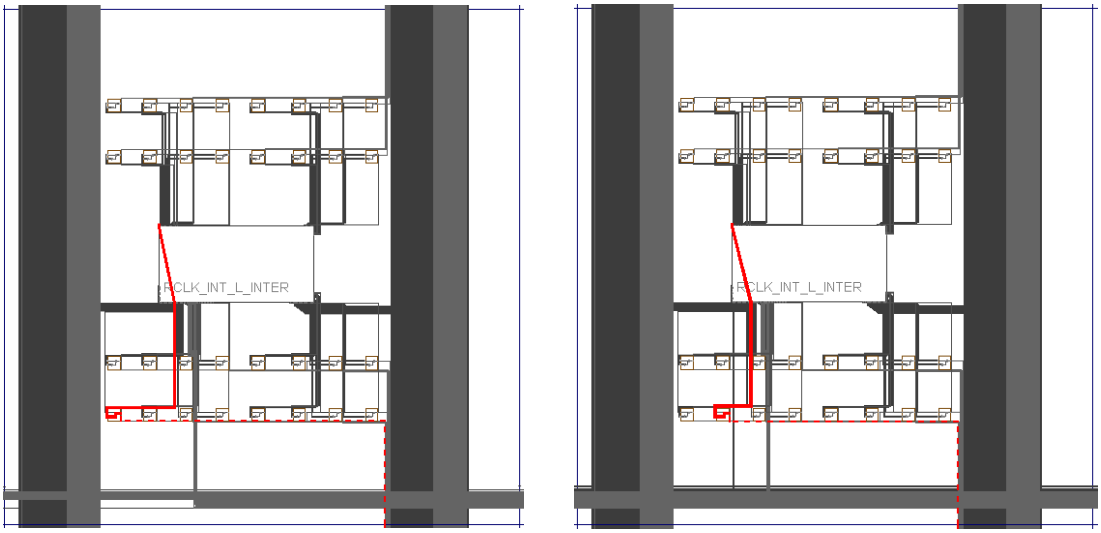
Figure 3.6: Example of bitstream encodings for routing in the UltraScale+ family.



HCLK_L_X32Y78/HCLK_CK_OUTIN_L7 -> HCLK_LEAF_CLK_B_TOPL5
CLB col 01 row 75
Frame 01 : 00 08 09 B3 00 00 00 00

HCLK_L_X32Y78/HCLK_CK_OUTIN_L1 -> HCLK_LEAF_CLK_B_TOPL5
CLB col 01 row 75
Frame 01 : 80 00 09 BF 00 00 00 00

Figure 3.7: Clock resource encodings of 7-Series FPGAs in bitstream.



BUFCE_LEAF_X0Y0

Row Address 0
CLK col 01
Frame 00 : 20 00 22 22 20 00 02 20 00 00 00 00
Frame 41 : 02 02 80 00 00 00 2A A0 00 00 00 00
Frame 42 : 00 AA A0 00 00 00 82 A8 00 00 00 00
Frame 45 : 00 02 22 22 00 00 02 02 00 00 00 00

BUFCE_LEAF_X1Y0

Row Address 0
CLK col 04
Frame 00 : 80 08 00 00 00 00 88 80 00 00 00 08
Frame 38 : 00 02 22 22 00 00 02 02 00 00 00 00
Frame 41 : 02 02 80 A0 00 00 8A A0 00 00 00 00
Frame 42 : 00 A0 00 80 00 00 A8 A2 00 00 00 00

Figure 3.8: Clock resource encodings of UltraScale+ FPGAs in bitstream.

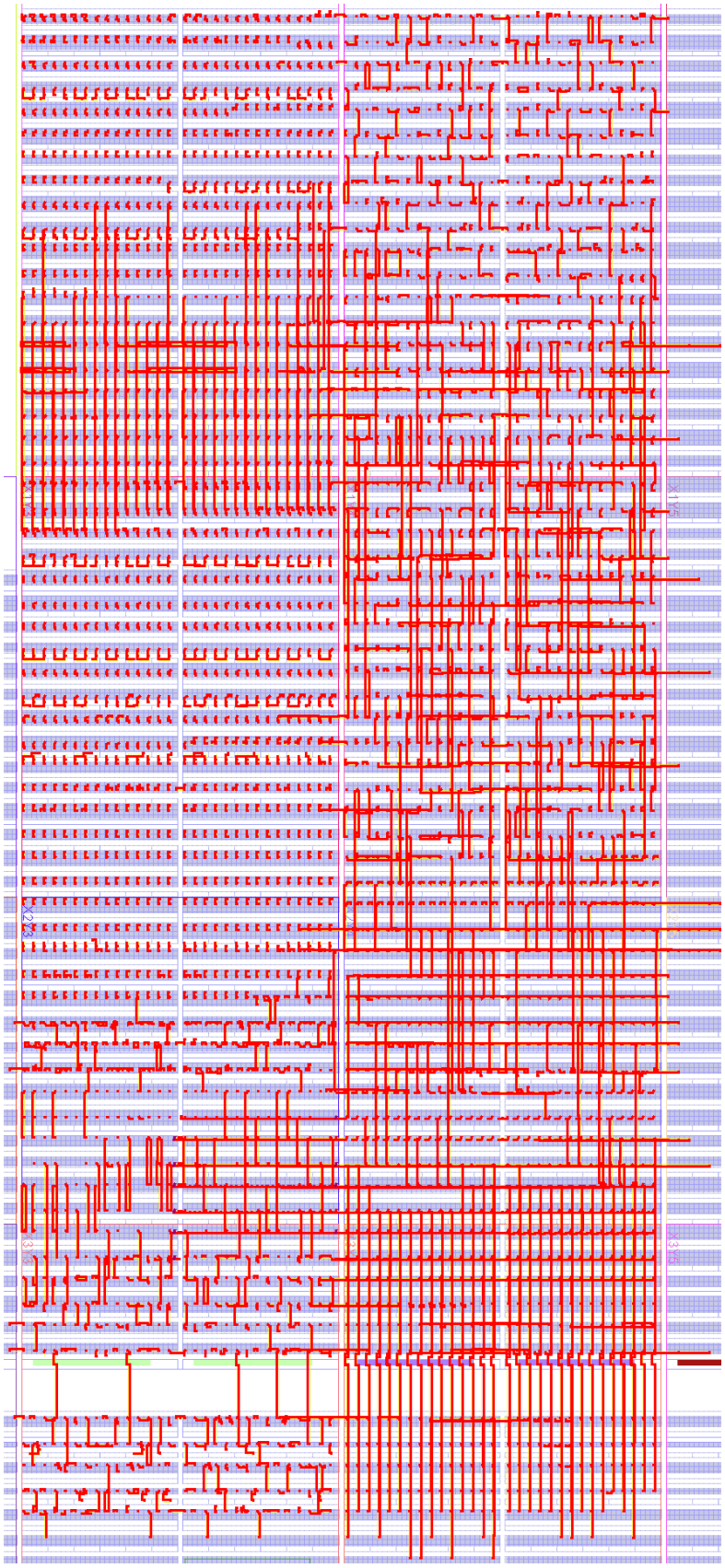


Figure 3.9: All possible connections in a switch matrix of a Xilinx UltraScale+ FPGA are configured in a Vivado design. The routing constraints in this design are generated by TCL commands from a list of potential connections, which is obtained from the GoAhead tool. The resulting bitstream of this design is then parsed through BitMan to get the corresponding bitstream encodings of these connections.

Device resource description: Xilinx FPGAs are organised in *resource columns* in terms of CLB, Block RAM, DSP, clock, and other I/O. Multiple columns are grouped as a *resource row*. Figure 3.3 illustrates a Kintex UltraScale XCKU025 device layout with details of one resource row.

Resource columns consist of *frames* for the configuration of the corresponding primitives and for routing. Every column provides routing resources such as switch boxes with routing multiplexers. These routing resources are used for implementing the signal wiring inside the FPGA fabric. The configuration bits controlling the routing resources are encoded in the bitstream file together with the configuration of all other primitives of the FPGA.

The resource's architecture as well as the number of frames per column in one row stays the same in a family, but commonly differs from family to family. For example, a CLB on a 7-Series device has **8** frames for its content, but **12** frames on UltraScale and **16** frames on UltraScale+ counterparts, respectively. The number of frames for routing is also different due to differences in the routing fabric. Tables 3.1 and 3.2 give a summary on the number of frames for a couple of device families that are all supported by BitMan.

Figure 3.5 and Figure 3.6 show how a connection in a switch matrix is encoded in the bitstream. Moreover, Figure 3.9 shows all possible connections in a switch matrix of a Xilinx UltraScale+ FPGA. We refer readers to see [LAB+05] for more details on the implementation of switch matrix multiplexers on modern FPGAs. Multiplexers on SRAM-based FPGAs are commonly implemented by pass transistors or transmission gates [BRM99] that are directly controlled by configuration SRAM cells. On 7-Series FPGAs, large multiplexer are organised in a row-column architecture with one set of configuration bits selecting the rows and another set for the columns, as discussed in [BKT10] and briefly illustrated in Figure 3.4. For instance, a $20:1$ multiplexer on a 7-Series FPGA is organised in a 5×4 layout and has two groups of configuration bits with $5+4=9$ configuration bits in total, as shown in Figure 3.5.

On the other hand, UltraScale+ devices expose both dimensions of the multiplexers for rows and columns as individual multiplexers directly to the user. Therefore, UltraScale+ devices appear to have more but smaller multiplexers which in turn are typically encoded by a one-hot scheme in the bitstream, as illustrated in Figure 3.6.

Figure 3.7 and Figure 3.8 show how clock resources are encoded in the bitstreams of 7-Series and UltraScale+ FPGAs. Note that the clock architectures have been changed significantly from 7-Series devices to UltraScale/UltraScale+ devices, see [Xil18a, Xil18d] for more details. By changing the configuration data, we could reroute clock signals. BitMan provides an API that allows reporting and manipulating clock tree and other routing resources by simply providing the resource column and the corresponding clock routing wires to be connected or deleted.

In BitMan, we only manipulate the CLB, BRAM contents, the internal configuration of

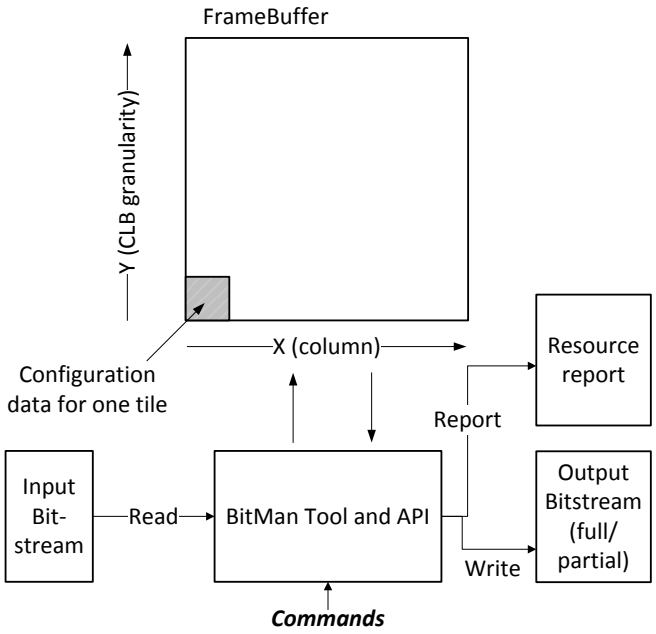


Figure 3.10: The BitMan process.

CLBs, BRAMs, and DSPs, routing, and clock resources because this is sufficient for dynamically reconfiguring a partially reconfigurable module. Other resources, including I/O blocks, Gigabit transceivers, or hardened cores are commonly part of a static system (shell) that usually does not require run-time adaptation through means of partial reconfiguration.

3.1.2 Module Placement and Relocation

From this point, we have to solve the problem of how to model the different resources such that module placement can be carried out transparently with respect to overlaying software stacks. We were inspired by the alphabetical string representation method in [GK15], which uses a *resource string* abstraction for modeling the FPGA device resources. In our case, the used symbols are as follows:

- `N: NULL` (NULL resource which we do not need to read and change the information such as IO fabric, hardcore CPU, ICAP port or PCIe interface),
- `L: CLBL` (a logic tile resource column),
- `M: CLBM` (CLBL plus distributed memory),
- `R: BRAM`,
- `D: DSP`,
- `B: BUFG` (clock resource).

For example, the *resource string* of the first 35 columns on a Kintex UltraScale XCKU025 is:

`NLMLRLMDMLMLRLMDMLMLMLMDMLMLRLMDBMLML`.

With this, reconfigurable regions as well as reconfigurable modules will be modeled as strings, and a feasible module placement requires that the resource footprint string of the module fits the resources available in the targeting reconfigurable region. BitMan can include the resource strings into a bitstream header (for both full and partial bitstreams) and uses this information for testing feasible module placement positions.

Module relocation is achieved by modifying address information fields inside the bitstream. Moreover, the clock resource configuration data of the module may need to be adopted to the specific targeting region as well.

3.1.3 Bitstream Manipulation Tool

BitMan can be used as an independent tool or integrated to a controller as a software API. Table 3.3 shows functions of BitMan API, which are exposed to higher level applications.

Figure 3.10 shows the operation of BitMan. The whole input bitstream will be read and stored in a 2-D array *FrameBuffer*. The *FrameBuffer* also removes the need for configuration readback as the buffer mirrors the configuration of the fabric. BitMan, however, can generate readback bitstreams for debug purposes. BitMan also receives commands from higher level applications. `(X, Y)` coordinate system refers to a grid at CLB granularity. Alternatively, a grid of the height of a clock region in vertical dimension can be used for convenience. All low-level details of the bitstream are hidden from the user and BitMan translates user-friendly commands into low-level bitstream manipulation. For example, Algorithm 1 shows how to use the *resource string* model and resource description information in Table 3.1 and Table 3.2 to specify a *Frame_Index*.

BitMan is written in `ANSI C` to be portable in literally any platform, from a desktop computer with Intel Core i7 to an embedded ARM Cortex-A9 or a soft-core CPU. Its performances in various examples will be evaluated in the next Section.

3.2 Applications and Evaluation

In this section, we discuss how applications could benefit from the proposed BitMan tool. For example, the run-time adaptation of a module such as the module relocation, duplication, rerouting or LUT/BRAM content modification can be applied to save a significant amount of logic overhead. Moreover, the binary-level stitching of an overlay design can boost up some aspect of the CAD toolflow at up to two orders of magnitude, as discussed in Section 3.2.2.

In a recent effort, BitMan has been integrated to an academic vendor-independent toolflow, called EFCAD [PVKH19], to generate Xilinx FPGA partial bitstreams from Verilog source

Table 3.3: BitMan functions. Note that `(X`$_i$`,Y`$_i$`)` are the horizontal and vertical coordination of a logical tile.

High-level APIs	Functionality
`replace_FPGA_region(X`$_0$`, Y`$_0$`, X`$_1$`, Y`$_1$`, X`$_2$`, Y`$_2$`)` `duplicate_FPGA_region(X`$_0$`, Y`$_0$`, X`$_1$`, Y`$_1$`, X`$_2$`, Y`$_2$`)`	Replace/duplicate a rectangular FPGA region bounded by bottom-left `(X`$_0$`,Y`$_0$`)` and top-right `(X`$_1$`,Y`$_1$`)` to a new region which starts at `(X`$_2$`,Y`$_2$`)`. Replace will clear configuration data in the old region.
`reroute_wire(X, Y, input, output)` `reroute_clock(X, Y, input, output)`	Change configuration data of switch box/clock multiplexer `(X, Y)` to connect `input` to `output`.
`change_LUT_content(X, Y, LUT, new_config)` `change_BRAM_content(X, Y, new_config)`	Change the content of LUT `(LUT)`/BRAM `(X, Y)` to the `new_config` data.

Table 3.4: Performances of BitMan (B) and Maverick (M)'s bitstream generation [Gli20] on an Intel Core i7 desktop environment and an embedded Dual-core ARM Cortex-A9 CPU with 512MB RAM (ZedBoard/PYNQ-Z1) platform.

	Processing time (μs)							
	LUT		CLB		BRAM		Routing	
System configuration	B	M	B	M	B	M	B	M
Quad-core Intel Core i7	372	225	1069	717	2604	—*	487	311
Dual-core ARM Cortex-A9	45	93	94	316	229	—	54	146

* BRAMs are not supported by Maverick (M).

Algorithm 1 Algorithm to find *Frame_Index*. To compute a *Frame_Index* of a specific frame (i.e. the absolute position of that frame in the whole bitstream), we have to count the weighted sum of frames with respect to the resource column type.
* *RowAddress*, *MajorAddress*, and *MinorAddress* are fields in the Frame Address Register (FAR), as illustrated in Figure 3.1.
* *RAMInterFrames*, *BUFGFrames*, *CLBFrames*, *DSPFrames*, and *RAMContentFrames* are the number of frames per resource column, see Table 3.1 and Table 3.2 for details.
* *RowCLBFrames* and *RowRAMFrames* are the number of frames per CLB row and RAM row. *RAMOffsetFrames* is the number of frames before the configuration data of BRAM contents. These values are device-specific.

```
1: if NOT a BRAM − content column then
2:     for i = 0 to MajorAddress do
3:         if a BRAM − interconnect column then
4:             Increase NoColRAM
5:         else if a BUFG column then
6:             Increase NoColBUFG
7:         else if a CLBM column then
8:             Increase NoColCLBM
9:         else if a CLBL column then
10:            Increase NoColCLBL
11:        else if a DSP column then
12:            Increase NoColDSP
13:        end if
14:    end for
15:    return (MinorAddress + NoColRAM * RAMInterFrames + NoColBUFG *
       BUFGFrames + (NoColCLBM + NoColCLBL) * CLBFrames + NoColDSP *
       DSPFrames + RowAddress * RowCLBFrames)
16: else if a BRAM − content column then
17:    return (MinorAddress + MajorAddress * RAMContentFrames + RowAddress *
       RowRAMFrames + RAMOffsetFrames)
18: else
19:    return (-1)
20: end if
```

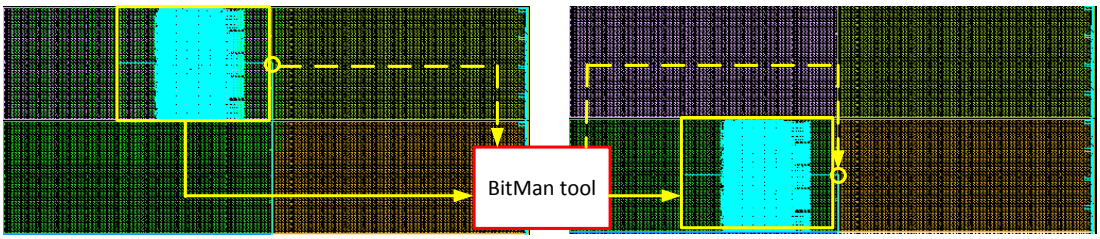


Figure 3.11: An example of module relocation. There are 2 steps to achieve this moving: 1) relocate the whole module resource (fixed arrow), and 2) reroute clock signals for the relocatable module (dashed arrow).

codes. The whole process including synthesis, implementation and bitstream generation runs totally on the Xilinx Zynq UltraScale+ MPSoC in which 64-bit ARM CPUs are tightly coupled with UltraScale+ FPGAs on the same die, and the resulting bitstream will then be downloaded onto FPGA fabric via PCAP to reconfigure the FPGA part of the chip, so called programming logic (PL), to change the system behaviour. Therefore, in the user's point of view, it looks like the chip compiles itself.

Note that plain un-encrypted Xilinx FPGA bitstreams were used in below examples. BitMan supports compressed bitstreams as generated by the Xilinx vendor tools, but it does not support encryption. The later can be implemented by a system providing a secure storage mechanism.

3.2.1 Run-time Adaptation

Module relocation and duplication: A partial module might spread across a number of CLB and/or BRAM columns, and its reconfiguration can be carried out without affecting surrounding modules or the static system. In particular if some of the routing resources within a reconfigurable region implement static routing (e.g., for crossing signals of the surrounding system through a reconfigurable area, as discussed in Section 4.2.3), the relocation and duplication are permitted and will have no side-effects due to a partial reconfigurable process. The relocation and duplication require routing constraints on the static routing through reconfigurable regions that can be generated with the GoAhead tool [BKT12], as presented in Chapter 4.

Rerouting: BitMan is able to reroute clock signals by reconfiguring clock multiplexers in BUFG or BUFHCLK cells. By doing this, a relocatable module could be disabled/enabled or maintains its operation at a different frequency. This is also needed to keep the routing of the clock resources that belong to the static system untouched when partially reconfiguring a module, as discussed more detailed in Chapter 4.

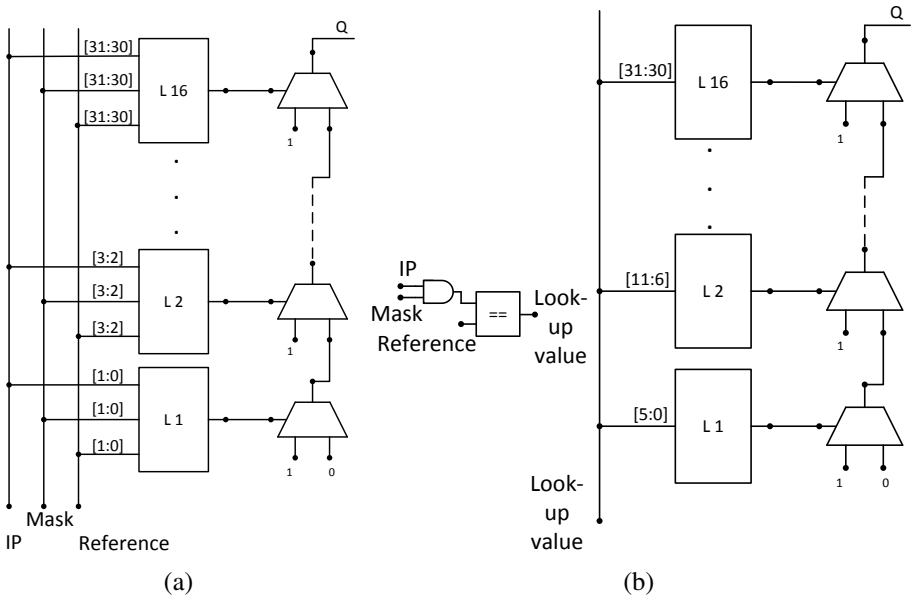


Figure 3.12: Conventional CAM (a) *vs* LUT-modifiable CAM (b).

Figure 3.11 shows an example of module relocation. While simple systems may use only a single clock, BitMan is designed for complete real-world systems that use a plurality of clock networks (e.g., for different memory controllers [Ama], NIC interfaces [CCP+16], PCIe [VKVF16]). Any other routing resource including interconnection could be changed accordingly.

Table 3.4 shows BitMan performances on an ARM Cortex-A9 embedded platform and an Intel Core i7 desktop. In this experiment, a 3.85MB bitstream of the Zynq-7000 XC7Z020 device (ZedBoard) was used, and we have manipulated configuration data of an LUT, a CLB, a BRAM content, or a routing primitive, respectively. Moreover, we found that BitMan's performance was comparable with another academic toolflow Maverick [Gli20], which uses Project X-Ray [Sym] to generate bitstreams for 7-Series FPGAs.

LUT/BRAM content modification: BitMan supports updating the content of LUTs and BRAMs of an FPGA fabric on-the-fly. Application examples for this are changing coefficients in digital filters, updating keys in cryptography systems or swapping binaries stored in on-FPGA memory using the configuration interface rather than some extra user logic. An example with LUT update for FIR coefficients was mentioned in [KS16], while another example with nullifying S-Box contents storing in BRAMs to compromise the AES encryption scheme of a high-security commercial USB flash drive was presented in [SFK+17].

For demonstration the usefulness of LUT content modifications, we looked into an application where we compare an IP address with a masked reference IP. As it can be seen, the

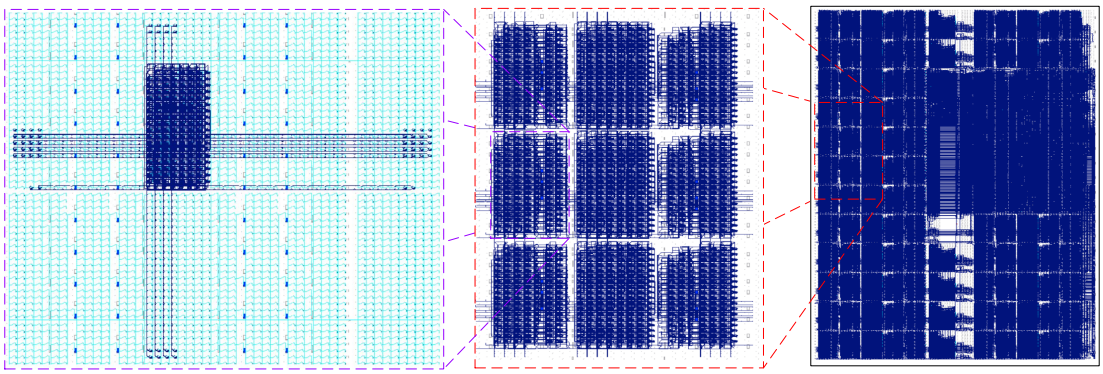


Figure 3.13: Stitching processing elements (PEs) to build a large Coarse Grained Reconfigurable Array (CGRA) at the netlist level in rapidly building overlay (ROB) work [YKL15]. With BitMan, we are able to perform this stitching at the binary (bitstream) level.

Table 3.5: BitMan performance on overlay architecture's support.

	BitMan	Rapid Overlay Builder [YKL15]
Numbers of PEs	101	101
Time (seconds)	2.24	2259

CAM approach in Figure 3.12b results in a carry chain that is only about a third as long as the conventional approach in Figure 3.12a. This saving in both resources and latency can be substantial for large CAMs as usual in latest routers.

3.2.2 Hardware Mapping and Linking for the Overlay Architecture

In [YKL15], an approach for rapidly building an overlay CGRA (Coarse Grained Reconfigurable Array) is presented where a number of physically implemented processing element (PE) modules were replicated for building large CGRAs with a hundred or more PEs (see Figure 3.13). This approach tries to amortise CAD tool time for one PE to build large scale systems. By stitching together fully placed and routed PE tiles, CAD tool times could be reduced by 9.3× in the original paper [YKL15].

However, the stitching itself in [YKL15] was carried out at the netlist level which requires a time-consuming netlist translation process that we circumvented by stitching PE tiles directly at the bitstream level. To demonstrate BitMan, we repeated the same experiments but instead of stitching at the netlist level, the stitching was performed at the bitstream level.

As shown in Table 3.5, this reduces the whole bitstream generation process into the range of seconds. The performance of BitMan would allow a just-in-time stitching of CGRAs. For example, optimised PEs (e.g., integer, floating-point, or logic operations) are stitched together

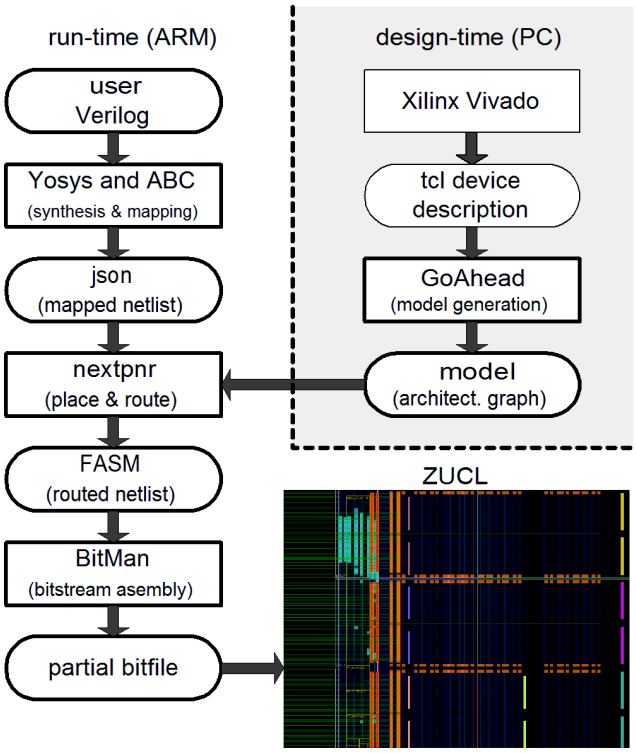


Figure 3.14: The EFCAD flow. The figure is adopted from [PVKH19].

at run-time for a new application without using the vendor CAD tools.

3.2.3 Bitstream generation for Enabling On-chip Self-compilation

EFCAD [PVKH19] is one of the latest efforts from FPGA community to introduce an end-to-end vendor-independent FPGA development toolflow which takes Verilog source codes as inputs, and then, generates (partial) bitstreams for education and research purposes. The flow is targeting Xilinx Zynq UltraScale+ FPGAs with the ARM-FPGA heterogeneous computing architecture so that the synthesis, implementation, and bitstream generation are executed on the CPU side, so called processing system (PS), then the resulting (partial) bitstreams are deployed

Table 3.6: Overheads of bitstream generation for a partial region of 8 CLB columns (i.e. 3840 LUTs) in a XCZU3EG Zynq UltraScale+ FPGA.

System configuration	Time (seconds)
Vivado 18.2.1 on Quad-core Core i7 @ 3.40GHz and 64GB RAM - Windows 7	24.87
BitMan on Quad-core ARM Cortex-A53 @ 1.5GHz and 2GB RAM - Linux 4.18	1.86

on the FPGA side (PL). As the mentioned transformation from Verilog source codes to FPGA bitstreams all occurs inside the chip transparently from the users and results at changing system behaviours, EFCAD gives the users a sense that the chip is able to compile itself.

Multiple academic open-source projects, including Yosys+nextpnr [SHW+19] for logic synthesis and implementation, GoAhead [BKT12] for generating architectural model, ZUCL [PVVK18] for infrastructure framework, and BitMan for bitstream generation, have been integrated to create the flow, as illustrated in Figure 3.14. Bitstream generation for a partial region of 8 CLB columns (i.e. 3840 LUTs) was measured at 1.86 seconds on average on the UltraZed platform, which is an order of magnitude less than the latency of Vivado's bitstream generation on a desktop machine, as shown in Table 3.6. More detail about EFCAD can be found at [PVKH19].

3.3 Chapter Summary

In this chapter, we have introduced the BitMan tool and API that permits complicated bitstream manipulation tasks to be carried out at run-time for all latest FPGAs of the vendor Xilinx. Various use cases were demonstrated and discussed to show BitMan tool's advantages. This includes module relocation and duplication, modifying switch matrix and LUT settings as well as stitching together CGRAs from a PE library. The results of BitMan are configuration bitstreams that can be directly sent to the FPGA through any available configuration port (e.g., ICAP, or PCAP). Moreover, BitMan plays an essential role to fully exploit the flexible run-time reconfiguration, which is a key feature in FPGA-virtualised systems, as discussed detail in Chapter 4, Chapter 5, and demonstrated in Chapter 6.

Chapter 4

Decoupled Compilation Flow for FPGA Virtualisation

In England, everything is permitted except what is forbidden.
In Germany, everything is forbidden except what is permitted.
In France, everything is allowed, even what is prohibited.
In the USSR, everything is prohibited, even what is permitted.
Winston Churchill

As the design methodologies and toolflows from major FPGA vendors face significant obstacles as pointed out in Chapter 2, we need to adopt an alternative methodology to overcome these obstacles and to achieve the objectives which we have setup for the methodology. For example, one of the goals is to enhance the design productivity which can be achieved by decoupling the design of the basic infrastructure (shell) and the hardware accelerators (roles), i.e. the development abstraction mentioned in Chapter 1. By this, we can compile the application independently from the whole system implementation, and hence, it can be compiled dramatically faster than what we are able to do with the vendors' flows. Like in software compilation where the complexity of the operating system is hidden by conveniently usable APIs that allow compiling application individually, we propose the decoupled module compilation flow that hides the complexity by standardised hardware interfaces for hardware synthesis and physical implementation.

The decoupled compilation flow is described in Section 4.1 of this chapter, while details of how to utilise this methodology and corresponding academic tools to build up the systems are being discussed in the other Section 4.2 (for shell design) and Section 4.3 (for the module compilation). A number of FPGA-virtualised systems have been implemented and deployed on top of different platforms using FPGAs ranging from Xilinx Zynq-7000 to Zyng UltraScale+ families by using this proposed methodology.

As part of this PhD, the decoupled compilation flow for CPU+FPGA heterogeneous computing systems was designed as well as integrated necessary tools into the flow. With this, users were able to compile partial bitstreams for their HLS applications running on heterogeneous systems before Xilinx has introduced the Vitis flow [Xil20] to offer a similar software-centric compilation experience. Parts of this Chapter have been published in [PVVK18, PPV+19, VPPK20] and represented here with additional materials for the sake of completeness.

4.1 Design Methodology

4.1.1 Overview

There are fundamentally differences in the state-of-the-art design methodology from vendor tools and the here proposed design methodology. In vendor tools, *every possible routing is permitted except what is forbidden*; whereas in our counterpart, *every possible routing is forbidden except what is permitted*. Hence, to provide the infrastructure, development, and deployment abstractions as mentioned in Chapter 1, the following strict requirements need to be satisfied:

1. Communication interfaces between modules and remaining parts of the system must be identical, not only with respect to the logical protocol but also with respect to the physical implementation such that relative positions of connection wires are the same in all PR regions. This requirement guarantees that modules are able to receive operation commands from the host CPU and to transfer data back and forth to the main memory, regardless of a module placement position;

2. Clock splines must be distributed in the same regular pattern across every PR regions. A module will need to use these clock splines to provide clock signals to slice flip-flops, BRAMs, and DSPs[1];

3. *No routing from the static part is allowed in the reconfiguration part and vice versa* (except the *pre-defined* interface and communication signals) to ensure that module relocation does not interfere with other parts of the system.

The introduced methodology is used to design an FPGA-virtualised heterogeneous computing system which provides such necessary abstractions.

The proposed design methodology was built upon the standard FPGA development flow (Vivado toolchain for Xilinx FPGAs) to implement the basic infrastructure, acting as the OS

[1]Note that while clock routing including clock splines could be carried out by BitMan at run-time, this would require running static timing analysis again which is not trivial to perform at bitstream level.

shell, and the hardware applications. Our contributions, however, included all the additional design steps and the corresponding automatic tools that customise and adapt the default flow to get the final systems. This compilation flow integrates the entire shell and module development process into a unified design framework, as illustrated in Figure 4.1. Note that our method ultimately does not require the usage of vendor's partial reconfiguration design flow, which is a challenge for even experienced FPGA designers.

The main steps of the decoupled compilation flow are described as following:

1. **Planning**: In this step, a system architect needs to be involved to make a series of system-level design decisions.

   (a) **Resource budgeting**: Based on the number of available resources on the chip, an architect has to decide which parts of the system will be static for the FPGA OS shell and which parts will be reconfigurable for the deploying hardware accelerators. Here, the trade-off is that the architect should allocate as many resources as possible for the reconfigurable part as it affects the system throughput, but should leave sufficient resources to the static part for system upgrade or maintenance. This trade-off is case-dependent, therefore difficult to be automated entirely. This step is the only place in the entire framework that requires a thorough understanding of FPGA domain knowledge. However, this step is only carried out once for a system on a specific FPGA device. This step results in 1) the static system floorplan and 2) bounding boxes of the reconfigurable regions.

   (b) **Static/partial partitioning**: The resulting resource budgets for the static and the partial parts from the previous step will be used for partitioning the chip layout to clearly divide boundaries between these parts. It also clarifies the size of each PR region on which the High Level Synthesis (HLS) tools [CLN+11, CCA+13, Xil19b] can operate the Design Space Exploration (DSE) for throughput optimisation [MMRL17].

   (c) **Communication interface definition**: The communication between the static and the partial regions has to be specified as well in terms of protocol, mode, and data-width. This step results in the wrapper templates for PR modules for system implementation.

2. **Routing Constraint Generation**: The system floorplan and the PR wrapper template from the previous step will be used to generate the routing constraints. Once the allocated static/partial parts and communication protocol are known, we can put implementation rules to academic PR frameworks, such as GoAhead [BKT12] and a TCL library, called TedTCL [Ves18], to generate routing constraints automatically. These constraints

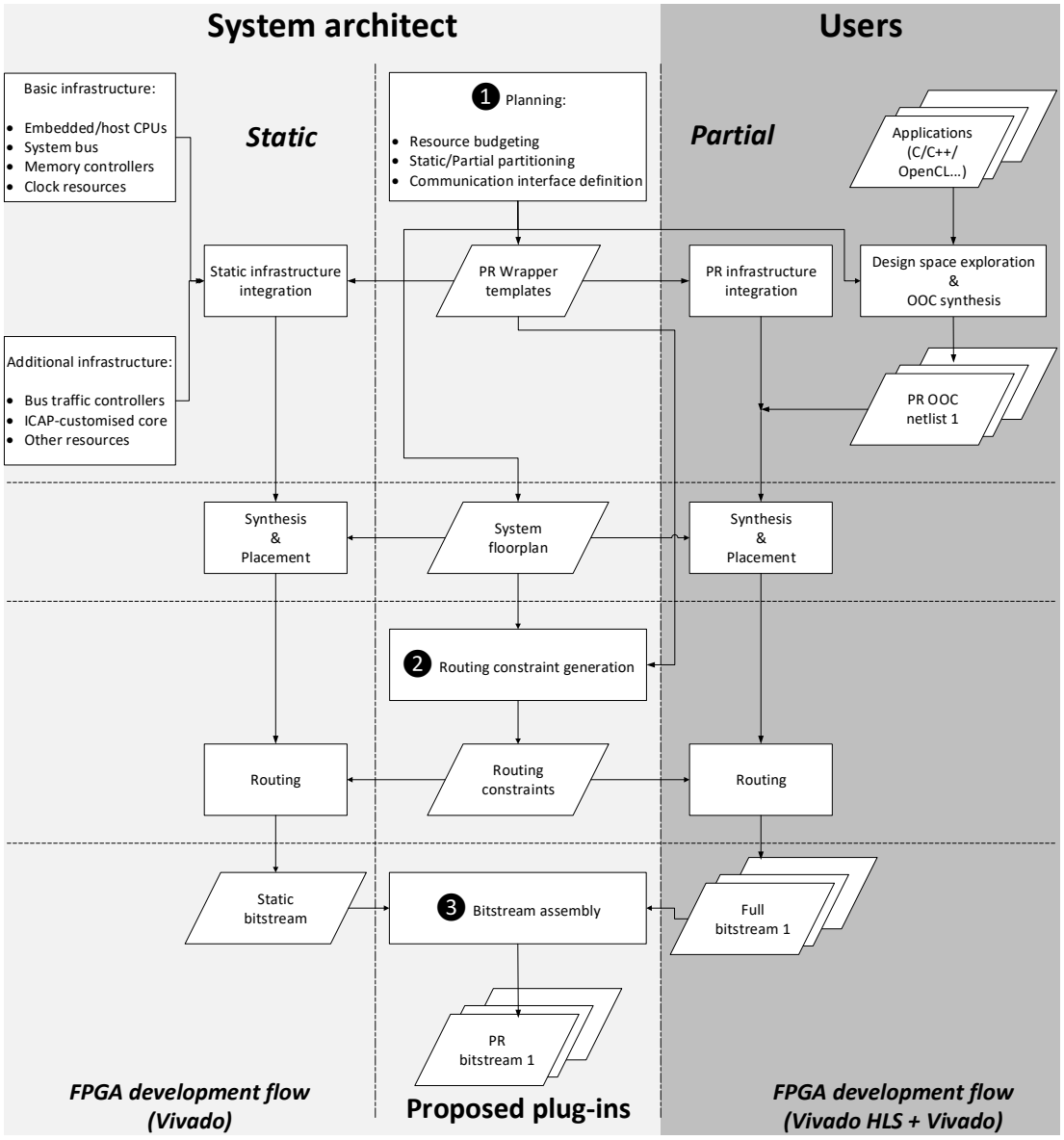


Figure 4.1: The decoupled compilation flow. The left part is performed once for a specific version of a shell while the right part is performed once for each module (role).

are given as generated TCL files, which are then used by Vivado to guide the routing stage.

3. **Configuration Bitstream Generation**: The decoupled compilation flow hereby results in *full configuration bitstreams* of static and module designs. To compose *partial bitstreams* for modules, bitstream manipulation described in Chapter 3 has been used. This bitstream assembly step can be done either offline to create the hardware library or online to relocate the hardware accelerators.

4.1.2 Academic Tools for Routing Constraints Generation and Bitstream Manipulation

Routing constraints are useful to implement systems which fulfil the requirements for FPGA virtualisation, yet vendor toolflows [Xil14, Alt10] have not supported to generate such required constraints automatically. There are various choices of academic PR frameworks beyond the state-of-the-art vendor ones such as RapidSmith2 [HNH15], RapidWrigth [LK18], ReCoBus-Builder [KBT08], GoAhead [BKT12], or an unpublished TCL library, called TedTCL [Ves18].

Throughout this PhD project, GoAhead and TedTCL have been used intensively as they support UltraScale/UltraScale+ Xilinx FPGAs and output TCL files to be used with the Vivado toolflow to guide the physical implementation stage. GoAhead was first released in 2012 and is being maintained by Christian Beckhoff and Dirk Koch, while TedTCL has been developed by Malte Vesper during his PhD project in the University of Manchester.

Finally, for the bitstream transformation at design time to compose the hardware library and at run-time to relocate the module, we use the BitMan tool and API, which was introduced in Chapter 3.

4.2 Shell Design

This section discusses how to apply the proposed methodology to implement the static infrastructure (shell). As a case study, we are targeting the Xilinx Zynq-7000 and latest Zynq UltraScale+ devices that include hardened ARM CPUs in the Processing System (PS) part, coupled with a 7-Series/UltraScale+ FPGA fabric in the Programmable Logic (PL) part. Other hardened primitives in the targeting FPGAs are the AXI interfaces, memory controllers, and clock domain crossing, of which the systems are utilising to build a light-weight yet feature-rich FPGA operating system services. Like it is common in the software world to use an operating system without changes, our shells are designed to meet the requirements of most users without reconfiguration.

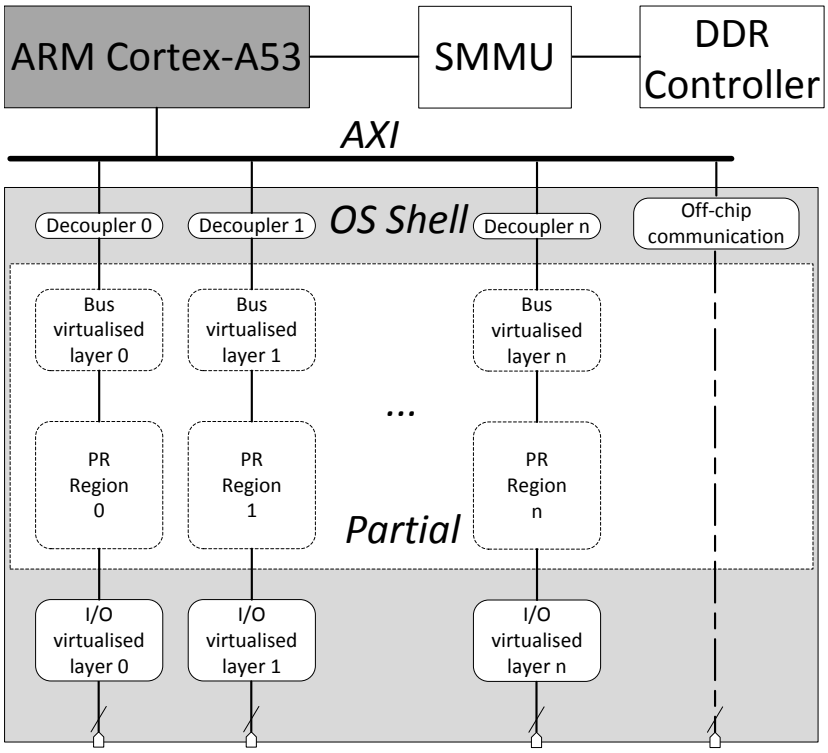


Figure 4.2: The overall organisation of an FPGA-virtualised system.

4.2.1 Implementation of the Shell

The shell design starts with the integration of basic infrastructure and additional infrastructure to the unified top-level shell design (see Figure 4.2). Shell includes the hard 32-bit ARM Cortex-A9 (in the Zynq-7000 platforms) or the 64-bit ARM Cortex-A53 (in the case of Zynq UltraScale+ platforms) CPU cores, memory controllers, AXI4 interconnects, Xilinx PR Decouplers for disabling/enabling static and module communication, clock management tiles for tuning module frequencies, and PR Wrapper instantiations.

The PR Wrapper module contains two VHDL files: a wrapper to connect the AXI4 bus to the PR Module Interface and a file describing this interface in terms of FPGA primitives. The PR Module Interface has an AXI4-Lite Slave for control register access via the CPU and an AXI4 Master for memory access. This interface corresponds directly to the interface that is used by default for OpenCL kernels when compiling with Vivado HLS. However, other module specifications (e.g., C/C++, Verilog, or VHDL) are supported as well, and modules can directly access the system DDR memory.

This fixed interface between the hardware module and the static system is implemented using the available routing resources in the Zynq-7000 and Zynq UltraScale+ FPGA devices. In particular, we keep this interface identical for all PR slots to serve relocatable hardware modules by pre-placing and pre-routing these communication signals. This reassembles the

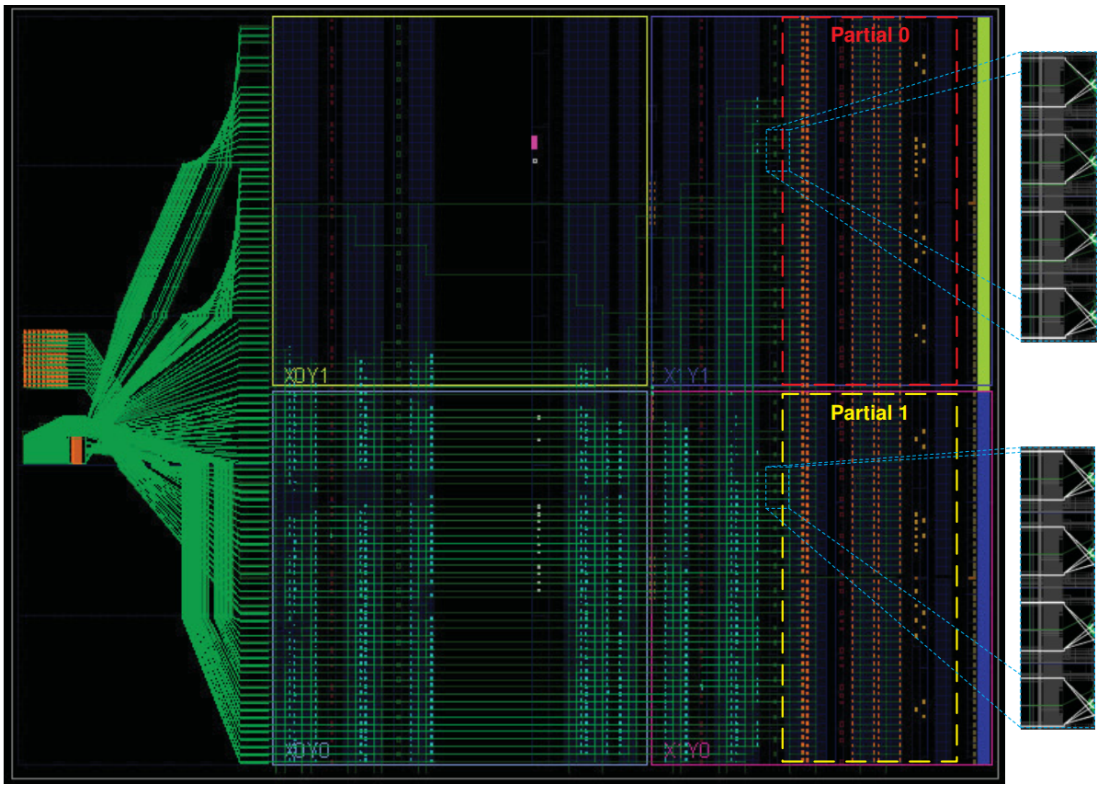


Figure 4.3: The physical implementation on the Zybo platform. This version has two *slots* which can host up to two FPGA applications simultaneously.

bus macro approach, which was popular for Xilinx Virtex-II devices [LBM+06].

Moreover, clocking resources for PR slots must be kept identical for relocatable hardware modules. We achieve this condition by blocking all except for a defined subset of the `BUFCE_LEAF` primitives inside PR slots. This forces the router to use only the defined subset of these primitives that each drives a specific vertical clock spline which connects to the flops, BRAMs, and DSPs. However, we only route the clock for the PR regions this way. When routing the static system, we firstly route the PR module clocks with `prohibit` constraints on the `BUFCE_LEAF` primitives, then we remove these constraints and route the rest of the system. This allows us to route further clock nets as needed by the static system (e.g., for future implementing high-speed peripheral modules).

Finally, to prevent any static signal from violating PR regions, a blocker macro is inserted before routing the static system. This blocker macro is generated by the GoAhead tool [BKT12] or TedTCL library [Ves18] according to the system floorplanning.

The above steps result in the final static designs shown in Figure 4.3 for the Zybo platform, Figure 4.4 for the ZCU102, and Figure 4.5 for the UltraZed/Ultra96 platforms. Note that the Zybo has a Zynq-7000 FPGA while the others (ZCU102, UltraZed, and Ultra96) feature Zynq UltraScale+ FPGAs. We can see that PR Module Interfaces at the borders between the

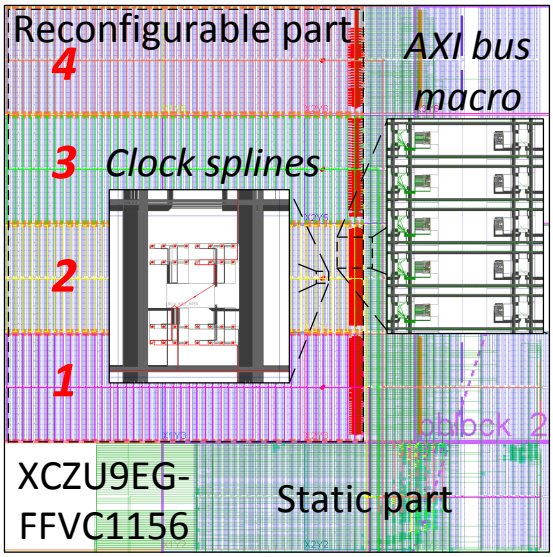


Figure 4.4: The implemented static system on the ZCU102 platform. Module Interfaces and Clock splines are kept identical for all PR regions (slots). There are four *slots* for hosting a maximum of up to four FPGA applications at the same time. The AXI bus macro ensures that all slots provide the same physical interface implementation and the reconfigurable part routes the clock signal to the same vertical clock spines.

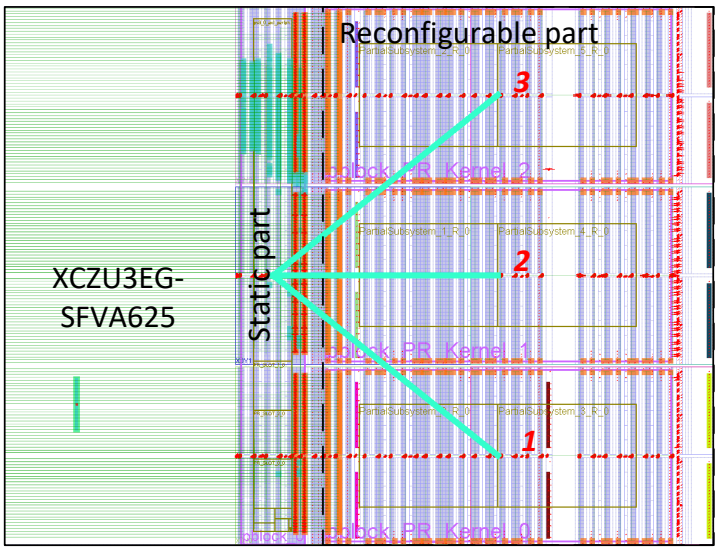


Figure 4.5: The physical implementation on the UltraZed and Ultra96 platforms. This version has three *slots* which can host up to three FPGA applications simultaneously.

reconfigurable part and the static part have the same relative physical positions in all PR slots. Furthermore, clock splines are distributed in the same pattern across PR slots.

Software users may not need to concern about this static shell design as it can be provided pre-built and can be used immediately. Moreover, there is no need to update their hardware

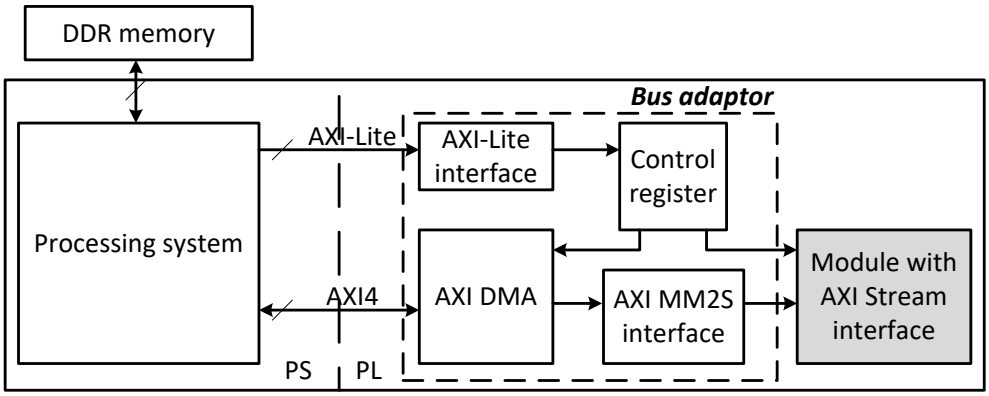


Figure 4.6: An example for bus virtualisation: the module has a 32-bit AXI-Lite interface and a 32-bit AXI Stream interface without DMA engine. In this case, the *bus adaptor* with AXI DMA and AXI MM2S IPs are chosen to carry out the communication with the rest of system.

modules if there is an upgrade for the static system implementation, thanks to the unique way of implementing accelerators independent to the shell. The shell also supports combining multiple consecutive slots for hosting larger monolithic modules. In this case, only one PR module interface will be used.

4.2.2 Bus Virtualisation

Operating a hardware accelerator needs communication with the host CPU to issue commands as well as access to memory for the data to process. A module kernel may have a wide range of bus width, such as 32/64/128-bit width, and various bus protocols, such as AXI4 Master/Stream in the proposed approach. Moreover, HLS kernels are often equipped with DMA engines for fetching data from memory by default, which is not always the case with hand-crafted RTL or customised netlist accelerators. This fact requires another component in the proposed shell as we do not want to replace or adjust the current static infrastructure to support another interface.

We have tackled this issue by providing another level of abstraction for bus interfaces between the FPGA applications and shells, where the interface of the 32-bit AXI-Lite protocol and the 128-bit AXI4 protocol are fixed. Depending on what exact physical interface will be used by a module, we use a set of *bus adaptors* that will be instantiated in a module wrapper such that a module can communicate with the rest of the system as required by the given individual FPGA application. An example of this process is illustrated in Figure 4.6. With this, shells can remain light-weight, operational, and unchanged, while the FPGA applications can be wrapped up with the provided *bus adaptor* at design-time or run-time. The hereby shells support up to 128-bit wide datapaths for memory accesses because this is the native width to the ARM SoC. The advantage of the here proposed *bus adaptor* concept is that an adaptor is only integrated into a module if needed and not speculatively provided by the shell.

Table 4.1: Resource overheads for bus virtualisation at the logical and physical levels.

Module interface	Shell Interface	Bus adaptor's services	Resource overhead		
				Logical Level	Physical Level
32-bit AXI-Lite and 32-bit AXI4 Master	32-bit AXI-Lite and 128-bit AXI4 Master	AXI Inter-connect	LUTs	153	2400
			FFs	284	4800
			BRAMs	0	12
32-bit AXI -Lite and 32-bit AXI Stream	32-bit AXI-Lite and 128-bit AXI4 Master	Control register, AXI MM2S, and AXI DMA	LUTs	1952	2400
			FFs	2694	4800
			BRAMs	2.5	12

We also use a *bus adaptor* to translate between AXI Master and AXI Stream protocols. We provide different versions of AXI Stream adaptors to be used depending on the AXI Stream channel width. A user can either re-compile their modules with a logical wrapper of the appropriate bus adaptor at design phase or stitch their modules with a pre-built binary of that bus adaptor at run-time. The latter option is similar to the case of run-time module stitching as presented in Section 3.2. However, because a fixed portion of the slot is allocated to those bus adaptors in common scenarios, the resource overhead is significant, as shown in Table 4.1. Figure 4.7 shows the implementation of the bus adaptor with control register for wrapping the module registers with the AXI protocol, AXI MM2S, and AXI DMA services for the module which has 32-bit AXI-Lite and 32-bit AXI Stream interface.

4.2.3 I/O FPGA Virtualisation

Ideally, FPGA applications should be implemented transparently without the detailed knowledge of I/O layouts in FPGA fabrics to allow cross-deploying of hardware modules (roles) without re-compiling. This comprises a two-fold problem:

1. A module which has to deal with I/O pins does not need to care about PCB layout details across platforms.

2. A computing-intensive module which does not need access to I/O pins should occupy as many resources as possible without being blocked by other system's off-chip communications.

To resolve these issues, we constrained a set of dedicated physical wires, which were crossing the partial region boundary, in an orderly and regularly routed pattern for the static crossing signals. In the first case, as illustrated in Figure 4.8, those physical wires between an FPGA

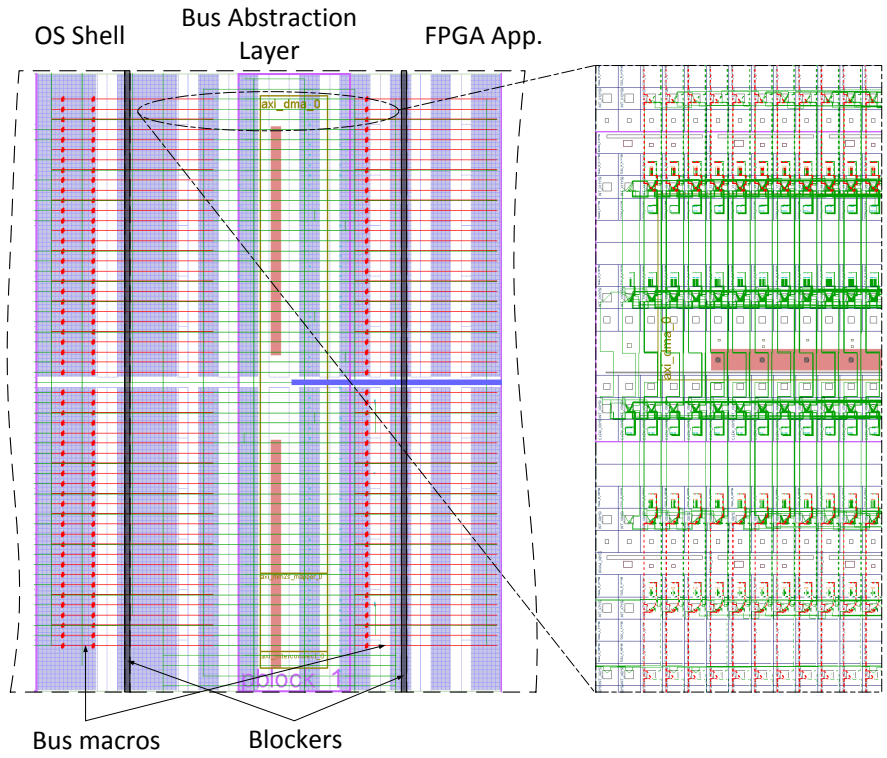


Figure 4.7: Implementation of a bus abstraction layer on UltraZed/Ultra96 platforms [PPV+19]. The bus adaptor is implemented as a module binary and stitched to the system at run-time by partial reconfiguration. The adaptor is a partial module that in turn interfaces to other partial modules. This technique helps users avoid re-compiling the modules but pays extra area overhead of the implemented bus adaptor.

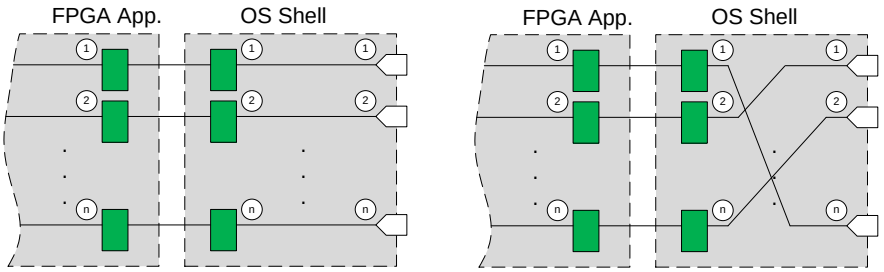


Figure 4.8: I/O FPGA virtual pins: physical connections from FPGA application to OS shell are fixed while the OS shell can be implemented variously to adapt to the new I/O layouts.

application, and different shell regions are constrained to act as fixed virtual pins, such that the module can be relocated to different locations and still be fully functional as it accesses modules using the same wires with respect to its module bounding boxes. Moreover, inside shells, those virtual pins can be connected to different FPGA I/O pins to adapt to different PCB layouts.

In the second case, long physical wires, which are hardly used in practice to implement

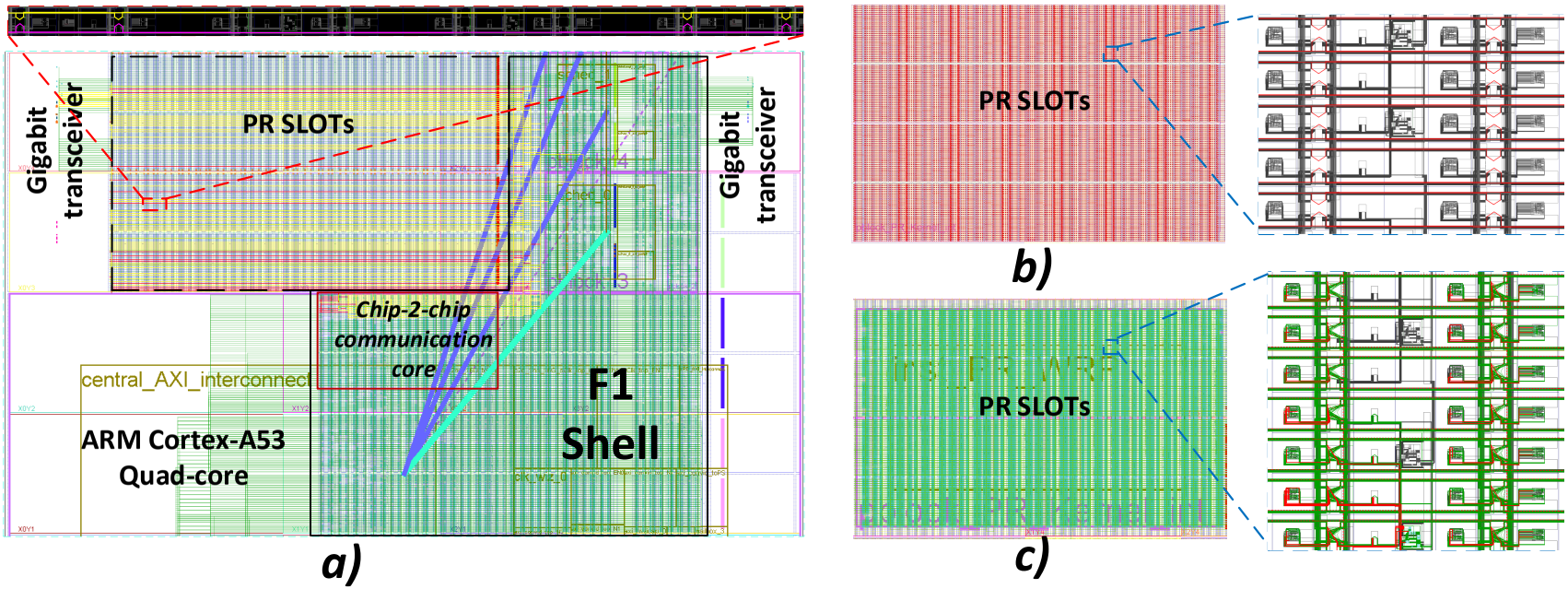


Figure 4.9: Implementation of the I/O virtualisation technique on ZCU102. (a) is the implemented F1 shell, in which the static communication signals crossing the PR slots in an orderly and regular manner using horizontally 12-long wires (yellow for transmitting signals and red for receiving signals); (b) is a template, in which all horizontally 12-long wires used by the static communication signals are pre-occupied (see red dotted lines) to implement the relocatable accelerators; (c) is the implementation of the Michelsen accelerator using the template to generate a single partial bitstream for all PR slots in the QFDB-based system.

the module, were reserved to carry out the communication crossing a partial module region. To this Xilinx Zynq MPSoC system, horizontally 12-long wires such as `EE12_*` and `WW12_*`, which route 6 switch matrices far per East/West direction, were deliberately exploited to carry the static communication signals for both directions of transmitting and receiving data from the 10-Gigabit transceiver. However, we reserved those wires in a way that routing could only be implemented in a straight channel through the reconfigurable region. These constraints were implemented by sparing out tunnels in the GoAhead-generated blocker hard-macros [BKT12], which were placed in the boundaries of the reconfigurable part and the static part to avoid unwanted routing from each part violating onto the other so that only reserved physical 12-long wires could be used for routing of the static communication signals. Those 12-long wires, which start from switch matrices close to the vertical boundary of the static part and PR part, would end in switch matrices inside the PR part. Therefore, additional constraints must be placed inside the PR part to force the static communication signals to continue with the predefined routing pattern, as shown in Fig. 4.9a.

The module was then not allowed to use those horizontally 12-long wires as they belong logically to the static system. Indeed, we pre-occupied the `EE12_*` and `WW12_*` wires, which were being used by the static communication signals, and hence, force the module implementation not to use them, as shown in Fig. 4.9b. Therefore, we can place reconfigurable modules in regions that contain static routing paths without interference. In fact, modules left a more extensive set of possible long wires unused such that there was no interference regardless at which PR region the module is placed. With this, we allowed static signals crossing the reconfigurable regions while still supporting module relocation, as shown in Fig. 4.9c.

This approach is well-suited to Xilinx UltraScale+ FPGAs that provide in each switch matrix 8 12-long distance wires. This means that each horizontal CLB row provides 48 long distance wires per direction ($48 \times 60 = 2880$ wires per East/West direction in total per clock region). We can therefore anticipate that reserving a few hundred wires will have little impact on routability.

4.3 Module Compilation

This section introduces the compilation flow for hardware modules which could be deployed on top of the proposed shells at run-time. The hereby compilation flow is a direct result of the proposed design methodology and analogous to software compilation, as shown in Figure 4.1. This decoupled compilation flow has been verified in several publications such as [PVVK18, PPV+19, HSPK17, PHK+18] in multiple FPGA platforms including Zybo, ZedBoard (Zynq-7000 family), ZCU102, UltraZed, and Ultra96 (Zynq UltraScale+ family).

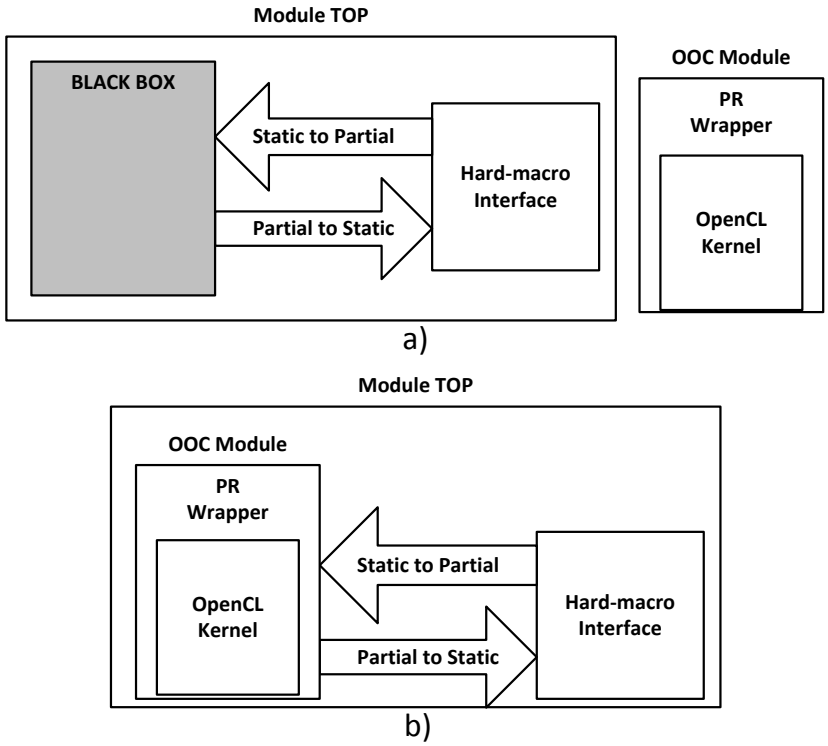


Figure 4.10: a) is hardware module synthesis, and b) is hardware module implementation.

4.3.1 Overview of Module (Role) Design

The module design starts with either high level language (HLL) source codes (C/C++, OpenCL) or hardware descriptions (RTL/netlist). If HLL source codes are used, they will go through High Level Synthesis (HLS) step [CLN+11, CCA+13] to generate the RTL source codes. One further step of Design Space Exploration (DSE) might be conducted based on the allocated resources to the module to optimise the module throughput as presented in [MMRL17]. This optimisation step will generate different versions of the same kernel (i.e. design alternatives with different resource-performance trade-offs) by choosing the appropriate `pragma` options in the Vivado HLS tool. The resulted RTL source codes will be out-of-context (OOC) synthesised. This process is well established by various industry tools [Xil14, Alt10], and the contribution of this PhD project is to integrate them in an automated design flow for building partially reconfigurable systems to offer users a software-centric compilation experience for the first time.

The PR Wrapper templates are used to create a minimal top-level placeholder for the module implementation. This temporary placeholder acts as sink/source connection points and substitute the surrounding static system. The module OOC netlists are integrated to this placeholder for synthesis and placement stages.

Blockers will be generated based on TCL routing constraints to oblige all partial module's

primitives and routing resources according to strict implementation rules as mentioned in Section 4.1. These blockers will be placed around the selected area, therefore, acting as a wall to implement hard module bounding box constraints. Routing tunnels are included for the communication to and from the temporary placeholder. The position of these tunnels match exactly the tunnels as used in the static design to implement the communication between static and partial areas.

As a module is implemented in a separated design from the static system, the final result generated by Vivado is a *full configuration bitstream*. This bitstream is passed on to BitMan that cuts out the configuration data that corresponds to the module only. BitMan can also include header information starting metadata such as maximum clock frequency and the exact resources used. We repeat these steps for all modules to compose a hardware module library. At run-time, BitMan manipulates those partial bitstreams to relocate modules to a desired position inside a partial region of the static system.

4.3.2 Module Synthesis

After selecting the minimum number of resource slots according to available primitives per slot as in Table 4.2, the RTL code (produced by Vivado HLS or manually instantiated by the user) is compiled all the way to the final partially reconfigurable bitstream without any manual intervention. To allow a module to be integrated into the static system, the accelerator modules have to be implemented with exactly the same physical interface as being used by the static system. This correspondence of static system and partial module interface is the key for encapsulating the partial module compilation. The implementation of this flow for the module is accomplished by a TCL script that synthesises the hardware accelerator as an Out-Of-Context (OOC) module, i.e. a module without external connection pins, synthesised as a standalone module.

In practice, to keep user interaction to a minimum, we intend to provide compilation templates for different sized modules (e.g., a 1-slot, 2-slot, 3-slot version). It is believed in this thesis that only a rather small number of slots will be available in the here assumed reconfigurable systems. This is contrary to previous approaches such as ReCoBus-Builder [KBT08] that aimed to accommodate many tens of modules with fine adjustable bounding boxes. However, with the progressing towards HLS and DSE, the problem of internal fragmentation of a resource slot (described in a previous research in [Koc12]) is actually solved by a HLS tool [Xil19b] that can optimise performance for a given resource target.

Before synthesising the OOC module, an accelerator TOP design is synthesised containing only a black box (a module containing only the description of its I/O signals, with no information about its function) to host the OOC module and the interface between the module and the whole system. This approach detaches the OOC module from the TOP design, allowing the

Table 4.2: Available resources for 1 slot of on the ZCU102 platform and the UltraZed & Ultra96 platforms. The version on ZCU102 has 4 slots in total while the other platforms provide 3 slots in total.

Resources on ZCU102	Number of resources (1 slot)	Slot utilisation (%)	Total utilisation (%)
CLB LUTs	32640	11.70	46.80
CLB Regs.	65280	11.90	47.60
BRAMs	108	12.10	48.40
DSPs	336	13.30	53.20
Resources on UltraZed & Ultra96	Number of resources (1 slot)	Slot utilisation (%)	Total utilisation (%)
CLB LUTs	17760	25.17	75.51
CLB Regs.	35520	25.17	75.51
BRAMs	60	27.78	83.33
DSPs	96	26.67	80

utilisation of the RTL folder generated by Vivado HLS with no manual intervention. The OOC is then integrated by the synthesis tool into standardised PR wrappers and to the hierarchically higher level of the hardware accelerator design, as shown in Figure 4.10a. The resulting design checkpoint (a proprietary file from Xilinx), containing the module kernel connected to the interface, is saved to be used by the implementation phase. As we use the Xilinx Vivado toolchain in this step, there is no extra overhead compared to the default vendor synthesis. Note that all these tools specific details are hidden from the user by Makefile-like compilation scripts for providing a user experience similar to software compilation.

4.3.3 Module Implementation

The first step of the hardware module implementation is merging the design checkpoint containing the accelerator top-level design to the OOC module containing the module synthesised core, as shown in Figure 4.10b. After that, a TCL script, which uses the fixed bounding box coordinates according to how many slots are chosen, is executed to:

- define the partially reconfigurable area for the OOC positioning;
- translates the geometric bounding box information to be used by GoAhead [BKT12] or TedTCL library [Ves18];
- generates the coordinates used by BitMan [PHK17] to extract the partial region used by the accelerator module from the full bitstream at the end of the Vivado implementation and bitstream generation.

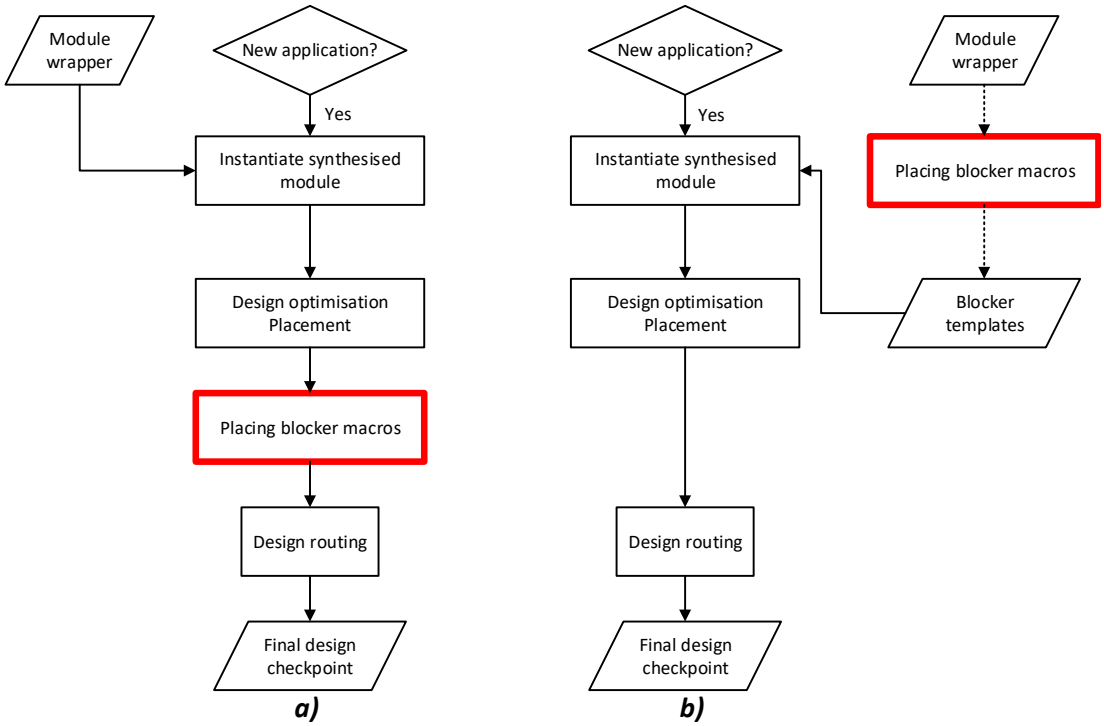


Figure 4.11: Physical module implementation flow; (a) is the conventional implementation flow, versus (b) is using blocker checkpointing.

Table 4.3: The overheads of two implementation options on the ZCU102 and UltraZed/Ultra96 platforms. Option 1 is to insert blockers through TCL scripts, as shown in Figure 4.11a). Option 2 is to insert blockers through design checkpoints, as shown in Figure 4.11b).

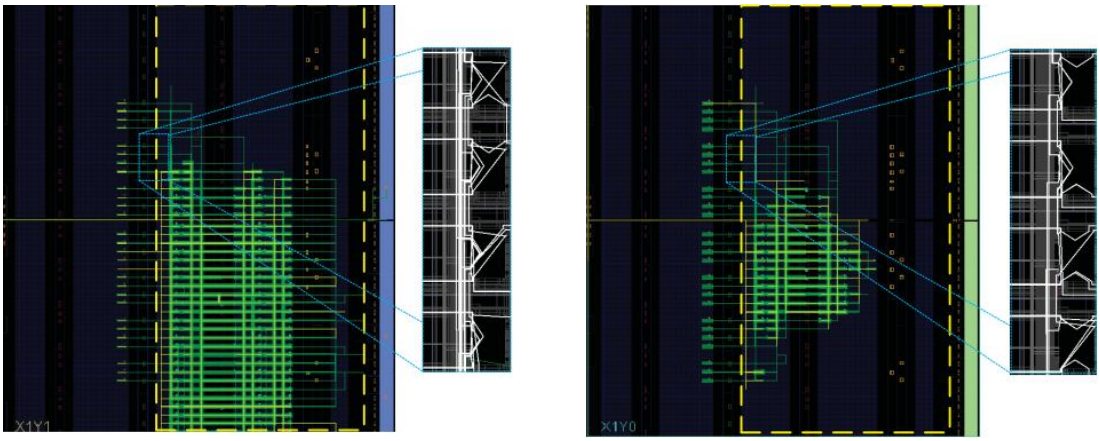
No. of slots on ZCU102	Module	Option 1 t_1 (seconds)	Option 2 t_2 (seconds)	Speed-up
1	SPMV	622	379	1.64×
2	Sobel Filter	1788	1076	1.66×
4	MM	3151	1476	2.13×
No. of slots on Ultra96	**Module**	**Option 1 t_1 (seconds)**	**Option 2 t_2 (seconds)**	**Speed-up**
1	SPMV	624	357	1.75×
2	Sobel Filter	1323	658	2.01×
3	DCT	1863	724	2.57×

The next step is the placement and routing of the interface. This is a critical step due to the fact that the interface wires must match exactly the same relative physical positions of the interface defined in the static design. Otherwise, the partial bitstream will not function because communication with the AXI interconnect cannot be carried out, or the whole system may even freeze. The implementation script is used to route the interface sets an attribute to keep the routing in fixed positions, even when the tool starts routing the remainder of the design. Moreover, the horizontally 12-long wires are pre-occupied for I/O virtualisation if necessary, as explained in Section 4.2.3.

GoAhead/TedTCL is then executed to generate blockers around the partially reconfigurable area to prevent any signals from the hardware accelerator to use wires outside the partial area. The only exceptions to this rule are the pre-routed interface wires and the pre-occupied long wires for I/O virtualisation. This step in module implementation is almost identical to the step in static shell implementation mentioned in Section 4.2.1. The differences are that in this step, we are blocking routing of the reconfigurable module (role) to go out of the defined bounding box rather than blocking static routing to violate the reconfigurable part as in the static shell implementation.

However, we have realised that the TCL scripts of blocker hard-macros generated by GoAhead/TedTCL take a subtantial amount of time to be fully placed into a Vivado design, so it adds an extra overhead into the already lengthy implementation process. Even worse, this overhead is repeated when we need to compile a new module. Therefore, we have improved the proposed implementation process by placing the blocker macros into the empty module wrapper to create a fence around the area given to the module. These pre-generated blocker macros are provided for modules in the form of design checkpoints and automatically added by our compilation scripts, as shown in Figure 4.11b. Hence, the revised implementation process has improved significantly, as experiments taken on a Windows 7 machine with Intel Core i7-4930K CPU at 3.4GHz, 64GB RAM and 512GB SSD have shown in Table 4.3. This improvement has boosted up an implementation by up to 2.57$\times$ than the conventional one. Moreover, the bigger the module is, the more we save.

Ultimately, a module can be built independently from other modules and from shell, and then can be deployed on top of shell at run-time in a hot plug-and-play manner by exploiting the partial reconfiguration feature on FPGA devices. In contrast, state-of-the-art FPGA-virtualised approaches [YB18, AGV+17, WBP13, WPAH16, BSB+14, KLP+18, CSZ+14] are still suffering in lengthy module compilation overhead. For example, a recent approach may take several weeks to compile a set of 30 applications [KLP+18]. Therefore, the decoupled module compilation boosts up design productivity dramatically in an analogous way of the software compilation approach which compiles software applications independently from each other. Moreover, as long as the interface between shell and the slot as well as the slot resource footprints remain, shell is free to change without affecting the built modules.



"*grep*" command – partial module

"*translate*" command – partial module

Figure 4.12: Physical implementation designs of "grep" and "translate" Linux bash commands as partial modules. All modules were implemented on the Zybo platform with the decoupled compilation flow [HSPK17].

4.3.4 Hardware Module Library

The decoupled compilation flow has been verified with different designs in various domains on multiple platforms with several FPGA families ranging from the Xilinx Zynq-7000 SoCs (on Zybo and ZedBoard platforms) to the Zynq UltraScale+ MPSoCs (on ZCU102, UltraZed and Ultra96 boards). For example, Figure 4.12 show the implementation of partially reconfigurable modules with the *grep* and *translate* Linux bash commands on the Zybo platform.

Figure 4.13 shows implemented designs of OpenCL applications: Discrete Cosine Transform (DCT), Finite Impulse Response (FIR) filter, 3D Normal Estimation, Sparse Matrix-Vector Multiplication (SPMV), Histogram and Sobel filter applications from the Spector benchmark [GAPK16] as well as Black Scholes application from an academic project [MMRL17]. Moreover, Matrix Multiplication was further optimised to achieve better throughput. Note that the Sobel filter is a 2-slot version and that Matrix Multiplication has 1-slot and 4-slot variants, while DCT has 2-slot and 4-slot variants and other applications have 1-slot version only. This optimisation is done by using design space exploration (DSE) and HLS `pragma` keywords to generate the corresponding RTL versions of an application.

Figure 4.14 shows designs including signal processing (DCT and FIR filter), data analytics (Histogram and Black Scholes), machine learning (SPMV), security (AES and SHA-3), and arithmetic (VADD, Mandelbrot set, and MM) on the UltraZed and Ultra96 platforms. Note that the SHA-3 source code is in RTL [PHK+18], the Mandelbrot set is written in C, while the remaining are provided in OpenCL from the academic benchmarks [GAPK16, MMRL17].

Table 4.4: The compilation flow's bitstream XML keyword description.

XML keywords	Descriptions
$\langle hw \rangle \langle /hw \rangle$	Name of the hardware module
$\langle target \rangle \langle /target \rangle$	FPGA device on which the hardware module will be placed
$\langle resource\_string \rangle \langle /resource\_string \rangle$	*Resource String* for resource footprint verification
$\langle size \rangle \langle /size \rangle$	Number of slots are occupied by the hardware module
$\langle n\_args \rangle \langle /n\_args \rangle$	Number of hardware module registers
$\langle offset\_address\_X \rangle \langle /offset\_address\_X \rangle$	Address Offset of register X
$\langle clk\_freq \rangle \langle /clk\_freq \rangle$	Maximum clock frequency of the hardware module
$\langle length \rangle \langle /length \rangle$	Length of the bitstream in 32-bit words
$\langle bitstream \rangle \langle /bitstream \rangle$	Bitstream of the hardware module

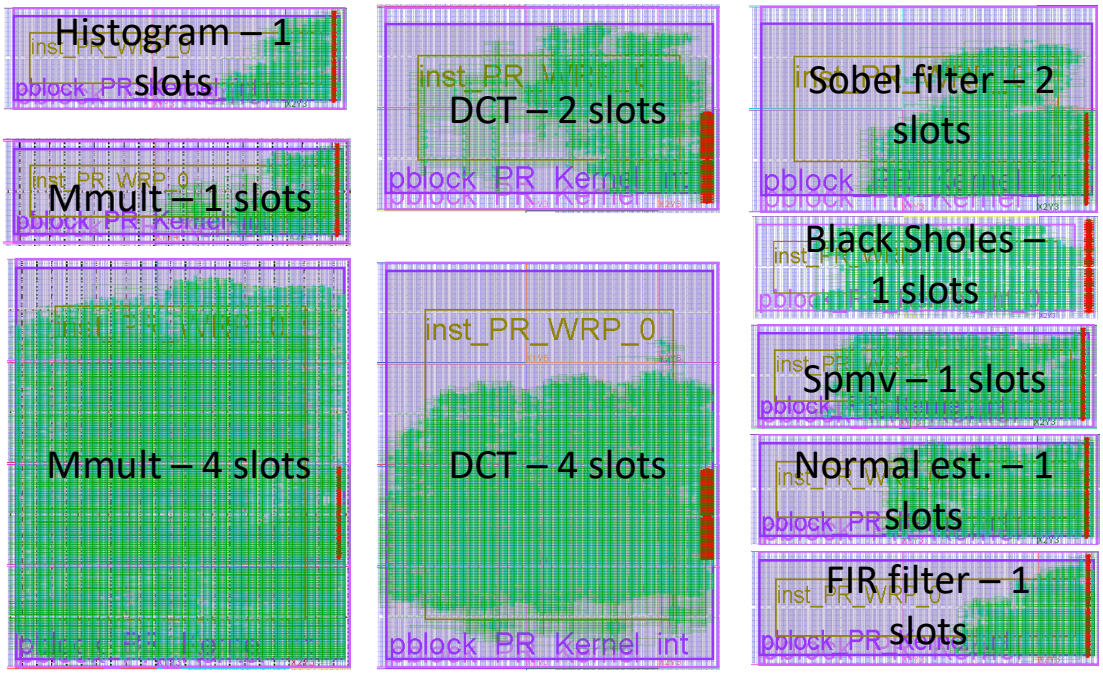


Figure 4.13: Physical implementation designs of Matrix Multiplication, Discrete Cosine Transform (DCT), Finite Impulse Response (FIR) filter, 3D Normal Estimation, Sparse Matrix-Vector Multiplication (SPMV), Histogram, Sobel filter, and Black Scholes. All modules were implemented on the ZCU102 platform with the decoupled compilation flow [PVVK18].

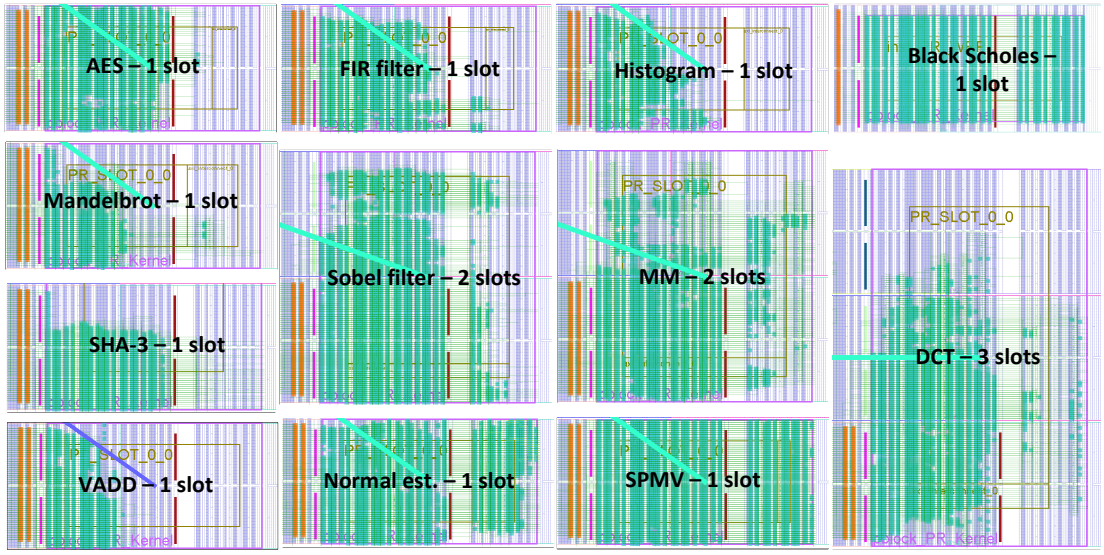


Figure 4.14: 11 designs, which are written various design languages such as RTL, OpenCL, and C, are physically implemented by the decoupled compilation flow on UltraZed and Ultra96 boards [PPV+19].

As a distinct feature of the proposed methodology (i.e. I/O virtualisation as described in Section 4.2.3), both UltraZed and Ultra96 shells can host exactly the same partial bitstream, hence allowing cross-deploying of accelerator modules without re-compilation of the same module for different platforms, which is common in state-of-the-art approaches [Xil18c].

The full bitstream generated during the module implementation phase is used by BitMan to cut-out a region containing the implemented kernel along with its interface. The hardware module will get its partial bitstream wrapped in an XML file which contains necessary meta information such as the accelerator name, target device, and resource string, as described in Table 4.4. The module name in the XML file is managed by the hardware task scheduler (see Section 5.3). This XML file generation is automatically done by this decoupled compilation flow and an example containing metadata for run-time resource management and the module partial bitstream is given in Listing 4.1. Finally, a collection of XML files for various hardware modules comprise a hardware module library.

To evaluate the efficiency of this decoupled compilation flow, we have compared against the Xilinx PR flow [Xil18c]. As the Xilinx PR flow requires compiling both static and accelerator designs together, performing place and route (P&R), and generating a dedicated bitstream for *each* slot, it introduces additional latency. In our decoupled compilation flow, we first generate a *full* bitstream for the compiled accelerator, and then, a *relocatable* partial bitstream for *all* slots using BitMan. To quantify this latency, we used accelerators with increasing resource utilisation: AES, Normal Estimation (Normal Est.), and Black Scholes. The utilisation of each is reported in Table 4.5 for ZCU102 and Ultra96 platforms. Implementation of the Black

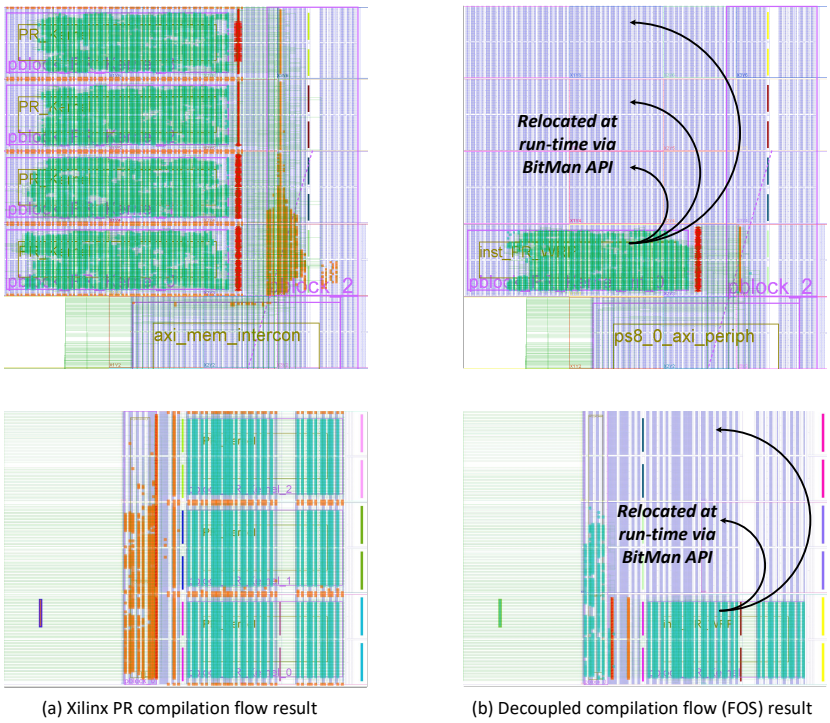


Figure 4.15: Implemented results of the Black Scholes accelerator [MMRL17] on ZCU102 (top) and Ultra96 (bottom) for the Xilinx PR flow and the decoupled compilation flow.

Scholes accelerator is shown as an example in Figure 4.15 with comparisons with the designs generated by the Xilinx PR flow. As the decoupled compilation flow is integrated to a novel FPGA Operation System (FOS) [VPPK20], we will call the flow FOS for short in this Figure and the below comparison table. Note that FOS uses BitMan [PHK17] to generate a relocatable bitstream, which is copied to the other PR slots at run-time.

Table 4.5 shows the latency breakdown for place and route (P&R) and bitstream generation for both the Xilinx PR flow and the decoupled compilation flow on ZCU102 and Ultra96. The results show that the average P&R latency per region is higher for the decoupled compilation flow as it adds additional constraints for supporting relocatability, as presented in this chapter. However, when compiling for multiple regions (i.e. four for ZCU102 platform and three for Ultra96 platform), our flow outperforms the Xilinx PR flow (by up to 2.34×) by duplicating a single slot accelerator. Especially, the latency of bitstream generation from our flow is up to 3× faster than from the Xilinx PR flow, as the latter will generate *all* partial bitstreams for all slots. Overall, when the number of slots increases, the latency of the decoupled compilation flow stays constant, whereas the latency of the Xilinx PR flow increases linearly. This relationship turns into an exponential increase in compilation time when compiling several applications with the state-of-the-art Xilinx PR flow, as it needs to compile *each module for every slot*.

4.4 Chapter Summary

In this chapter, the design and implementation methodology to build up FPGA-virtualised systems, including FPGA shells and the hardware module libraries deployed on top of shells, has been introduced. Shell and its corresponding hardware module libraries are built independently, so they can be modified with limited side effects to the whole system thanks to the decoupled compilation flow. The hardware module library consists of a number of hardware modules compiled from the HLS source codes to the partially reconfigurable module by a novel module compilation path, a process that originally required very specific FPGA domain knowledge. Moreover, the decoupled compilation circumvents the lengthy module implementation overhead, which is a major obstacle of state-of-the-art FPGA-virtualised approaches. Various FPGA-virtualised case studies have been implemented and deployed on top of multiple FPGA platforms featuring different FPGA device families by utilising the proposed design methodology. The following Chapter 5 will discuss the system run-time management, and Chapter 6 evaluate the resulting systems against targeting goals and requirements of FPGA virtualisation.

Note that the proposed methodology has also been applied to introduce an extended custom design flow which enables the partially reconfigurable capability for the Xilinx Isolation Design Flow on Xilinx 7-Series FPGAs. This flow combined for the first time partial reconfiguration capabilities of FPGAs with the isolation requirements [NIS01] for building secure systems on FPGAs (see Appendix A for details).

Table 4.5: Place and route (P&R), bitstream generation, and total latency for AES, Normal Estimation (Normal Est.) [GAPK16], and Black Scholes [MMRL17] accelerators using the Xilinx PR flow and the decoupled compilation flow, called FOS in the table for short as the flow is an integral part of FOS (FPGA Operation System [VPPK20]).
Note that the accelerators are compiled for all *four* slots on ZCU102 and all *three* slots on Ultra96 using the Xilinx PR flow; whereas the accelerators are compiled for *one* slot on ZCU102 and *one* slot on Ultra96 using the decoupled compilation flow (see Figure 4.15).
The evaluation is conducted using Vivado 2018.2.1 on an Intel core i7-4930K CPU running at 3.4 GHz with 64 GB of RAM and 512GB SSD.

Applications	Slot Utilisation (on ZCU102)	P&R (seconds)			Bitgen (seconds)			Total (seconds)		
		Xilinx	FOS	Speed-up	Xilinx	FOS	Speed-up	Xilinx	FOS	Speed-up
AES	16%	978.45	700.70	1.39×	403.81	157.15	2.57×	1382.26	857.85	1.61×
Normal Est.	33%	1421.50	822.59	1.73×	438.85	163.09	2.69×	1860.35	985.68	1.89×
Black Scholes	48%	1826.11	936.07	1.95×	483.17	168.15	2.87×	2309.28	1104.22	2.09×
Applications	Slot Utilisation (on Ultra96)	P&R (seconds)			Bitgen (seconds)			Total (seconds)		
		Xilinx	FOS	Speed-up	Xilinx	FOS	Speed-up	Xilinx	FOS	Speed-up
AES	33%	429.40	284.18	1.51×	176.19	64.06	2.75×	605.59	348.24	1.74×
Normal Est.	63%	747.75	387.41	1.93×	201.21	70.09	2.87×	948.96	457.50	2.07×
Black Scholes	81%	1296.26	574.56	2.26×	231.27	77.11	2.99×	1527.53	651.67	2.34×

Listing 4.1: An XML snippet of a 2-slot DCT module targeting the ZCU102 featuring the XCZU9EG FPGA. Metadata in the header part will be used for run-time resource management.

```
<?xml version="1.0" encoding="utf-8"?>
<data>
  <header>
    <hw> DCT </hw>
    <target> XCZU9EG </target>
    <resource_string>
        ImsLMsLRsLMsDMsLMsLRsLMsDMsLMsDMsLMsLRsLMsDMsLMsLRsLMsDMsLMsDMsLMsL
    </resource_string>
    <size> 2 </size>
    <n_args> 8 </n_args>
    <offset_address_0> 0x00 </offset_address_0>
    <offset_address_1> 0x28 </offset_address_1>
    <offset_address_2> 0x30 </offset_address_2>
    <offset_address_3> 0x38 </offset_address_3>
    <offset_address_4> 0x40 </offset_address_4>
    <offset_address_5> 0x44 </offset_address_5>
    <offset_address_6> 0x4c </offset_address_6>
    <offset_address_7> 0x50 </offset_address_7>
    <clk_freq> 100 <clk_freq>
  </header>
  <body>
    <length> 144586 </length>
    <bitstream>
        FFFFFFFF FFFFFFFF 000000BB 11220044
        FFFFFFFF AA995566 20000000 20000000
        30002001 000C0000 30008001 00000001
        20000000 20000000 30004000 5006C24F
        ...
        30008001 00000005 30002001 07FC0000
        30008001 00000007 30008001 0000000D
        20000000 20000000 20000000 20000000
    </bitstream>
  </body>
</data>
```

Chapter 5

Run-time Management

A ship in harbour is safe, but that is not what ships are built for.

John A. Shedd

This chapter presents the run-time management what mainly provides the *deployment abstraction* such that the modules compiled by the proposed design methodology in Chapter 4 can be deployed on top of a shell (also designed by the same methodology as described in Section 4.2). The run-time management consists of a configuration controller, a hardware task scheduler, a module device driver, a memory management to orchestrate the user applications and the available resources of heterogeneous CPU+FPGA computing systems dynamically and transparently from user's point of view. The configuration controller is an original contribution of this PhD thesis as an application of BitMan API, while other components are provided from other MPhil and PhD students (K. Paraskevas and A. Vaishnav) but reproduced here for the sake of completeness.

5.1 Workflow Overview

As described in Chapter 4, a library of XML files containing the partial bitstreams of hardware modules is compiled from HLS source code by using the proposed compilation flow. The metadata in the final XML files (that replace the original Xilinx configuration bitstreams in the `.bit/bin` format, as shown in Listing 4.1) are used by the run-time execution to manage the deployment of each hardware module by a user application (e.g., the mapping of control registers that will be used by the HLS kernel or the potentially fastest clock frequency).

When the hardware task scheduler receives a function call from the user application, it looks for the hardware module in the hardware library and requests a module placement to the configuration controller. If the module placement is successful, the scheduler dispatches

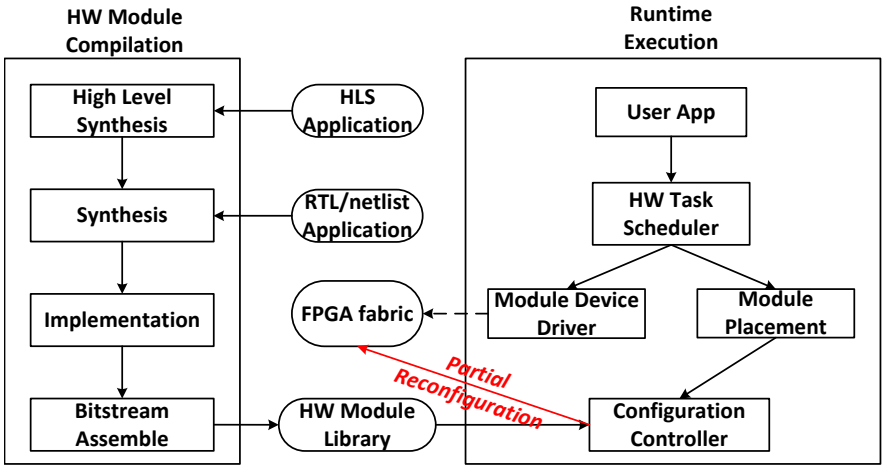


Figure 5.1: Hardware (HW) module compilation steps and the run-time execution environment for hardware modules.

acceleration tasks to the hardware module via the module device driver. Otherwise, the fail-safe execution scheme to run the application task on CPUs will be deployed. Note that any components in the run-time execution can be changed for higher efficiency or additional functionalities by users, if necessary. The roles of module compilation and the run-time execution phases are summarised in Figure 5.1 while the components of the run-time management is illustrated in Figure 5.2.

5.2 Configuration Controller

5.2.1 Overview

The configuration controller is in charge of the physical hardware module placement on the FPGA fabric at run-time. It uses a subset of the functionality that is available in the BitMan tool and API [PHK17]. The configuration controller receives requests from the hardware task scheduler and sends commands to the configuration port driver as well as to the bus traffic control drivers to place an accelerator kernel to the system without interfering with the rest of the AXI buses. This is a sensitive operation and therefore automated, because without bus decoupling a module reconfiguration could potentially corrupt the bus protocol or interfere with other running bus transactions, and hence, freeze the entire system including the ARM SoC. In our FPGA-virtualised systems, either the Processor Configuration Access Port (PCAP) or the Internal Configuration Access Port (ICAP) can be utilised to write configuration data to the FPGA fabric. In the case of using PCAP for reconfiguration, the software implementation is more trivial as we can exploit the built-in Xilinx FPGA Manager [Xile] to deliver bitstreams to the FPGA fabric, whereas we need a custom kernel driver in the case of ICAP. Bus traffic

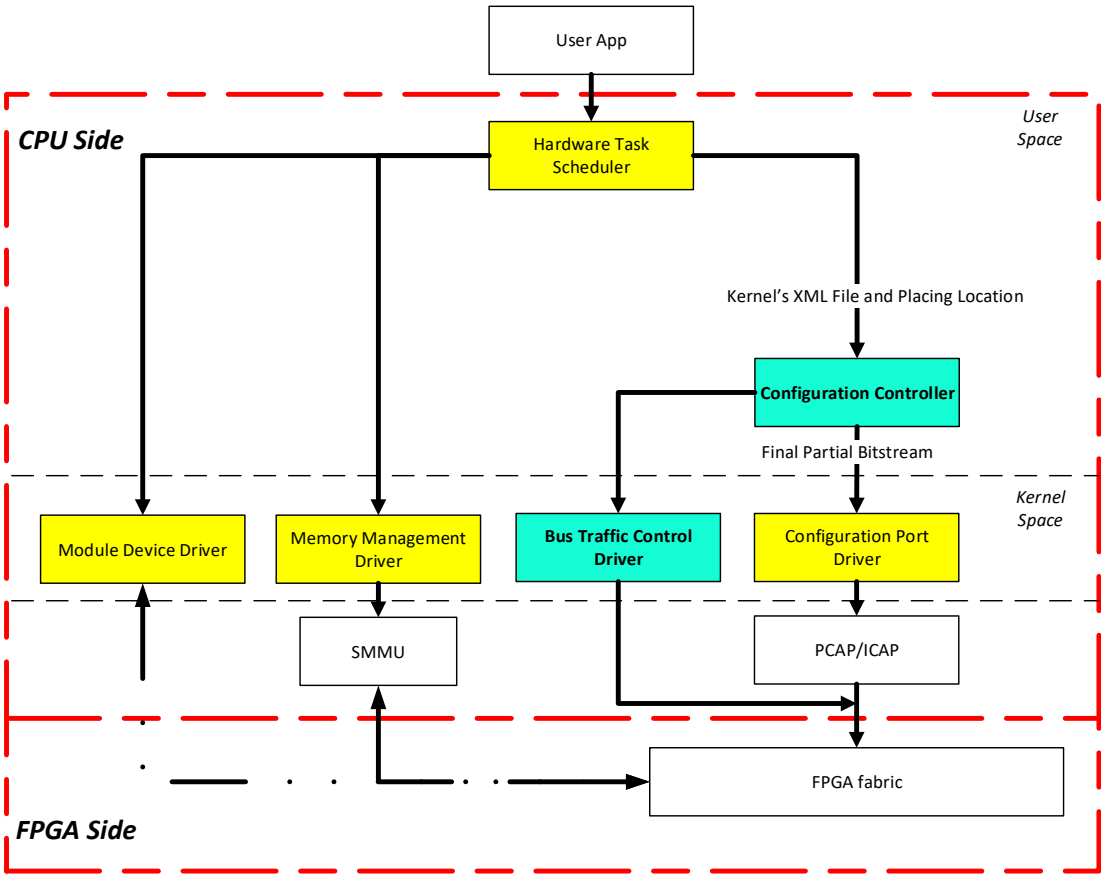


Figure 5.2: Run-time execution and management. Components in the blue colour are contributions of this PhD project.

control is implemented by means of Xilinx PR Decoupler IP core [Xil16b] which essentially multiplexes/de-multiplexes AXI bus signals when requested. The role of the configuration controller and the bus traffic controller/decoupler are illustrated in Figure 5.2.

5.2.2 Hardware Module Placement Process

When the configuration controller receives a request from the hardware task scheduler with an XML file name and a placement position as arguments, the configuration controller will extract the hardware module information such as how many slots will be needed. With this information, the configuration controller will check if there are enough available PR slots to host the hardware module. The next step is to verify if the resource footprint of the hardware module matches the footprint of the PR slot. This checks if the relative layout of primitive columns (slices, BRAMs, and DSPs) correspond to the resources found at the target position on the FPGA fabric.

As the present system uses un-encrypted, no-CRC partial bitstreams for module relocation,

there is a chance that a user may try to corrupt configuration data in the non-assigned regions, such as shell or other user running roles, with malicious data. This can be done by deliberately providing either a partial bitstream with more configuration data than necessary for a slot or reports an incorrect number of slots in the metadata. For example, in the metadata, the module informs it has only one slot, but the attached partial bitstream actually contains configuration data for two slots. However, as the bounding boxes of slots in a shell are pre-defined, the valid number of configuration data for each slot is known. Therefore, the configuration controller parses the partial bitstream to check if the actual amount of configuration data is valid as well as if the actual number of slots shows a mismatch within the metadata.

Another important step automatically done by the configuration controller is to update the bitstream of a partially reconfigurable module if there is any relocation needed. As mentioned earlier, a hardware module can be implemented in any slot (for example **Slot 1**) and be placed in the same slot or any other slot (for example **Slot 2**). In the case of placing to a different slot, we automatically perform module relocation. Placing a module incorporates two manipulations that will be done through the BitMan API:

1. updating address information inside the bitstream according to the target position on the FPGA;

2. modifying the clock configuration data of the module according to the targeting location[1].

Before loading the configuration data onto the FPGA fabric, the communication between the targeting slot(s) and the shell needs to be disabled to prevent any unexpected interference with the rest of the system bus, which may corrupt other bus transactions and/or freeze the whole system. This communication disabling can be done via the bus traffic controller/decoupler and orchestrated by the configuration controller. After that, the configuration data is loaded to the targeting slot, and then, the communication is enabled again. Finally, the status of all slots in the FPGA is updated for the purpose of resource management. The whole module placement flow is summarised in Figure 5.3.

5.2.3 Configuration Overhead

ICAP theoretical throughput is up to 800MB/s and PCAP throughput is measured at 256MB/s, while the software overhead for the scheduler to call the configuration controller and the configuration controller process is measured at 13ms on average.

[1]Unless module relocation is performed, the configuration controller is using a feature provided in BitMan that prevents the clock configuration data from being changed. This reflects the requirements of complex real-world systems that use a plurality of clock networks on the FPGA (e.g. for various peripheral controllers). With this, each clock region or each slot on shell can have its own clock configuration while still allowing to place modules freely.

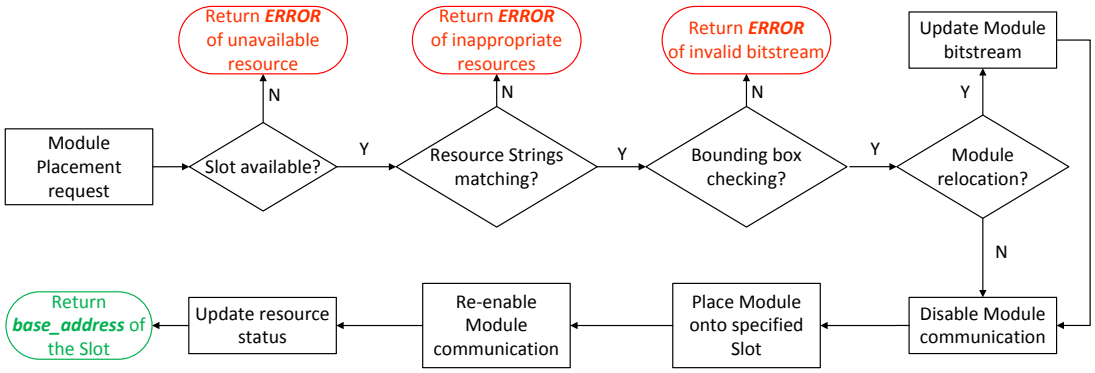


Figure 5.3: The configuration controller's hardware module placement process.

5.3 Hardware Task Scheduler

5.3.1 Overview

The hardware task scheduler is responsible for managing accelerators on the FPGA-virtualised systems and is provided as a daemon running under PetaLinux (the Xilinx distribution of Linux kernel). An application user can send acceleration jobs to that daemon in the same way (using an identical API) as used for Xilinx SDSoC systems. The scheduling of kernels is performed using a first-come-first-serve (round-robin, RR) scheduler as the default implementation, which internally maintains the state of the FPGA occupancy along with a waiting queue of the kernels and their status. The default scheduler dynamically selects the accelerators based on the number of slots available at run-time. This feature allows the scheduler to maximise the utilisation of the FPGA and improves the execution time of kernels based on run-time conditions, as evaluated in Section 6.1.

Moreover, thanks to the aforementioned abstractions provided by this PhD project, other advanced scheduling policies such as resource elastic scheduling (RES) [VPKG18] and, especially, heterogeneous resource elastic scheduling (HRES) [VPK19] are enabled on our FPGA-virtualised systems. With *resource elasticity*, we refer to *the ability of a task to change its resource allocation dynamically and transparently from the user*, while the actual resource allocation is decided by the scheduler. When combined with a data-parallel threaded programming model (for example, the OpenCL programming model [KHR]), the scheduler is able to take into account the resources available (FPGA resources only in the case of RES, or all the compute resources including CPUs and FPGAs in the case of HRES), the accelerators available in the module library and estimated remaining execution times, as summarised in Figure 5.4. The latter one ensures that reconfiguration is only performed if it pays off due to a possible speed-up of the reaming jobs available in the system. Overall dynamic resource allocation allows scheduling of accelerators in both the *time domain* and the *spatial domain* to maximise

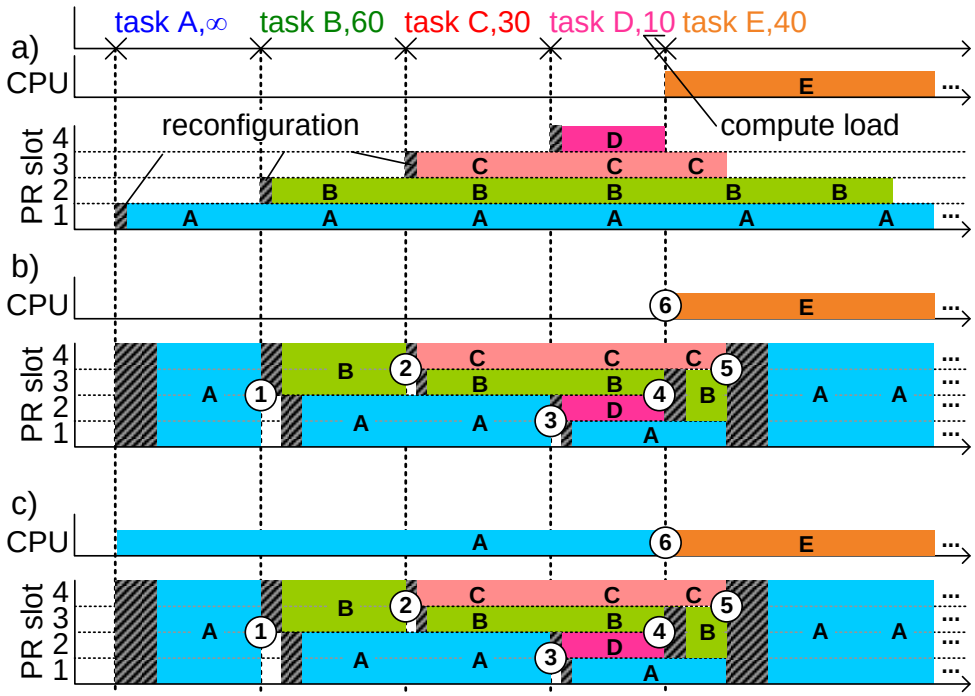


Figure 5.4: Resource allocation for tasks (A-E) in time when using a) run-to-completion scheduling, b) resource-elastic scheduling (RES) [VPKG18], and c) heterogeneous resource-elastic scheduling (HRES) [VPK19] on a CPU+FPGA architecture. The circled events highlight cases where resources accommodate newly arriving tasks (1, 2, 3, 6) or cases where tasks complete (4, 5).

the utilisation and system performance, as evaluated in Section 6.2.

This hardware task scheduler and its corresponding scheduling policies are the work of A. Vaishnav and are briefly reproduced here to show the benefit of the system implementation.

5.3.2 Scheduling Overhead

The overhead of the Round Robin scheduler is 1μs while the HRES takes 2.9μs on the UltraZed board when using a Quad-core ARM Cortex-A53 CPU at 1.5GHz and 2GB of DDR4 memory [PPV+19]. This time needed for taking a scheduling decision is much less than the configuration time which is in range of milliseconds for a slot on our targeting platforms.

5.4 Module Device Driver

Interaction with the accelerator is performed using a module device driver which provides APIs for accessing the accelerator status and issuing threads for execution given the memory address

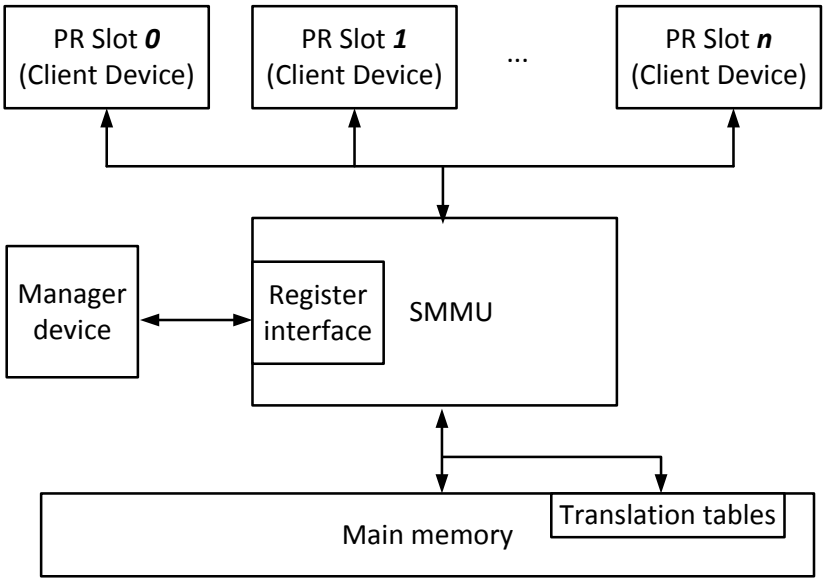


Figure 5.5: The implementation of the ARM SMMU in the memory system. Client devices in the PR slots are connected through the memory interconnect to the SMMU in the upstream bus. The connection between the SMMU and the rest of the memory system is the downstream bus. When the SMMU is set, the client devices agnostically issue transaction requests to the SMMU. After a successful translation, the SMMU performs the memory access and returns a valid response, or faults otherwise.

where the kernel is mapped. The driver usage is hidden from the user when using the default scheduler but if the user wishes to implement their own schedulers or gain more direct control of the kernel execution, it can also be accessed from the user code using standard C `#include` functionality rather than installation of driver in the operating system.

This module device driver is implemented by A. Vaishnav. However, a contribution of this project is integrating the module device driver in the run-time management.

5.5 Memory Isolation/Management

5.5.1 Overview

As the FPGA interface is capable of addressing the entire memory space of a system, the memory isolation should be provided in a multi-tenancy computing environment and be able to guarantee non-interference between virtual address spaces of user applications. Access control has been an important aspect for FPGAs that are to be deployed in cloud systems

for security and management purposes and is often addressed with various ring level privileges for stakeholders provided by Virtual Machine Monitors (VMMs) and run-time systems [WBP13, WPAH16, BSB+14], and hence, we embrace the technique in our FPGA-virtualised systems, as shown in Figure 5.5.

Memory isolation is achieved by utilising the System Memory Management Unit (SMMU) IP core which is an industrial standard of the Input-Output Memory Management Unit (IOMMU) developed by ARM [ARMb] and available in the hardened CPU core (i.e. Processing Element or PS for short in Zynq UltraScale+ terminology). As all memory references pass through that SMMU core, it provides a common view on the memory to all system components and takes care of all memory management issues including caching, translation of virtual memory addresses to physical addresses as well as providing memory protection and isolation when configured. If left unconfigured, no checks are performed and the SMMU is essentially bypassed. To use the SMMU, we have built and integrated the following memory management framework onto the run-time execution and management.

This memory management work is contributed by K. Paraskevas and briefly reproduced here for the sake of completeness. The memory management work is also integrated into the here presented run-time management.

5.5.2 Memory Management Framework for FPGA:

The framework consists of two components: the *kernel driver* and the *user-space library*. The kernel driver can be called by the wider application flow to allocate memory resources to the application, virtual machine, or container that the accelerators can access, while the user-space library provides a user-friendly pre-processed interface.

Kernel Driver: The kernel driver is responsible for:

- Associating new Stream IDs or invalidating existing entries in the Stream Match Registers (SMR) registers of the SMMU.
- Configuring the Page Table pointer of a particular SMMU context to point to the user-space application page table.
- Setting the appropriate cacheability attributes for the user page table.
- Flushing the Page Table Entries from the CPU cache, due to lack of memory coherence support.

Usually during the SoC design process, IOMMU implementations such as the SMMU on the Zynq UltraScale+ MPSoC utilise a non-coherent Page Table Walker (PTW). This decision

Table 5.1: The overhead of using the SMMU. In the first iteration of a DMA transfer, DMA read was completed in 90 cycles (900 ns) on average, where all following iterations took 20 cycles (200 ns) to complete. When SMMU is not used, the completion times are identical, but the anomaly of the first iteration is absent. (This Table is adopted from [PPV+19]).

	With SMMU (ns)	Without translation (ns)
First iteration	∼900	∼200
Next iterations	∼200	∼200

is usually taken to save resources on the die as the coherency mechanisms require additional complexity in the cache to implement a coherent interconnect. In addition, memory regions accessed via the IOMMU are usually static and long-lived in (e.g., kernel allocated ring buffers for devices). Additionally, memory regions committed to the accelerators are usually non-cacheable, and therefore, coherency of the PTW is deemed unnecessary. This poses a challenge when providing accelerator access to user-space memory where allocations can be dynamic during the life cycle of the application. To overcome this we provide the user-space with an API for page table management, such as flushing.

User-space library: The user-space library binds to the endpoint created by the driver and wraps a number of system calls to provide an abstraction to the user to:

- Create handles that are used to associate the user's page table and accelerators with the SMMU.
- Allocate memory and pin it to RAM so it can be used by user accelerators.
- Free and de-associate user-space memory and accelerators.

5.5.3 Memory Management Overhead

The total overhead of the procedure of registering an application to the SMMU by using the driver is measured at 1.3ms, where the driver function of page table flushing takes up the significant portion of 1.26ms. Given the fact that page table flushing takes place only once before any translation occurs, we consider that this is outside the critical path of the overhead.

The overhead of using the SMMU was determined in two experiments by measuring the number of clock cycles in the FPGA logic for the read part of a DMA transfer. In the first experiment, the DMA uses virtual addresses and the SMMU is configured to translate, where in the second run, the SMMU was set to bypass and physical DMA addresses were used. The results that are shown in Table 5.1. The increased latency of the first iteration (“cold miss")

can be attributed to the initial page table miss from the TLBs of the SMMU and the subsequent fetching operation by the PTW.

5.6 Chapter Summary

This chapter introduces run-time management, the last component of FPGA-virtualised systems, which is in charge of abstracting the deployment of applications. With this, modules developed by the proposed methodology in Chapter 4 can be deployed in various execution scenarios on top of the resulting shells to maximise the resource utilisation and the system throughput. Chapter 6 provides experiments showing the benefits of *infrastructure abstraction*, *development abstraction*, and *deployment abstraction* in CPU+FPGA heterogeneous computing systems.

Chapter 6

System Evaluation

Work hard in silence, let your success be your noise.

Frank Ocean

In this chapter, five showcases are presented to evaluate our systems against the objectives of FPGA virtualisation mentioned in Chapter 1. Case Study 1 shows how the infrastructure, development and deployment abstractions were used to improve design productivity and flexibility in module deployment, hence, enable the multi-tenancy support as well as to achieve better resource utilisation and higher system throughput, while Case Study 2 discusses how clever and sophisticated hardware task scheduling policies can achieve even much better resource utilisation and system throughput. The cost of multi-tenancy support to provide flexibility and security in terms of bus virtualisation and memory isolation is analysed in Case Study 3. Case Study 4 demonstrates how the proposed approach improves fault tolerance, resilience, and maintenance in a small multi-node system with 8 Zynq UltraScale+ MPSoCs. Finally, in Case Study 5, we examine the proposed approach in a large scale multi-node system (with 64 MPSoCs and 1TB memory in total) in terms of scalability and energy efficiency.

The results presented in the first four showcases were a collaboration effort with the PhD student A. Vaishnav who provided the different schedulers and run-time application drivers, while the author of this PhD project provided the hardware platform and low-level FPGA abstractions [PVVK18, VPKG18, VPK18b, VPK19, PPV+19]. Moreover, the experimental results presented in the last case study have been conducted by K. Georgopoulos, A. Ioannou, and P. Malakonakis from the ECOSCALE project, whereas the author of this PhD thesis provided the shell designs including workarounds of the Vivado PR flow, and the implemented accelerators [GBM+19].

6.1 Case Study 1: Design Productivity and Deployment Flexibility [PVVK18]

In this case study, we implement and execute Matrix Multiplication (MM) and Discrete Cosine Transform (DCT) OpenCL kernels in various instantiation scenarios. MM has two implemented variants which occupy 1-slot and 4-slot FPGA resources while DCT has 2-slot and 4-slot variants, as shown in Figure 4.13. This demonstration has been conducted on Xilinx ZCU102 Evaluation Board [Xilh].

The static shell design was implemented using Vivado 2017.4 (see Section 4.2) is explicitly *decoupled* from the hardware module designs (roles), which were deliberately compiled using Vivado 2016.2 (see Section 4.3). Note that users are free to use any Vivado version that is 2016.1 or later to implement their hardware modules. To the best of the author's knowledge, despite of providing users much more flexibility to utilise their available EDA tools, this feature is not supported by other state-of-the-art works [Alt17, Xil18c, KLP+18].

Moreover, users can use standard HLS optimisation methods to tune an OpenCL kernel's throughput and area for generating different RTL source codes for module variants. For example, performing vectorisation, loop unrolling and increasing work-group size (batch size) on the DCT kernel results in 3.68× throughput at twice the resource consumption. This means that at the resource cost of the large DCT kernel, we could host two small DCT ones. However, the large DCT kernel provides almost double the throughput of the aggregated small kernels, which we refer as *super-linear speed-up* with regard to the resource usage. This super-linear speed-up is achieved by micro-architectural optimisation, resource sharing, and reusing of intermediate values amongst computing elements in the kernel. Ultimately, this case study supports implementation variants to provide the *entire best module layout possible* towards the super-linear speed-up, as shown in Table 6.1. The different module variants can be implemented by the provided TCL templates.

6.1.1 Analysis of Design Productivity

The compile latency of the 1-slot, 2-slot and 4-slot module variants are 379, 1076 and 1476 seconds respectively by using our decoupled compilation flow, as reported in Table 6.1. Note that to provide the same level of flexibility in module deployment as in our approach, the state-of-the-art vendor PR flows [Xil18c, Alt17] have to implement all *permutations of module and placing slots*, since building relocatable modules is not officially supported in these vendor toolflows, as discussed and evaluated in Section 4.3.4.

A recent approach suggests to implement all permutations of deploying applications to

Table 6.1: Matrix Multiplication (MM) and Discrete Cosine Transform (DCT) show cases.

Application	Matrix Multiplication (MM)		
Implemented variant	1-slot		4-slot
Compilation time (seconds)	379		1476
Bitstream size (MBytes)	1.7		6.8
Instantiation scheme (No. of instantiations × No. of slots per instantiation)	1×1	4×1	1×4
Compute throughput (MBps)	0.067	0.268	0.339
Compute throughput speed-up	1×	4×	5.06×
Time to maximum throughput t_{TMT} (ms)	21.24	84.96	45.96

Application	Discrete Cosine Transform (DCT)		
Implemented variant	2-slot		4-slot
Compilation time (seconds)	1076		1476
Bitstream size (MBytes)	3.4		6.8
Instantiation scheme (No. of instantiations × No. of slots per instantiation)	1×2	2×2	1×4
Compute throughput (MBps)	1.37	2.74	5.04
Compute throughput speed-up	1×	2×	3.68×
Time to maximum throughput t_{TMT} (ms)	29.48	58.96	45.96

achieve high resource utilisation and system throughput [KLP+18]. However, the whole implementation process of 30 applications may take weeks in a workstation machine, and a single modification in any of applications will force this lengthy process restarted, as this ArmophOS work is built upon the Xilinx PR flow. In addition, any change in the static shell infrastructure also results in the same restart of module implementation. Thus, it is important to adopt a scalable and decoupled compilation flow when targeting multiple applications and shell versions, which are common in high-performance and cloud computing environments, as discussed in Section 4.3.4.

6.1.2 Analysis of Configuration Overhead

In order to quantify configuration overhead, which may vary because we have various alternatives to configure accelerators on the FPGA, we introduced the concept of *Time to maximum throughput (t_{TMT})*. t_{TMT} summarises the entire configuration time and software overhead that is needed to load the maximum number of accelerators to the FPGA. t_{TMT} is, therefore, an upper bound and can be estimated by Equation 6.1, where n is the number of instantiations, t_{conf} is the time in which PCAP partially reconfigures the FPGA fabric, and t_{sw} is the software overhead for the scheduler to call the configuration controller and the configuration controller process. PCAP throughput was measured at 256 MB/sec while the t_{sw} is measured at 13ms on average, as reported in Section 5.2.3. For n instantiations, the time to maximum throughput is:

$$t_{TMT} = n \times (t_{conf} + t_{sw}) \tag{6.1}$$

The time, t_{TMT}, was measured on average to be 45.96ms for 4 slots when using PCAP to partially reconfigure the FPGA fabric (see also Table 6.1).

6.1.3 Analysis of Deployment Flexibility and System Completion Time

In the experimental scenario, a long-running MM task starts at the time zero for a workload of 400 work-groups while DCT arrives at an arbitrary time. Various module instantiation schemes (Instantiations $\times$ slots per variant) are deployed and evaluated to demonstrate how flexible the proposed approach can support executing multiple kernels simultaneously, as shown in Figure 6.1. Figure 6.1a shows the scheme in which MM has occupied 1 slot, therefore when DCT arrives it can occupy 2 available slots and start executing to completion without delay. In Figure 6.1b, 1-slot MM has been instantiated twice to occupy 2 slots, but DCT can still get the other available slots for its execution and hence, minimise its waiting time. Further, 1-slot MM has been instantiated 4 times to occupy all 4 slots, so DCT must wait to get enough resources,

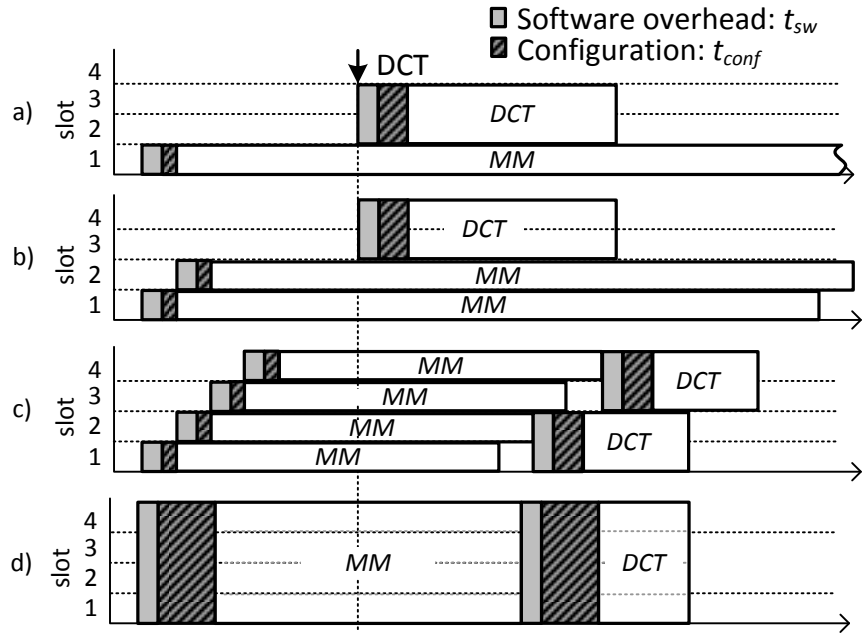


Figure 6.1: Various instantiation and execution schemes for Matrix Multiplication (MM) and Discrete Cosine Transform (DCT) kernels.

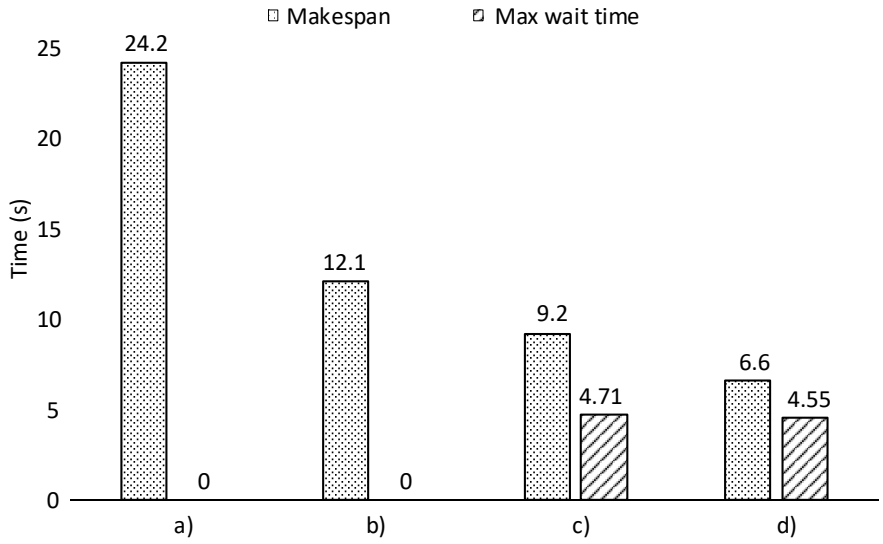


Figure 6.2: Completion time in various execution schemes in Figure 6.1.

as shown in Figure 6.1c. In Figure 6.1d, the biggest and fastest variants of MM and DCT have been used to achieve the highest throughput and the shortest *makespan* (i.e. end-to-end execution time) at 6.6 seconds .

The last scenario is indeed the best for performance (3.67× in performance) but is not good for fairness as the *wait time* of the second application (DCT application in this case) is the *second longest* among the four scenarios, as shown in Figure 6.2. Although the makespan metric

is critical to most of high-performance computing applications, the fairness metric is equally important in the scenario of multi-tenant clould computing systems, in which applications from different users are competing to get the necessary resources to execute.

6.1.4 Summary

This case study demonstrates how the *development abstraction* enables the decoupled implementation of the static shell and the application roles in order to enhance the design productivity. Moreover, the flexibility of application deployment improves the resource utilisation and system completion time (i.e. makespan) beyond what the state-of-the-art approaches can offer thanks to the *deployment abstraction*, which is a major contribution from this PhD project. The next case study will show how the techniques enabled in this PhD project can help to utilise available resources on a heterogeneous computing system in terms of CPUs and FPGAs to further enhance the resource utilisation and reduce the overall completion time.

6.2 Case Study 2: Resource Utilisation and System Performance [VPKG18, VPK19]

This case study shows how the flexibility of module deployment (i.e. deployment abstraction) enabled by the proposed approach combining with the OpenCL programming model and the cooperative scheduling utilises the resources in a heterogeneous computing system even higher, and hence, provides the whole system a further significant performance improvement.

Four OpenCL applications including a Finite Impulse Response (FIR) filter, a Discrete Cosine Transformation (DCT), a 3D Normal Estimation (eNormal) and a Matrix Multiplication (MM) (see Section 4.3) are deployed on top of the shell, which is built for a Xilinx ZCU102 development board (see Section 4.2). Note that on one hand, eNormal is CPU favoured where it can run 4× faster on CPU than on FPGA, and on the other hand, the FIR filter is FPGA favoured with a 3.4× boost up over CPU implementation. Moreover, DCT is an interesting case where the DCT implementation on CPU is slightly faster (1.18×) than the 2-slot FPGA DCT and considerable slower (3×) than the 4-slot one. The implementation difference between 2-slot or 4-slot DCT modules is based on larger buffer sizes and loop unrolling for better data reuse. In particular, depending on the scheduler's decision, it may be CPU favoured or FPGA favoured. Finally, 4-slot FPGA MM is 3.8× faster than its CPU counterpart.

The following arrival times are considered for the sake of simplicity: eNormal at 0ms, DCT at 100ms, FIR at 200ms and MM at 300ms. Moreover, five different execution scenarios

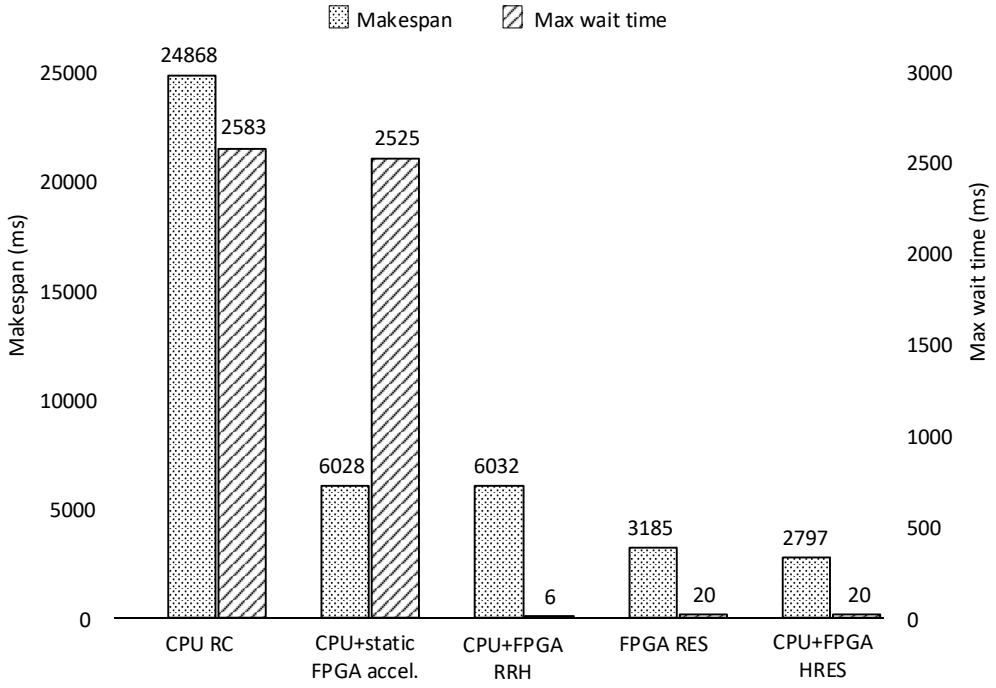


Figure 6.3: Comparison of *makespan* (total execution time of the entire benchmark) and *maximum wait time* for different scenarios including software execution only (CPU RC), static accelerators (CPU+static FPGA accel.), round-robin scheduling(CPU+FPGA RRH), resource elastic scheduling (FPGA RES), and the heterogeneous resource elastic scheduling (CPU+FPGA HRES), which uses FPGA and CPU resources in orchestration for acceleration.

according to different scheduling policies are examined as following:

1. Kernels are deployed only on CPUs (using POCL [JdLLS+15], an OpenCL run-time API);

2. On CPU and using one static FPGA accelerator (similar to Xilinx SDSoC [Xild] using FPGA drivers) where the FPGA accelerator is MM (biggest kernel);

3. On CPU and FPGA using Round-Robin scheduling (RR-H) (similar to the Case Study 1 in Section 6.1);

4. Only on FPGA using resource elastic scheduling (RES) [VPKG18];

5. on CPU and FPGA using heterogeneous RES (HRES) [VPK19].

In this case study, we use *makespan* (total execution time of the entire benchmark) as the metric to illustrate the system performance and *maximum wait time* as the metric of fairness.

6.2.1 Analysis of Resource Utilisation and Performance

Figure 6.3 shows comparison of *makespan* (total execution time of the entire benchmark) and *maximum wait time* for the aforementioned execution scenarios. As compared to a reconfigurable run-to-completion model using full reconfiguration (which is the model used by the Xilinx SDSoC run-time), our RES approach is reducing the total execution time of the benchmark from over 6000ms down to 3185ms (1.88× in performance). Moreover, using HRES can take full advantage of the heterogeneous resources available in Zynq UltraScale+ FPGAs by scheduling to FPGA and CPU resources simultaneously, which reduces makespan even further down to 2797ms (i.e. the speed-up at 2.15×). It is important to understand that this performance boost come entirely from a more competent scheduling without the need of an application developer to change anything in the application or any accelerator kernel module. However, our scheduler can take advantage of implementation alternatives (which are modules implementing the same functionality, but with different micro-architectures in order to trade resources for computational throughput). Therefore, an application developer can help the resource elastic scheduler by providing corresponding accelerator implementation alternatives with an assist from a Design Space Exploration (DSE) approach [MMRL17].

6.2.2 Summary

In this case study, the available computing resources in a heterogeneous system in terms of CPUs and FPGAs are fully utilised to speed-up the system performance by up to 2.15×. This improvement is enabled by using resource elastic scheduling policies such as RES and HRES which are only feasible because of the *development abstraction* (via implementation variants, as discussed in Section 4.3.4 and the Case Study 1) and the *deployment abstraction* (via the dynamic and flexible resource allocation), as introduced in this PhD thesis.

6.3 Case Study 3: Cost of Flexibility and Security for Multi-tenancy Support [PPV+19]

Multi-tenancy support is a useful feature of any FPGA-virtualised system yet difficult to achieve as various users may come with their own requirements of module designs and deployment schemes. For example, as the proposed shell is designed with the default 128-bit AXI4 Master interface for modules to fetch data from the main memory, a user module with other AXI interfaces, such as AXI Stream, or different data-width, such as 32- or 64-bit, will need a *bus adaptor* to be integrated to our shell instead of re-designing the module itself or the

shell. Moreover, multi-tenancy support requires a basic level of logical isolation between users, which is provided in the proposed approach by means of memory virtualisation, as presented in Section 5.5.

6.3.1 Analysis of Resource and Performance Overhead

Although the bus adaptor approach provides more flexibility for module development and deployment, logic overhead of the bus adaptor is inevitable, as reported in Section 4.2.2.

To evaluate run-time system overhead of bus and memory virtualisation, we use a memory bound application rather than compute bound application as the performance of a compute bound application is primarily tied to the logic resources and not to the run-time overhead caused by the shell. To this advent we use an application example with probably the most common access pattern of 2 read operations and 1 write operation: vector addition (VADD). We evaluated the system with 32-bit, 64-bit, and 128-bit data-width memory bound VADD accelerators, over memory accesses of 512kB and 1 MB and numbers of users ranging from 1 to 3, which access the system DDR memory concurrently. In this scenario, we use the same application type to identify and quantify the worst-case overhead caused by the communication interface and the memory virtualisation.

The end-to-end time of module operation includes overheads of the hardware scheduler, configuration controller, memory management, and the module execution itself, as summarised in Equation 6.2:

$$t_{total} = t_{sched} + t_{conf} + t_{mem} + t_{exe} \tag{6.2}$$

The overhead of the hardware scheduler, t_{sched}, is the time to make its decision of which hardware module is selected to launch. t_{sched} ranges from 1μs to 2.9μs, as mentioned in Section 5.3.2. However, this t_{sched} is marginal for the total overhead. t_{conf} summarises the configuration time and software overhead to load a hardware module to the FPGA. The throughput of the Processor Configuration Access Port (PCAP) was measured at 256MB/s while the software overhead is measured at 13ms on average, as shown in Section 5.2.3. Because these overheads are the same on all experimental cases, we omit them in the final comparison. t_{mem} is the penalty we pay to integrate the memory isolation stack into the run-time management layer. The overhead of registering a new application was measured at 1.3ms (see Section 5.5.3) while the overhead of using SMMU is 0.9μs for the first iteration is negligible, as mentioned in Section 5.5.3. The module execution time, t_{exe}, is measured from fetching the processing data to its completion.

As mentioned above, VADD computation is lightweight but memory intensive. Thus, by using the 32-bit communication bus-width instead of the default 128-bit one, a reduction of 3.15$\times$ in performance was observed. This reduction includes the penalty for the memory

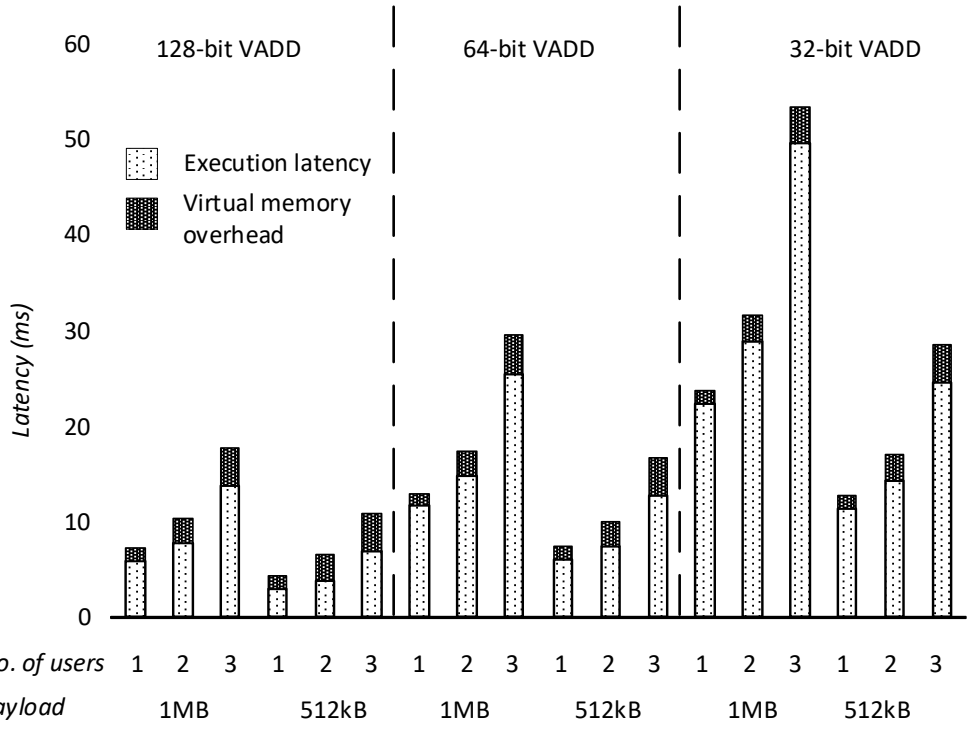


Figure 6.4: The Vector Addition (VADD) design which has 2 inputs and 1 output (i.e. conducting 2 operand reads and 1 operand write at a time) is being used for this experiment. The experiment is conducted on the Ultra96 platform [AVN19].

isolation, which is required for guaranteeing the security and robustness of a system. The reason why the reduction is not 4× (as we reduced the bus-width by 4×) is because the 128-bit version cannot saturate the full memory bandwidth. The exact benchmarking results are shown in Figure 6.4.

6.3.2 Summary

In this case study, the cost for a flexible and secure multi-tenant support on heterogeneous computing systems is analysed. The here proposed bus adaptor technique along with the decoupled compilation flow plays an essential role to provide the flexibility of this multi-tenant support.

6.4 Case Study 4: Fault Tolerance/Resilience, and Maintenance of Multi-node Systems [VPK18b]

We have examined our proposed approach against multiple objectives for FPGA virtualisation of a single-node system as presented in previous sections. For example, it was shown how flexibility in module deployment enables the proposed approach to speed up the whole system performance as well as to utilise a significant amount of available computing resources, as had been discussed in Section 6.1 and Section 6.2. However, modern data centres consist of a great number of FPGA systems, therefore, fault tolerance, maintenance and high availability of FPGA acceleration services are highly recommended [MPL+16]. Hence, application migration is necessary for large-scale multi-node systems for following reasons:

- **Fault tolerance:** In case of a fault, an application needs to be migrated to another node where it can resume execution from the last known consistent state (possible by using established techniques such as check-pointing).
- **Maintenance:** Consider a scenario where a node needs a hardware/software upgrade but is currently executing a long running application. With the help of migration, the application can be temporarily moved to a remote node while the upgrade is performed transparently.
- **Resource management:** Migration permits dynamic load-balancing to redistribute works across a network as the workload changes.

In this section, we discuss how the here developed platform helps improve the fault tolerance, resilience, and maintainability of multi-node heterogeneous computing systems. Let us consider a multi-node system in which a number of Zynq UltraScale+ MPSoCs are used, as shown in Figure 6.5b. As before in Section 6.2, we assume the here presented system running OpenCL workload where large jobs are decomposed into small jobs (called work-groups) that run-to-completion and that can be dispatched to the compute resources in the system in any arbitrary order.

A master hardware task scheduler can be deployed on top of the system to monitor and orchestrate running applications for a unified CPU+FPGA computing resource pool, as illustrated in Figure 6.5a. Given the ability to perform context switching between applications across various computing resources and the data-parallel nature of the application's execution model (for example, the OpenCL execution model [KHR]), we can migrate a running application between nodes without disrupting its execution by overlapping the module execution on the target node with the one in the source node. This overlapping is possible as the execution order of different work-groups (a group of threads in the OpenCL terminology) are undefined, as shown in

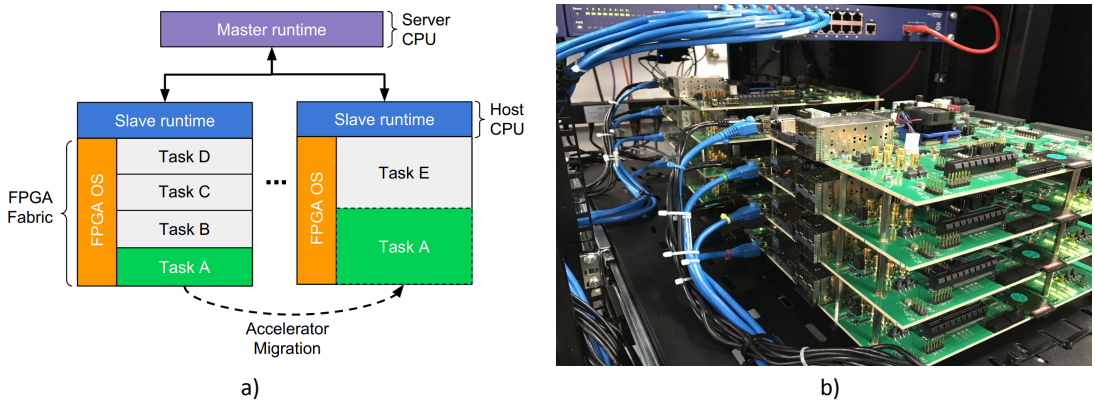


Figure 6.5: a) is an example of the live migration process of FPGA accelerators across computing nodes and b) is the experimental setup with eight Xilinx ZCU102 platforms [Xilh].

Figure 6.6. For interested readers, we suggest to read the paper [VPK18b] for more details.

6.4.1 Mitigation Scenarios for Fault Tolerance/Resilience

The proposed approach provides support for fault tolerance at node level in which a running application of a faulty node can be migrated and continue its execution on another healthy node without any interruption from the user's point of view. The fault scenario may occur due to a software bug, a hardware fault (e.g. power failure), or problems in the local operating system which requires the node to be rebooted. Since the main bottleneck for migration is the data transfer of kernels' partial bitstreams across the multi-node system, as discussed in the paper [VPK18b], the relocatable partial bitstreams are highly desired to save the number of bitstreams needed to be transferred.

Migration for fault tolerance can be combined with load-balancing algorithms to redistribute workloads across multiple nodes to get a high utilisation for large-scale systems, as briefly illustrated in Figure 6.5a. To achieve this objective, the ability to merge multiple adjacent PR regions to host modules with various resource requirements, which is also supported by the proposed approach, is essential.

Moreover, this migration can actively happen while the system infrastructure (a.k.a shell) is maintained or upgraded. Note that such kind of system upgrade usually requires not only the whole system to be shutdown but also all application modules to be rebuilt, as discussed in Section 2.2. In contrast, the proposed approach can perform the shell update at run-time, and hence, cause limited side-effects in the execution of user applications.

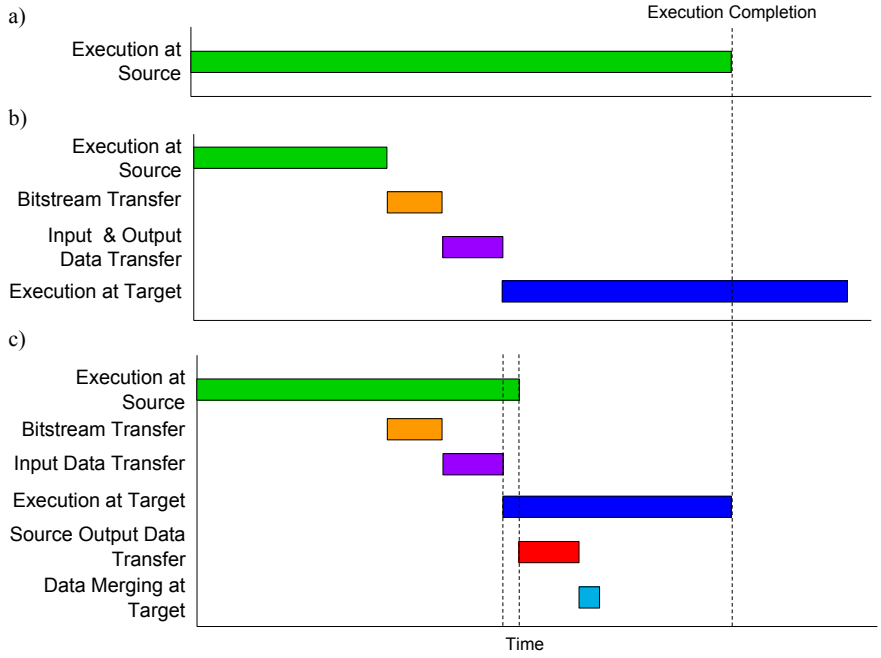


Figure 6.6: Execution trace of an accelerator in different scenarios of data movement. (a) is the case of no migration; (b) is the case of accelerator migration *without* overlapping of data movement; and (c) is the case of accelerator migration *with* overlapping of data movement. Note that these run-time data management schemes are the contribution of A. Vaishnav but reproduced here for completeness [VPK18b].

Table 6.2: Shell update latency breakdown.

Phase	Latency (ms)
Write-to-SD-Card + Migration to Temp. Node	852
Shutdown Period	600
Boot-up Period	29420
Static Logic Reconfiguration	104
Run-time Initialisation	2010
Migration to Source	278
Total Time Taken	33264

6.4.2 Analysis of System Maintenance

To update a shell at run-time in our demonstrating system, the following steps need to be performed:

1. A new Linux image with updated static logic needs to be written to SD-Card of the source node while in parallel the migration to another node is performed. This is required to allow the ARM core on the Zynq UltraScale+ chip to boot correctly with the new device tree information which is needed if hardware components change.

2. A reboot sequence needs to be initiated on the source node using the new booting image in the SD-Card.

3. Migrating back the applications to the source node.

The latency of each of these steps for the Matrix Multiplication benchmark using 4-slot through replication is shown in Table 6.2. The total latency for the shell update is 33.26 seconds, the majority of which corresponds to the Linux boot up period at 29.42 seconds (about 88.45% total latency). In terms of the total execution time of the Matrix Multiplication benchmark, the overhead represents 13.25% of the total execution time. During this period live migration can allow continuous provisioning of acceleration service by moving to another node without pausing the kernel execution.

6.4.3 Summary

This case study demonstrates novel techniques of fault tolerance/resilience and system maintenance for a multi-node heterogeneous system with limited side-effects on other running applications in the system. The here presented techniques are infeasible without the support of module relocation (i.e. *deployment abstraction*), which helped to reduce the bitstream transfer latency as only a single bitstream is needed, and the decoupled compilation flow (i.e. *development abstraction*), which helped to reduce the overall system update latency. Both the *deployment abstraction* and the *development abstraction* are original contributions of this PhD thesis.

6.5 Case Study 5: Scalability and Energy Efficiency of Multi-node Systems

In this case study, we examine the deployment of the proposed abstractions on a large scale multi-node heterogeneous system with 64 FPGAs and 1TB memory in total. Moreover, to achieve the requirement of energy efficiency for high-performance computing with large data-sets [oE10], we have adopted the approach of moving compute towards data instead of the conventional approach of moving data by leveraging partial reconfiguration.

This case study uses Quad-Daughter FPGA Board (QFDB) which is a custom multi-FPGA board including 4 Xilinx ZU9EG MPSoCs and provides a very dense, tightly connected FPGA placement [KAB+18, CIK+19], as illustrated in Fig 6.7a. In addition, the QFDB was constructed with UNIMEM (Unified Memory) [MGF+16] and its successor UNILOGIC [GBM+19] in mind, which provides a uniform partitioned global memory address space (PGAS) and virtualised access of reconfigurable logic across multiple HPC nodes. Further, multiple QFDBs can be connected to form up a large pool of computing resources with a low power budget. Together, the QFDB cluster and the UNILOGIC architecture provide a foundation to deploy FPGA applications at large scale.

A cluster of QFDBs is used as the testing platform in this case study. This cluster is designed to scale for low-power, high-performance computing consists of 64 Xilinx ZU9EG Zynq MPSoCs, tightly connected together by 10-Gigabit Ethernet connections, as shown in Fig 6.7b. Each MPSoC is attached to 16GBs of ECC-protected DDR4 memory, and the DDR controller is configured to run at DDR4-PC2133 speed. Each QFDB has a master node (known as F1) responsible for the management and work-offload to other slave nodes (F2-F4). The management layer executes on the CPU cores of F1 with Linux kernel at its base. In contrast, the CPU cores of F2-F4 operate in a bare-metal mode for high performance and energy efficiency. The FPGA fabric in each node has four partial regions (labelled slots 0 to 3 in Fig 6.7a) and adopt the FPGA Operating System (FOS) shell [VPPK20] by adding UNIMEM and UNILOGIC to the static system infrastructure. In this system, UNIMEM and UNILOGIC are offering a unified address scheme for memory, CPUs, and the reconfigurable fabric of FPGAs, as if they were connected to the same bus.

The high-level connectivity between these QFDBs to form a bigger cluster is established via the F1 node on each board. Figure 6.7b shows an upscaled cluster with 64 MPSoCs in which only one F1 node is boot up with the Linux kernel acting as the *cluster master node*.

For such a large scale system, to minimise the energy consumption, we need to:

1. limit the amount of data transfer between nodes in the cluster; and
2. reduce the configuration overhead when switching accelerators as much as possible.

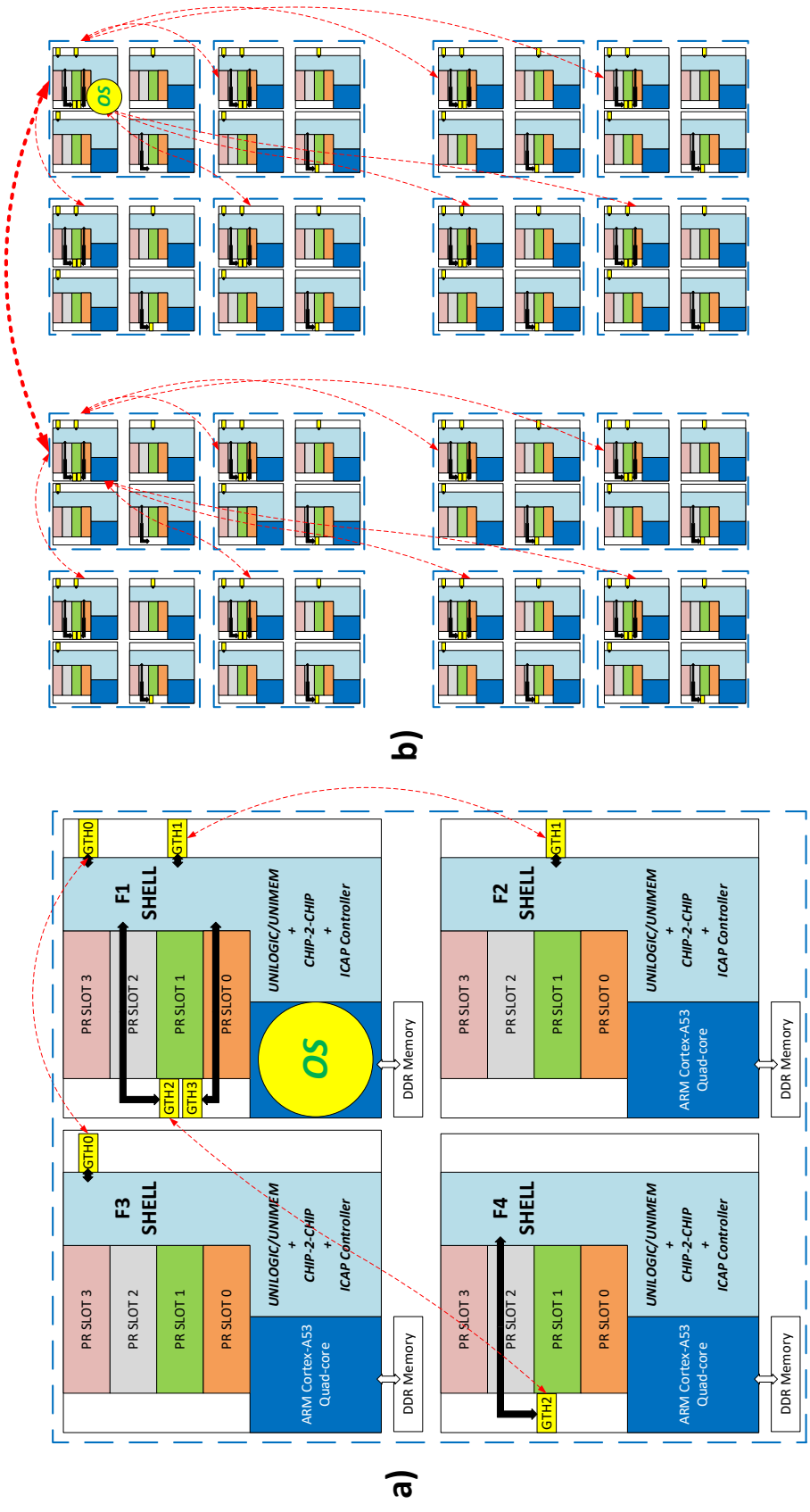


Figure 6.7: A cluster of QFDB (Quad-Daughter FPGA Board) featuring Xilinx ZU9EG Zynq MPSoCs. (a) cluster architecture including the nodes F1 (the master/network node), F2, F3, and F4 acting as slave nodes; (b) large scale test platform with 16 QFDBs (64 FPGAs and 1TB RAM) connected together, while there is only one F1 node running the full Linux kernel, marked by the yellow circle.

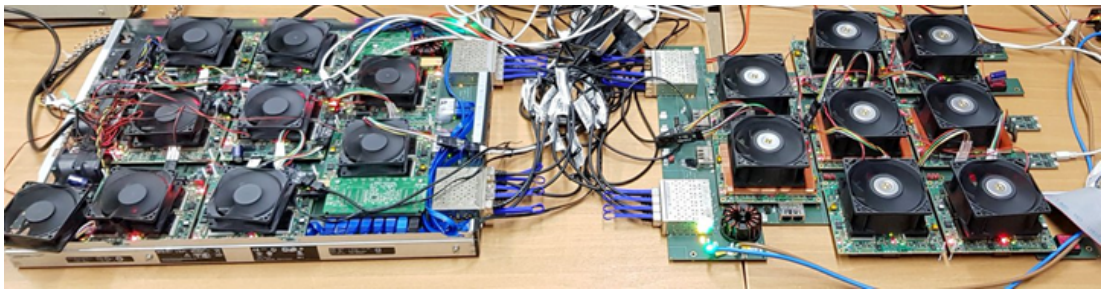


Figure 6.8: The ECOSCALE platform with 16 QFDBs (64 FPGAs and 1TB RAM) is used in this case study.

For the first requirement, we have adopted the approach of moving compute accelerators towards nodes where data are located, as the size of an accelerator (i.e. an FPGA bitstream) is in the range of a few MBs compared to the typical range of GBs of the data-set to be processed. For the second requirement, in this case study, we have chosen to use the Internal Configuration Access Port (ICAP), integrated inside the FPGA fabric of the MPSoC, instead of the processor-based reconfiguration port (i.e. PCAP), as ICAP has better throughput (800 MB/s for ICAP and 256 MB/s for PCAP). Moreover, as ICAP is located in the FPGA fabric of the MPSoC, it can be connected directly to the UNILOGIC infrastructure for (remote) chip-to-chip bitstream transfer. Therefore, using ICAP introduces less software-stack overhead than using PCAP, which is about 13ms (see Section 5.2.3).

However, using ICAP is lacking of support from both vendor design tool and software API. For providing ICAP configuration, we adopted the work in [VF14b] to our system and clocked it at maximum frequency at $200MHz$. In addition, our custom ICAP was able to manipulate partial bitstreams to relocate an accelerator to various targeting PR slots, as inspired by [KLPR05]. With this, the resulting throughput of intra-chip configuration reached 770.27 MB/s (8.64$\times$ improvement compared to the Xilinx HWICAP IP [Xil16a]) while the throughput of inter-chip configuration reached 482.23 MB/s (6.76$\times$ improvement) with the loss of 59.73% due to the network latency, as reported in Table 6.3. The higher network latency compared to the Xilinx IP is due to the network latency becoming a more significant bottleneck when the configuration throughput is improved.

In two out of four MPSoCs, static signals need to be routed across the slots to access the physical 10-Gigabit transceiver primitives for chip-to-chip communication, as illustrated in Figure 6.7a. Since the MPSoCs have the same FPGA fabric but with different I/O layouts, existing approaches would have different static system flooplans which, in turn, require different static system-specific accelerator module implementations. We targeted the diversity of I/O layouts by providing individual shell for each FPGA in a QFDB (i.e. *infrastructure abstraction*) to offer a unified application development process and deployment framework for users by leveraging the methodology proposed in Section 4.2.3. Consequently, users do not need to

Table 6.3: Throughput of ICAP-based controller in intra- and inter-chip configuration.

Type of Configuration	HWICAP [Xil16a] (MB/s)	Custom ICAP (MB/s)	Speed-up
Intra-chip (local)	89.13	770.27	8.64×
Inter-chip (remote)	71.3	482.23	6.76×
Network overhead	25%	59%	

deal with every single FPGA with different I/O layouts to develop their applications, which is infeasible with other state-of-the-art design methodologies [Alt17, Xil18c, KLP+18]. Moreover, the I/O virtualisation technique introduced in Section 4.2.3 was incorporated to provide more flexibility and to reduce the amount of data transfer as a single partial bitstream of an application is executable across the platform.

6.5.1 Analysis of Scalability and Energy Efficiency

The accelerators implementing the Michelsen algorithm [IMG+19] were used to demonstrate and evaluate the system. The Michelsen algorithm is widely used in the field of oil-reservation simulation to predict oil and gas flows, and hence, its hardware accelerators represents a complex real-world application. In this case study, an optimised version of the Michelsen algorithm with higher parallelism for FPGAs was compiled directly from its OpenCL source code to a single partial bitstream using the decoupled compilation flow (see Section 4.3). The implemented Michelsen accelerator has used 40630 LUTs, 57 BRAMs, and 176 DSPs in the targeting ZU9EG FPGA (2 slots).

The decoupled compilation process took 55 minutes using Vivado 2018.2.1 on Intel Core i7-4930K CPU running at 3.4 GHz with 64 GB of RAM due to high utilisation of available resources and routing congestion, as shown in Figure 4.9. Note that this Michelsen accelerator needed to be compiled only once and were able to be deployed across the entire platform thanks to supporting module relocation and replication at the bitstream level. The implementation of the same Michelsen accelerator using Vivado PR flow on the same machine took 336 minutes for all possible slots in a QFDB, i.e. it took 6× longer than our approach.

To evaluate the system performance and energy consumption, we used a data set of earth substrate analysis with 240 million grid points. This data set, which was about 1.5GBs in total, was solved by the Michelsen algorithm deploying on different computing platforms, including:

1. CPU-based platform with Intel Core i7 at 3.4GHz and 64GB RAM,

2. GPU-based platform equipped with an Nvidia GTX980,

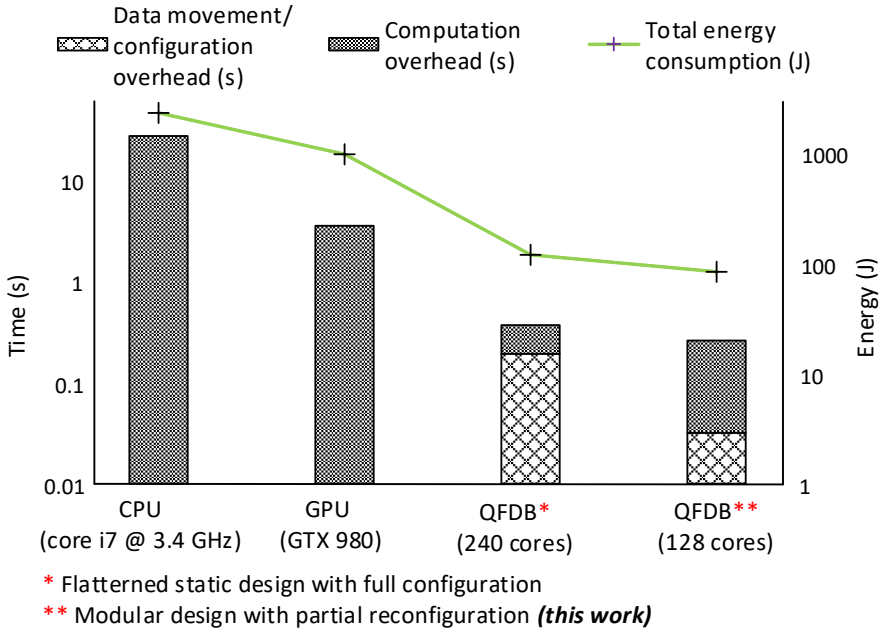


Figure 6.9: Execution latency and energy consumption of Michelsen on different deploying platforms.

3. QFDB-based platform with Michelsen accelerators statically integrated (240 cores) using full configuration,

4. QFDB-based platform with Michelsen accelerators built as a partially reconfigurable module and deployed on the platform using intra- and inter-chip partial reconfiguration (128 cores in total).

The total execution time, including configuration overhead, computation overhead, and energy consumption per platform, is reported in Figure 6.9. For this highly parallel application, the QFDB-based deployments outperformed the other software-based deployments (28.1 seconds and 2389 Joules for the CPU-based platform, and 3.6 seconds and 1008 Joules for the GPU-based platform) as expected. The interesting comparison was performed between the QFDB-based deployments themselves with and without utilising partial reconfiguration.

In the deploying case without partial reconfiguration, the Michelsen accelerators were statically integrated to the system and hence, the final system implementation was better optimised to deploy a higher number of cores. This led to faster computation at 0.179 seconds. However, the data-set (1.5GB) needs to be transferred across the platform, and the full configuration for each node needs to be performed when switching the applications. Therefore, the configuration and data movement (of 1.5GB of the data-set) overhead was at 0.194 seconds, and the total energy consumption was 121.5 Joules resulting in performance to per kilo Joules ratio of 13.17.

In the case with utilising partial reconfiguration, the Michelsen accelerators were built as a relocatable partial bitstream and launched on the deploying platform at run-time thanks to the reconfiguration infrastructure developed in this work. Moreover, the data-set is manually partitioned across the platform. Hence, the FPGA partial bitstream of the Michelsen accelerator (5.7MB) is the only data that needs to be transferred across the platform, which is two order of magnitudes less than the previous case. Therefore, the longest data movement (of 5.7MB of the partial bitstream) and configuration overhead was at 0.031 seconds in the case of remote reconfiguration. However, this approach caused more resource overhead in the static part of each FPGA as one floorplan was used for all FPGAs. This disadvantage led to fewer resources being available for the accelerators and lower number of deploying cores. It took 0.23 seconds to finish the computation and consumed an energy of 85.4 Joules resulting in performance to per kilo Joules ratio of 18.74 (42% improvement). This improvement means that in this case study, remote partial reconfiguration and I/O virtualisation were beneficial even with some resource overhead, because it reduced the need for time-consuming data movement.

6.5.2 Summary

To enhance the system throughput and reduce the energy consumption, it is essential to limit the amount of data transfer. Therefore, in this case study, we adopted the approach of moving the compute accelerator (in range of a few MBs) towards the data (in ranges of a few GBs) by utilising partial reconfiguration. For such a complex multi-node heterogeneous computing system, the *infrastructure abstraction* was used to virtualise the underlying FPGAs with diversified I/O layouts via implemented shells to provide a transparent process of application development. Further, an I/O virtualisation technique in the decoupled compilation flow (i.e. *development abstraction*) was utilised to build a single partial bitstream of the accelerator which can be deployed in different FPGAs across the system.

6.6 Chapter Summary

Five examples have been discussed to evaluate the resulting systems against the overall objectives of FPGA virtualisation mentioned in Chapter 1. Objectives including: 1) design productivity and flexibility (discussed in Section 6.1); 2) resource management and performance (in Section 6.2); 3) multi-tenancy, isolation, and security (in Section 6.3); 4) fault tolerance/resilience, and maintenance/upgradability (in Section 6.4); and 5) scalability and energy efficiency (in Section 6.5); have been achieved. The next chapter is concluding this PhD thesis as well as provisioning future works which are enabled by this PhD project.

Chapter 7

Conclusion

'Why so hard?' the charcoal once said to the diamond; 'for are we not close relations?'
'Why so soft? O my brothers', thus I ask them: 'for are you not my brothers?'

Friedrich Nietzsche

This chapter summarises contributions of this PhD project and discusses future research directions enabled by this PhD project.

7.1 Summary of Contributions

7.1.1 A Model of FPGA Virtualisation on Heterogeneous Computing Platforms

Based on the rigorous examination of state-of-the-art FPGA virtualisation trends and techniques, this thesis has proposed a model of FPGA virtualistion on heterogeneous computing platforms that the **FPGA resources in heterogeneous systems should be abstracted, managed, and virtualised in an analogous way as the way that is known from software operating systems (OSs)**. This model should essentially provide multiple level of abstractions including 1) infrastructure abstraction, 2) development abstraction, and 3) deployment abstraction, as introduced in Chapter 1.

7.1.2 A Tool and API for Bitstream Abstraction

Partial reconfiguration [Koc12] can be utilised to provide the aforementioned abstractions for the proposed model of FPGA virtualisation. However, the state-of-the-art design toolchains

and techniques have several significant limitations that prevents us to achieve our goals, as pointed out in a recent survey of partial reconfiguration [VF18]. Among the limitations, the ability of design adaptation at binary (bitstream) level for latest FPGA devices is the crucial missing. Therefore, this PhD project has provided a tool and API, called BitMan [PHK17], to deliver this ability of bitstream adaptation on latest Xilinx FPGAs. The BitMan tool can be used at the design phase to merge multiple FPGA designs together (a.k.a hardware linking) while the BitMan API can be utilised to relocate hardware modules at run-time, as discussed in Chapter 3.

7.1.3 Design Methodology and System Implementation

A novel design methodology for FPGA virtualisation is proposed to utilise academic design tools, including BitMan [PHK17], GoAhead [BKT12] and TedTCL [Ves18], and vendor toolflow such as Xilinx Vivado [Xil14] in order to implement FPGA systems which follow the proposed model to achieve the goals of FPGA virtualisation. Moreover, the detailed implementation of the static infrastructure (shell) and the compilation flow of the hardware modules (roles) have been discussed in Chapter 4.

7.1.4 System Prototype and Evaluation

The system run-time management which is in charge of abstracting module deployment has been discussed in Chapter 5. This run-time management includes multiple components such as a configuration controller, a hardware task scheduler, a module device driver, and a memory management framework. Furthermore, five case studies have been presented to evaluate the resulting systems against the aforementioned goals of FPGA virtualisation in Chapter 6.

7.2 Future Works

7.2.1 Security for Multi-tenancy FPGA-virtualised Systems

Integrating FPGA resources into data centres and cloud computing systems along with CPUs and GPUs is opening a new surface of attack which has not been fully studied. Especially in the multi-tenancy environment where many users are sharing the same FPGA fabric, protecting data privacy of all users or preventing a user to shutdown the system is a new challenge. Although this PhD project has provided some basic services such as bounding box checking, memory management, and physical module isolation (see Appendix A), more intensive researches need to be conducted to fully support secure multi-tenancy FPGA-virtualised systems.

Bitstream verification to mitigate electrical-level threats on multi-tenancy computing environments: Sharing configuration bitstreams rather than netlists is a very desirable feature to protect IP or to share IP without long CAD tool times and it can potentially remove the need to run CAD tools at all in a hardware app ecosystem. However, that comes with security concerns including side-channel attacks for leaking information, denial-of-service attacks for impacting proper execution of a system, or even triggering FPGA aging or damaging effects. A security mechanism is required to tackle most of the existing malicious circuits directly at the bitstream level, as recently mentioned in [KGT19]. With a detailed understanding of architecture and bitstream format of Xilinx UltraScale+ FPGAs (which are being widely deployed in data centres and cloud computing systems [MPL+16, Ama]) provided by BitMan, we could intensively explore more possible malicious circuits as well as provide a truly protecting mechanism on large-scale data centres and cloud computing systems. This research is being conducted in an ongoing research project called *rFAS - reconfigurable FPGA Accelerator Sandboxing*.

7.2.2 High-speed Configuration Infrastructure

Reconfiguring FPGAs may take tens to hundreds of milliseconds depending on the size of the partial bitstream which is unacceptable in many applications. To overcome this issue, several techniques have been proposed before such as overclocking the configuration port [HKT11], on-chip buffering the configuration data [VF14b], and compressing the bitstream [KBT07]. We are going to deploy the combination of these existing techniques on the Xilinx UltraScale+ devices [Xil17b] to examine how much configuration throughput we can achieve. Moreover, with the introduction of multi-die FPGAs [Xilg] in which several identical FPGAs are integrated on the same chip, the possibility of utilising the physically existing configuration ports for parallel configuration will be explored.

7.2.3 Vendor-independent FPGA Platforms for Education and Research Purposes

With the current progresses of academic open-source FPGA design toolflows [LGW+14, VVS17, SHW+19, VVS19, GGNW19], users can synthesise, implement their FPGA designs, and then, generate bitstreams without using vendor toolflows. Such kind of academic toolchains help remove the burden to install, maintain and operate heavy FPGA vendor tools, and hence, these toolchains could be utilised for building lightweight self-contained out-off-the-box education and research solutions which essentially allow users to experiment and test their FPGA designs on real hardware. This aligns with the PYNQ initiative from Xilinx that is a Python-based educational platform for SoC design [Xilb]. Our initial work in this research direction is the

EFCAD toolflow [PVKH19] which integrated Yosys+nextpnr [SHW+19], GoAhead [BKT12], BitMan [PHK17], and ZUCL framework [PVVK18]. The next step will be the integration of PYNQ and EFCAD to provide an entire stack of vendor-independent FPGA platforms for education and research purposes.

Bibliography

[AGV+17] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne. Virtualized Execution run-time for FPGA Accelerators in the Cloud. *IEEE Access*, 5, 2017.

[Alt09] Altera. AN 567: Quartus II Design Separation Flow. 2009.

[Alt10] Altera. Introduction to the QuartusÂő II Software. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/intro_to_quartus2.pdf, 2010.

[Alt17] Altera. Partial Reconfiguration. https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/partial-reconfiguration.html, 2017.

[Ama] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[AOC+17] U. Aydonat, S. O'Connell, D. Capalija, A. Ling, and G. Chiu. An OpenCL™Deep Learning Accelerator on Arria 10. In *FPGA*, 2017.

[ARMa] ARM Ltd. AMBA Specifications. https://www.arm.com/products/system-ip/amba-specifications.

[ARMb] ARM Ltd. System Memory Management Units. https://developer.arm.com/ip-products/system-ip/system-controllers/system-memory-management-unit.

[AS03] M. Abramovici and C. Stroud. BIST-Based Delay-Fault Testing in FPGAs. *Journal of Electronic Testing*, 2003.

[AVN19] AVNET. Ultra96-V2 Development Board. http://zedboard.org/product/ultra96-v2-development-board, 2019.

[BCB18] J. Bishop, J. Chareau, and F. Bonavitacola. Implementing 5G NR Features in FPGA. In *EuCNC*, 2018.

[BKT10] C. Beckhoff, D. Koch, and J. Torresen. Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration. In *FPL*, 2010.

[BKT11] C. Beckhoff, D. Koch, and J. Torresen. The Xilinx Design Language (xdl): Tutorial and Use Cases. In *ReCoSoC*, 2011.

[BKT12] C. Beckhoff, D. Koch, and J. Torresen. GoAhead: A Partial Reconfiguration Framework. In *FCCM*, 2012.

[BKT14] C. Beckhoff, D. Koch, and J. Torresen. Design Tools for Implementing Self-Aware and Fault-Tolerant Systems on FPGAs. *TRETS*, 2014.

[BRM99] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[BSB+14] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *FCCM*, 2014.

[CCA+13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. Brown, and J. Anderson. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.*, 2013.

[CCP+16] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *MICRO*, 2016.

[CIK+19] F. Chaix, A. Ioannou, N. Kossifidis, N. Dimou, G. Ieronymakis, M. Marazakis, V. Papaefstathiou, V. Flouris, M. Ligerakis, G. Ailamakis, T. Vavouris, A. Damianakis, M. Katevenis, and I. Mavroidis. Implementation and impact of an ultra-compact multi-fpga board for large system prototyping. In *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 34–41, 2019.

[CLN+11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[CSPJ03] S. Choi, R. Scrofano, V. Prasanna, and J. Jang. Energy-efficient Signal Processing Using FPGAs. In *FPGA*, 2003.

[CSZ+14] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. Enabling FPGAs in the Cloud. In *CF*, 2014.

[DPP02] M. Dyer, C. Plessl, and M. Platzner. Partially Reconfigurable Cores for Xilinx Virtex. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *FPL*, 2002.

[EMHK16] M. Eckert, D. Meyer, J. Haase, and B. Klauer. Operating System Concepts for Reconfigurable Computing: Review and Survey. *International Journal of Reconfigurable Computing*, 2016, 2016.

[FVS15] S. A. Fahmy, K. Vipin, and S. Shreejith. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *CloudCom*, Nov 2015.

[GAPK16] Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner. Spector: An OpenCL FPGA benchmark suite. In *FPT*, 2016.

[GBLV12] L. Gantel, M. E. A. Benkhelifa, F. Lemonnier, and F. Verdier. Module relocation in Heterogeneous Reconfigurable Systems-on-Chip using the Xilinx Isolation Design Flow. In *ReConFig*, 2012.

[GBM+19] K. Georgopoulos, K. Bakanov, I. Mavroidis, I. Papaefstathiou, A. Ioannou, P. Malakonakis, K. Pham, D. Koch, and L. Lavagno. *A Novel Framework for Utilising Multi-FPGAs in HPC Systems*, pages 153–170. Taylor and Franchis Group, 2019.

[GGNW19] D. Glick, J. Grigg, B. Nelson, and M. Wirthlin. Maverick: A Stand-Alone CAD Flow for Partially Reconfigurable FPGA Modules. In *FCCM*, 2019.

[GK15] N. Grigore and D. Koch. Placing Partially Reconfigurable Stream Processing Applications on FPGAs. In *FPL*, 2015.

[Gli20] D. Glick. *Maverick: A Stand-Alone CAD Flow for Partially Reconfgurable FPGA Modules*. PhD thesis, Brigham Young University, 2020.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. JBits: Java based Interface for Reconfigurable Computing. In *MAPLD*, 1999.

[GPK18] B. Gottschall, T. Preusser, and A. Kumar. Reloc - An Open-Sourced Vivado Workflow for Generating Relocatable, Out-Of-Context End-User Configuration Tiles. https://github.com/bgottschall/reloc, 2018.

[Gra18] Grand View Research, Inc. Field Programmable Gate Array (FPGA) Market Analysis By Technology (SRAM, EEPROM, Antifuse, Flash), By Application (Consumer Electronics, Automotive, Industrial, Data Processing, Military & Aerospace, Telecom), And Segment Forecasts, 2018 - 2024. https://www.grandviewresearch.com/industry-analysis/fpga-market, 2018.

[GRE18] I. Giechaskiel, K. Rasmussen, and K. Eguro. Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires. In *ASIACCS*, 2018.

[HKT11] S. G. Hansen, D. Koch, and J. Torresen. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro. In *IPDPS*, 2011.

[HLK02] E. Horta, J. Lockwood, and S. Kofuji. Using PARBIT to Implement Partial Run-Time Reconfigurable Systems. In *FPL*, 2002.

[HNH15] T. Haroldsen, B. Nelson, and B. Hutchings. RapidSmith 2: A Framework for BEL-level CAD Exploration on Xilinx FPGAs. In *FPGA*, 2015.

[HSPK17] E. Horta, X. Shen, K. Pham, and D. Koch. Accelerating Linux Bash Commands on FPGAs Using Partial Reconfiguration. In *FSP*, 2017.

[Huf] J. Huffstetler. Intel Processors and FPGAs – Better Together. https://itpeernetwork.intel.com/intel-processors-fpga-better-together/#gs.vaugnk.

[HW13] E. Hung and S. Wilton. Towards Simulator-like Observability for FPGAs: A Virtual Overlay Network for Trace-buffers. In *FPGA*, 2013.

[IMG+19] A. Ioannou, P. Malakonakis, K. Georgopoulos, I. Papaefstathiou, A. Dollas, and I. Mavroidis. Optimized fpga implementation of a compute-intensive oil reservoir simulation algorithm. In D. Pnevmatikatos, M. Pelcat, and M. Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 442–454, Cham, 2019. Springer International Publishing.

[JCP+10] S. Jin, J. Cho, X. D. Pham, K. M. Lee, S. Park, M. Kim, and J. W. Jeon. FPGA Design and Implementation of a Real-Time Stereo Vision System. *IEEE Transactions on Circuits and Systems for Video Technology (TCSVT)*, 20(1):15–26, Jan 2010.

[JdLLS+15] P. Jääskeläinen, C. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. POCL: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, 43(5):752–785, Oct 2015.

[JRHK15] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner. RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators. *TRETS*, 8(4), 2015.

[KAB+18] M. Katevenis, R. Ammendola, A. Biagioni, P. Cretaro, O. Frezza, F. Lo Cicero, A. Lonardo, M. Martinelli, P. Paolucci, E. Pastorelli, F. Simula, P. Vicini, G. Taffoni, J. Pascual, J. Navaridas, M. Lujan, J. Goodacre, B. Lietzow, A. Mouzakitis, N. Chrysos, M. Marazakis, P. Gorlani, S. Cozzini, G. Brandino, P. Koutsourakis, J. van Ruth, Y. Zhang, and M. Kersten. Next generation of exascale-class systems: Exanest project and the status of its interconnect and storage development. *Microprocessors and Microsystems*, 61:58 – 71, 2018.

[KBL13] D. Koch, C. Beckhoff, and G. Lemieux. An Efficient FPGA Overlay for Portable Custom Instruction Set Extensions. In *FPL*, 2013.

[KBT07] D. Koch, C. Beckhoff, and J. Teich. Bitstream Decompression for High Speed FPGA Configuration from Slow Memories. In *FPT*, 2007.

[KBT08] D. Koch, C. Beckhoff, and J. Teich. Recobus-builder - A Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In *FPL*, 2008.

[KGS17] O. Knodel, P. Genssler, and R. Spallek. Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures. *Reconfigurable Architectures, Tools and Applications*, 2, 2017.

[KGT19] J. Krautter, D. Gnad, and M. Tahoori. Mitigating Electrical-level Attacks Towards Secure Multi-Tenant FPGAs in the Cloud. *ACM Trans. Reconfigurable Technol. Syst.*, 12(3), August 2019.

[KHR] KHRONOS Group. OpenCL Overview. https://www.khronos.org/opencl.

[KLP+18] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AMORPHOS. In *OSDI*, 2018.

[KLPR05] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *IPDPS*, 2005.

[Koc12] D. Koch. *Partial Reconfiguration on FPGAs: Architecture, Tools, and Applications*. 2012.

[KS16] A. Kulkarni and D. Stroobandt. How to Efficiently Reconfigure Tunable Lookup Tables for Dynamic Circuit Specialization. *International Journal of Reconfigurable Computing*, 2016, 2016.

[KVS+16] A. Kulkarni, E. Vansteenkiste, D. Stroobandt, A. Brokalakis, and A. Nikitakis. A Fully Parameterized Virtual Coarse Grained Reconfigurable Array for High Performance Computing Applications. In *IPDPS*, 2016.

[LAB+05] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose. The Stratix II Logic and Routing Architecture. In *FPGA*, 2005.

[LBM+06] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *FPL*, 2006.

[LGW+14] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2), July 2014.

[LK18] C. Lavin and A. Kaviani. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *FCCM*, 2018.

[LMW+07] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *MSE*, June 2007.

[LPL+14] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, and M. Wirthlin. RAPIDSMITH - A Library for Low-level Manipulation of Partially Placed-and-Routed FPGA Designs. Technical report, Brigham Young University, 2014.

[MGF+16] M. Marazakis, J. Goodacre, D. Fuin, P. Carpenter, J. Thomson, E. Matus, A. Bruno, P. Stenstrom, J. Martin, Y. Durand, and I. Dor. Euroserver: Share-anything scale-out micro-server design. In *DATE*, 2016.

[MLG+20] K. Matas, T. La, N. Grunchevski, K. Pham, and D. Koch. Invited tutorial: Fpga hardware security for datacenters and beyond. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA'20, pages 11–20, New York, NY, USA, 2020. Association for Computing Machinery.

[MMRL17] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access*, 5:2747–2762, 2017.

[MPL+16] I. Mavroidis, I. Papaefstathiou, L. Lavagno, D. Nikolopoulos, D. Koch, J. Goodacre, I. Sourdis, V. Papaefstathiou, M. Coppola, and M. Palomino. ECOSCALE: Reconfigurable Computing and Runtime System for Future Exascale Systems. In *DATE*, 2016.

[MTAB07] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. 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):15–31, Apr 2007.

[NIS01] NIST. Security Requirements for Cryptographic M. https://csrc.nist.gov/publications/detail/fips/140/2/final, 2001.

[NK16] T. Nguyen and A. Kumar. Prfloor: An automatic floorplanner for partially reconfigurable fpga systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA'16, pages 149–158, New York, NY, USA, 2016. Association for Computing Machinery.

[NR08] J. Note and E. Rannaud. From the Bitstream to the Netlist. In *FPGA*, 2008.

[oE10] U. S. Department of Energy. The opportunities and challenges of exascale computing. Technical report, 2010.

[PA12] O. Pell and V. Averbukh. Maximum Performance Computing with Dataflow Engines. *Computing in Science Engineering*, 14(4):98–103, July 2012.

[PCC+14] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA*, 2014.

[PHK17] K. D. Pham, E. Horta, and D. Koch. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *DATE*, 2017.

[PHK+18] K. D. Pham, E. L. Horta, D. Koch, A. Vaishnav, and Thomas Kuhn. IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs. In *MCSoC*, 2018.

[PPV+19] K. D. Pham, K. Paraskevas, A. Vaishnav, A. Attwood, M. Vesper, and D. Koch. ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs. In *FSP*, 2019.

[PVKH19] K. D. Pham, M. Vesper, D. Koch, and E. Hung. EFCAD – An Embedded FPGA CAD Tool Flow for Enabling On-chip Self-Compilation. In *FCCM*, 2019.

[PVVK18] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *FSP*, 2018.

[QRDC+15] H. Quinn, D. Roussel-Dupre, M. Caffrey, P. Graham, M. Wirthlin, K. Morgan, A. Salazar, T. Nelson, W. Howes, E. Johnson, J. Johnson, B. Pratt, N. Rollins, and J. Krone. The Cibola Flight Experiment. *TRETS*, 2015.

[Red09] K. Red. *Single Event Upsets in SRAM FPGA based readout electronics for the Time Projection Chamber in the ALICE experiment*. PhD thesis, The University of Bergen, Bergen, Norway, 2009.

[RFG16] J. Rettkowski, K. Friesen, and D. Goehringer. RePaBit: Automated Generation of Relocatable Partial Bitstreams for Xilinx Zynq FPGAs. In *ReConFig*, 2016.

[RPD+18] C. Ramesh, S. Patil, S. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier. FPGA Side Channel Attacks without Physical Access. In *FCCM*, 2018.

[RS02] A. Raghavan and P. Sutton. JPG - A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs. In *IPDPS*, 2002.

[SAFW11] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood. OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In *IPDPSW*, 2011.

[SFK+17] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar. Interdiction in practice—Hardware Trojan against a high-security USB flash drive. *Journal of Cryptographic Engineering*, 7(3):199–211, Sep 2017.

[SHW+19] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *FCCM*, 2019.

[SVC+16] D. Stroobandt, A. L. Varbanescu, C. B. Ciobanu, M. Al Kadi, A. Brokalakis, G. Charitopoulos, T. Todman, X. Niu, D. Pnevmatikatos, A. Kulkarni, E. Vansteenkiste, W. Luk, M. D. Santambrogio, D. Sciuto, M. Huebner, T. Becker, G. Gaydadjiev, A. Nikitakis, and A. J. W. Thom. EXTRA: Towards the Exploitation of eXascale Technology for Reconfigurable Architectures. In *ReCoSoC*, 2016.

[Sym] SymbiFlow. Project X-Ray - Xilinx Series 7 Bitstream Documentation. https://github.com/SymbiFlow/prjxray.git.

[TLF+17] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow. Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *FPGA*, 2017.

[Uni19] Princeton University. Princeton Reconfigurable Gate Array, 2019.

[Ves18] M. Vesper. *Dynamic Stream Processing Pipelines on FPGAs Examplified on the PostGreSQL DBMS*. PhD thesis, The University of Manchester, 2018.

[VF14a] K. Vipin and S. A. Fahmy. DyRACT: A Partial Reconfiguration Enabled Accelerator and Test Platform. In *FPL*, 2014.

[VF14b] K. Vipin and S. A. Fahmy. ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq. *IEEE Embedded Systems Letters*, 2014.

[VF18] K. Vipin and S. A. Fahmy. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Comput. Surv.*, 51(4):72:1–72:39, July 2018.

[VFBS14] E. Vansteenkiste, B. A. Farisi, K. Bruneel, and D. Stroobandt. TPaR: Place and Route Tools for the Dynamic Reconfiguration of the FPGA's Interconnect Network. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(3):370–383, 2014.

[VKP17] M. Vesper, D. Koch, and K. D. Pham. PCIeHLS: an OpenCL HLS framework. In *FSP*, 2017.

[VKVF16] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy. JetStream: An Open-source High-performance PCI Express 3 Streaming Library for FPGA-to-Host and FPGA-to-FPGA Communication. In *FPL*, 2016.

[VOK17] A. Vaishnav, J. R. G. Ordaz, and D. Koch. A Security Library for FPGA Interlays. In *FPL*, 2017.

[VPK18a] A. Vaishnav, K. D. Pham, and D. Koch. A Survey on FPGA Virtualization. In *FPL*, 2018.

[VPK18b] A. Vaishnav, K. D. Pham, and D. Koch. Live Migration for OpenCL FPGA Accelerators. In *FPT*, 2018.

[VPK19] A. Vaishnav, K. D. Pham, and D. Koch. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *HEART*, 2019.

[VPKG18] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. Resource Elastic Virtualization for FPGAs using OpenCL. In *FPL*, 2018.

[VPMK19] A. Vaishnav, K. D. Pham, K. Manev, and D. Koch. The fos (fpga operating system) demo. In *FPL*, pages 429–429, 2019.

[VPPK20] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch. Fos: A modular fpga operating system for dynamic workloads, 2020.

[VVS17] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. Liquid: High Quality Scalable Placement for Large Heterogeneous FPGAs. In *FPT*, 2017.

[VVS19] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. CRoute: A Fast High-Quality Timing-Driven Connection-Based FPGA Router. In *FCCM*, 2019.

[vZ13] D. van den Heuvel and R. Zenden. Middleware Turns Zynq SoC into Dynamically Reallocating Processing Platform. *Xcell*, 85, 2013.

[WAHH15] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. In *UIC-ATC-ScalCom*, Aug 2015.

[WBP13] W. Wang, M. Bolic, and J. Parri. pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment. In *CODES+ISSS*, 2013.

[Wir15] M. Wirthlin. High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond. *Proceedings of the IEEE*, 2015.

[WPAH16] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner. Network-Attached FPGAs for Data Center Applications. In *FPT*, 2016.

[Xila] Xilinx. PlanAhead Design and Analysis Tool. https://www.xilinx.com/products/design-tools/planahead.html.

[Xilb] Xilinx. PYNQ. http://www.pynq.io.

[Xilc] Xilinx. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[Xild] Xilinx. SDSoC Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

[Xile] Xilinx. Solution ZynqMP PL Programming. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841847/Solution+ZynqMP+PL+Programming.

[Xilf] Xilinx. Xilinx Runtime (XRT) Architecture. https://xilinx.github.io/XRT/master/html/index.html.

[Xilg] Xilinx. Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/vcu118.html.

[Xilh] Xilinx. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[Xil09] Xilinx. *UG621 - Virtex 5 Libraries Guide for HDL Designs*. 2009.

[Xil14] Xilinx. *UG910 - Vivado Design Suite User Guide*. 2014.

[Xil15a] Xilinx. *UG470 - 7 Series FPGAs Configuration User Guide*. 2015.

[Xil15b] Xilinx. *UG570 - UltraScale Architecture Configuration User Guide*. 2015.

[Xil16a] Xilinx. Axi hwicap v3.0: Logicore ip product guide, 2016.

[Xil16b] Xilinx. Partial Reconfiguration Decoupler v1.0. 2016.

[Xil16c] Xilinx. XAPP1222 - Isolation Design Flow for Xilinx 7 Series FPGAs or Zynq-7000 AP SoCs (Vivado Tools). 2016.

[Xil16d] Xilinx. XAPP1256 - Zynq-7000 AP SoCs or 7 Series FPGAs Isolation Design Flow Lab (Vivado Design Suite). 2016.

[Xil17a] Xilinx. Reconfigurable Acceleration in the Cloud. https://www.xilinx.com/products/design-tools/cloud-based-acceleration.html#alibaba, 2017.

[Xil17b] Xilinx. Zynq ultrascale+ mpsoc data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2017.

[Xil18a] Xilinx. 7 Series FPGAs Clocking Resources - v1.14. 2018.

[Xil18b] Xilinx. UG908 - Vivado Design Suite User Guide: Programming and Debugging. June 2018.

[Xil18c] Xilinx. UG909 - Vivado Design Suite User Guide Partial Reconfiguration. April 2018.

[Xil18d] Xilinx. UltraScale Architecture Clocking Resources - v1.8. 2018.

[Xil18e] Xilinx. Zynq-7000 soc: Technical reference manual. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2018.

[Xil19a] Xilinx. Versal: The first adaptive compute acceleration platform (acap). https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf, 2019.

[Xil19b] Xilinx. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2019.

[Xil20] Xilinx. Vitis: Unified software platform for all developers, 2020.

[YB18] S. Yazdanshenas and V. Betz. Interconnect Solutions for Virtualized Field-Programmable Gate Arrays. *IEEE Access*, 6, 2018.

[YKL15] M. X. Yue, D. Koch, and G. G. F. Lemieux. Rapid Overlay Builder for Xilinx FPGAs. In *FCCM*, 2015.

[ZHA+17] Q. Zhao, Hendarmawan, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi. hCODE 2.0: An Open-source Toolkit for Building Efficient FPGA-enabled Clouds. In *FPT*, 2017.

[ZNA+18] Z. Zhao, N. Nguyen, D. Agiakatsikas, G. Lee, E. Cetin, and O. Diessel. Fine-Grained Module-Based Error Recovery in FPGA-Based TMR Systems. *ACM Trans. Reconfigurable Technol. Syst.*, 11(1), January 2018.

[ZXX+17] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda. The Feniks FPGA Operating System for Cloud Computing. In *APSys*, 2017.

Appendix A

IPRDF

Throughout this PhD project, we had been contacted by the company HTV Halbleiter-Test & Vertriebs-GmbH, Bensheim, Germany to implement a secure reconfigurable system that fulfils module isolation requirements [NIS01]. This was carried out by simply modifying the constraints generated by GoAhead [BKT12] to incorporate a fence and to ensure that no signals will be routed through that fence except for direct connections between reconfigurable accelerator modules. These are just minor additional constraints to the compilation scripts used for this PhD project but help to solve a real-world embedded system design problem. A distinct feature of that system is that partially reconfigurable modules include I/O pins. This feature is not supported in the Xilinx FPGA vendor toolchain [Xil18c] yet is essential in this demonstrator to implement module encapsulation.

A.1 Introduction

There are strong needs to use FPGAs in other application domains such as in automotive, aerospace, defense, cyber-security and in hazardous environments (e.g., in space [QRDC+15] or high-energy physics [Red09]) which often require secure and safety-critical system implementations. However, these applications are challenging to implement with SRAM-based FPGAs. For instance, when FPGAs are working in space, an ionized particle hit can cause a single-event upset (SEU), resulting in a bit-flip in one or more configuration cells or a communication signal and consequently in catastrophic behavior [Wir15]. Moreover, in information assurance applications, hardware modules must be independently and solitarily implemented in a single chip to satisfy isolation, reliability, and security concerns[Xil16c].

Finally, with the trend of using FPGAs as accelerators in cloud environments[BSB+14, Xil17a], physical insulation of reconfigurable modules may become important for granting

multiple users simultaneous access to the same FPGA device. For example, recent studies [GRE18, RPD+18] have demonstrated how side channel attacks using delays on long wires can allow leaking information from FPGA cryptographic modules *without physical access*. However, a physical isolation fence between wires of different modules can reduce side-channel effects significantly [GRE18], and therefore can eliminate the risk of data leakage through this sort of attack.

Xilinx, a major FPGA vendor, addresses these concerns by providing an Isolation Design Flow (IDF)[Xil16c], which comprises the following requirements:

- Each hardware module has to be isolated and must be in its own level of hierarchy.
- A fence must be used to separate isolated hardware modules within a single chip. Further, it is not allowed to use any primitive or routing resources in any directly adjacent tile (e.g., a Configurable Logic Block (CLB) or Block RAM (BRAM)). Therefore, a fence is at least one tile wide.
- Input/output buffers (IOBs) must be instantiated *inside* isolated modules for proper isolation of the IOBs. Thus, having full control over the routing of the signals from IOBs to the module is essential to establish off-chip trusted communications.
- On-chip communication between isolated hardware modules is achieved through the use of Trusted Routing, which has to follow restrictions:
  - There is no entry or exit point in the fence between isolated regions.
  - There is one source and one destination for each routing path (only point-to-point routing).
  - The entry and the exit points must stay in the source and destination regions, respectively.
  - Its entirety stays contained in the source/destination regions
  - It does not touch a fence tile from another isolation region.

Xilinx IDF supports the implementation of isolated modules which satisfy all mentioned requirements as well as an automatic mechanism to verify module isolation using the Xilinx Vivado Isolation Verifier (VIV). However, even the latest Vivado tool suite (version 2018.1) does not provide any partial reconfiguration (PR) capability together with IDF.

Although isolation design is not possible with the Xilinx Partial Reconfiguration (PR) flow as clocks and IOBs must remain in the static logic part[Xil18c] when using 7-Series and earlier devices, the redundancy needed for safety-critical systems and the requirements in the physical implementation of these systems would strongly benefit from partial reconfiguration (PR). For

Table A.1: Isolation Design Flows' features and supports.

Features	Xilinx IDF[Xil18c]	Altera SDF[Alt09]	Academic Customized IDF[GBLV12, RFG16]	*IPRDF*
FPGA families	Xilinx	Altera	Xilinx	Xilinx
Isolated modules	✓	✓	✓	✓
Fences between modules	✓	✓	✓	✓
IOB assignments	automatic	automatic	?	manual
Secured communication	✓	✓	✓	✓
Partial reconfiguration	✗	✗	✓	✓
Support for module relocation, multiple module instantiation	✗	✗	✓	✓
Direct communication between PR modules	✗	✗	✗	✓
Off-chip trusted communications in PR regions	✗	✗	✗	✓

example, the fences around modules or the redundant instances of modules for TMR put more pressure on resource utilization which can be in some cases mitigated with the help of PR. Furthermore, cryptographic military systems often use multiple different cyphers (e.g., DES, AES, Blowfish) in a round-robin fashion which can be implemented in a resource efficient manner using PR. Moreover, PR is an effective countermeasure against single-event upsets (SEUs) by correcting configuration swiftly. Therefore, by integrating a PR flow into IDF, we are able not only to mask a SEU, but also to recover the malfunctioned area at run-time.

To implement this, we propose an alternnative flow, named *Isolated Partial Reconfiguration Design Flow (IPRDF)*, which applies partial reconfiguration (PR) design practices into the isolation design flow (IDF). Moreover, we provide a design rule check to ensure that our approach fulfills all requirements for a standard isolation design flow as specified in[Xil16c]. In addition to the Xilinx vendor IDF, systems designed by our flow are partially reconfigurable. This enables designing self-aware and fault-tolerant mechanisms, which gives systems higher probability to detect and mask errors by moving relocatable modules to other regions for alleviating defects in the FPGA fabric. Thus, systems designed by the proposed *IPRDF* use less resources (i.e. are cheaper), are potentially less vulnerable, more reliable, and more suitable in secure and safety-critical applications than static counterparts.

IPRDF includes several phases, which are realized by a combination of commercial Xilinx Vivado[Xil18b], open source tools such as GoAhead[BKT12] and BitMan[PHK17] as well as our own tools and scripts adding the isolation capability to reconfigurable systems. In detail, placement and routing for static and partial designs are constrained by GoAhead, and are physically implemented by Vivado. Moreover, full bitstreams are generated by Vivado, but are later manipulated by BitMan to compose relocatable partial bitstreams.

Additionally, we present two case studies: 1) an isolated Triple Modular Redundancy (TMR) system and 2) a single-chip cryptographic (SCC) design that both run on a ZedBoard using a Xilinx *XC7Z020* FPGA to demonstrate capabilities of the proposed *IPRDF* tool flow.

The contributions of this work include:

- A design flow which guarantees isolated modules, trusted communication channels and separated IOBs for partial reconfigurable modules (Section A.3).
- Trusted regions which are partially reconfigurable and able to host multiple relocatable modules in time and space manners (Section A.3).
- Error detection and recovering techniques with considering isolation as required for TMR (Section A.4).
- Two case studies on *IPRDF* (Section A.4 and Section A.5).

Further sections include an overview on related work in Section A.2 and a conclusion in Section A.6.

A.2 Related Work

A.2.1 Isolation Design Flows

The major FPGA vendors have introduced proprietary isolation flows such as Xilinx Isolation Design Flow (IDF)[Xil16c], and Altera Separation Design Flow (SDF)[Alt09]. However, as mentioned in Section A.1, they lack partial reconfiguration capability.

Related research papers [GBLV12, RFG16] used Xilinx IDF to design relocatable modules. Those works, however, are not targeting design isolation as needed to implement secure or reliable systems. Instead, some IDF mechanisms were used to prevent static system routing to cross partial regions. This property allowed those approaches to relocate modules. However, none of those related approaches could support off-chip trusted communications for partial regions.

Comparisons of the proposed *IPRDF* and other state-of-the-art tool flows are compared in Table A.1. As we can realize, the limitation of *IPRDF* is that IOB assignment has to be carried out manually[1].

A.2.2 Partial Reconfiguration Tools

There are several partial reconfiguration (PR) design flows, available both from industry players such as Xilinx[Xil18c], Altera[Alt17] and from academia, for instance, OpenPR[SAFW11], and GoAhead[BKT12]. OpenPR and GoAhead can generate blocker macros that allow to prohibit the Xilinx vendor router to use a defined set of wires only (e.g., this allows it to implement an IDF conform fence around a module). When physically implementing a module, blocker macros will occupy all possible connections to and from modules. However, we include tunnels into these blockers to carry out the top-level routing. The here presented *IPRDF* is a frontend for the GoAhead tool because it is the only academic PR tool that is currently supporting latest FPGAs from Xilinx.

A.2.3 Designing for Reliability

Wirthlin in [Wir15] summarized common design practices for high reliable FPGA systems including hardware redundancy, configuration scrubbing, error-correction coding, flip-flop mitigation, and system-level mitigation of FPGA single-event effects (SEEs). That work highlights the importance of combinations of hardware redundancy, especially Triple-Modular Redundancy (TMR), and configuration scrubbing as a recovering mechanism in systems being used

[1] However, this process is relatively easy to carry out and is normally done before the PCB design (which implies that designing secure systems include aspects beyond the actual FPGA design).

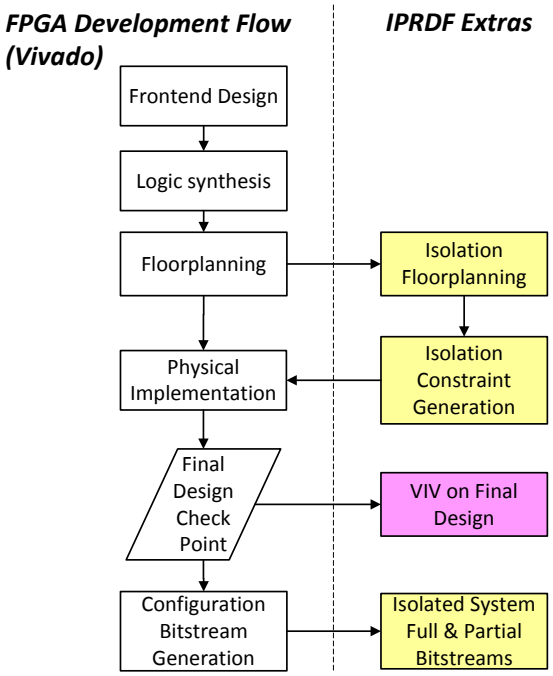


Figure A.1: Isolated Partial Reconfiguration Design Flow *(IPRDF)*.

in satellites [QRDC+15]. Moreover, Abramovici et al. proposed the idea of rotating functional units via PR for testing and repairing in [AS03].

A.3 The *IPRDF* Flow

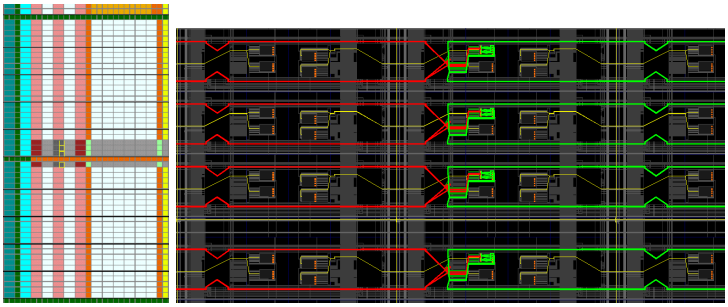
In Section A.4, we will demonstrate a combination of TMR, isolation design and partial reconfiguration that can act as a recovering scheme for highly available and secure systems.

In this section, *IPRDF* is presented step-by-step. The flow is described in Subsection A.3.1. Static and Partial designs are presented in Subsection A.3.2 and Subsection A.3.3.

A.3.1 Overview

We are using standard Xilinx Vivado for our front-end design and logic synthesis. This allows us to take advantages from all input specification methods that are available in Vivado including RTL, schematic entry, or even High Level Synthesis (HLS). *IPRDF* requires to carry out more steps in the floorplanning stage which will consecutively affect physical implementation as well as bitstream generation. The flow in comparison with the default Xilinx Vivado flow is presented as in Figure A.1 from a developer's point of view.

IPRDF's steps are described as follows:



(a) (b)

Figure A.2: An example of a 16-bit bus for system communication when using physical constraint generation shown in (a) GoAhead and in (b) after routing by Vivado. Note that red signals are highlighting inputs while green signals are highlighting outputs.

- **Isolation Floorplanning:** based on the resource utilization retrieved from synthesis reports, we define regions on the FPGA, either to host reconfigurable modules (in the static system) or to implement reconfigurable modules (partial modules). In module floorplanning, an automatic placement exploration[GK15] starts finding all possible positions. Bounding boxes according to all those positions are generated without user's operation. Static system's floorplanning is done manually but assisted by our tool flow by an automatic check that ensures that those regions provide the necessary number of resources, even if some resources are not allowed to be used due to module isolation.

- **Isolation Constraint Generation:** position and bounding box information is then used for static and partial designs. Physical constraints for placement and routing are generated by the GoAhead tool. These constraints are written into TCL files, which are then used by Vivado to guide the physical implementation stage.

  IPRDF adds rules to this process to match the IDF requirements (see Section A.1). This is implemented through scripts that are written for GoAhead.

- **Final Design Check Point:** Design Check Points from previous steps will be ran through the Xilinx Vivado Isolation Verifier (VIV) to ensure that our designs comply with the isolation design rules.

- **Configuration Bitstream Generation:** *IPRDF* results in full bitstreams of static and module designs. To compose partial bitstreams for modules, we use the tool BitMan [PHK17].

A.3.2 Static Design

The implementation of the static design starts with a floorplanning step where we define the placement of the static system components, communication infrastructure and reserved areas

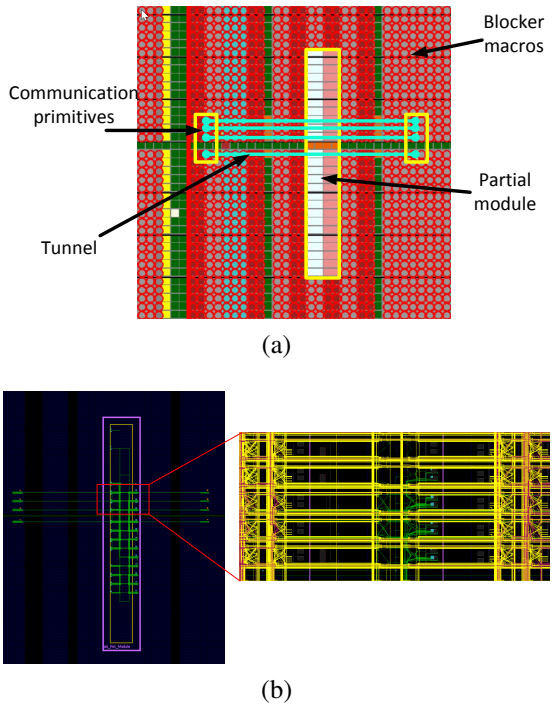


Figure A.3: Module placement, communication tunnels, and blockers for the selected partial module in (a) GoAhead and in (b) after routing by Vivado. Note that green signals are highlighting module's wires while yellow and red signals are highlighting blocker hard-macros.

for the partial modules. With these parameters, we instruct GoAhead to create top-level routing and placement constraints as TCL scripts for Vivado.

The goal at this stage is to define a region which hosts partial modules. In order to leave as many resources as possible for the actual application, we aim at maximizing this area. In the case of using *IPRDF* for TMR systems, we define 3 regions of identical size (and to be more precise, regions where the relative layout of primitive columns (e.g., CLBs, BRAMs) is identical[2]). A blocker macro is then generated to prohibit any routing or logic resource to be used by reconfigurable modules.

The blocker macro will prevent all FPGA primitives and routing resources to be used in the selected partial region. Therefore, we will leave holes in the blocker macro (called tunnels) that are used to constrain module interface signals to specific wire resources on the FPGA following isolation rules. See Figure A.2 for a 16-bit bus example.

[2]It is not mandatory that all TMR regions are identical because it is possible to generate different module implementations for each region. However, by using identical regions, we could even share the place and route result including the final partial bitstream.

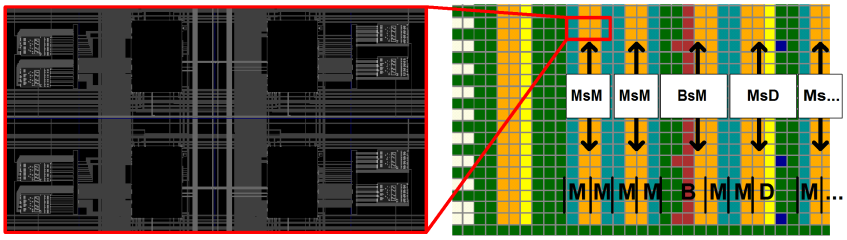


Figure A.4: Partial slots with various FPGA primitive slots such as *MsM*, *BsM* and *MsD*. The left screenshot is from the Vivado and the right screenshot is taken from the GoAhead's floorplanning GUI.

A.3.3 Module Design

The implementation of the partial module design also starts with a floorplanning step which includes placing communication primitives around the partial module. These macros act as sink/source connection points and substitute the surrounding static system. Blockers will be generated to prohibit all partial module's primitives and routing resources. These blockers will be placed around the selected area, hence, acting as a fence to implement strict module bounding boxes as well as the isolation fence. Routing tunnels are included for the communication to and from the primitives. The position of these tunnels match exactly the tunnels as used in the static design to implement the communication between static and partial areas.

The result of this stage is shown in Figure A.3. As we can see, blockers are placed around the partial module to ensure all primitives must be used inside the bounding box area.

As a module is implemented in a separated design from the static system, the final results generated by Vivado is a full configuration bitstream. This data is passed on to BitMan that cuts out the configuration data that corresponds to the module only. We repeat these steps for all modules to build a partial module library.

At run-time, BitMan manipulates those partial bitstreams to relocate modules to a desired position inside a partial region of the static system.

A.4 Case Study I: Triple Modular Redundancy

In this example, a TMR system, as in Figure A.5, will be designed on a ZedBoard to demonstrate our *IPRDF*'s capability. This system includes 3 video background generators, 3 video streaming channels, and a *Quality Assurance Unit*, which contains a *Majority Voter* and a *Configuration Controller*. All these components are implemented isolated from each other by physical fences.

Partial regions are tiled into multiple adjacent slots that are two resource columns (CLB, BRAM, or DSP) wide. Modules are one or more slots wide. Implemented modules include a

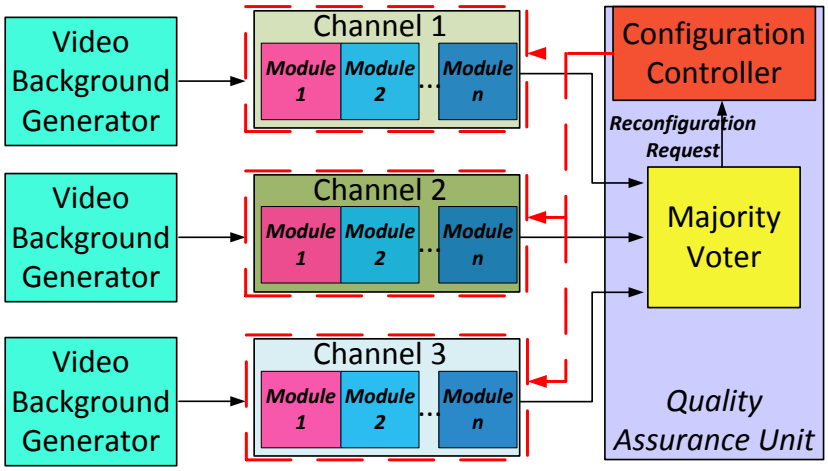


Figure A.5: Block diagram of the TMR system.

video overlay generator, a DES encryption, and a SHA1 hash function. A module could have multiple alternatives as it might be placed on different primitives (e.g., one alternative providing a RAM column in the left and another providing RAM in the right half of the module). For TMR operation, all channels must host the same modules in the same order to guarantee fully redundant execution over all channels.

Outputs from video streaming channels will be routed to the *Majority Voter* inside the *Quality Assurance Unit*. This *Majority Voter* will guarantee that any SEU in any channel would not impact the final system's output. Moreover, a single difference in a channel's output will trigger the *Configuration Controller* to dynamically partially reconfigure this specific channel to mitigate any SEU effect with a low guaranteed latency. The *Configuration Controller* can stay either on-chip by utilizing reconfigurable ports such as *ICAP* and *PCAP*, or off-chip in a host machine and using *JTAG* port to reconfigure the FPGA fabric[3]. This redundant system with repair mitigates against multiple upsets which may occur and potentially impact the TMR outputs.

This case study has ultimately extended the system in [ZNA+18] with additional physical isolation fences and trusted communication between modules. These implementation practices are essential as an SEU may otherwise impact two TMR instances at the same time with eventually fatal consequences.

Specific information of this example's implementation is provided in Subsection A.4.1. Subsection A.4.2 describes how the error detection and recovering schemes work. In Subsection A.4.3, the fully implemented system is presented, and achieved goals for this isolated design are discussed.

[3]In this case study, we are using the *PCAP* port that is controlled by the available ARM core only. For full fault tolerant operation, it would need another port (e.g., *JTAG*) to remove a single point of failure or a watchdog mechanism.

Table A.2: Available primitives on various resource slots and required elements for different modules.

Resource Slot	Region Size		Available Resources		
	Columns	Rows	LUTs	BRAMs	DSPs
MsM	2	46	736	0	0
BsM	2	46	368	8	0
MsD	2	46	368	0	16
Module	**Module Size**		**Required Resources**		
	Columns	Rows	LUTs	BRAMs	DSPs
Video Overlay Generator	2	46	207	0	0
2-round DES in CRC mode [VOK17]	2	46	226	0	0
2-Stage SHA1 [VOK17]	2	46	235	0	0
3-Stage SHA1 [VOK17]	2	46	295	0	0

A.4.1 System Implementation

FPGA resources on the *XC7Z020* device are aligned column-wise. We represent the relative layout of primitive columns by a *Resources String* which is simply a string of symbols which denote the particular column types, as introduced in Chapter 3. For example, starting from the bottom left corner, we can model the FPGA with the device resource string *MsM MsM BsM MsD* ..., as shown in Figure A.4, with:

- *s*: two switch matrices between primitives (any *L M B* or *D* type),
- *L*: a CLB column providing *SLICE_L* primitives (supports only logic and arithmetic),
- *M*: a CLB column providing *SLICE_M* primitives (supports logic, arithmetic, and distributed memory),
- *B*: a column providing *BRAM* primitives,
- *D*: a columns providing *DSP48* primitives.

A resource slot (our smallest atomically reconfigurable unit) is defined by a primitive column followed by two switch matrices and another primitive column. To incorporate this in our string matching abstraction, we model our resource slots with symbols like *MsM* or *BsM* or *MsD* to indicate the different combinations of primitives that may exist for a resource slot, as illustrated in Figure A.4. A partial module with the type *MsM* can only be loaded into *MsM*-slots. Therefore, a logical module may need different implementations (e.g., *MsM*, *BsM*, or

MsD-compatible versions) if relocation is used. In this case study, each channel provides 6 resource slots.

To establish horizontal fences, we have reserved one CLB row at the top and another at the bottom of a resource slot. This action results in less available primitives within each resource slot to host a module, and this is an expected overhead when using module insulation.

Numbers of FPGA primitives in various slots and the required resources for different functional modules are listed in Table A.2.

As long as sufficient resources are found in the bounding box, horizontally physical fences at the top and bottom of the module can be established without much effort. However, routing of communication tunnels must be carefully analyzed and strictly constrained to a predefined set of wires that will carry out the top-level module communication through the fence. A physical fence following IDF rules[Xil16c] must leave all logical and routing primitives unused.

This means that at least one resource column must be left totally empty at each side of the module's borders. Therefore, when creating the communication infrastructure, we will use *double*-wires, which span a distance of 2 resource columns, or *quad*-wires, which jump from the current column to another one that is 4 columns away.

This ensures that we can bypass the switch matrices of the fence while still implementing all top-level signals. When using *double*-wires, this allows us to route up to four bit signals per CLB row and respectively 3×4 bit signals in the case of *quad*-wires. Considering the usable $50-2$ CLB rows, this allows wide interfaces of up to $(50-2) \times (4+3 \times 4) = 768$ bits in total per signal direction for a module that is one clock region in height on a Xilinx Zynq FPGA.

The reconfiguration overhead per resource slot is measured at 0.4 ms on average while the size of a partial bitstream is 59 kBytes and the PCAP throughput is 128 MBytes/second.

A.4.2 Error Detection and Recovering Schemes

At run-time, errors can be caused by SEUs or physical ageing and have to be mitigated differently. Therefore, we have developed two schemes to mask them, as shown in Figure A.6. In this work, we used the *Majority Voter* as an error detection unit. It will detect any mismatch happening over all channels' outputs, specify the malfunctioning channel, and send requests to the *Configuration Controller* for reconfiguration actions. These recovering schemes are implemented as a software application on the embedded ARM CPU and PCAP is used for FPGA reconfiguration.

SEU Recovering Scheme

In the first flow, as shown in Figure A.6a, any difference in a channel's output is detected by the *Majority Voter* and the error causing channel is recognized. A *Mismatch* counter will be

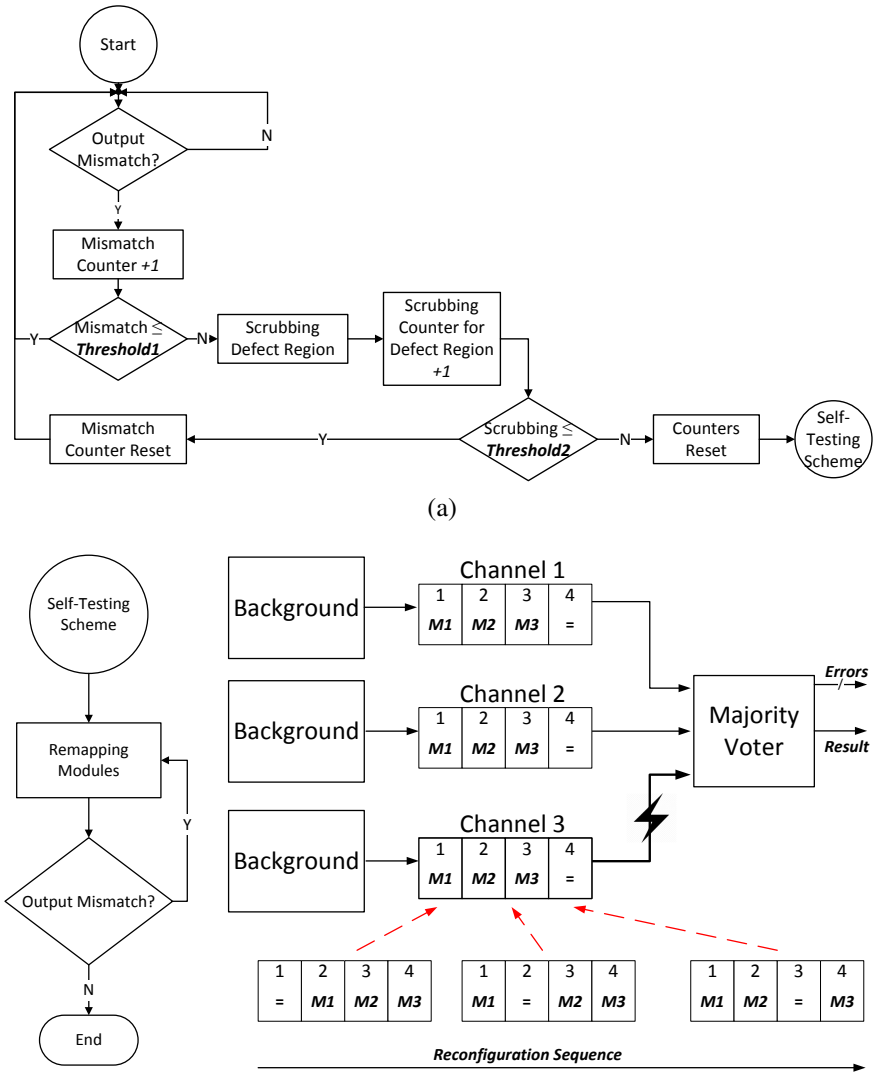


Figure A.6: Two different error detection and recovering schemes. (a) is to prevent impacts from single-event upsets (SEUs), and (b) is to reduce impacts from permanently physical damages such as ageing or device imperfections.

incremented. This allows distinguishing between SEUs on the datapath (transient errors) from SEUs hitting configuration SRAM cells. Former issues can be mitigated by the majority result from the *Voter* while later ones must be repaired by reconfiguring the modules of the impacted channel.

However, permanent FPGA defects cannot be recovered by configuration scrubbing regardless of reconfiguration efforts. Consequently, a *Scrubbing* counter for each defect channel will be incremented until it reaches a threshold which in turn triggers the second flow, called *Self-Testing Scheme*.

Self-Testing Scheme

The example in Figure A.6b shows three channels for TMR, each hosting three modules that are one slot wide. Each channel provides one spare slot that is bypassed (symbolized with '='). For *Self-Testing*, we generated a reconfiguration sequence such that the first configuration bypasses the first slot, the second configuration bypasses the second slot, etc., until we find a working sequence that is eventually recovering the defective slot.

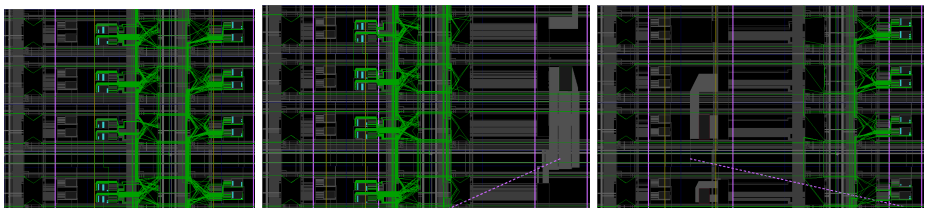
Self-Testing is carried out for a single channel (e.g., channel 3 in Figure A.6b) while using the other channels as a reference for testing correct operation. The actual configurations are composed by BitMan from relocatable modules which in some cases may involve implementation alternatives to deal with the heterogeneous layout of resource columns. Note that in this case study, each channel has 6 slots, and *Self-Testing* is being conducted in parallel to the operation of the system.

In general, the modules in the channels may have an internal state that would be out of synchronization after partial reconfiguration and it needs somehow a mechanism to resynchronize all TMR instances. In this case study working on a video stream, after each row of pixels (in our case, 1024 pixels), all modules start with the same initial state. We therefore wait after reconfiguration for at least this time before evaluating the *Majority Voter* output.

A.4.3 Result

Details of an implemented module designed by *IPRDF* are shown as in Figure A.7. The generated blocker macros ensure that all module primitives are placed inside the bounding box. Ultimately, no routing violation was found by the Vivado Isolation Verifier (VIV) when using our design methodology.

Moreover, with our design methodology, a module can be implemented in different resource slots. Thus, it is more flexible to place and relocate modules across the FPGA fabric in order to mitigate physical vulnerabilities that may happen during its lifetime. Figure A.7 shows implemented alternatives of the *Video Overlay Generator* in three resource slots providing different resource footprints. These three implemented alternatives are sufficient to place this two



(a) (b) (c)

Figure A.7: Implemented options of a *Video Overlay Generator* module, with (a) is the *MsM*-compatible option, (b) is the *MsD*-compatible option, and (c) is the *BsM*-compatible option.

column-wide module to any slot inside any reconfigurable region of the system.

The whole demonstration system is designed with *IPRDF* and is shown in Figure A.8. The static parts include the *Video Background Generator* and the *Quality Assurance Unit* which are physically isolated from other partial regions and from each other. In addition, partial channels are also separated by horizontal and vertical fences between system elements.

For recovering schemes, we tested the correctness of transient fault masking by injecting errors into channels' inputs via push buttons. Moreover, the mitigation technique from SEU in configuration data was verified by flipping random bits from a LUT table through partial reconfiguration.

Finally, to emulate permanent errors, we used a feature from BitMan that prohibits a defined resource (a LUT for our experiments) from reconfiguration. Considering modules that are one-slot wide and a channel which has s slots hosting m modules, the worst case time for the *Self-Testing* procedure is:

$$\binom{s}{m} \times (t_{config} \times s + t_{test}),$$

with t_{config} being the time to reconfigure one slot and t_{test} be the time for testing one configuration of the test sequence.

In this case study, reconfiguration overhead per slot was 0.4 ms on average, and the testing scheme took about 0.02 ms per configuration. The worst case was 14.5 ms, where resource verification needed to be conducted for 6 slots per channel.

The whole error mitigation was running in bare-metal on the ARM using PCAP partial reconfiguration. Please note that this case study cannot handle defects in the static system or on the communication channels used to integrate partially reconfigurable modules. This can be solved using the techniques described in [BKT14].

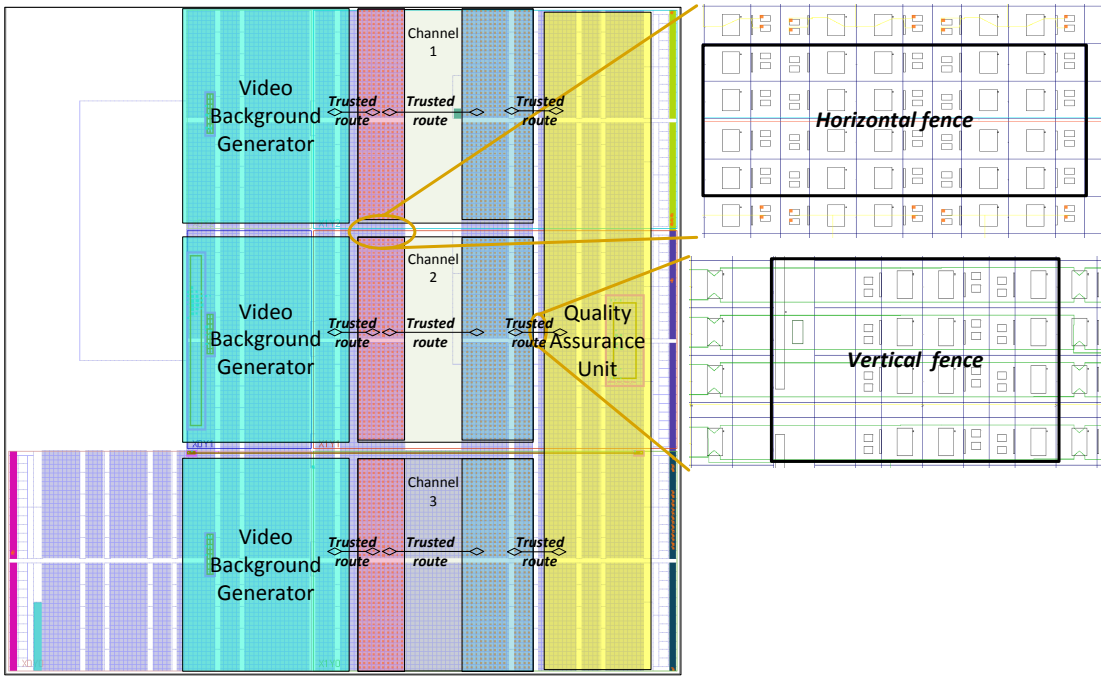


Figure A.8: System layout of the TMR design implemented on a *XC7Z020* FPGA. Each channel, which has 6 resource slots and can host the modules from Table A.2, as long as their resource primitives match the targeted resources. There are no wires in the horizontal fence between the isolated regions and only trusted routes are crossing the vertical fence.

A.5 Case Study II: Single-chip Cryptographic Design

To directly compare our *IPRDF* against Xilinx IDF, we took the example of the Single-chip Cryptographic (SCC) design from the Xilinx's XAPP1256 application note for IDF [Xil16d]. We then implemented its modules not only isolated but also partially reconfigurable by using our *IPRDF* methodology. The isolation of cryptographic modules satisfies information assurance requirements while the partial reconfiguration enables hardware module replacement or operation maintenance at run-time.

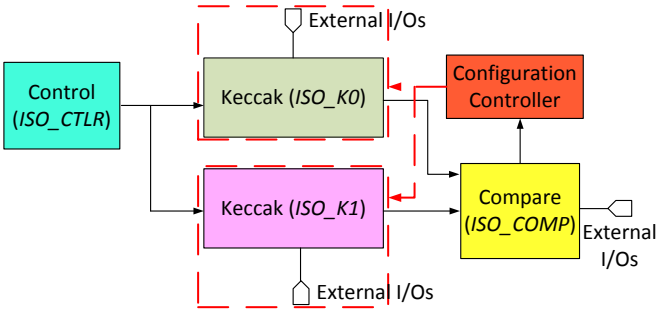


Figure A.9: The single-chip cryptographic (SCC) system's block diagram.

The example design consists of two redundant Keccak cryptographic hash modules[4] (*ISO_K0* and *ISO_K1*), whose outputs are sent to a comparator (*ISO_Compare*) block, and a processor control (*ISO_Controller*) module is used to supply clocks and resets, as shown in Figure A.9. This case study is utilizing Double Module Redundancy (DMR) technique to guarantee system's functional correctness. Moreover, as the main difference to Case Study I in Section A.4, it requires that module IOBs stay inside partially isolated partitions for off-chip communication, which is officially not supported for 7-Series devices according to the latest Xilinx PR documentation [Xil18c] at the time of writing this work.

In addition to XAPP1256, we have developed SHA-2 and AES-based hash modules as alternative solutions in order to demonstrate PR on this case study. These modules can be loaded to change cryptographic algorithms without shutting down the whole system.

A.5.1 System Implementation

We have revised the Floorplan for the Xilinx IDF reference to reserve two partially reconfigurable regions for Keccak cryptographic modules. One module, *ISO_K0*, is placed in the top-right of the chip layout whereas the other, *ISO_K1*, is at the bottom-left corner. The *ISO_Controller* and *ISO_Compare* stay in the static part of the system as shown in Figure A.10. All IOBs for module off-chip communication are reserved inside the isolation partitions. IOBs are connected directly to the modules for full control over the routing of the signals from IOBs to the module.

Moreover, to partially reconfigure a 7-Series FPGA without interference its IOBs, BitMan [PHK17] has been used to wrap up the partial bitstreams generated by this *IPRDF* in the following steps:

1. asserts the `GHIGH_B` signal: places all interconnects in a `High-Z` state to prevent contention when writing new configuration data;
2. writes new configuration data;
3. de-asserts the `GHIGH_B` signal: activates all interconnects.

A.5.2 Result

The final design is shown in Figure A.10. Physical fences are realized by our *IPRDF* methodology. Moreover, trusted routing is used between isolated partitions for secured communication.

Off-chip inputs and outputs for each Keccak hash module are instantiated and assigned into its isolated partition for trusted communication requirements. Moreover, they are reconfigurable along with the partial modules, which is not feasible with Xilinx IDF and PR flow.

[4] Keccak is the superset of the SHA-3 standard.

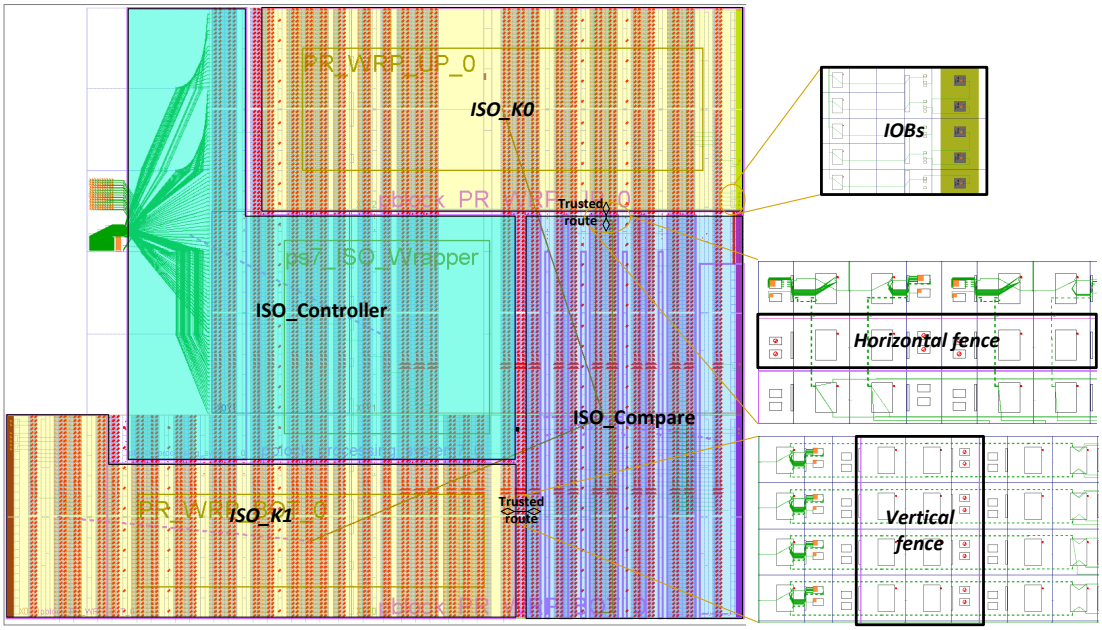


Figure A.10: System layout of the SCC design implemented on the *XC7Z020* FPGA. There are 2 partially reconfigurable regions *ISO_K0* and *ISO_K1* which could host Keccak hash modules, as in this example following the Xilinx's XAPP1256 [Xil16d], or other cryptographic ones at run-time.

This partially reconfigurable capability allows isolated partitions to host different cryptographic algorithms at run-time. Table A.3 shows available resources of each partial region and sizes of their partial bitstreams.

The outputs of modules in *ISO_K0* and *ISO_K1* are passed to the *ISO_Compare* module for quality assurance purpose. Any difference in these outputs triggers an alarm.

This final design was verified successfully for isolation-compatibility by the Xilinx Vivado Isolation Verifier (VIV). In addition, the demonstration of this partially reconfigurable SCC design could be seen on a YouTube video *(https://www.youtube.com/watch?v=YsQGaTUy1pM)*.

A.6 Conclusion

In this work, we have proposed an alternative design flow, named *IPRDF*, to build fully isolated and reconfigurable systems.

Two case studies have been presented to demonstrate details on how to use *IPRDF* for implementing fully isolated designs. The first case study uses this to implement a safety-critical TMR system that provides mitigation strategies for transient faults, configuration SEUs as well as for permanent FPGA defects using partial reconfiguration. The second case study enhances a single-chip cryptographic (SCC) system from [Xil16d] with partial reconfiguration

Table A.3: Available resources in *ISO_K0* and *ISO_K1* partial regions and size of partial bitstream to reconfigure each region.

Partial Region	Slice LUTs	BRAMs	DSPs	IOBs	Bitstream Size (kBytes)
ISO_K0	13600	30	60	50	1375
ISO_K1	13248	31	42	50	1425

capabilities which would allow changing ciphers at low resource cost.

It should be mentioned that module insulation is a requirement for implementing modules in certain security concerned systems (e.g., military applications) and that our entire tool flow is generating physical constraints for the Xilinx vendor tools. As a consequence, no IP details (neither code nor netlist) has to be presented to our tool flow, because our physical implementation scripts can be generated independently, even before the application development. This means that the *IPRDF* is not adding any security threat to the already established insulation flow for static only systems.

In practice, the security may be even higher as different modules are developed and physically implemented entirely separated from each other. With this, we enable partial reconfiguration in secure and safety-critical systems including a cleaner and more secure design flow.