FPGA Structures for High Speed and Low Overhead Dynamic Circuit Specialization

FPGA-structuren voor dynamische circuitspecialisatie aan hoge snelheid en lage kost

Amit Kulkarni

Promotor: prof. dr. ir. D. Stroobandt
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: elektrotechniek

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2016 - 2017

ISBN 978-94-6355-035-2
NUR 959, 987
Wettelijk depot: D/2017/10.500/70

Examination Commission

Prof. Dr. Patrick De Baets, chairman
Department of Electrical energy, systems and automation
Faculty of Engineering and Architecture
Ghent University, Belgium

Prof. Dr. Guillaume Crevecoeur, secretary
Department of Electrical Energy, Metal,
Mechanical Structures and Systems
Faculty of Engineering and Architecture
Ghent University, Belgium

Prof. Dr. Michael Huebner
Chair for Embedded Systems Information Technology
Ruhr-University Bochum, Germany

Dr. Mohamed El-Hadedy
Research Scientist
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, USA

Prof. Dr. Nele Mentens
Department of Electrical Engineering - ESAT
Faculty of Engineering Technology
Katholieke Universiteit Leuven, Belgium

Prof. Dr. Francis wyffels
Department of Electronics and Information Systems - ELIS
Faculty of Engineering and Architecture
Ghent University, Belgium

Em. Prof. Dr. Erik D'Hollander
Department of Electronics and Information Systems - ELIS
Faculty of Engineering and Architecture
Ghent University, Belgium

Prof. Dr. Dirk Stroobandt, advisor
Department of Electronics and Information Systems - ELIS
Faculty of Engineering and Architecture
Ghent University, Belgium

M.Sc. Amit Kulkarni

Tel.: +32-9-264.33.78
Fax.: +32-9-264.35.94
Email: Amit.Kulkarni@UGent.be

Advisor: Prof. Dr. Dirk Stroobandt

Ghent University
Faculty of Engineering and Architecture
Electronics and Information Systems (ELIS) department
Hardware and Embedded Systems (HES) research group
iGent, Technologiepark - Zwijnaarde 15, B-9052 Ghent, Belgium

Tel.: +32-9-264.34.01
Fax.: +32-9-264.35.94
Email: Dirk.Stroobandt@UGent.be

This work was supported by the European Commission in the context of the FP7 FASTER project (#287804) (www.fp7-faster.eu) and the H2020 EXTRA project (#671653) (www.extrahpc.eu)

Dissertation submitted in fulfillment of
the requirements for the degree of
Doctor of Electrical Engineering

Department of Electronics and Information Systems
Chairman: Prof. Dr. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2016 - 2017

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Dirk Stroobandt for giving me an opportunity to pursue the doctoral studies at the HES research group. His guidance and reviews have guided me to follow right path during my research studies. I would like to acknowledge Dr. Karel Bruneel and Dr. Karel Heyse for their support during my Ph.D. I would like to extend my gratitude to the jury members for their valuable feedback on this work. I cherish memorable moments with friends and colleagues: Abhishek, Ajeya, Alexandra, Andre, Ark, Brahim, Dries, Fatma, Florian, Helena, Mohamed, Poona, Prasun, Samir, Shruti Soultana, Vijay, Yun, and all the staff members of the ELIS department.

I would like to thank the Ghent University and the EU Commission for funding my research work. My special thanks to the people and professors at the chair for Embedded Systems Information Technology at the Ruhr-University Bochum for sharing a great rapport during my internship days.

I would like to acknowledge Prof. Sébastian Bilavarn and Dr. Robin Bonamy from the University of Nice, Sophia-Antipolis for their support for our research collaboration.

Many thanks to my family members who encouraged me to stand through thick and thin situations.

I owe a big thank you to all the people in the research community whom I met at the conferences and meetings in Austin, Chicago, California, Cancun, Stockholm, Amsterdam, Amersfoort, London, Cambridge, Bochum, and Nice.

Last but not least, I want to thank everyone else involved in this thesis. This includes you, the reader, for showing interest in the dissertation.

Ghent, August 19, 2017
Amit Kulkarni

Samenvatting

Een FPGA (Field Programmable Gate Array) is een programmeerbare digitale elektronische chip. De FPGA wordt door de producent niet geleverd met een vooraf gedefinieerde functie. De toepassingsontwikkelaar moet de functionaliteit zelf nog bepalen door een digitaal circuit op de FPGA-basisblokken te implementeren. Doordat de functionaliteit van een FPGA naar wens geherprogrammeerd kan worden, kreeg de component de naam "in het veld programmeerbaar" ("Field programmable"). FPGA's zijn bruikbaar voor digitale elektronische producten die in kleine oplage nodig zijn omdat het ontwerpen van een specifieke digitale chip zo duur is. De functionaliteit van een FPGA wijzigen (wat ook "de FPGA configureren" genoemd wordt) doet men door de configuratiebitstromen te wijzigen die de functionaliteit van de FPGA definiëren. Deze bitstromen worden bewaard in een FPGA-geheugen dat het configuratiegeheugen genoemd wordt. De SRAM-cellen of opzoektabellen (LUT's), Block Random Access Memories (BRAM) en DSP-blokken vormen samen het configuratiegeheugen van een FPGA. De configuratiedata kunnen aangepast worden aan de noden van de gebruiker om gebruikersspecifieke hardware te implementeren. De eenvoudigste manier om het configuratiegeheugen te programmeren, is om de bitstromen te downloaden via de JTAG-interface. Moderne technieken zoals **Partiële Herconfiguratie** laten ons toe een deel van het configuratiegeheugen tijdens het gebruik te configureren met partiële bitstromen. De herconfiguratie gebeurt door partiële bitstromen in het configuratiegeheugen op te laden via een configuratie-interface genaamd de **Interne Configuratietoegangspoort (ICAP)**. De ICAP is een hardware-primitieve (macro) die beschikbaar is in de FPGA en gebruikt wordt om intern toegang te krijgen tot het configuratiegeheugen vanuit een ingebedde processor. De herconfiguratietechniek voegt flexibiliteit toe om gespecialiseerde circuits te gebruiken die compacter en meer efficiënt zijn dan grotere algemene circuits. Een voorbeeld van zo'n implementatie is het gebruik van gespecialiseerde vermenigvuldigers in plaats van grote generische vermenigvuldigers in een FIR-implementatie met constante coëfficiënten. Om dergelijke circuits te specialiseren en tijdens het gebruik te herconfigureren, hebben onderzoekers van de HES-groep een nieuwe techniek voorgesteld genaamd **geparameteriseerde herconfiguratie**. Deze techniek kan gebruikt worden om **Dynamische CircuitsSpecialisatie (DCS)** efficiënt en op een automatische manier te implementeren. De techniek bouwt verder op de partiële herconfiguratiemethode en gebruikt de run-time-herconfiguratietechniek die aangepast is om een geparameteriseerd ontwerp te implementeren. Een toepassing wordt geparameteriseerd genoemd als sommige van haar ingangswaarden

veel minder frequent veranderen dan de rest. Deze ingangswaarden worden parameters genoemd. In plaats van deze parameters te implementeren zoals normale inputs, implementeren we ze in DCS als constanten en optimaliseren we de toepassing voor deze constanten. Voor elke verandering in parameterwaarden wordt het ontwerp dan geheroptimaliseerd (gespecialiseerd) tijdens de werking en geïmplementeerd door het geoptimaliseerde ontwerp te herconfigureren voor een nieuwe set van parameterwaarden.

In de DCS-methode worden de bitstromen van het geparameteriseerde ontwerp uitgedrukt als Boolese functies van de parameters. Voor elke infrequente verandering in parameterwaarden, wordt een gespecialiseerde FPGA-configuratie gegenereerd door de overeenkomstige Boolese functies te evalueren en wordt de FPGA geherconfigureerd met de gespecialiseerde configuratie. Het primaire doel van dit doctoraat is een gedetailleerde studie te maken van de overhead van DCS en geschikte oplossingen te vinden om de overhead te verminderen door aangepaste FPGA-structuren voor te stellen. Ik stel ook verschillende verbeteringen voor aan de configuratiegeheugenarchitectuur van de FPGA. Na het aanreiken van aangepaste FPGA-structuren, onderzocht ik ook de rol van DCS op FPGA-overlays en het gebruik van aangepaste structuren om de overhead van DCS in deze FPGA-overlays zo klein mogelijk te houden. Door dat te doen, hoop ik ontwikkelaars van toepassingen te overtuigen DCS (met een minimale kost) te gebruiken in hedendaagse toepassingen. Ik begin het onderzoek over de overhead van DCS met de implementatie van een adaptief FIR-filter gebruik makende van DCS en voor drie verschillende FPGA-platformen van Xilinx: de Virtex-II Pro, Virtex-5 en Zynq-SoC. De studie van het gedrag van DCS en de overhead die ermee gepaard gaat in de evolutie van de drie FPGA-platformen vormt de niet-triviale basis om de kosten van de DCS-methode te ontdekken. Daarna stel ik aangepaste FPGA-structuren voor (herconfiguratiecontrollers en herconfiguratiedrivers) om de belangrijkste overhead van DCS (herconfiguratietijd) te verminderen. Deze structuren verminderen niet alleen de herconfiguratietijd maar helpen ook om het meest vermogenverslindende deel van het DCS-systeem te omzeilen. Na deze hoofstukken bestudeer ik de rol van DCS op **FPGA-overlays**. Ik onderzoek het effect van de voorgestelde FPGA-structuren op **Virtuele Herconfigureerbare Matrixstructuren met grove granulariteit (VCGRAs)**. Ik heb twee varianten van VCGRA-roosters ontworpen voor HPC beeldverwerkingstoepassingen, met name het MAC-rooster en Pixie.

Ten slotte probeer ik de herconfiguratietijdsoverhead aan de hardwarekant van de FPGA neer te halen door het aanpassen van de configuratiegeheugenarchitectuur van de FPGA. In dit deel van mijn onderzoek stel ik voor een parallelle geheugenstructuur te gebruiken om de herconfiguratietijd van DCS drastisch te verbeteren. Deze verbetering komt echter met een significante overhead op hardwaregebruik die zal moeten opgelost worden in toekomstig werk op commerciële configuratiegeheugenarchitecturen op FPGA.

Summary

A **Field Programmable Gate Array (FPGA)** is a programmable digital electronic chip. The FPGA does not come with a predefined function from the manufacturer; instead, the developer has to define its function through implementing a digital circuit on the FPGA resources. The functionality of the FPGA can be reprogrammed as desired and hence the name "field programmable". FPGAs are useful in small volume digital electronic products as the design of a digital custom chip is expensive.

Changing the FPGA (also called configuring it) is done by changing the configuration data (in the form of bitstreams) that defines the FPGA functionality. These bitstreams are stored in a memory of the FPGA called configuration memory. The SRAM cells of LookUp Tables (LUTs), Block Random Access Memories (BRAMs) and DSP blocks together form the configuration memory of an FPGA. The configuration data can be modified according to the user's needs to implement the user-defined hardware. The simplest way to program the configuration memory is to download the bitstreams using a JTAG interface. However, modern techniques such as **Partial Reconfiguration (PR)** enable us to configure a part in the configuration memory with partial bitstreams during run-time. The reconfiguration is achieved by swapping in partial bitstreams into the configuration memory via a configuration interface called **Internal Configuration Access Port (ICAP)**. The ICAP is a hardware primitive (macro) present in the FPGA used to access the configuration memory internally by an embedded processor.

The reconfiguration technique adds flexibility to use specialized circuits that are more compact and more efficient than their bulky counterparts. An example of such an implementation is the use of specialized multipliers instead of big generic multipliers in an FIR implementation with constant coefficients. To specialize these circuits and reconfigure during the run-time, researchers at the HES group proposed the novel technique called **parameterized reconfiguration** that can be used to efficiently and automatically implement **Dynamic Circuit Specialization (DCS)** that is built on top of the Partial Reconfiguration method. It uses the run-time reconfiguration technique that is tailored to implement a parameterized design. An application is said to be parameterized if some of its input values change much less frequently than the rest. These inputs are called *parameters*. Instead of implementing these parameters as regular inputs, in DCS these inputs are implemented as constants, and the application is optimized for the constants. For every change in parameter values, the design is re-optimized (specialized) during run-time and implemented by reconfiguring the optimized design for a new set of

parameters.

In DCS, the bitstreams of the parameterized design are expressed as Boolean functions of the parameters. For every infrequent change in parameters, a specialized FPGA configuration is generated by evaluating the corresponding Boolean functions, and the FPGA is reconfigured with the specialized configuration.

A detailed study of overheads of DCS and providing suitable solutions with appropriate custom FPGA structures is the primary goal of the dissertation. I also suggest different improvements to the FPGA configuration memory architecture. After offering the custom FPGA structures, I investigated the role of DCS on FPGA overlays and the use of custom FPGA structures that help to reduce the overheads of DCS on FPGA overlays. By doing so, I hope I can convince the developer to use DCS (which now comes with minimal costs) in real-world applications.

I start the investigations of overheads of DCS by implementing an adaptive FIR filter (using the DCS technique) on three different Xilinx FPGA platforms: Virtex-II Pro, Virtex-5, and Zynq-SoC. The study of how DCS behaves and what is its overhead in the evolution of the three FPGA platforms is the non-trivial basis to discover the costs of DCS.

After that, I propose custom FPGA structures (reconfiguration controllers and reconfiguration drivers) to reduce the main overhead (reconfiguration time) of DCS. These structures not only reduce the reconfiguration time but also help curbing the power hungry part of the DCS system.

After these chapters, I study the role of DCS on **FPGA overlays**. I investigate the effect of the proposed FPGA structures on **Virtual-Coarse-Grained Reconfigurable Arrays (VCGRAs)**. I classify the VCGRA implementations into three types: the conventional VCGRA, partially parameterized VCGRA and fully parameterized VCGRA depending upon the level of parameterization. I have designed two variants of VCGRA grids for HPC image processing applications, namely, the MAC grid and Pixie.

Finally, I try to tackle the reconfiguration time overhead at the hardware level of the FPGA by customizing the FPGA configuration memory architecture. In this part of my research, I propose to use a parallel memory structure to improve the reconfiguration time of DCS drastically. However, this improvement comes with a significant overhead of hardware resources which will need to be solved in future research on commercial FPGA configuration memory architectures.

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

ADAS	Airborne Data Annotation System
ADC	Analog to Digital Converter
AES	Advanced Encryption System
AGU	Address Generator Unit
AIG	AND-Inverter-Graph
ALU	Arithmetic and Logic Unit
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface

B

BBRAM	Battery Backup Random Access Memory
BLE	Basic Logic Element
BPI	Byte Peripheral Interface
BRAM	Block Random Access Memory
BUF	Buffer

C

CAB	Configuration Access Bus
CAM	Content Addressable Memory
CB	Connection Block
CGRA	Coarse-Grained Reconfigurable Arrays
CLB	Configurable Logic Block
CPU	Central Processing Unit
CW	Channel Width

D

D2PR-EDAC	Dynamic Partial Reconfiguration with Error Detection and Correction
DCS	Dynamic Circuit Specialization
DMA	Direct Memory Access
DOR	Dimension-Order Routing
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random Access Memory
DRC	Design Rule Check
DSP	Digital Signal Processing
DT	Destination-Tag

F

FF	Flip-Flop
FIFO	First-In-First-Out
FinFET	Fin Field-Effect Transistor
FIR	Finite Impulse Response
FLIT	FLow control digIT
FPGA	Field Programmable Gate Array

G

GP	General Purpose
GPU	Graphic Processing Unit

H

HDL	Hardware Description Language
HES	Hardware and Embedded Systems
HP	High Performance
HPC	High Performance Computing
HWICAP	Hardware Internal Configuration Access Port

I

IBM	International Business Machine
IC	Integrated Chip
ICAP	Internal Configuration Access Port
IOB	Input Output Block
IP	Intellectual Property
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors

J

JTAG	Joint Test Action Group

L

LCD	Liquid Crystal Display
LUT	LookUp Table

M

MAC	Multiply Accumulate
MAF	Module Assignment Function
mCW	minimum Channel Width
MiCAP	Micro-reconfigurable Configuration Access Port
Min AD	Minimal Adaptive
MM2S	Memory Mapped to Stream
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPSoC	Multiprocessor System-on-Chip
MRMW	Multi-Read-Modify-Write
MROMW	Multi-Read once-Modify-Write
MUX	Multiplexer

N

NoC	Network-on-Chip

O

OPB	On-Chip Peripheral Bus

P

PCAP	Processor Configuration Access Port
PCIe	Peripheral Component Interconnect express
PE	Processing Element
PL	Programmable Logic
PLB	Processor Local Bus
PLD	Programmable Logic Device
PPC	Partial Parameterized Configuration
PR	Partial Reconfiguration
PRF	Polymorphic Register File
PRM	Partial Reconfigurable Module
PRR	Partially Reconfigurable Region
PS	Processing System

R

RAM	Random Access Memory
RegEx	Regular Expression
RISC	Reduced Instruction Set Computer
RPU	Real-time Processing Unit
RTL	Register-Transfer Level

S

S2MM	Stream to Memory Mapped
SB	Switch Block

SCG	Specialized Configuration Generator
SD	Secure Digital
SIMD	Single Instruction Multiple Data
SoC	System on Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory

T

TC	Template Configuration
TCAM	Ternary Content Addressable Memory
TCON	Tunable Connection
TLUT	Tunable LookUp Table
TLUTMAP	Tunable LookUp Table Mapper
TPAR	Tunable Place and Route
TPLACE	Tunable Placer
TROUTE	Tunable Router

U

UGAL	Universal Globally-Adaptive Load-balanced

V

VAL	Valiant
VC	Virtual Channel
VCGRA	Virtual Coarse-Grained Reconfigurable Arrays
VHDL	Very-high-speed Integrated Circuit Hardware Description Language
VPR	Versatile Placement and Routing
VSB	Virtual Switch Block

W

WL	Wire Length

1 Introduction

This chapter starts with a brief introduction to digital electronics and implementations on CPU, GPU, ASIC and FPGA. It compares one implementation form over the other, with emphasis on the importance of heterogeneous computing and the justification on why it has emerged as the most recent trend in electronic implementations. Reconfigurability plays an important role in heterogeneous computing and therefore, this chapter will focus most of the discussion on reconfigurable computing. Furthermore, a brief introduction to reconfiguration in FPGAs and classification of the reconfiguration followed by introduction to Parameterized Reconfiguration technique is presented. After, a brief overview of each chapter presented in this theses. This chapter concludes with a description on the structure of this dissertation.

1.1 Introduction to Digital Integrated Circuits

In today's world digital electronics or electronic chips have made significant contribution to the evolution of modern electronic devices. The electronic chips are the brains of almost every electric device ranging from a digital thermometer to a personal computer to an international space station.

Digital electronic chips (also called Integrated Chip - IC) consist of a network of Boolean logic gates built from a basic building block called transistor. The transistor count in a typical IC could range from a few thousands to millions to billions (depending on the functionality to be implemented) connected together to form a

huge network of Boolean logic gates. The task of these networks is to process digital signals, which are in the form of binary 0s or 1s, to produce a meaningful output to the user. For example, consider a digital thermometer that can sense the temperature using a thermal sensor. The data sensed by the thermal sensor is converted into binary (using ADC) form 0s and 1s. A digital electronic chip is designed in such a way that it takes the binary values as input and drives the LCD unit to display the current temperature in a human readable format (either in Celsius or Fahrenheit). The design of such IC is application specific and cannot be used for any other purpose and hence they are called Application Specific Integrated Circuit (ASIC). After fabrication, the function of an ASIC will remain the same throughout its life span and cannot be changed at all.

The digital thermometer is a very simple example to illustrate the use of a digital IC. However, there exist stringent performance requirements for the high-performance computing applications that demand the processing of huge data (petascale) within considerably less execution time. To handle such requirements the technology has evolved towards heterogeneous computing platforms.

1.2 Heterogeneous computing platforms

A high-performance computing machine contains a digital brain called the microprocessor built with billions of transistors dedicated to perform useful computations for the user. According to Moore's law, the transistor count on chip doubles every 18-24 months, leading to the prediction that the number of on-chip cores of a processor (but not the performance) doubles for every two years [1].

Dennard scaling (also known as MOSFET scaling law) states that the power density stays constant as transistor sizes become smaller hence the power use remains in proportion with the area of the transistor [2].

In today's transistor technology node, we cannot continue to ride the transistor count growth curve as per Moore's law since the Dennard scaling law on power density has failed massively for the technology nodes below 65 nm. The principal challenge turns out to be the increase in power density that prevents all the processor cores to be switched on at the same time. This phenomenon is called dark silicon. Due to the failure of power density, embedding more transistors on a single chip to work does no longer help therefore, we need a strategy to use them more efficiently. This has led to the evolution of heterogeneous computing where the transistors can be efficiently used as every problem gets its own optimized implementation [3].

The heterogeneous computing architectures can be broadly classified into two categories: performance heterogeneity and functional heterogeneity.

1. Performance heterogeneous multi-core. The multi-core architecture shares a common Instruction Set Architecture but consists of cores with different

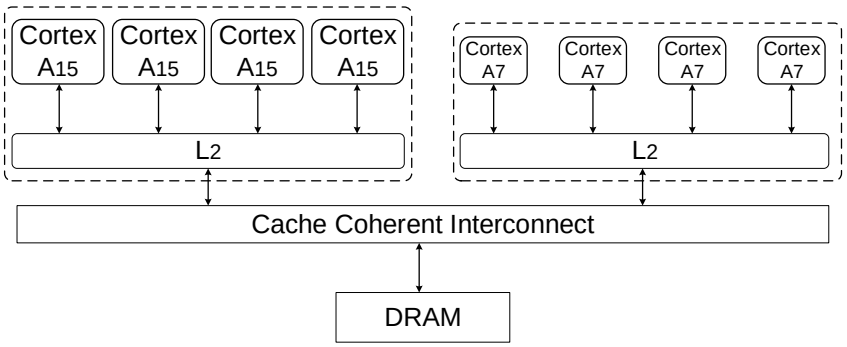


Figure 1.1: Asymmetric multi-core: ARM big.LITTLE

power per performance characteristics. The distinct micro-architectural features such as in-order vs out-of-order instruction execution will affect the power performance of each core. Therefore, the complex core can provide high performance with the cost of high power while the simpler cores can provide low performance with an advantage of low power consumption. These cores can be further classified into static *asymmetric multi-core* and *dynamic asymmetric multi-core*.

(a) Static asymmetric multi-core. In this type of architecture, the heterogeneity (mix of different cores) is fixed at fabrication time. For example, ARM big.LITTLE [4] belongs to the asymmetric multi-core category that has high-performance out of order cores integrated with low-power in-order cores as shown in Figure 1.1. It consists a set of high performance quad-core ARM Cortex-A15 RISC processors integrated with low power quad-core ARM Cortex-A7. This kind of multi-core appears in Samsung's Exynos 5 Octa SoC used in Samsung galaxy S4 smart phones.

These multi-cores are introduced to accommodate software diversity such as a mix of instruction level parallelism and thread level parallelism and prove to be much better than homogeneous multi-cores. The low-power (Cortex-A7) core can take care of uncomplicated applications such as handling the email client or a modest messenger. Keeping the high-performance core off the shelf thus it proves the power efficiency. However, the complex core Cortex-A15 is switched on only when there is a demand to handle compute intensive tasks such as 3D gaming, high stream video applications, etc. trading the energy with the performance. The main drawback of such architecture is they are not flexible enough to adjust themselves to the dynamic nature of a

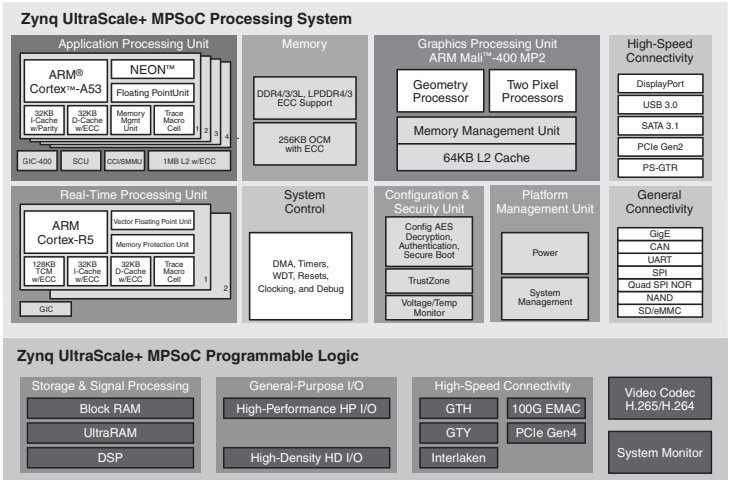


Figure 1.2: Zynq UltraScale+ MPSoC [6]

workload.

(b) Dynamic asymmetric multi-core. This type of multi-core architecture is designed to provide flexibility so that they can dynamically tailor themselves according to the application demand. These architectures are fabricated as a set of homogeneous cores. At run-time two or more homogeneous cores are collated together to form one big complex virtual core. Also, at run-time one complex virtual core can be split to form two or more simple cores. An example of the dynamic asymmetric multi-core architecture is the Bahurupi architecture [5].

2. Functional heterogeneous multi-core. This multi-core architecture comprises of cores with distinct functionality. The functionality of each core together introduces heterogeneity to meet the performance requirements under a stringent power budget. For example, a Multi-Processor System-on-Chip (MPSoC) used in embedded products consists of Central Processing Unit (CPU) cores, Graphics Processing Unit (GPU) cores, Real-time Processing Unit (RPU) cores, Digital Signal Processor (DSP) blocks and Programmable Logic (PL) accelerators.

   A commercially available functional heterogeneous computing platform for embedded space (Xilinx Ultrascale + MPSoC) is depicted in Figure 1.2.

The MPSoC platform is divided into two parts: Processing System (PS) and Programmable Logic (PL). The PS mainly contains a quad-core ARM Cortex-A53 RISC CPU running up to 1.5 GHz; a Mali-400 embedded GPU, a dual-core ARM Cortex-R5 RPU with a clock running up to 600 MHz, memory controllers, high speed connectivity interfaces, etc.

A Graphics Processing Unit is a specialized processor designed to handle complex mathematical and geometrical calculations (vector processing, floating-point operations and matrix operations) needed for graphics rendering. They excel in number crunching using extreme data parallelism built by large-scale Single Instruction Multiple Data (SIMD) computer architectures.

A Real-time Processing Unit is used for time-bound computations. The processor has to be active enough to receive the data, perform computation and respond back within a fixed time frame. Such processors work independently (without involving in dependencies with other sub-systems) and are inherently fault tolerant.

Digital Signal Processing blocks are dedicated hardware used for efficient signal processing of digital signals (mainly audio). They have a fixed arithmetic data path that can handle a huge amount of numerical calculations. For example, a DSP block contains a hard coded multiply-accumulate (MAC) operator unit that is much more efficient and faster than a MAC operation on a RISC processor.

The PL is a Field Programmable Gate Array (FPGA) fabric that contains up to 1M logic elements, 1K DSP slices, a total block RAM of 35 Mb and other high performance communication blocks such as gigabit transceivers. The PL is used to implement application specific specialized hardware that is efficient to perform tasks requested by the PS in order to accelerate and obtain real time responsiveness. Thus, the PL acts as an accelerator while the PS is suitable for implementing the applications whose features or specifications change frequently.

1.3 Reconfigurable Computing

Reconfigurable computing is a computer paradigm that combines the flexibility a software program running an a CPU with the performance of dedicated hardware [7]. All cores (except PL) described in the previous sections are traditional fixed-function ASIC accelerators that provide high efficiency but offer zero flexibility.

In a broader spectrum, on one end a general-purpose processor provide full flexibility via software programming, but the performance and energy efficiency

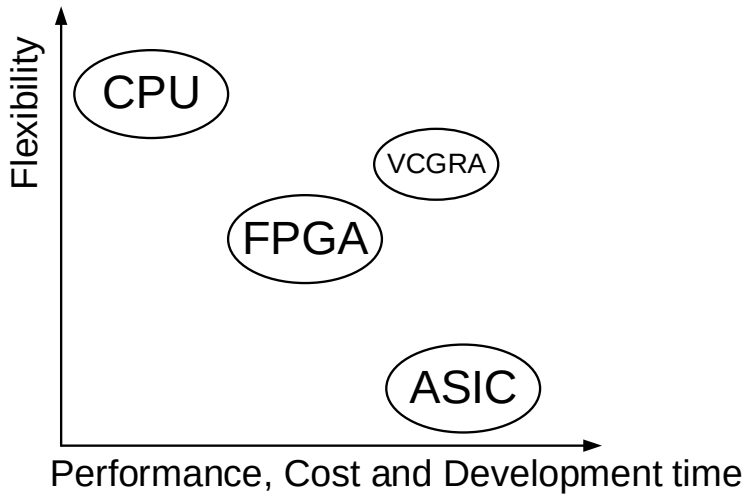


Figure 1.3: A spectrum of different computing platforms

is much lower than in an ASIC; on the other end, an ASIC provides high performance and energy-efficient implementations with no flexibility. Reconfigurable computing fills the gap between the CPU flexibility and ASIC like performance as illustrated in Figure 1.3.

1.3.1 Field Programmable Gate Array

An FPGA is a semiconductor fabric that allows users to change (configure) its functionality (hardware behavior) not only at compile-time but also at run-time. The configuration data (bitstreams) of an FPGA define the hardware functionality. Therefore, the user can design the hardware and change it by modifying the bitstreams for a given set of requirements. As shown in Figure 1.3, an FPGA fills the gap between two ends (CPU and ASIC) in a spectrum of flexibility, development cost and time similar to the reconfigurable architectures. Therefore, the FPGA is a key player in the era of reconfigurable computing.

1.3.2 Coarse-Grained Reconfigurable Array

In Coarse-Grained Reconfigurable Arrays (CGRAs) each programmable logic component are defined at a higher abstraction level. These components are called Processing Elements (PEs) and the group of PEs along with the inter-connection network form an architecture that enables ease of programmability and result in

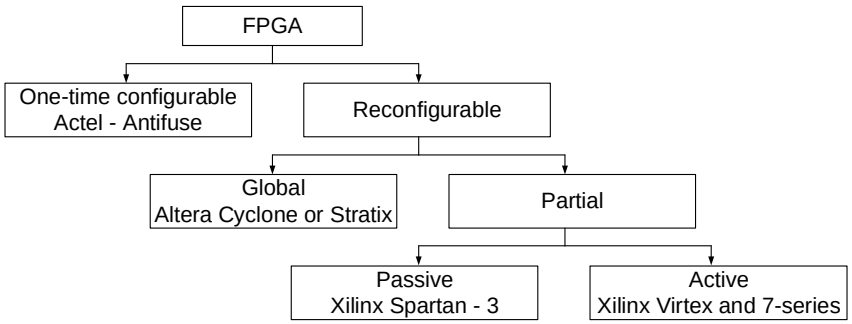


Figure 1.4: Classification of FPGA based on configurability [7]

low development costs. They enable the ease of use specifically in reconfigurable computing applications. The smaller cost of compilation and reduced reconfiguration overhead enables them to become attractive platforms for accelerating high-performance computing applications. The CGRAs are ASICs and therefore, expensive to produce. However, Field Programmable Gate Arrays (FPGAs) are relatively cheaper for low volume products but they are not so easily programmable. Combining the best of both worlds leads to a Virtual Coarse-Grained Reconfigurable Array (VCGRA) on FPGA. VCGRAs are a trade off between FPGA and ASICs.

1.4 Reconfiguration techniques and types

A reconfigurable device (chip) allows the user to change its functionality at any time (including run-time) in the system. Such systems require no manual interventions while changing the functionality or destructing the existing functionality of digital design. A CPU cannot be considered as a reconfigurable device since the hardware structure of the CPU remains the same while only the instructions change on a clock cycle basis. However, an FPGA is a reconfigurable device since the digital implementation on the FPGA can be changed at anytime.

Depending on the configuration capability, an FPGA architecture can be classified as illustrated in Figure 1.4. At the top level, FPGAs can be divided between one-time configurable devices (Antifuse-based) that can replace ASIC devices and (re)configurable FPGAs. Configurable FPGAs (SRAM-based) can be then classified into globally and partially reconfigurable categories. A complete FPGA configuration has to be swapped while performing global reconfiguration, which leads to a change in the internal state of the hardware and thus the FPGA has to restart its operation. This kind of reconfiguration is used for in-field updates. In partial reconfiguration, the user can change the function of part of the FPGA de-

vice (usually a region) while other sections remain operational. Furthermore, the partial reconfiguration can be performed either passive (static) by stopping the operation of the application (by disabling all clocks) or active (dynamic) where the operation of the application can continue during the reconfiguration. Henceforth we will focus only on active or dynamic partial reconfiguration.

1.4.1 Dynamic Partial Reconfiguration

Partial region-based reconfiguration [8] is the ability to modify a part of the logic blocks of an FPGA while the rest remains active. The modification of the logic is achieved by downloading partial bitstreams. The partial bitstreams represent Partial Reconfigurable Modules (PRMs). The reconfiguration process is triggered by the application software when a set of conditions at a given moment in time are met.

The system for conventional Partial region-based Reconfiguration is depicted in Figure 1.5. The system consists of a CPU (such as PowerPC, ARM Cortex-A9 or MicroBlaze, either on or off the chip), a configuration database (an SD card), a configuration interface (such as HWICAP) and Programmable Logic (PL- such as the logic blocks on an FPGA). A part of the PL region is segregated and dedicated to implement the reconfigurable logic. The "F_1" region shown in Figure 1.5 on the FPGA is a dynamic region dedicated for reconfiguration and is also called Partially Reconfigurable Region (PRR). The rest of the PL is called static region. The static and reconfigurable regions are connected physically using primitives called bus macros [9].[1] The user chooses the PRR well before the bitstream compilation. The remaining part of the FPGA is not a part of the reconfiguration and hence can be regarded as a static region.

The application software running on the CPU triggers the reconfiguration by sending a reconfiguration request to the configuration manager. The configuration manager downloads an appropriate partial bitstream by fetching it from the configuration database. The partial bitstream is downloaded via a configuration interface called Hardware Internal Configuration Access Port (HWICAP) [12]. The HWICAP is responsible for orchestrating the swap of partial bitstreams.

A set of PRMs are compiled during the bitstream compilation of the design. Each PRM has to undergo conventional FPGA toolflow steps: Synthesis, Technology Mapping and Place and Route to compile into Partial Bitstreams (Figure 1.6). These PRMs are stored in the configuration database which is located in a memory. The memory can be internal (DRAM) or it can be external to the FPGA (SD card).

[1] In the modern FPGAs the bus macro connections are called partition pins. The partition pins are the physical and logical connection between static logic and reconfigurable logic. Partition pins are automatically created and placed by the design tool such as Vivado [10] for all PRR [11].

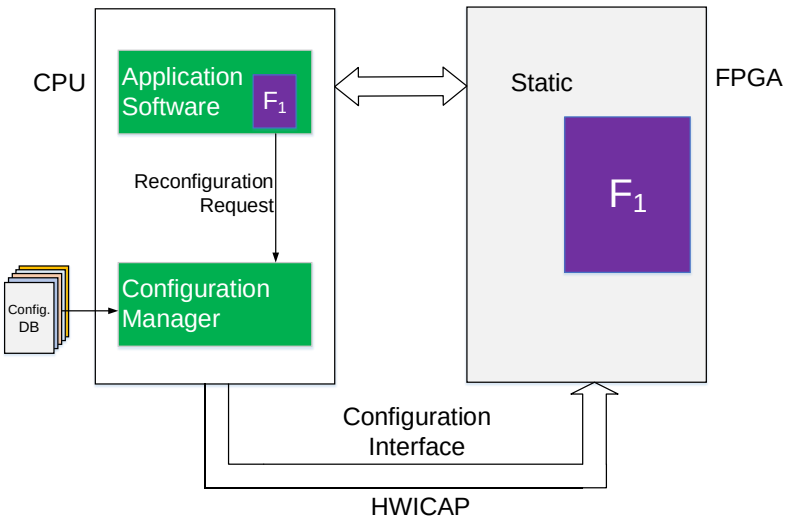


Figure 1.5: Partial region-based Reconfiguration System

The run-time Partial region-based Reconfiguration can be used to save a significant amount of silicon area and dynamic power by swapping only the required design into the FPGA. However, if the costs of the PR such as reconfiguration time and amount of memory needed to store the partial bitstreams is very high then the advantage of using PR diminishes. For example, an adaptive FIR filter (16-taps, 8-bit) whose taps can be reconfigured for any set of filter coefficients requires 2^{128} different PRMs to be stored in the memory which is practically impossible and therefore, PR is not feasible, even for such a simple application. Parameterized Reconfiguration can be used in such situations to overcome the constraints imposed by the overheads.

1.4.2 Dynamic Parameterized Reconfiguration

Parameterized Reconfiguration is suitable to implement parameterized applications. A parameterized application contains a set of inputs whose values change less frequently than the rest of the inputs [13]. These infrequently changing inputs are called *parameters*. In this approach, a part of the FPGA configuration bitstreams is expressed as Boolean functions of parameter inputs. For every change in parameter input values, the Boolean functions are evaluated to generate specialized bitstreams at run-time and the FPGA is reconfigured with the specialized bitstreams using partial run-time reconfiguration. This technique is also called Dynamic Circuit Specialization (DCS) [13].

The parameterized reconfiguration system is depicted in Figure 1.7. The pa-

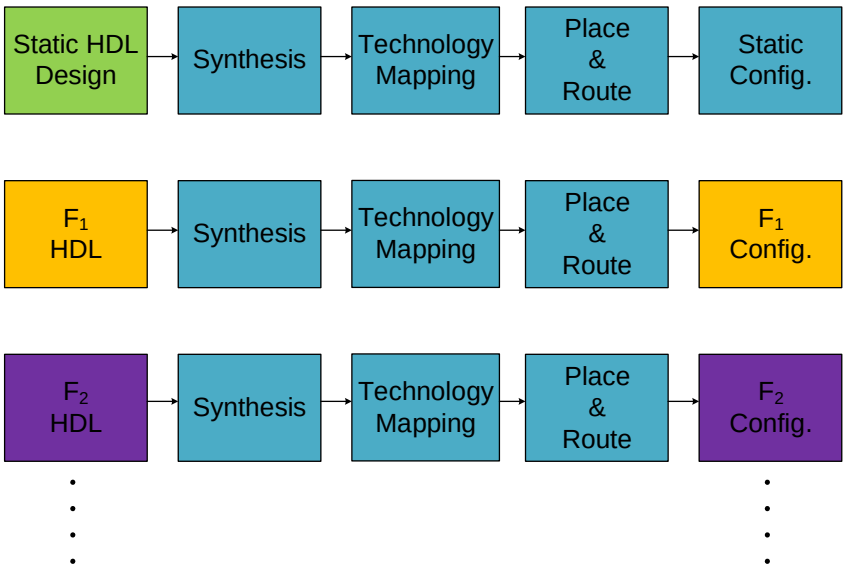


Figure 1.6: Tool flow for each PRM

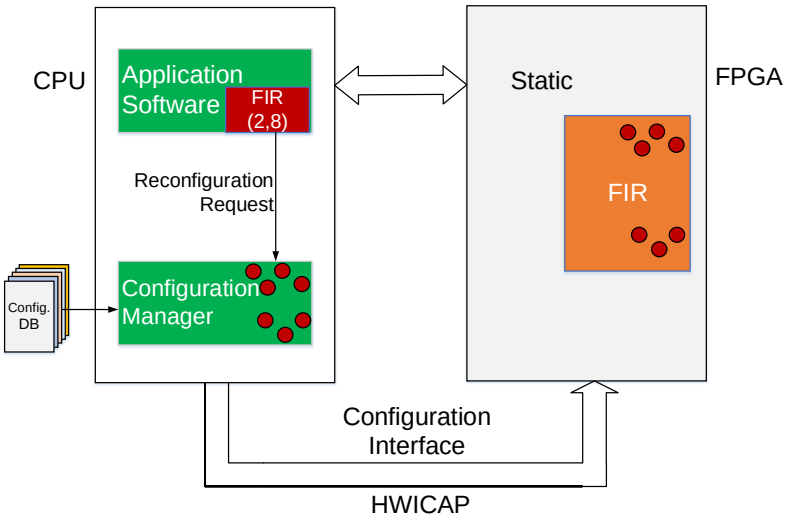


Figure 1.7: Parameterized Reconfiguration System

rameterized reconfiguration is built on top of partial region-based reconfiguration and hence most parts in the system remain the same. The application software

running on the CPU monitors the parameterized inputs. Once a change in parameter value is detected, the specialization is performed by the configuration manager by evaluating the Boolean functions (that are stored in the configuration database) for given parameter values, thus generating the specialized bitstreams. The stale frames of the FPGA are replaced with the specialized frames using *micro-reconfiguration*.

The adaptive FIR filter (16-taps, 8-bit), can be efficiently implemented using the parameterized reconfiguration technique. With this technique, an approximate reduction of 42% of resource utilization can be achieved when compared against generic (non-reconfigurable) implementation. [13]. The FIR filter contains filter taps containing multiplications that are parameterized. For every (infrequent) change in the coefficient value, a specialized bitstream is generated (evaluating the Boolean functions) and the filter taps containing multiplications are micro-reconfigured accordingly.

1.5 Introduction to the research and overview of the chapters

Although dynamic reconfiguration in FPGAs is an important feature that offers design flexibility under low-cost silicon area and power budgets, this flexibility comes with undesired overheads. One of the main overheads is the time taken during reconfiguration that is too high for the reconfiguration technology to be embraced as a standard. To handle the stringent performance requirements of future High Performance Computing (HPC) applications, HPC systems need ultra-efficient heterogeneous compute nodes. To reduce power and increase performance, such compute nodes will require reconfiguration as an intrinsic feature [14], so that the reconfiguration technology can help in optimal acceleration of a specific part of the HPC application even if they regularly change over time [15].

This dissertation presents a detailed study of overheads of DCS. To overcome the overheads I propose custom FPGA structures that are designed to implement efficient reconfiguration for DCS. I also propose different improvements to the FPGA architecture. To address the problems with ease of programming in FPGAs, I also study VCGRA architectures using DCS.

FPGA architecture and the tool flow

The background of an FPGA architecture and the conventional tool flow to generate configuration bitstreams are discussed in Chapter 2. I describe more on the reconfiguration infrastructure of the modern Xilinx FPGA followed by a comparison of key features of modern SoC FPGAs with their predecessors.

1.5.1 My contribution to the research

The following chapters give brief details of my contribution to this dissertation.

Overhead evaluation and measurement of Dynamic Circuit Specialization

Chapter 3 is more focused on the state-of-the-art of Dynamic Circuit Specialization and the tool flow used to generate parameterized configurations followed by an explanation of *micro-reconfiguration*. I have evaluated the overheads of the DCS technique and its performance on three different Xilinx FPGA platforms.

MiCAP and MiCAP-Pro

The reconfiguration controller (such as HWICAP) is a necessary component for dynamic reconfiguration. However, using the conventional HWICAP proves to be inefficient for DCS. Therefore, the custom reconfiguration controllers (MiCAP and MiCAP-Pro) for DCS are designed in order to reduce the reconfiguration overheads. More details on these controllers are presented in Chapter 4.

DCS for FPGA Overlay architectures

FPGAs have proven their potential in accelerating High Performance Computing (HPC) Applications. Conventionally such accelerators predominantly use FPGAs that contain fine-grained elements such as LookUp Tables (LUTs), Switch Blocks (SB) and Connection Blocks (CB) as basic programmable logic blocks. However, the conventional implementation suffers from high reconfiguration and development costs. In order to solve this problem, programmable logic components are defined at a virtual higher abstraction level. This result is an intermediate virtual architecture called FPGA overlays. FPGA overlays make the life of software programmers easier as they bypass the need to understand the hardware knowledge. The abstraction helps to reconfigure the parts of the hardware faster at the intermediate level than at the lower-level of an FPGA. In Chapter 5, I focus on the DCS technique for VCGRAs.

Custom FPGA configuration memory architecture for ultra-fast reconfiguration

The reconfiguration time overhead produced by the conventional configuration ports (such as ICAP) is too high for the reconfiguration technology to be embraced as a standard. Furthermore, the current FPGA configuration memory architecture restricts the access of configuration data to the frame level; this significantly delays the reconfiguration process. A custom FPGA architecture for ultra-fast micro-reconfiguration is presented in Chapter 6.

Finally, the concluding remarks and the future work of the dissertation is presented in Chapter 7.

Publications

Journal Papers

- **Amit Kulkarni** and Dirk Stroobandt. "MiCAP-Pro: A high speed custom reconfiguration controller for Dynamic Circuit Specialization".
  Design Automation for Embedded Systems, Vol. 20, Issue 4, pp. 341-359, 2016.

- **Amit Kulkarni** and Dirk Stroobandt. "How to Efficiently Reconfigure Tunable Lookup Tables for Dynamic Circuit Specialization".
  International Journal of Reconfigurable Computing, Vol. 2016, pp. 1-12, 2016.

- Mohamed El-Hadedy, **Amit Kulkarni**, Dirk Stroobandt and Kevin Skadron. "Reco-Pi: A Reconfigurable Cryptoprocessor for π-Cipher".
  Journal of Parallel and Distributed Computing: Special issue on Reconfigurable Computing Through the Looking Glass (Accepted, 2017)

Conference Papers

- **Amit Kulkarni**, Poona Bahrebar, Dirk Stroobandt, Giulio Stramondo, Catalin Bogdan Ciobanu, Ana Lucia Varbanescu. "A NoC-based custom FPGA configuration memory architecture for ultra-fast micro-reconfiguration".
  In 2017 International Conference on Field-Programmable Technology. under review (Submitted 2017)

- Alexandra Kourfali, **Amit Kulkarni**, Dirk Stroobandt. "SICDA: A Superimposed In-Circuit Debug Architecture for Virtual Coarse-Grained Reconfigurable Arrays".
  In 2017 International Conference on Field-Programmable Technology. under review (Submitted 2017)

- Alexandra Kourfali, **Amit Kulkarni**, Dirk Stroobandt. "SICTA: A Superimposed In-Circuit Fault Tolerant Architecture for SRAM-based FPGAs".
  In 2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS) Proceedings, pp. 1-4, 2017.

- **Amit Kulkarni**, Andre Werner, Florian Fricke, Dirk Stroobandt and Michael Huebner. "Pixie: A heterogeneous Virtual Coarse-Grained Reconfigurable Array for high performance image processing applications".

In 3rd International Workshop on Overlay Architectures for FPGAs (OLAF), Proceedings, pp. 1-6, 2017.

- **Amit Kulkarni**, Elias Vansteenkiste, Dirk Stroobandt, Andreas Brokalakis and Antonios Nikitakis. "A fully Parameterized Virtual Coarse-Grained Reconfigurable Array for High Performance Computing applications".
  In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, Proceedings, pp. 265-270, 2016.

- Mohamed El-Hadedy, **Amit Kulkarni**, Hristina Mihajloska, Danilo Gligoroski, Dirk Stroobandt and Kevin Skadron. "A 16-bit Reconfigurable Encryption Processor for π-Cipher".
  In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, Proceedings, pp. 162-171, 2016. (*Best paper award*).

- Dirk Stroobandt, Ana Lucia Varbanescu, Catalin Bogdan Ciobanu, Muhammed Al Kadi, Andreas Brokalakis, George Charitopoulos, Tim Todman, Xinyu Niu, Dionisios Pnevmatikatos, **Amit Kulkarni**, Elias Vansteenkiste, Wayne Luk, Marco D Santambrogio, Donatella Sciuto, Michael Huebner, Tobias Becker, Georgi Gaydadjiev, Antonis Nikitakis and Alex JW Thom.
  "EXTRA: Towards the exploitation of eXascale technology for reconfigurable architectures".
  In 2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2016) Proceedings, pp. 1-7, 2016.

- **Amit Kulkarni**, Vipin Kizheppatt and Dirk Stroobandt. "MiCAP: A custom Reconfiguration Controller for Dynamic Circuit Specialization".
  In International Conference on ReConFigurable Computing and FPGAs (ReConFig), Proceedings, pp. 1-6, 2015.

- **Amit Kulkarni**, Robin Bonamy and Dirk Stroobandt. "Power measurements and analysis for Dynamic Circuit Specialization".
  In International Conference on ReConFigurable Computing and FPGAs (ReConFig), Proceedings, pp. 1-6, 2015.

- **Amit Kulkarni**, Tom Davidson, Karel Heyse and Dirk Stroobandt. "Improving reconfiguration speed for Dynamic Circuit Specialization using placement constraints".
  In International Conference on ReConFigurable Computing and FPGAs (ReConfig), Proceedings, pp. 1-6, 2014.

- **Amit Kulkarni**, Karel Heyse, Tom Davidson, Dirk Stroobandt. "Performance Evaluation of Dynamic Circuit Specialization on Xilinx FPGAs".
  In FPGAworld Conference, Proceedings, pp. 1-6, 2014.

Abstract / Poster presentations

- **Amit Kulkarni** and Dirk Stroobandt, “Enabling HPC applications through Virtual Coarse-Grained Reconfigurable Arrays”. *In ICT.OPEN* 2017.

2

FPGA architecture and the tool flow

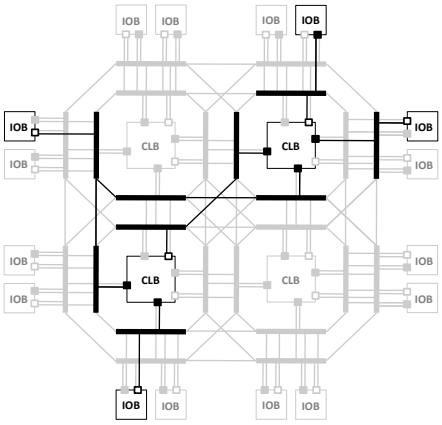
In this chapter, I present a detailed explanation of the FPGA architecture, discuss the conventional FPGA tool flow used to develop the FPGA configuration bitstreams, and I explain possible methods to download the bitstreams from a computer to the FPGA configuration memory. To discuss the evolution of commercial FPGAs, I consider three main FPGA platforms and compare their architecture specifications. These specifications are important factors that will affect the reconfiguration infrastructure of each FPGA.

2.1 FPGA architecture

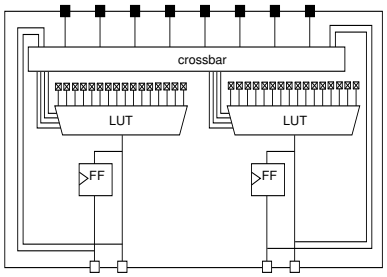
There exist two styles of FPGA architectures: island style and column-based style. The island style architecture covers a general, basic version of the FPGA architectures while the column-based style covers the modern commercial FPGA architectures.

Island style. The main building blocks of FPGAs are Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs) and a routing network. Typically, an FPGA contains a grid of CLBs ranging from 10,000s to 100,000s (even millions in today's FPGAs) in number and hundreds to thousands of IOBs [16].

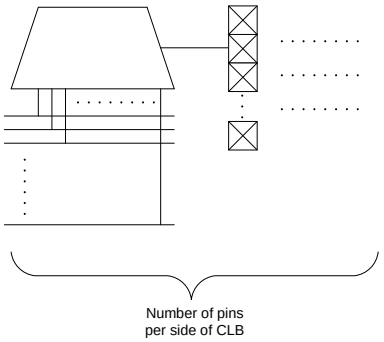
The CLBs include LookUp Tables (LUTs) and flip-flops that can be combined to realize any digital circuit. Each LUT can implement any arbitrary Boolean function for the given inputs depending on the truth table entries stored in the configuration memory. The truth table entries of the LUT define the combinatorial



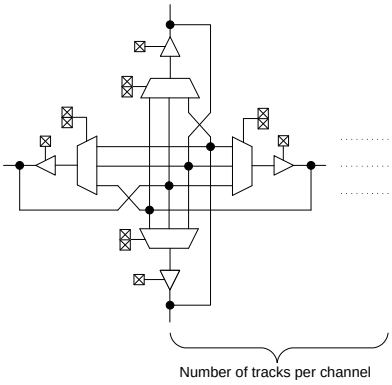
(a) Schematic of a basic FPGA with 4 CLBs, a routing network and IOBs



(b) Schematic of a CLB with 4-input LUT and two flip-flips. SRAM configuration memory cells are shown as ⊠ [17]



(c) Schematic of a Connection block (linking the channel wires to a CLB or IOB) with programmable multiplexer. SRAM cells are shown as ⊠



(d) Schematic of a Switch block (linking wires of horizontal and vertical wire channels) with programmable multiplexer. SRAM cells are shown as ⊠

Figure 2.1: FPGA architecture

logic for different values of the inputs. The flip-flops are used to store the output of a LUT and hence are used for implementing sequential logic.

The IOBs are connected to the external pins of the FPGA chip to establish communication with the external world, and therefore they are placed along the perimeter of the FPGA fabric.

The IOBs and the inputs and outputs of each CLB can be connected to each

other using the FPGA's routing network. The network consists of a lot of wires placed between the CLBs and form the routing channel width. Usually, the routing channel width is more than 100 in the commercially available FPGAs [18]. These wires are solely used for realizing the user applications and hence they do not interfere with the clock tree routing. Separate clock tree routing resources are provided to distribute the clock signal to each flip-flop in the CLBs. A schematic of a stripped down FPGA is depicted in Figure 2.1a. In this small example, each CLB has four inputs and two outputs and the routing channel width is two. The routing network is a mesh with vertical and horizontal wire channels that run alongside the CLBs and are meant to connect outputs of CLBs (or IOBs) to inputs of CLBs (or IOBs).

As a part of the routing network, there are two primitives called Switch block (SB) and Connection block (CB). The connections between wires are made using multiplexers in the SBs (Figure 2.1d), and the connections between wires and IOBs or CLBs are made using multiplexers in the CBs (Figure 2.1c).

As shown in Figure 2.1b, the CLBs contain crossbars to establish a connection between the input of the LUTs and the output of the other LUTs in the same CLB. The state of the multiplexers of CBs, SBs, and crossbars are also a part of the configuration memory. The routing network consumes most of the silicon area of the FPGA.

The configuration memory is a volatile memory, and hence the FPGA has to be configured again every time at boot-up. The FPGA has special infrastructure to write and read the configuration data from the configuration memory. Usually, the configuration memory cells are implemented using Static Random Access Memory (SRAM) technology, hence the name SRAM-based FPGAs.

Column-based style. Today's commercial[1] FPGA architectures can be best described with the column-based style. To understand better we will consider a commercial FPGA from Xilinx called Zynq-SoC. The Programmable Logic of the SoC is made up of an Artix-7 series FPGA. It contains an array of Configurable Logic Blocks (CLB) which encapsulate LUTs, flip-flops and multiplexers. Each CLB contains 8 LUTs and is capable of realizing combinatorial and sequential logic. The array of CLBs is divided into a number of Clock Regions. Each clock region contains CLB columns with a fixed number of CLBs and the height of the CLB column remains the same in all the clock regions. There are multiple CLB columns adjacent to each other thus forming CLB rows as shown in Figure 2.2.

Current Xilinx 7 series FPGAs and UtlraScale CLBs contain two types of slices (SLICE-M and SLICE-L). Each slice includes 6-input LUTs (6-LUT) and 16 registers [19]. Each LUT is fracturable into two 5-LUTs. Therefore, each LUT has two outputs; the extra output enables us to use a 6-LUT as two 5-LUTs sharing

[1] In my work, I focus on the commercial FPGAs from Xilinx because they have supported dynamic reconfiguration for a longer time and better than Altera.

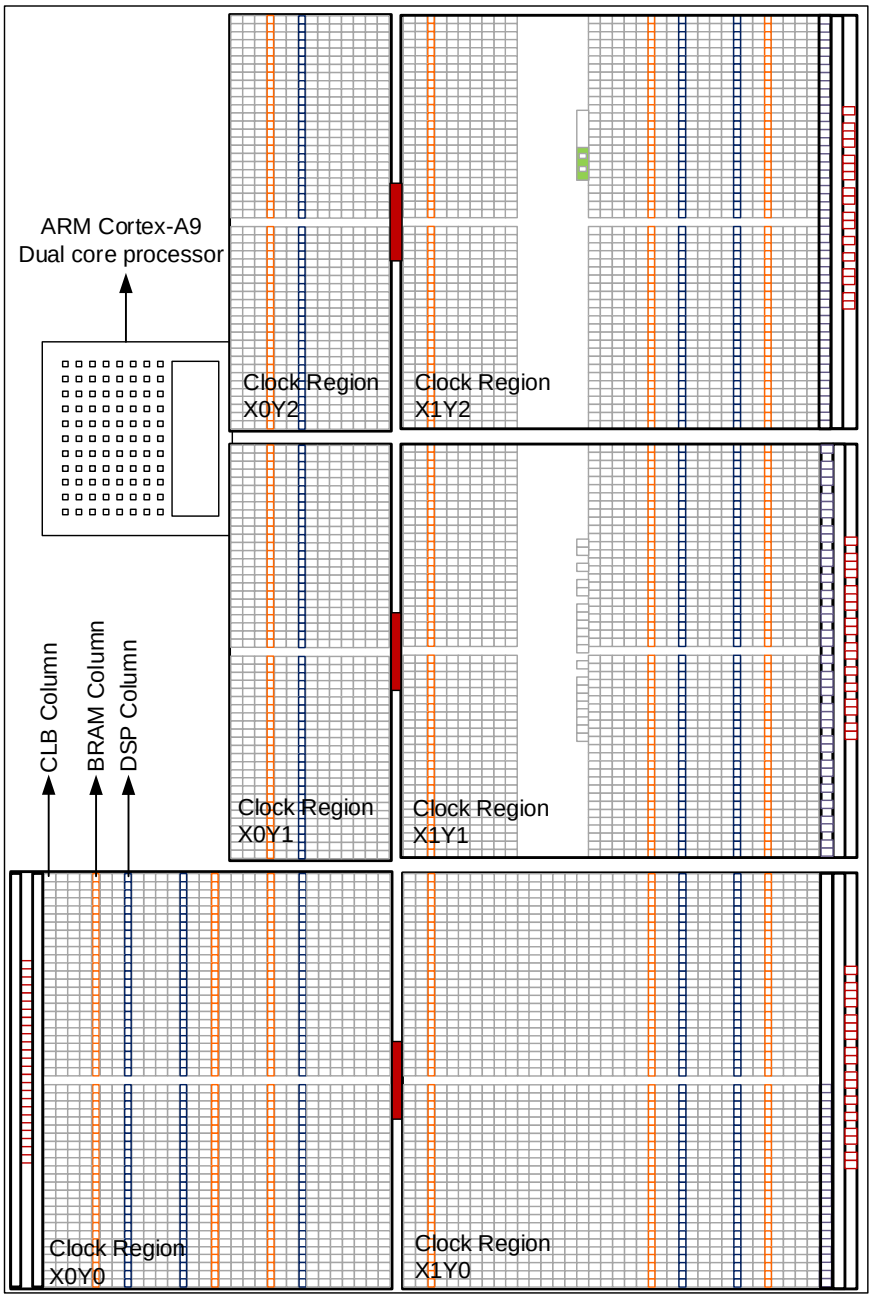


Figure 2.2: Column-based FPGA architecture: Zynq-SoC

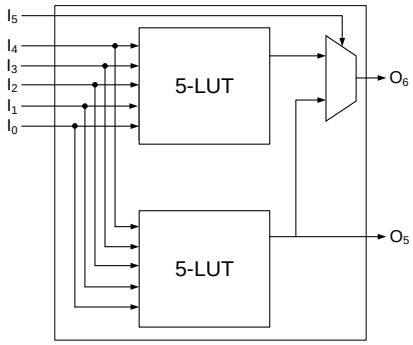


Figure 2.3: Schematic of a 6-input fracturable LUT

the same inputs or a combination of 6-input and 5-input function as shown in Figure 2.3.

Figure 2.4 shows the architecture of Slice-M. Each slice consists of four 6-LUTs enumerated alphabetically (A,B,C and D). There is a carry chain used for efficient implementation of adders and subtractors. The configuration memory LUTs can be used as a small memory called Distributed RAM or as a shift register LUT. The grey colored multiplexers in the figure are set via the configuration bits.

Figure 2.5 shows the architecture of Slice-L. This is the simple version of the Slice that does not contain additional circuitry such as carry chains and they cannot be used as Distributed RAM or shift registers. The grey colored multiplexers in the figure are set via the configuration bits.

There are other heterogeneous primitives available in the commercial FPGAs such as DSP (containing multiply accumulate operators) columns, Block RAM (BRAM) columns, high speed IO protocols (PCIe, Gigabit Ethernet, etc.), clock management resources, ADC. The most important primitive is the embedded processor such as ARM Cortex-A9 which is more powerful and efficient than the softcore processor (MicroBlaze) implemented on the programmable logic of the FPGA. These primitives help to meet the stringent performance requirements for a given application.

2.1.1 Xilinx commercial FPGA products

In this section, the Xilinx commercial FPGAs and their range of products are presented in order to understand the current reconfiguration architectures and see what the problems are to be solved in this thesis. The Xilinx FPGA products along with their technology node portfolios are listed in Table 2.1.

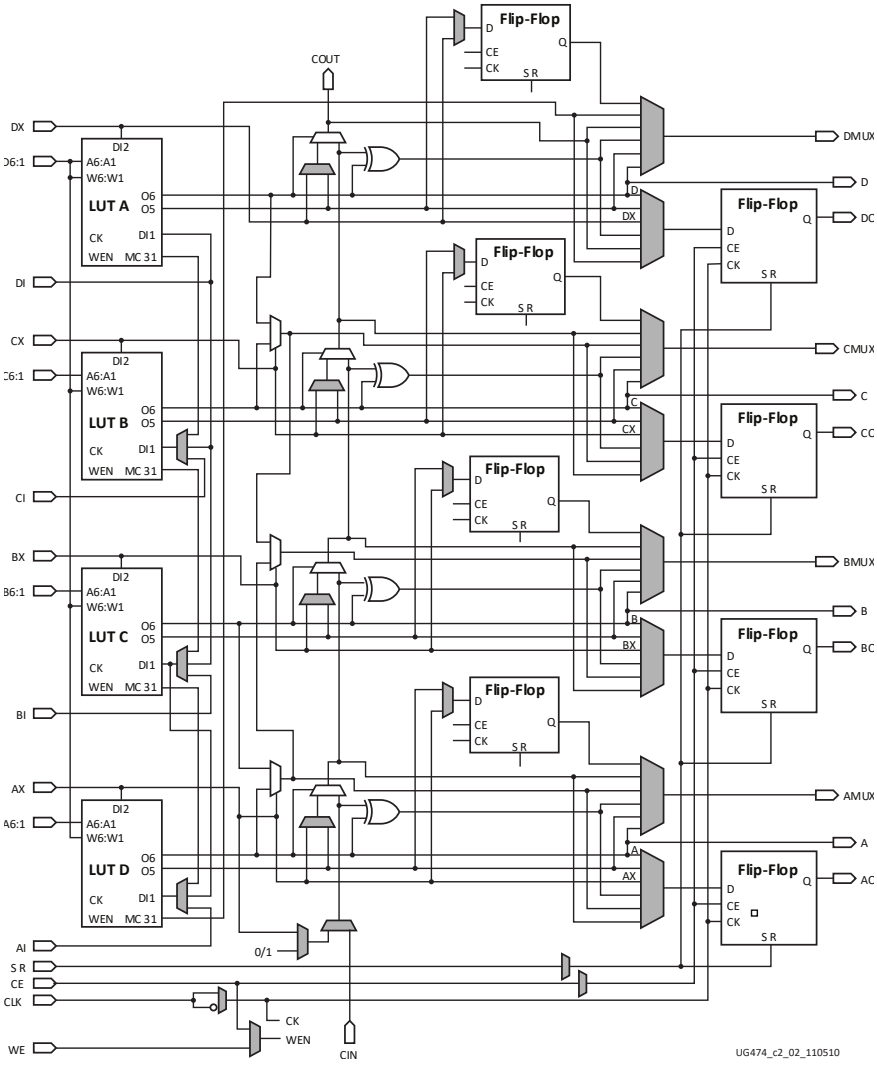


Figure 2.4: Schematic of the Slice-M (Xilinx Artix-7) [19]

2.1.1.1 Xilinx 7 series FPGAs

The Xilinx 7 series offers four FPGA families: Spartan-7, Artix-7, Kintex-7, and Virtex-7. The 7 series FPGAs address a range of system requirements ranging from low cost to ultra high-end connectivity bandwidth and signal processing capability for the high-performance applications [20]. The logic cell density of FPGAs ranging from Spartan-7 to Kintex-7 scales by a factor of two. The Virtex-7 FPGA is the largest FPGA among the 7 series FPGA family whose logic cell density is four

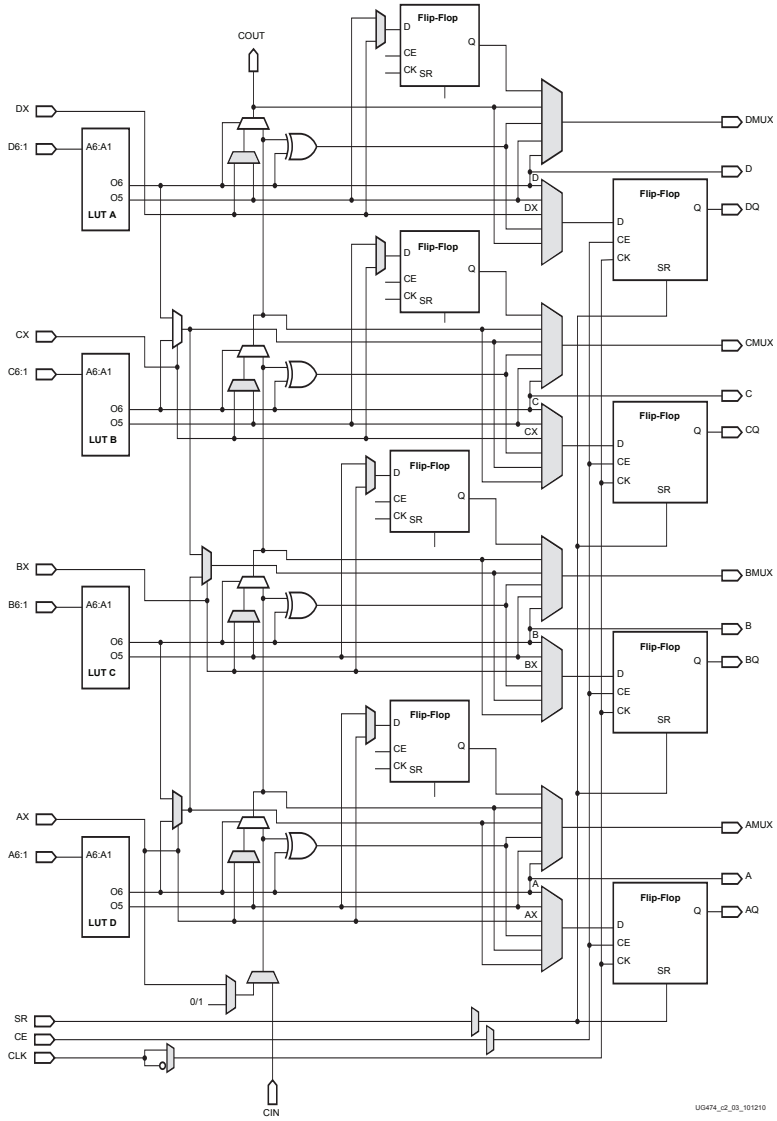


Figure 2.5: Schematic of the Slice-L (Xilinx Artix-7) [19]

Table 2.1: Xilinx FPGA products classification (2017)

7 series (28 nm)	UltraScale (20 nm)	UltraScale+ (16 nm)
Spartan7	Kintex UltraScale	Kintex UltraScale+
Artix7	Virtex UltraScale	Virtex UltraScale+
Kintex7		
Virtex7		

times bigger than Kintex-7.

2.1.1.2 Xilinx UltraScale and UltraScale+ FPGAs

The Xilinx UltraScale and UltraScale+ FPGAs are high-performance FPGAs, the architecture is optimized to focus on lowering the total power consumption (up to 40% lower power vs. 7 series generation) [21]. These device are built using both monolithic and next-generation 3D IC technology. The UltraScale products are fabricated with 20 nm technology and the UltraScale+ devices provide the highest performance and integration capabilities in a FinFET 16 nm technology.

The Kintex UltraScale FPGA has the high-performance architecture with a focus on price per performance. They have numerous power options that can balance between application requirements and the power envelope. The Virtex UltraScale FPGA is the industry's most capable high-performance FPGA. The Virtex families are optimized to address key market and application requirements [21].

2.1.1.3 Xilinx all programmable System-on-Chip products

In recent years, Xilinx has released series of System-on-Chip (SoC) FPGA products called Zynq all programmable SoC. The product combines FPGA fabric with a powerful embedded ARM processor, that can run a full-blown Linux OS. Thus, the Zynq provides a very robust heterogeneous processing system. These products can be classified based on their different range of applications: cost-optimized, mid-range and high-end as tabulated in Table 2.2.

Table 2.2: Xilinx SoC products classification (2017)

Cost-optimized	Mid-range	High-end
Zynq 7000**S** (Artix-7)	Zynq UltraScale+ MPSoC (CG)	Zynq UltraScale+ MPSoC (EG)
Zynq 7000 (Artix-7)	Zynq 7000 (Kintex-7)	Zynq UltraScale+ MPSoC (EV)

The Zynq 7000**S** (Artix) product is the low-cost SoC FPGA product. It has a single core ARM Cortex-A9 processor combined with the 28 nm Artix-7 FPGA that can be used as a coprocessor on the same die. This is the lowest cost entry point to the scalable Zynq-7000 platform. The product is suitable for industrial IoT applications such as embedded vision and motor control [22].

The Zynq 7000 (Artix) product has a dual-core ARM Cortex-A9 processor combined with the 28 nm Artix-7 or Kintex-7 FPGA for performance-per-watt and maximum design flexibility. The product can be used for a wide range of embedded applications including 4K2K Ultra-HDTV.

The Zynq UltraScale+ MPSoC devices come with more powerful 64-bit processor scalability along with hardcore engines for real-time, graphics, video, waveform, and packet processing.

- The Zynq UltraScale+ MPSoC (CG) comes with a dual-core Cortex-A53 and a dual-core Cortex-R5 real-time processing unit combined with the 16 nm FinFET+ FPGA. These devices are suitable for industrial motor control, sensor fusion, and industrial IoT applications.
- The Zynq UltraScale+ MPSoC (EG) comes with a quad-core ARM Cortex-A53 processor along with dual-core Cortex-R5 real-time processors and an ARM Mali-400 MP2 embedded GPU. The powerful SoC is combined with the 16 nm FinFET+ FPGA. The EG devices have specific processing elements that are suitable to implement next-generation wired and 5G wireless infrastructure, cloud computing, and Aerospace and Defense applications.
- The Zynq UltraScale+ MPSoC (EV) is built on top of the EG device with extra add-on integrated H.264 / H.265 video codec capable of handling high-speed multimedia applications to encode and decode up to 4Kx2K (60 fps). Therefore, the EV devices are suitable for multimedia, automotive ADAS, surveillance, and embedded vision applications.

2.1.2 Configuration bitstream

The configuration bitstream is a stream of bits (0s and 1s) that set all the LUT values and the multiplexer selection bits and thus define the functionality of a digital circuit on an FPGA. Besides the configuration data, the stream contains a set of commands that are used to orchestrate the programming of the FPGA. For example, the header of the bitstream can include a location address where to store the following configuration data. A detailed description of the construction of Xilinx FPGA (7 series) bitstreams is presented in [23]. The bitstream size of the largest Xilinx FPGA, Virtex-7 (7V2000T) is 56 MB.

2.1.3 Frame Structure

A frame of an FPGA is the smallest addressable element of an FPGA configuration. It can be viewed as a vertical stack of a fixed number of bits spanning a complete height of a row [24] [23]. A fixed data size of 2 words (1 word = 32 bits) are assigned to each CLB within the entire frame. This means that a set of LUT entries present in one CLB can be configured within those 2 words. However, the complete configuration data of an entire CLB containing multiple LUTs spans over multiple frames and each frame has its own unique frame address [24]. It should be noted that there exist one extra word called “HCLK config word” for each column within one frame as shown in Figure 2.6.

A single frame can contain truth table entries of multiple LUTs which are located in a single CLB column. In the Zynq-7000 family, there are 50 CLBs in one column, so a total of $50 \times 2 + 1 = 101$ words exist in one frame. The frame

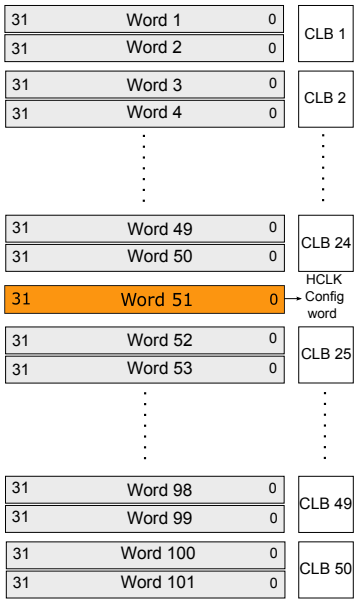


Figure 2.6: Frame structure of column-based Xilinx FPGA, Zynq-SoC (Artix-7)

size plays an important role during the reconfiguration process. Since a frame is the smallest addressable element, for every reconfiguration process, at least one frame has to be accessed via the Hardware Internal Configuration Access Port (HWICAP). A similar explanation applies for the previous versions of the Xilinx FPGAs: Virtex-II pro and Virtex-5. The device parameters that affect run-time reconfiguration for three different platforms of the Xilinx FPGA platforms are listed in Chapter 3, Table 3.1.

2.1.4 Configuration Interfaces on Xilinx FPGAs

The configuration ports are used to load the configuration bitstreams from a master onto an FPGA. There are multiple ports available on the FPGA chip.

- Serial, SPI, BPI and SelectMAP interfaces. These interfaces are used to configure the FPGA at boot-up. Several communication protocols are available through which an FPGA can be configured using an external master. For example, a microprocessor present in the desktop computer can be a master to program the FPGA. In some cases, an FPGA can act as a master itself and fetch the configuration bitstreams from an external configuration database.
- JTAG interface. The JTAG is a standard debugging port used by the external master. This port can be used for configuration and configuration read-back.

- ICAP/ICAPE2 interface. The ICAP interface is a primitive providing the embedded processor (present on the same chip) access to the internal configuration of the FPGA. Therefore, the ICAP is used for self-reconfiguration. The ICAP is compatible with the external SelectMAP interface.

  The HWICAP [25] is a hardware driver instantiated on the FPGA. It is built with the regular FPGA resources and encapsulates the ICAP primitive. The HWICAP connects the ICAP primitive with the bus so that it can be accessed by the embedded processor.

- PCAP interface. The Processor Configuration Access Port (PCAP) [26] is a reconfiguration controller used for Partial Reconfiguration on the Zynq-SoC. The PCAP is accessed through a device configuration interface (DevC) that has a DMA controller to transfer the bitstreams from the DRAM memory to the PCAP for reconfiguration. The PCAP is a configuration interface similar to the ICAP that is tightly coupled with the PS region of the Zynq-SoC. With the appropriate software drivers, the PCAP supports configuration read-back.

2.2 Conventional FPGA tool flow

Configuration bits define the functionality of FPGAs. Generating the right sequence of bitstreams is a crucial step in realizing the required digital circuit on FPGAs. This is accomplished using an automated tool flow called FPGA tool flow depicted in Figure 2.7. The corresponding intermediate results are shown in Figure 2.8.

Implementing a desired digital circuit starts from describing the hardware using a Hardware Description Language (HDL)[2] such as VHDL or Verilog. A VHDL code description of a 4-to-1 multiplexer is shown in Figure 2.8a. The conventional FPGA tool flow contains the following steps: Synthesis, Technology Mapping, Packing, Placement, and Routing.

2.2.1 Synthesis

The synthesis step transforms an HDL description into a Boolean network of logic gates (such as AND, NOT, flip-flop, etc.). The synthesis tool usually employs an optimization criterion such as reduced number of logic gates, lower logic depth, etc. The Boolean network of logic gates is represented by an AND-INVERTER-Graph (AIG) [27] as show in Figure 2.8b. In some cases the HDL code containing

[2]The hardware can also be described in a sequential programming language such as C, C++, SystemC, etc. Using suitable high-level synthesis tools, the sequential code is transformed into HDL.

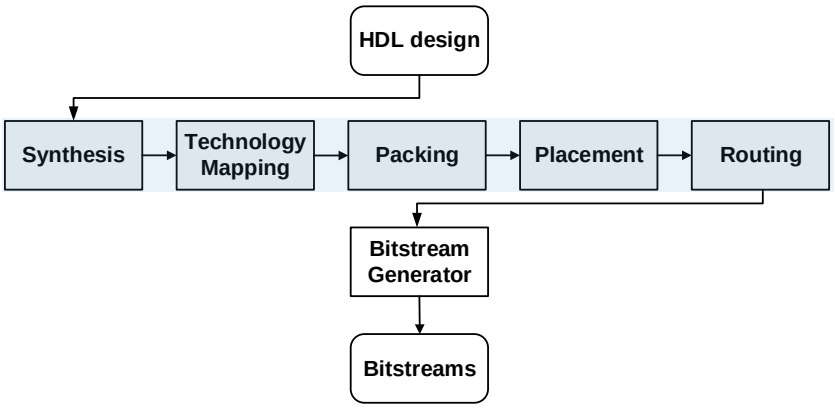


Figure 2.7: Conventional tool flow for FPGAs

concrete structures such as adders, multipliers or memory banks can be directly inferred into predefined primitives using LUTs, carry chains, DSPs and BRAMs.

2.2.2 Technology Mapping

During the technology mapping stage, the synthesized Boolean network is mapped onto the available resources of the target FPGA architecture such as LookUp Tables (LUTs) while optimization of circuit area and speed are being taken into consideration. Other primitives, such as DSPs, BRAM blocks, etc. are directly inferred from the HDL, and hence they are not mapped during technology mapping.

A LUT is a basic primitive of an FPGA that can implement any Boolean function of its input bits. The size of a LUT depends on the number of inputs and its truth table entries. Figure 2.8c shows the technology mapped netlist of the 4-to-1 multiplexer. In this example we used 3-input LUTs for clarity but commercial FPGAs use 4-input LUTs or 6-input LUTs.

2.2.3 Packing

In this step, the LUTs and flip-flops are clustered into CLBs without changing the interconnection structure. The packer tries to condense the interconnections between the LUTs so that they are grouped into the same CLBs (resulting in faster connectivity between LUTs) while it minimizes the interconnections between CLBs, which are relatively slower. The packed netlist after technology mapping is shown in Figure 2.8d.

In the Xilinx' latest tool flow, the packer is a part of the placement step [10].

2.2.4 Placement

In the placement step, the packed CLBs are placed or associated to specific blocks of the target FPGA architecture. Extensive optimization is considered so that interconnect wire length and interconnect delay are minimized. This is the most time consuming step in the FPGA tool flow. Figure 2.8e shows the result after the placement step.

2.2.5 Routing

The router configures the physical switch blocks and connection blocks to achieve the required interconnect according to the circuit netlist. The routed netlist will determine the critical path delay of the circuit. Therefore, multiple iterations of routing are performed by the router to meet the given timing constraints. For a large FPGA, the routing step can consume a huge amount of time. Figure 2.8f shows the result of the placed and routed circuit.

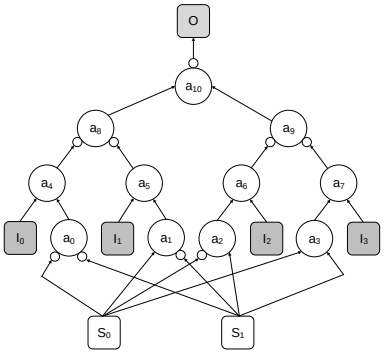
2.2.6 Bitstream generator

In the bitstream generation step, a series of bits (0s and 1s) is generated that corresponds to the exact implementation of the placed and routed circuit. The bitstream is called the *configuration*. The generated bitstream also consists of FPGA platform specific commands and settings that orchestrate the FPGA programming. The result of this step is a configuration bitstream.

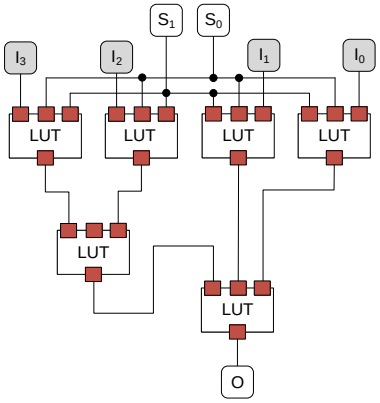
```
----------------------------------------
entity multiplexer is
port (
        I : in std logic vector (3 downto 0);
        S : in std logic vector (1 downto 0);
        O : out std logic
);
end multiplexer;

architecture behavior of multiplexer is
begin
        O <= I(conv_integer(S));
end behavior;
----------------------------------------
```

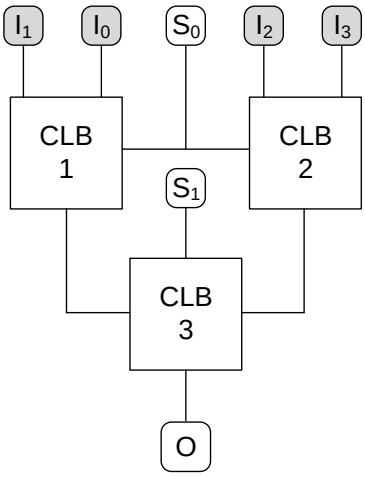
(a) HDL design: 4-to-1 multiplexer described in VHDL



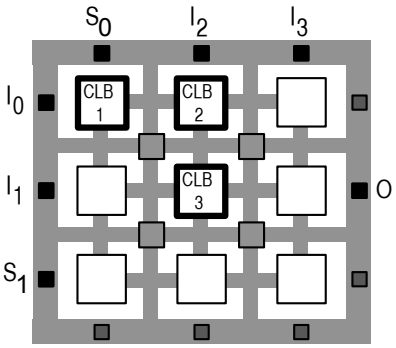
(b) After Synthesis: AIG of 4-to-1 multiplexer



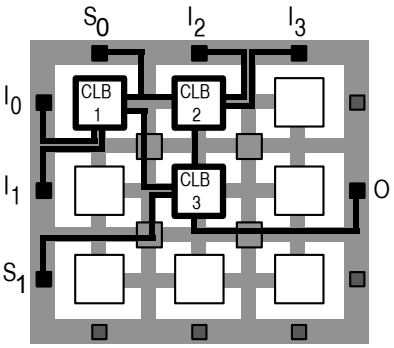
(c) After Technology Mapping: 4-to-1 multiplexer mapped on to 3-input LUTs



(d) After Packing



(e) After Placement [17]



(f) After Routing [17]

Figure 2.8: Intermediate results of the conventional FPGA tool flow

3

Dynamic Circuit Specialization

This chapter presents a novel technique for run-time FPGA reconfiguration called Dynamic Circuit Specialization (DCS). It is invented at the Hardware and Embedded Systems research group of Computer Systems Lab at Ghent University. DCS is suitable to implement parameterized applications on FPGAs. I also give the background of micro-reconfiguration and parameterized configuration that helps to understand DCS more precisely. My contribution to this technique begins by evaluating the performance of DCS on different Xilinx FPGA platforms, its power measurement, and analysis, and finally, I present a range of parameterized applications that could benefit from this novel technique.

3.1 What is DCS?

Dynamic Circuit Specialization (DCS) is a technique used to optimize parts of a parameterized application and switch between the specialized parts for the current specific conditions utilizing Partial Reconfiguration (PR) at run-time [28] [29]. This technique improves the functional density[1] of the FPGA.

The application is said to be parameterized when some of its inputs, called parameters, are infrequently changing compared to the other inputs. Instead of implementing these parameter inputs as regular inputs, in the DCS approach, these inputs are implemented as constants, and the design is optimized for these con-

[1]The functional density is defined as the number of computations that can be performed per unit area and unit time (for more details refer Section 3.4).

stants. When the parameter values change, the design is re-optimized for the new constant values by reconfiguring the FPGA.

In order to implement DCS for parameterized applications, I will present two intermediate steps: parameterized configuration and micro-reconfiguration. These steps are derived from the conventional FPGA implementation.

3.2 Parameterized configuration

A parameterized configuration contains bits that are not only static binary (0's and 1's) but also multi-valued Boolean functions of infrequently changing parameters as depicted in Figure 3.1. For specific parameter values, we can instantly derive specialized configurations by evaluating the Boolean functions for the given parameter values. In practice, the SRAM cells of the FPGA can hold only binary bits (0's or 1's). Therefore, the Boolean functions have to evaluated well before reconfiguring the FPGA configuration memory.[2] Thus, the functionality and property of the parameterized application can be instantly specialized using parameterized configuration once the parameter values are known.

For every change in parameter input values of a parameterized application, the functions are evaluated resulting in specialized bitstreams. Therefore, multiple specialized configurations can be generated by evaluating the Boolean functions instead of compiling the bitstreams from scratch using the conventional FPGA tool flow. This results in lower compilation costs per configuration and reduces the amount of storage needed in the configuration database. Generating a specialized configuration does not need to undergo time-consuming steps of the flow and to solve computationally hard problems such as placement and routing as is the case in the conventional tool flow. These problems are already solved when the parameterized configuration is generated.

3.2.1 Two-staged tool flow for parameterized configuration

The conventional FPGA tool flow cannot be used to generate parameterized configuration. Therefore, a new two-staged tool flow is needed [13]. The two-staged tool flow for generating a parameterized configuration is depicted in Figure 3.2 and consists the generic stage and the specialization stage.

In the generic stage, a HDL design with parameterized inputs (annotated by "`-PARAM`" in VHDL) is processed to yield a partial parameterized configuration. A Partial Parameterized Configuration (PPC) is a part of a FPGA configuration that contains bitstreams expressed as Boolean functions of parameters. In the specialization stage, these Boolean functions are evaluated for specific parameter values

[2] Parameterized configurations can be generated and used on existing FPGA hardware without the need of changes to the FPGA architecture.

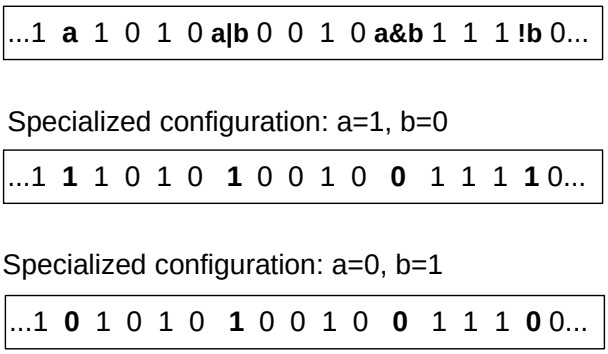


Figure 3.1: A parameterized configuration containing Boolean functions of two parameters 'a' and 'b' with two derived specialized configurations

to generate a specialized configuration. The following tool flow steps explain the generic stage and are adapted from the conventional tool flow.

3.2.1.1 Synthesis

In this step, the HDL design is converted into a network of logic gates. The parameter inputs described in the HDL are annotated parameter inputs and this annotation makes the difference between parameter inputs and regular inputs. The parameter inputs are also a part of the Boolean network of logic gates produced after synthesis and are not treated differently in the synthesis step.

3.2.1.2 Technology Mapping

During the mapping stage, the synthesized Boolean network is mapped onto the available resources of the target FPGA architecture such as LookUp Tables (LUTs), DSP blocks and BRAMs while optimization of circuit area and speed are being taken into consideration. The conventional mapping tool would map to the static LUTs and hence it would result in the conventional bitstreams after place and route. To generate a parameterized bitstream, authors of [13] change the conventional mapping tool to a tunable version, TLUTMAP, so that the Boolean functions of parameter inputs are mapped on to Tunable LookUp Tables (TLUTs). These are virtual LUTs that differ from conventional LUTs in the fact that their lookup entries are defined as the Boolean functions of the parameter inputs instead of static ones and zeros.

Presently, the parameterization of BRAM and DSP blocks is not yet possible but parameterization of the routing switches called TCONs[3] is established at the virtual FPGA level. However, the practical implementation in commercial FPGAs

[3]Parameterization of interconnect switches is achieved by using the TCONMAP [30] mapper.

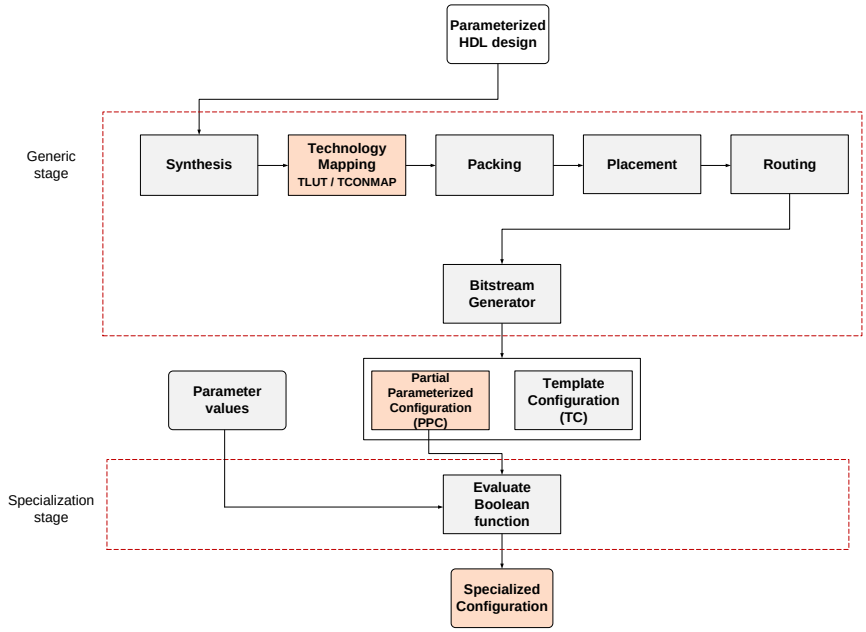


Figure 3.2: Two-staged tool flow for parameterized configurations

is yet to be done [30]. The TLUTMAP mapping algorithm is described in [13] and can be integrated with the conventional Xilinx tool flow which is explained in [31].

- A TLUT is a virtual LUT of which the truth table entries are defined as Boolean functions of parameters instead of ones and zeros.
- A TCON is a point-to-point connection which can be made or broken depending on the value of a Boolean function of parameters. It is implemented using the FPGA's routing network (i.e., the reconfigurable switch blocks and connection blocks). The parameterized configuration of the used switch and connection blocks will be derived from the Boolean function.

3.2.1.3 Placement, Routing and Bitstream generation

In the placement step, the mapped resources are placed or associated to specific blocks of the target FPGA architecture. Extensive optimization is considered so that interconnect wire length and interconnect delay is minimized. The router configures the physical switch blocks to achieve the required interconnect according to the circuit. As the parameters are already included in the LUT functionality through the Boolean functions, they are no longer present in the physical implementation of the netlist so the entire netlist can be placed and routed as if the

parameters where not present. Therefore, a conventional placer and router can be used.

The placed and routed netlist is used by the conventional Bitstream generator to generate FPGA configuration bitstreams.

The final output of the generic stage is the Template Configuration (TC) and Partial Parameterized Configuration (PPC). TC is a static bitstream which contains static ones and zeros, which are used for configuring during the start of the FPGA. The PPC contains sets of multi-output Boolean functions of the parameter inputs. The PPC needs to undergo the specialization stage, along with parameter values to produce an efficient specialized configuration.

The specialization stage consists of a Specialized Configuration Generator (SCG). The SCG takes the PPC and the parameter values as inputs and evaluates the Boolean functions of parameter inputs for given parameter values to produce a specialized configuration. After that, all the TLUTs are reconfigured by downloading the specialized configuration during run time and thus accomplishing run time reconfiguration. The SCG can be implemented on a hard-core embedded processor in the FPGA such as the ARM Cortex-A9, on a soft-core processor such as a MicroBlaze or on a custom processor (CP) which is more specifically designed to evaluate Boolean functions only. Different types of SCG implementations and their details are described in [32].

The SCG reconfigures the FPGA via a configuration interface called the Internal Configuration Access Port (ICAP) by swapping the specialized bitstreams into the FPGA configuration memory. The configuration controller such as the HWICAP (in a conventional DCS implementation, the HWICAP is used as a reconfiguration controller) encapsulates the ICAP primitive (port) of the FPGA and forms a controller that orchestrates the swapping of specialized bitstreams via the interface port ICAP. The bitstreams are accessed in the form of frames and a frame is defined as the smallest addressable element of the FPGA configuration data. Each frame contains reconfiguration bits of tens of LUTs and has its unique frame address that can be used to point to the frame during the reconfiguration.

3.2.2 Micro-reconfiguration

Micro-reconfiguration is a technique to change the configuration of minor or few resources of the FPGA. Therefore, micro-reconfiguration is the main application of DCS. The micro-reconfiguration can be best explained with the implementation of DCS on commercial FPGAs such as Xilinx' Zynq-SoC.

3.2.2.1 DCS on Xilinx FPGAs

The implementation of DCS on the Xilinx FPGAs, such as the Zynq-SoC, is shown in Figure 3.3. The SCG is realized on an embedded processor (ARM Cortex-A9

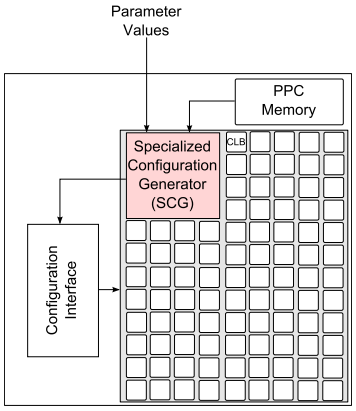


Figure 3.3: Dynamic Circuit Specialization on Xilinx FPGAs

dual core processor or a MicroBlaze soft core processor).

The PPC Boolean functions are stored in the memory such as DRAM memory of the Zynq-SoC. The ICAP is used as a configuration interface. The HWICAP reconfiguration controller is responsible for orchestrating the replacement of the stale frames with specialized frames present in the configuration memory of the FPGA.

3.2.2.2 The HWICAP driver "XhwIcap_setClb_bits" function

The HWICAP supports a software driver function called "XhwIcap_setClb_bits" to perform the reconfiguration. This function accepts two crucial function arguments:

1. Location co-ordinates of a LUT: this information is used to generate the frame address that is used to point to the frame that contains truth table entries of the TLUT that is implemented on this physical LUT.

2. Truth table entries: these are the specialized bits generated after the specialization stage of the DCS tool flow. The LUT truth table entries need to be overwritten with these specialized bits.

The reconfiguration takes place in 3 steps:

1. Read frames: using the frame address, a set of four consecutive frames containing the previous specialized truth table entries of a TLUT are read from the configuration memory.

2. Modify frames: the previous truth table entries of a TLUT are replaced by the specialized bits. The modified frames contain specialized bitstreams.

3. Write-back frames: using the same frame address, the modified four frames are written back to the configuration memory, thus accomplishing the micro-reconfiguration.

Micro-reconfiguration is a fine-grained form of reconfiguration used for DCS. Therefore, a reconfiguration controller in this case should be capable of reading, modifying and writing the frames from the configuration memory and a processor should take care of executing the cycle of read, modify and write-back of frames.

The micro-reconfiguration incurs 4 major costs compared to a generic, non-reconfigurable implementation. These costs are major drawbacks of DCS:

1. PPC memory size: memory space required to store all the Boolean functions of the parameterized application.

2. Evaluation time: time taken by the SCG to evaluate the Boolean function for a specific set of parameter values.

3. Reconfiguration time: time taken to update all the TLUTs of a parameterized design with the specialized bits. In other words, time taken to accomplish the micro-reconfiguration.

4. Power consumption: the idle (static, independent of whether or not we are reconfiguring) and dynamic (during the micro-reconfiguration) power consumed by the reconfiguration infrastructure.

The performance of DCS is entirely dependent on the trade-off between the benefits and the costs of micro-reconfiguration. The changes in the FPGA architecture directly influence the micro-reconfiguration costs. Therefore, it is important to understand how this trade-off for DCS evolves with the evolution of Xilinx FPGA platforms.

3.2.3 DCS on a self-reconfigurable platform for the Zynq-SoC

The self-reconfigurable platform is a soft layer on the FPGA configured to realize a DCS system on the Zynq-SoC. The self-reconfigurable platform consists of four main components: an embedded processor (ARM Cortex-A9 or a MicroBlaze), a reconfiguration controller (or reconfiguration interface), a parameterized application and a system bus to connect all these components.

Figure 3.4 shows a self-reconfigurable platform implemented on the Zynq-SoC for an adaptive parameterized FIR filter application. The PPC Boolean functions are stored in the DRAM memory of the Processor System (PS), and all the actions of the micro-reconfiguration are controlled by the ARM Cortex-A9 processor (clocked at 667 MHz). Therefore, the user can use a simple program to run software on the processor to monitor and measure the reconfiguration activity. The

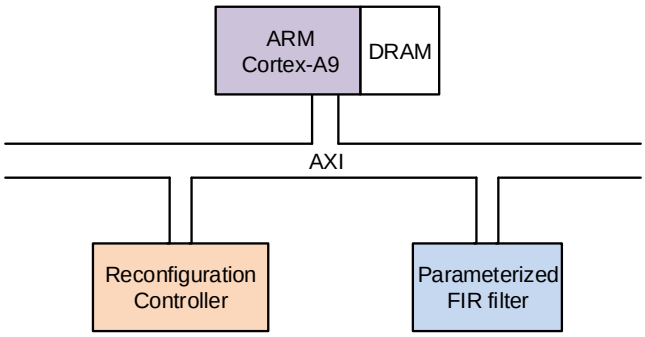


Figure 3.4: Dynamic Circuit Specialization on a self-reconfigurable platform for the Zynq-SoC

whole system is connected using the AXI bus (clocked at 100 MHz) for the data transfer.

3.3 Examples of parameterized applications

This section presents a set of applications that benefit from DCS.

- Ternary Content-Addressable Memory (TCAM): a regular Content-Addressable Memory (CAM) is a memory where a piece of data (bits) can be efficiently located. This means that, for a given input of data, the CAM returns the address of the data if the data is present in the memory. The TCAM [33] can contain not only the bits, but it can also store don't care values.

  A parameterized 16-bit TCAM memory with 32 entries was implemented using DCS. The data content of the TCAM memory change infrequently and therefore, the data can be considered as the parameters for a DCS implementation. Therefore, to change the content of the memory, the FPGA is reconfigured with the new data entry (specialized data). A 32-bit TCAM implemented with 128 entries showed a 35% reduction in LUT resources and 40% increase in clock frequency over the conventional implementation of the same TCAM [13].

- Multiply and Accumulate (MAC): the MAC operator is used as a basic Processing Element (PE) of the Virtual Coarse-Grained Reconfigurable Array (VCGRA) grid. The VCGRA grid is designed to implement a high performance computing filter application [34]. Each PE has a settings register that configures the function of the PE. The PE was implemented using

parameterized configuration with the settings register value as a parameter input and thus, for every change in settings value the FPGA is reconfigured with the new settings. In [34], a fully parameterized implementation of a floating-point MAC operator PE showed a reduction of 30% in LUT resources compared to the conventional implementation of the MAC operator. The maximum clock frequency was increased by 13% when implemented on the Zynq-SoC. A detailed explanation on VCGRA and examples of VCGRA grids is presented in Chapter 5.

- Advanced Encryption Standard (AES) encoder: the AES encoder [35] is a streaming encryptor that encrypts 128 bits of data per clock cycle. We used a pipelined implementation of 10 rounds of AES encryption. We used DCS to optimize the AES encoder for a specific encryption key. The encoder is specialized for every change in key (parameter) input. The parameterized AES encoder is useful if a large amount of data needs to be encoded with the same the encryption key. In [36] several implementations of the AES encoder are presented. The parameterized AES encoder implemented with DCS (on Virtex-II pro) showed a maximum LUT resource saving of 20.3% and 5.6% of increase in clock frequency.

- Encryption processor for π-Cipher: the processor encrypts 16-bit user input messages using a π-function. The key generator module of the processor is implemented in a parameterized configuration. The user key input is a parameter value for the key generator, for every change in user key, a detailed explanation of the parameterized key generator module is described in [37].

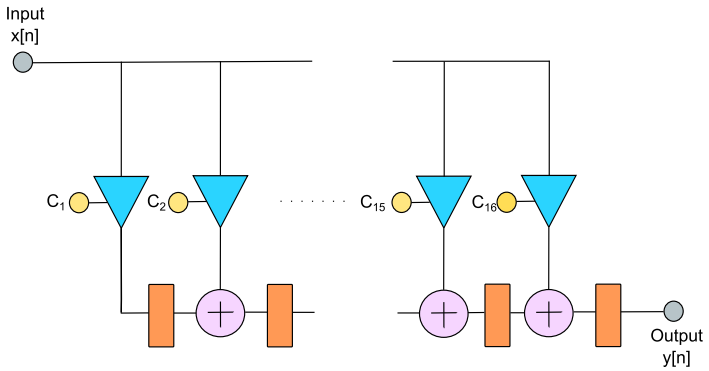


Figure 3.5: 16-tap, 8-bit FIR filter

- Finite Impulse Response (FIR) filter: an adaptive 16-taps, 8-bit FIR filter (Figure 3.5) is implemented using DCS. The filter taps are parameterized. Hence, for every infrequent change in coefficient input values, a specialized

bitstream is generated. The FIR filter is built with sixteen 8-bit multipliers (implemented using LUTs of the device), and they consume 384 TLUTs of the Zynq-SoC FPGA. This filter can be used for a DSP application in which FIR filters are used to realize the filtering of unwanted bandwidth of signals. If the frequency of the bandwidth is required to change infrequently, then a parameterized FIR implementation would suit better than the classic FIR implementation. In [13], a resource reduction of 42% in the LUT resources and 23% increase in clock frequency is obtained. In the remaining chapters, unless stated otherwise, I use the FIR filter as the main application for my experiments.

3.4 Functional Density

To compare DCS overheads and gain (optimized implementation) we use a metric that combines performance and cost of the design. The metric is called Functional density [38].

The functional density (F_d) is defined as the number of (useful) computations (N) that can be performed per unit area (A) and unit time (T) as shown in Equation 3.1.

$$F_d = \frac{N}{A \times T} \tag{3.1}$$

I elaborate this equation for a design implemented with and without DCS technique.

3.4.1 Functional density for generic implementation

For a design implemented without DCS (generic implementation), the functional density simply results in:

$$F_d^{generic} = \frac{N}{A_{circ} \times T_{exec}} \tag{3.2}$$

where T_{exec} is the time it takes to perform N computations using the total resource cost of the circuit A_{circ}.

3.4.2 Functional density for DCS implementation

For a design implemented with DCS technique, the functional density results in:

$$F_d^{DCS} = \frac{N}{(A'_{circ} + A_{DCS}) \times (T'_{exec} + n_{DCS} \times T_{DCS})} \tag{3.3}$$

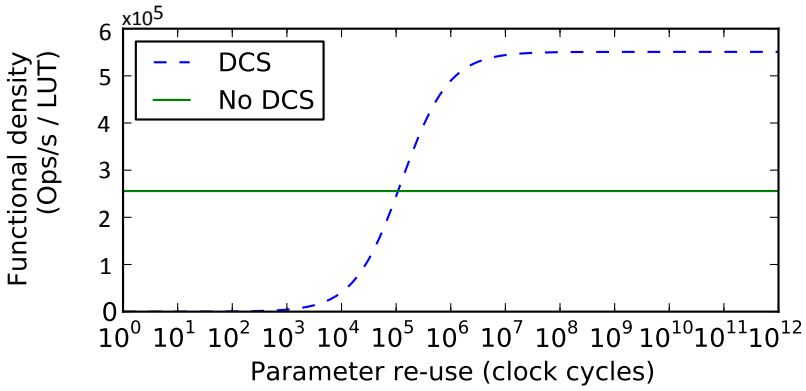


Figure 3.6: Functional Density curves for TCAM

where T_{DCS} is the time overhead of one reconfiguration procedure (time taken to evaluate Boolean functions and reconfigure the TLUTs of the design) and A_{DCS} the cost of hardware overhead of DCS (this mainly includes, FPGA resources utilized to implement SCG and reconfiguration controller). The variable n_{DCS} represents the number of times the DCS design needs to be optimized for new conditions while running the parameterized application.

Also, T'_{exec} can be smaller than T_{exec} if the optimized design can be clocked at higher clock frequency. A'_{circ} is smaller than A_{circ} since the optimized circuit needs fewer resources.

The overall gain is achieved only if $F_d^{generic} < F_d^{DCS}$. Therefore, it is very important to minimize A_{DCS} and T_{DCS}.

The functional density curve for the parameterized TCAM is depicted in Figure 3.6. The functional density is plotted against the rate of change of parameter values (i.e., the number of clock cycles in between two parameter changes as the reconfiguration overhead strongly depends on how many times a reconfiguration is needed). The functional density shows the efficiency of the implementation as a function of how fast the parameter values change.

Clearly, the magnitude of the functional density of the DCS implementation is higher that that of the generic implementation (no DCS) when the average time between parameter change is more than 10^5 clock cycles.

3.5 Performance evaluation of DCS on Xilinx FPGAs

In this section I present the evaluation of the performance of DCS on three different Xilinx FPGA architectures: Virtex-II Pro, Virtex-5 and Zynq SoC. Each of

these architectures has their own pros and cons. Our main objective is to evaluate how DCS will perform under the new Xilinx FPGA architectures. I evaluate the reconfiguration time, the specialization time and the size of the memory occupied by the design while using DCS on the three Xilinx FPGA architectures and compare them accordingly.

We used a 16-tap FIR filter with 8-bit wide coefficient values as a parameterized design for showing the benefits of DCS. All coefficients are parameterized inputs and hence for each infrequent change in the coefficient value, a specialized bitstream is generated and the filter taps containing multiplications are reconfigured accordingly.

The multiplications for the FIR filter are designed for a (T)LUT implementation keeping in mind that they should suit for all three FPGAs. Since Virtex-II Pro has a LUT input size of 4, we make use of 4-bit multiplications only. The filter requires 16 8-bit multiplications. Two 4-bit multiplications are combined to form one 8-bit multiplier and therefore, a total of 32 4-bit multiplications were used to build a complete FIR filter design. It is known that a 4-bit multiplier is mapped onto 12 TLUTs, therefore 1-tap of the FIR filter contains 24 TLUTs [32].

Table 3.1 shows the device names and the corresponding boards we used for our experiments. The number of inputs to a LUT for the corresponding FPGAs is also present in the same table. The number of inputs will influence the memory size of the PPC functions and the time taken to evaluate the Boolean functions within the LUT entries.

The embedded processors of both type (soft-core and hard-core) and their respective clock configurations, that are used in the Xilinx FPGAs are tabulated in Table 3.1. These processors are used to generate specialized configurations by evaluating the Boolean functions. To describe the specialization procedure, we use a standard C program with Xilinx SDK.

The ARM Cortex-A9 processor within the Zynq SoC is a dual core processor but we use only a single core. It has instruction and data caches each of size 32 KB [22]. The PowerPC processor for both Virtex-II Pro and Virtex-5 was configured with instruction and data caches each of size 32 KB [39] [40]. The MicroBlaze can also be configured to enable instruction and data caches. However, these caches in a softcore processor are just a reserved memory space in BRAMs and are not an actual or physical dedicated cache memory. Since the PPC functions already reside in BRAMs we did not enable caches for the MicroBlaze.

The HWICAP is used as a configuration interface and is responsible for loading the specialized bitstreams into the FPGA configuration memory. Table 3.1 shows the configurations for the HWICAP that we used for our experiments. The HWICAP throughput is also tabulated and it shows the rate at which the frames are read from the configuration memory and the rate at which the frames are written into the configuration memory.

Table 3.1: Xilinx FPGA device details

	Virtex-II Pro (XC2VP30)	**Virtex-5**	**Zynq SoC**
Device name	XC2VP30 -FF896-7C	XC5VFX70T -FFG1136	XC7Z020 -CLG484-1
Board name	XUPV2P Development System	ML507 Evaluation Platform	ZedBoard
LUT inputs (k)	4	6	6
LUT entries	16	64	64
Hard-core Processor	PowerPC 405 Core	PowerPC 440 Core	ARM Cortex-A9
Soft-core Processor	MicroBlaze (6.00.b)	MicroBlaze (8.20.b)	MicroBlaze (8.40.a)
Hard-core CPU clock (MHz)	300	400	667
Soft-core CPU clock (MHz)	100	100	100
HWICAP type	OPB HWICAP (1.00.b)	XPS HWICAP (5.01.a)	AXI HWICAP (2.03.a)
HWICAP clock (MHz)	66.67	100	100
HWICAP throughput (non-DMA) (MB/s)	10	19	19
HWICAP port width (bits)	8	32	32
Frame size (32-bit words)	206	41	101
Bus type	OPB + PLB	PLB	AXI
Bus clock (MHz)	66.67	100	100

Table 3.2: PPC evaluation time in microseconds

	Virtex-II Pro	**Virtex-5**	**Zynq SoC**
Hard-core Processor	6.6	4.6	1.7
Soft-core Processor	18.7	28.3	28.9

The size of one frame for an individual FPGA and the bus that we used to connect our parameterized design on a self reconfigurable platform, are tabulated in Table 3.1. The size of a frame describes the minimum number of words (1 word = 32 bits) in a bitstream that needs to be replaced for each reconfiguration process and has a direct influence on the reconfiguration time. For the Virtex-II Pro family, the frame size is not the same for the entire family and varies for different FPGA devices [41]. The specification of the bus can be seen in [42] [43] [44] and the bus interconnections in a typical embedded design are described in [31]. In the following sections, I present the results of my experiments and compare them to evaluate the TLUT-based DCS.[4]

3.5.1 Boolean function evaluation time

Table 3.2 shows the time taken by the processors to evaluate the Boolean functions for one multiplier (one coefficient) during the specialization phase.

3.5.1.1 Evaluation time - Hard-core Processors

We compare the performance of evaluating Boolean functions on all 3 hard-core processors - PowerPC of Virtex-II Pro, PowerPC of Virtex-5 and ARM Cortex-A9 of the Zynq SoC. Figure 3.7 shows the plots and it is clear that the ARM Cortex-A9 takes the least amount of time compared to both PowerPCs and hence it can claim to be very efficient. The PowerPC in a Virtex-5 is more efficient than the PowerPC in a Virtex-II Pro, showing the improvements in the newer processor architectures with higher clock frequency support. It is to be noted that the number of Boolean functions that needs to be evaluated increases for the Zynq SoC and the Virtex-5 compared to the Virtex-II Pro because of an increase in LUT inputs and corresponding LUT entries.

3.5.1.2 Evaluation time - Soft-core Processors

We also compare the performance of evaluating Boolean functions on all 3 soft-core processors - MicroBlaze of Virtex-II Pro, MicroBlaze of Virtex-5 and MicroBalze of Zynq SoC. Figure 3.8 shows that the efficiency of the MicroBlaze of

[4]The tool flow to implement TLUT-based DCS can be accessed at [45].

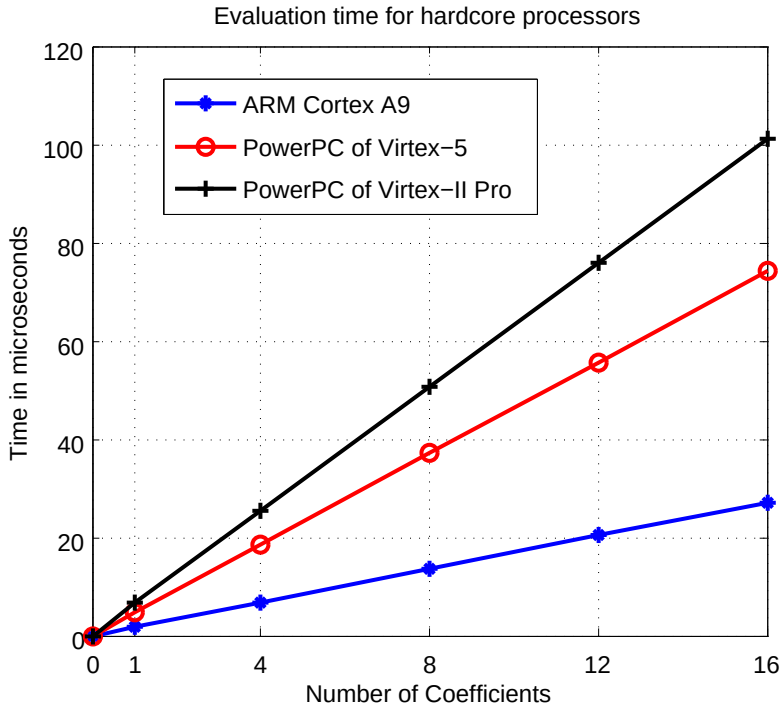


Figure 3.7: Evaluation time comparison of hard-core processors

both the Virtex-5 and the Zynq SoC are almost the same. Interestingly, the MicroBlaze in the Virtex-II Pro consumes less time and hence proves to be a very efficient soft-core processor. However, the main reason for this behaviour is that the number of LUT entries for the Virtex-II Pro (16) is smaller than for the Virtex-5 and Zynq SoC (64).

Comparing the hard-core processor with the soft-core processor in each FPGA shows that hard-core processor is always more efficient than the soft-core. The main advantage of using the hard-core processors is that they support a very high clock frequency which influences the evaluation of Boolean functions and also slightly the reconfiguration time. However, in some FPGAs it is inevitable to use soft-core processors due to the lack of availability of embedded hard-core processors in their FPGA architecture.

3.5.2 Reconfiguration time

Table 3.3 shows the time spent during reconfiguring one multiplier of the FIR filter which is composed of 12 TLUTS.

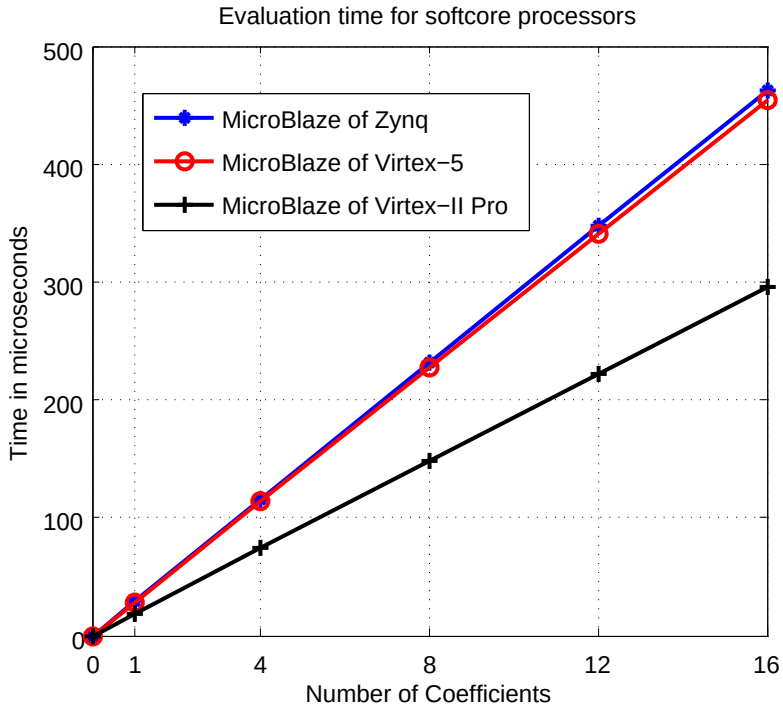


Figure 3.8: Evaluation time comparison of soft-core processors

Table 3.3: Reconfiguration time in milliseconds

	Virtex-II Pro	**Virtex-5**	**Zynq SoC**
Hard-core Processor	0.5	1	2.8
Soft-core Processor	1.89	1.4	4.0

The reconfiguration time is defined as the time required to update the configuration of all TLUTs using new specialized bitstreams. The updating process involves read, modify and write steps as explained in Section 3.2.2.2. Figure 3.9 shows the bar graph of the reconfiguration time for the 3 different FPGAs using respectively hard-core and soft-core processors. The normalized values are tabulated in Table 3.4, the normalization is with respect to FPGAs mentioned in the parenthesis. It is clear that the reconfiguration time overhead is not the same for Virtex-II Pro and Virtex-5. The resources that influence the reconfiguration time overhead are the frame size, interconnect bus speed, HWICAP port width and HWICAP clock. The Virtex-II Pro has a lower capacity of these resources compared to the Virtex-5. However, the number of frames to be reconfigured for a TLUT

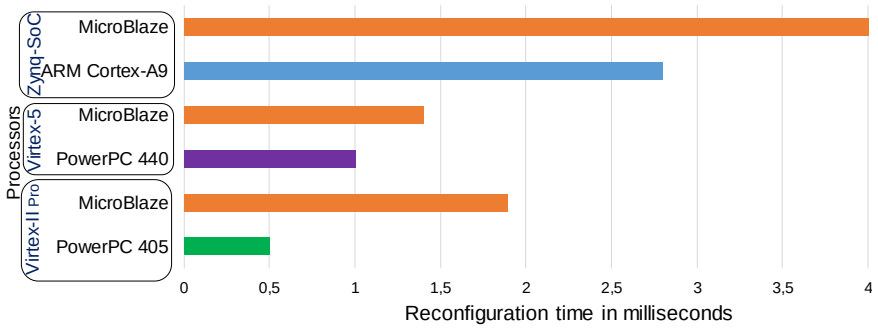


Figure 3.9: Reconfiguration time comparison

also has a direct influence on the reconfiguration time and should be considered during the comparison. Clearly, the number of frames to be reconfigured in the Virtex-5 is higher than for the Virtex-II Pro and therefore it consumes more time during reconfiguration.

The reconfiguration time overhead of the Zynq SoC is almost double the overhead of the Virtex-5. The main reason for the increase in reconfiguration time is the increase in the number of words per frame for the Zynq SoC compared to the Virtex-5. It is to be noted that the interconnect bus speed, HWICAP port width and HWICAP clock frequency are the same for the Virtex-5 and the Zynq SoC. The same note applies to the MicroBlaze results in all 3 FPGAs. The maximum clock frequency of the HWICAP that can be used for a reliable implementation is shown in Table 3.1. The HWICAP proves to be the bottleneck for the reconfiguration process. The hard-core processor clock speed is much higher than for the HWICAP, so obviously the HWICAP cannot process and synchronize directly with the processor. The processor will stall some clock cycles until the frames are swapped in and out of the configuration memory at the speed of the HWICAP clock.

Observing these results, we see that the DCS tool flow encounters a higher reconfiguration time overhead for the newer Xilinx FPGA architectures. In order to reduce the reconfiguration time overhead, one would have to improve the bus architecture and the HWICAP speed. The hard-core processors are fast enough to reduce the evaluation time and processing the frames during read, modify and write-back operation during reconfiguration, even when considering the increase in the LUT entries size. The Virtex-II Pro has a reconfiguration time overhead that is relatively smaller compared to the Virtex-5 and the Zynq SoC but it has a lower number of LUT entries compared to the newer FPGAs.

Normalizing the reconfiguration time with respect to the respective hard-core processors in all 3 FPGAs, shows that the use of hard-core processors is more efficient in reconfiguration than using the respective soft-cores. Table 3.5 shows

Table 3.4: Normalized Reconfiguration time (FPGAs)

	Virtex-II Pro	**Virtex-5**	**Zynq SoC**
Hard-core Processor(Virtex-II Pro)	1	2	5.6
Soft-core Processor (Virtex-II Pro)	1	0.7	2.1
Hard-core Processor (Virtex-5)	0.5	1	2.8
Soft-core Processor (Virtex-5)	1.35	1	2.8

Table 3.5: Normalized Reconfiguration time (hard-core processors)

	Virtex-II Pro	**Virtex-5**	**Zynq SoC**
Hard-core Processor	1	1	1
Soft-core Processor	3.5	1.4	1.4

Table 3.6: PPC memory size in KB

	Virtex-II Pro	**Virtex-5**	**Zynq SoC**
Hard-core Processor	7	10	11
Soft-core Processor	8	11	12

the normalized values of the reconfiguration time with respect to their hard-core processors.

3.5.3 PPC memory size

Table 3.6 shows the PPC memory size values. These values are the size of PPC functions only. They are compiled with "-O2" optimization and without debug option.

Figure 3.10 shows the bar graph of the memory size of the PPC functions. The normalized values with respect to the corresponding FPGAs shown in parenthesis are tabulated in the Table 3.7. We believe that the increase in PPC memory size of the Virtex-5 and the Zynq SoC compared to the Virtex-II Pro is caused by the increase in number of LUT entries from 16 to 64 for individual LUTs. It is also observed that the code density of the PowerPC is almost the same as that of the ARM Cortex-A9.

Table 3.7: Normalized PPC memory size

	Virtex-II Pro	**Virtex-5**	**Zynq SoC**
Hard-core Processor(Virtex-II Pro)	1	1.4	1.6
Soft-core Processor (Virtex-II Pro)	1	1.4	1.5
Hard-core Processor (Virtex-5)	0.7	1	1.1
Soft-core Processor (Virtex-5)	0.7	1	1.1

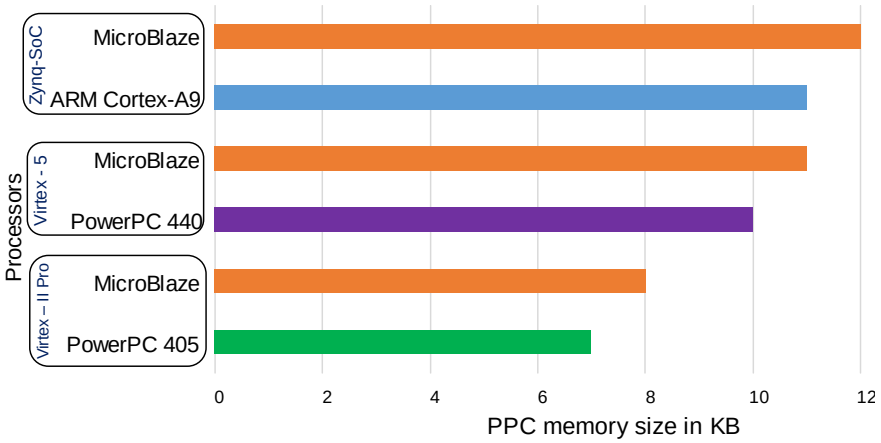


Figure 3.10: PPC memory size comparison

Table 3.8: Normalized PPC memory size

	Virtex-II Pro	**Virtex-5**	**Zynq SoC**
Hard-core Processor	1	1	1
Soft-core Processor	1.1	1.1	1.1

The normalized PPC memory size with respect to the hard-core processor is tabulated in Table 3.8. It reveals that the code density of the hard-core processors is a little bit higher than that of the soft-core processor which is in agreement with [46]. It can be seen that the increase in LUT entries affects the PPC memory size, reconfiguration time and the evaluation time. This effect can be negated by using an efficient bus structure, high speed HWICAP and highly sophisticated processor architecture.

From the results, it is clear that newer FPGA architectures tend to include more LUT resources and features. The added features increase the size of the LUT entries and the frame size. This creates additional overhead in reconfiguration time, evaluation time and the PPC memory size. The MicroBlaze soft-core processor proves to be inefficient in newer FPGA architectures for DCS because the processor clock speed, memory and ISA are limited and do not compete with those of hard-core processors. However for the edge cases, MicroBlaze in old FPGA architecture such as Virtex-II pro would be more suitable to implement the DCS.

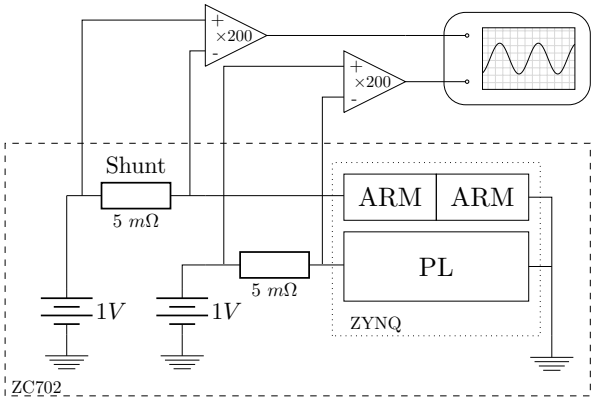


Figure 3.11: Current measurement schematics of Zynq-Soc on ZC702 board

3.6 Power measurement analysis of DCS

In this section, I present the power measurement analysis of the DCS technique used to implement a parameterized adaptive FIR filter described in Section 3.3. Due to the lack of power estimation tools for the Partial Reconfiguration technique, the authors of [47] proposed power consumption models for Dynamic Partial Reconfiguration (DPR). Similarly, I analyze the energy needed for Dynamic Circuit Specialization and compare this to the energy required to run the parameterized design. I also consider the static FPGA implementation of the same parameterized application and compare the power consumption by performing a power analysis. Defining the energy models for DCS and comparing the power behavior between the static and the DCS FPGA implementations, are the main contributions in this section.

3.6.1 Power measurement setup

The Xilinx ZC702 board is used for the power measurements and the DCS approach is implemented on the XC7Z020 Zynq-SoC. Ten power rails are present on this platform. Each rail is equipped with a shunt resistor on which current consumption can be monitored. Two channels are more interesting for the experiment. They separately supply the ARM cores and the Programmable Logic core. An external board is designed for power measurement purposes and two high-precision amplifiers are used to enhance the signal levels. The amplified signals are then sent to a digital oscilloscope for visualization and power trace analysis as shown in Figure 3.11. With this procedure, it is possible to measure power consumption variations as low as $0.1mW$. This accuracy is good enough for our energy analysis.

3.6.2 Zynq-SoC configuration setup

To obtain the energy models for DCS we used a clock frequency of 100MHz to drive the Programmable Logic (PL) and the same clock frequency of 100MHz for the HWICAP. The HWICAP is configured to be of the FIFO type with read and write buffer depth of 128 bytes. We used these parameters as a default project configuration in our following experiments.

The Specialized Configuration Generator (SCG) is implemented on the ARM Cortex-A9 dual core processor that operates at a clock frequency of 667 MHz. Therefore the evaluation of Boolean functions is expected to be faster than any of the tasks in the DCS.

3.6.3 Power Characterization for DCS

Using the power measurement setup we were able to measure the average power values on the Zynq ZC702 platform with the default project configuration explained in Section 3.6.2. There are three different power consumption parts that we need to consider:

1. The FPGA Idle Power is the power consumed by the silicon area of the FPGA even if it is unused and this state of the FPGA is called the idle state.
2. The FPGA Run Power is the power consumed by the silicon area of the FPGA when the FIR filter was triggered to execute the filter function and this state of the FPGA is called the run state.
3. The FPGA Reconfiguration Power is the power consumed by the silicon area of the FPGA during DCS reconfiguration and this state of the FPGA is called the reconfiguration state.

It is to be noted that both the CPU and the PL part of the Zynq-SoC consume power in all of the above three states.

I propose an energy analysis that is based on the energy required for reconfiguring one TLUT. For this, we need to consider the time τ_{tlut} needed to reconfigure one TLUT. According to our experiments, τ_{tlut} = $230\mu s$.

3.6.3.1 Energy consumed by the reconfiguration state on top of the idle state energy:

If E^{tlut}_{reconf} denotes the energy consumed during the reconfiguration of a TLUT, on top of the idle state energy then,

$$E^{tlut}_{reconf} = (P^{CPU}_{reconf} - P^{CPU}_{idle} + P^{FPGA}_{reconf} - P^{FPGA}_{idle}) \times \tau_{tlut} \tag{3.4}$$

where, P^{CPU}_{reconf} is the average power consumed by the CPU during DCS reconfiguration to perform the read, modify and write-back cycles of the frames. P^{CPU}_{idle}

Table 3.9: Average power consumed by the CPU and the PL fabric

	Notation	Average Power (mW)
CPU idle	P_{idle}^{CPU}	291
CPU run	P_{run}^{CPU}	384.1
CPU reconfiguration	P_{reconf}^{CPU}	390.3
FPGA idle	P_{idle}^{FPGA}	73.3 (45.6)
FPGA run	P_{run}^{FPGA}	74.7 (46.8)
FPGA reconfiguration	P_{reconf}^{FPGA}	68.7

Note: Power values after gating the HWICAP clock are mentioned between brackets.

is the power consumed by the CPU during its idle state. P_{reconf}^{FPGA} is the average FPGA reconfiguration power and P_{idle}^{FPGA} is the FPGA idle power. The idle power is defined as the power used when no reconfiguration, nor application execution is performed.

3.6.3.2 Relative power consumed by the reconfiguration state compared to the run state:

I also propose a relative power ratio between the reconfiguration state and the run state. If R_p denotes the relative power ratio then,

$$R_p = \frac{(P_{reconf}^{CPU} - P_{idle}^{CPU}) + (P_{reconf}^{FPGA} - P_{idle}^{FPGA})}{(P_{run}^{CPU} - P_{idle}^{CPU}) + (P_{run}^{FPGA} - P_{idle}^{FPGA})}$$
$$R_p = \frac{P_{reconf}}{P_{run}} \tag{3.5}$$

where P_{run} is the power consumed by the run state on top of the idle state power and depends on the size of the parameterized application. Indeed for a large parameterized design the value of P_{run} is much larger than P_{reconf} where, P_{reconf} is the power consumed by the reconfiguration state on top of the idle state.

3.6.4 Power measurements

From our measurements, I was able to extract the average power consumption of the Programmable Logic (PL) and the ARM Processor (CPU) of the Zynq-SoC. The average power values are tabulated in Table 3.9.

From equation 3.4 the estimated average energy E_{idle}^{tlut} is 21.8 mJ. And the relative power ratio R_p (equation 3.5) is 4.1. Since the relative power ratio is

greater than 1, the power consumption during the reconfiguration is higher than the power consumption during the execution of the FIR filter function. This is not a desired situation and I will further investigate this ratio later.

3.6.5 FPGA PL power drop during reconfiguration

Interestingly, in Table 3.9, the FPGA reconfiguration power is smaller than the FPGA idle power. In order to understand this phenomenon, the power curve is extracted and is shown in Figure 3.12. The reconfiguration happens between time units 0 and 90. Before and after that time, the system is running the FIR filter application. Clearly, the CPU power increases during the reconfiguration phase because the CPU has to perform the Boolean evaluation and the reconfiguration by swapping the specialized frames into the FPGA configuration memory via the HWICAP.

However, for the FPGA PL power we notice a significant power drop during the DCS reconfiguration phase compared to the FIR run state. An average power drop of 6.2 mW was observed. Further investigation revealed that there is a power drop only during frame read activity of the DCS reconfiguration as shown in Figure 3.13.

During the frame read activity, the configuration data (bitstream) is fetched from the FPGA configuration memory. The fetched bitstreams are first stored in the HWICAP read FIFO buffer. The maximum data that the read FIFO buffer can hold is a user configurable parameter and in our experiment it is set to 128 bytes. Once the read FIFO buffer is full, the ICAP has to wait until all the data in the FIFO buffer is received by the CPU via the AXI bus. This waiting state is established by turning off the ICAP's clock. Once the ICAP clock is turned off there will be no transaction of data between the ICAP port and the FPGA's configuration memory. Turning off the ICAP's clock causes the significant drop in the FPGA PL power and therefore it proves to be the main reason for the power fluctuations as depicted in Figure 3.13. The power drop is hence due to a mismatch between the computation bandwidth and the communication bandwidth (communication bandwidth limited design).

As a communication limited design (with wait cycles for data movement) increases the total time needed for the reconfiguration, the power drop does not necessarily result in a lower energy usage as well. Indeed, the increased time in fact results in a higher energy usage.

The best solution to avoid the HWICAP to stall is to increase the FIFO depth and clock the AXI bus much faster than the HWICAP. To simulate this situation, I performed the experiments with the HWICAP clock of 20 MHz. The average power gradient for the different configurations of the HWICAP clock, FIFO depth and the PL fabric is tabulated in Table 3.10. We observe that the AXI bus (with

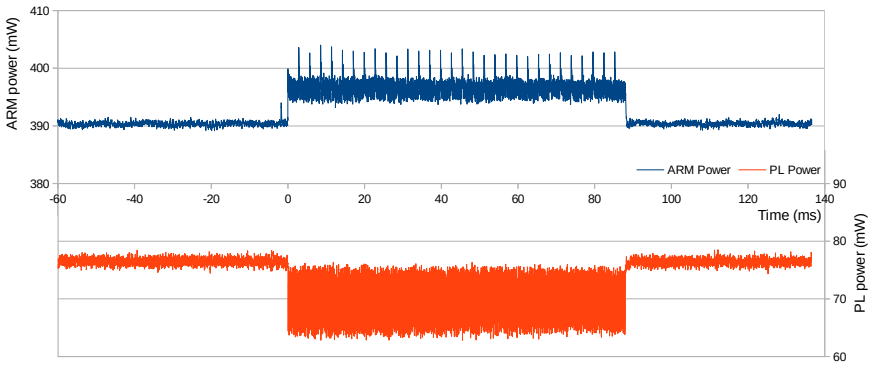


Figure 3.12: Average power consumption of CPU and FPGA during run and reconfiguration state

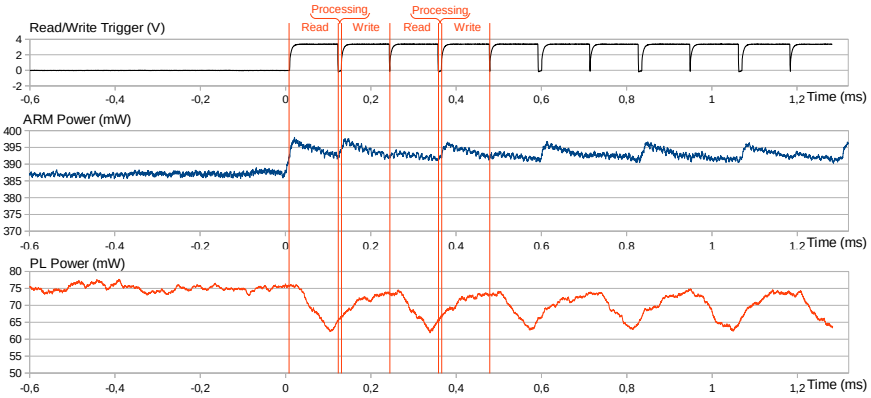


Figure 3.13: Average power consumption of CPU and FPGA during Frame read and Frame write activities

Table 3.10: FPGA PL Power gradient

PL fabric AXI bus clock (MHz)	**HWICAP clock (MHz)**	**FIFO depth**	**Average Power gradient (mW)**
100	100	128	- 6.2
100	20	128	+ 0.07
100	100	256	- 3
100	20	256	+ 1

Note: a “+” indicates an increase in power consumption and a “-” indicates a decrease in power consumption.

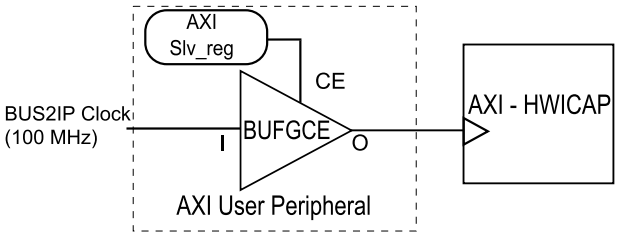


Figure 3.14: Clock gating for the AXI-HWICAP

100 MHz) is fast enough to receive the data from the HWICAP so the read FIFO is less likely full, therefore the HWICAP fetches the data as fast as possible. Also, the average power gradient values are halved for the experiments with the FIFO depth 256. This confirms the reason for the PL power drop during the frame read activity.

During the frame write activity, the HWICAP does not stop the ICAP's clock because the HWICAP constantly expects the ICAP's attention and makes it receive the data from the write FIFO buffer irrespective of whether the write FIFO buffer is full or not.

The Xilinx AXI-HWICAP IP consumes a huge idle power of 31.2 mW because the IP lacks clock gating and is active even if the reconfiguration process is unused. Therefore, to make DCS functional an extra power of 31.2 mW is required irrespective of whether or not the HWICAP is used for reconfiguration during the operation of the FIR filter. In order to make DCS more power efficient, we include the clock gating technique to the AXI-HWICAP IP and reduce the HWICAP idle power.

3.6.6 Xilinx HWICAP with Clock gating

The clock of the AXI-HWICAP IP is gated with the help of a user AXI-lite peripheral. The required clock gating for the AXI-HWICAP is depicted in Figure 3.14. The CE line is controlled by a user accessible AXI slave register. The slave register is software controlled and hence we can turn ON/OFF the HWICAP clock during the power measurements. After gating the AXI-HWICAP clock, we were able to reduce the idle power of the AXI-HWICAP with 27.9 mW (31.2 - 3.3 = 27.9) ($\approx$ 90%). The HWICAP still consumes a power of 3.3 mW during its idle state as tabulated in Table 3.12. The corresponding FPGA idle power was reduced to 45.6 mW.

The relative power ratio of equation 3.5 changes after introducing the clock gating for the HWICAP, and can be expressed as a function of the number of FIR

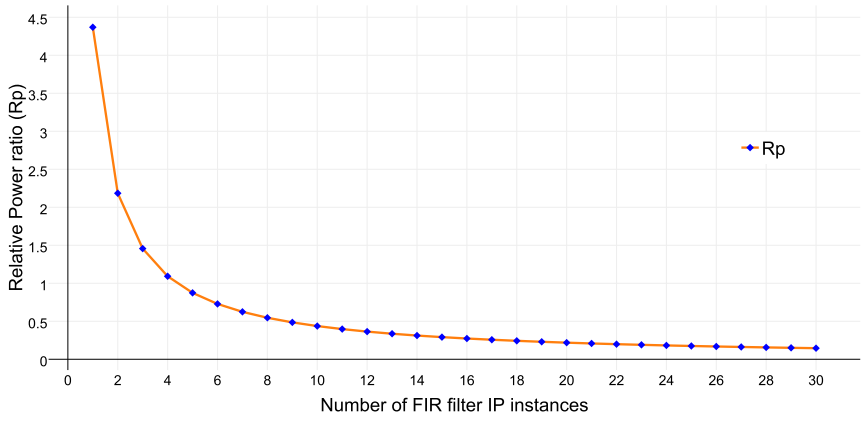


Figure 3.15: Relative Power ratio as a function of the number of FIR filter IP instances

filter instances N_{FIR}.

$$R_p = \frac{P_{reconf} + P_{idle}^{HWICAP}}{P_{run} \times N_{FIR}} \tag{3.6}$$

As shown in Figure 3.15, an increase in the number of FIR IP instances decreases the magnitude of the ratio R_p.

3.6.7 DCS Power Analysis

3.6.7.1 Power consumption of a DCS versus static implementation

In this section, I investigate how DCS affects the global power consumption of the system. The main objective of this experiment is to compare the global power consumption of the FIR using two different implementations:

1. FIR with static implementation: the FIR was implemented without using the reconfiguration technology. Instead, the coefficient inputs of the FIR are connected directly to the slave registers of the AXI bus and with the help of the CPU, the user can change the coefficient values at the software level. Therefore, I do not make use of the HWICAP and the DCS reconfiguration technology.

2. FIR with DCS implementation: the FIR (of one IP instance) was implemented using the reconfiguration technology. We use the DCS reconfiguration technology to change the FIR coefficients by reconfiguring the multiplications of the filter taps. Therefore, the user can change the coefficient values of the FIR filter using the CPU at the hardware level.

Table 3.11: FIR power consumption comparison Static vs DCS

Row no.	Project	$\mathbf{P}_{run}^{CPU}$ (mW)	$\mathbf{P}_{reconf}^{CPU}$ (mW)	$\mathbf{P}_{run}^{FPGA}$ (mW)	$\mathbf{P}_{reconf}^{FPGA}$ (mW)
1	No FIR, No HWICAP	382.8	Nil	39.5	Nil
2	Static FIR, No HWICAP	383.1	Nil	45.4	Nil
3	DCS FIR, No HWICAP	383.4	Nil	43.3	Nil
4	DCS FIR, HWICAP	384.1	390.3	74.5	68.5
5	DCS FIR, HWICAP with clock gating (Clock OFF)	384.1	390.3	46.6	68.5

To get a clear picture of the comparison, I measured the power consumption of the CPU and the PL for projects with different configurations given in Table 3.11. The differential power and the energy consumption of the idle and run power combined together are tabulated in Table 3.12. These values are the difference in power values between different rows of Table 3.11. For example, the static FIR (idle + run) power is obtained by the difference in corresponding power values of row no.2 and row no.1 of Table 3.11.

The power consumed by the HWICAP during the reconfiguration process is obtained by the difference between P_{reconf}^{FPGA} of row no.4 and P_{run}^{FPGA} of row no.3 of Table 3.11 i.e. (68.5 - 43.3 = 25.2 mW). The CPU power consumption during interaction with HWICAP is obtained by the difference between P_{reconf}^{CPU} of row no.4 and P_{run}^{CPU} of row no.1 i.e. (390.3 - 382.8 = 7.5 mW). Therefore total HWICAP reconfiguration power consumption is 32.7 mW (25.2 + 7.5 = 32.7). The DCS FIR power together with HWICAP reconfiguration power is obtained by the difference P_{reconf}^{FPGA} of row no.5 and P_{run}^{FPGA} of row no.1 of Table 3.11 i.e. (68.5 - 39.5 = 29 mW). The CPU power is obtained by the difference between P_{reconf}^{CPU} of row no.5 and P_{run}^{CPU} of row no.1 i.e (390.3 - 382.8 = 7.5 mW).

To investigate the power consumption by the DCS FIR (LUTs) compared to the static FIR (LUTs), I have removed the HWICAP from the DCS implementation and measured the power consumption. The FIR with DCS implementation consumes less FPGA idle and run power without HWICAP compared to the static FIR implementation. We observe a difference of 2.3 mW (6.1 - 3.8 = 2.3) ($\approx 36\%$).

Table 3.12: Differential Power Results

Extracted Power Results	**CPU (mW)**	**PL fabric (mW)**	**Total Energy (μJ)**
Static FIR, idle + run power (without HWICAP) (row no. 2 - row no. 1)	0.3	6.1	0.06 (1.86)
DCS FIR without HWICAP, idle + run power (row no. 3 - row no. 1)	0.6	3.8	0.044
HWICAP idle power, with Clock OFF (row no. 5 - row no. 3)	0.7	3.3	NA
HWICAP idle power, with Clock ON (row no. 4 - row no. 3)	0.7	31.2	NA
HWICAP reconf power	7.5	25.2	2838.3
DCS FIR (run) with HWICAP idle (row no. 5 - row no. 1)	1.3	7.1	0.08 (1.72)
DCS FIR with HWICAP, reconf power	7.5	29	3175.7

Note 1: The energy values for 3 FIR filter instances are mentioned between brackets.
Note 2: The row numbers mentioned in the table are the row numbers of Table 3.11 .

This difference is because of the reduction in FPGA resources (LUTs) utilized by the FIR filter. However, there is a huge FPGA reconfiguration (PL + CPU) power difference of 30.1 mW (29-6.1 + 7.5-0.3) between the static FIR (without any reconfiguration) and the FIR with DCS which proves to be an unavoidable overhead. During the reconfiguration process, the CPU consumes a maximum of 7.5 mW and this power is considered to be an overhead.

The corresponding average energy consumption (CPU + PL) is also listed. For one FIR IP instance the DCS implementation consumes more energy (0.02μJ) than the static FIR filter implementation. The HWICAP consumes extra energy in the DCS implementation (plus 0.036μJ) and this energy is constant irrespective of the number of FIR IP instances. Therefore, we need bigger designs (more FIR filter implementations) before the energy calculations start to be in favor of reconfigu-

ration. We investigated that DCS becomes energy efficient for 3 or more FIR IP instances and the corresponding energy values are shown within the brackets in Table 3.12. We observe an energy gain of 0.14μJ. More details are discussed in Section 3.6.7.2.

3.6.7.2 Power efficient DCS implementation and its reconfiguration rate

The results from the previous section show that the reconfiguration process using the HWICAP is power-hungry. However, the reconfiguration process is triggered only if the parameters (coefficients of the FIR filter) change. It is interesting to investigate the reconfiguration rate (expressed as the reconfiguration time over the total execution time), allowed under the constraint that the DCS energy is less than or equal to the static energy as a function of number of the FIR filter IPs. On the one hand only 950 LUTs are used to implement the FIR filter with DCS, and on the other hand 2525 LUTs are used for the static FIR filter implementation.

There are two important parameters that need to be considered to evaluate the global average energy (E_{static} and E_{DCS}): the number of FIR filter IPs (N_{FIR}) and the relative amount of time spent for reconfiguration (the reconfiguration rate R_{rate}) which is $R_{rate} = \frac{T_{reconf}}{T_{reconf}+T_{run}}$, where T_{reconf} is the time taken to reconfigure all the TLUTs of the FIR filter and T_{run} is the time taken to execute the FIR filter function. Accordingly, we deduce equation 3.7 and equation 3.8 for the energy needed for the execution of the implementation for a single round of constant coefficient values.

$$E_{static} = N_{FIR} \times P_{FIR}^{static} \times T_{run}^{static} + P_{coef} \times T_{coef} \tag{3.7}$$

$$\begin{aligned} E_{DCS} = N_{FIR} \times P_{FIR}^{DCS} \times T_{run}^{DCS} + P_{reconf} \times T_{reconf} \\ + P_{idle}^{HWICAP} \times (T_{reconf} + T_{run}) \end{aligned} \tag{3.8}$$

where, P_{reconf} and P_{coef} are the power consumption during the change of coefficient values for the DCS and static implementation of the FIR respectively. T_{run}^{static} and T_{run}^{DCS} are the time taken to execute the filter functions for the FIR with static and DCS implementations respectively.

Assuming P_{coef} is negligible, we can solve for the variable R_{rate} for a worst case scenario[5] where $T_{run}^{static} = T_{run}^{DCS}$.

$$R_{rate} = \frac{(P_{FIR}^{static} - P_{FIR}^{DCS}) - (\frac{P_{idle}^{HWICAP}}{N_{FIR}})}{(P_{FIR}^{static} - P_{FIR}^{DCS}) + (\frac{P_{reconf}}{N_{FIR}})} \tag{3.9}$$

This ratio provides the reconfiguration rate as a function of the number of FIR IP instances (reconfigured) for the condition that the average energy of DCS and static

[5] the DCS implementation usually runs about 20% faster than the static implementation.

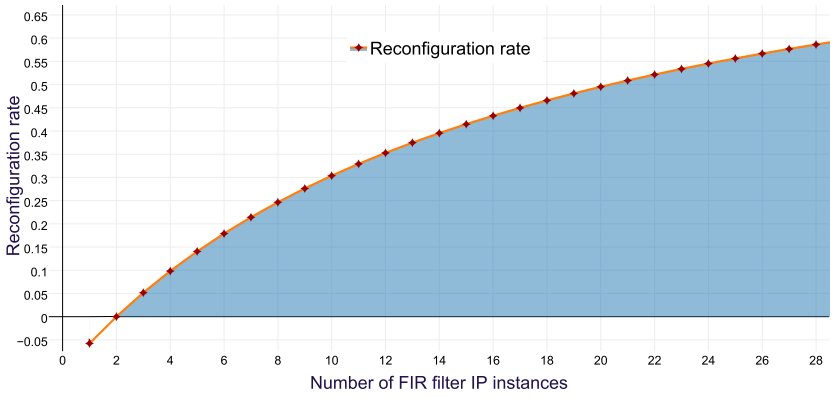


Figure 3.16: Reconfiguration rate as function of number of FIR filter instances

implementations are equal. Accordingly, we can plot a graph shown in Figure 3.16. Clearly, for less than 3 FIR IP instances DCS is inefficient in energy since the reconfiguration rate is negative. The DCS reconfiguration is energy efficient for 3 or more FIR IP instances if it has a reconfiguration rate within the shaded region. For example, suppose if the reconfiguration rate is 0.3, then we need to run at least 10 FIR filter IPs before the DCS reconfiguration becomes energy efficient. Vice versa, if we have 10 FIR filters, the reconfiguration time should not take more than 30% of the total time in order to remain energy efficient.

In a broader spectrum, the reconfiguration time is the backbone of power dissipation during the micro-reconfiguration. The larger the reconfiguration time the more power is dissipated and hence the more energy consumption.

In the next chapter, I propose custom reconfiguration controllers: MiCAP and MiCAP-Pro. These controllers are light weight, faster and much more energy efficient than the HWICAP.

4

MiCAP and MiCAP - Pro

In this chapter, I introduce custom reconfiguration controllers designed specifically to implement a DCS system on a self-reconfigurable platform for the Zynq-SoC. The results of the design of custom controllers are presented and compared with the standard reconfiguration controller HWICAP. The power measurement results of the controllers and the trade-off between their reconfiguration speed and area (resource utilization) are also explained. With the existing Xilinx FPGA column-based architectures, I propose to reconfigure multiple LUTs at the same time. To do this I propose to use design placement constraints to cluster the bits that have to be changed in the same reconfiguration columns and customizing the standard "XhwIcap_setClb_bits" function. This gives us a significant improvement in reconfiguration speed.

4.1 Why custom reconfiguration controllers?

In contrast to other FPGAs, Xilinx FPGAs have been partially reconfigurable since quite some years. The FPGA architectures contain a set of components to execute the reconfiguration, such as the Internal Configuration Access Port (ICAP), a data access bus (Processor Local Bus or Advanced eXtensible Interface bus) and an embedded processor (PowerPC or ARM Cortex-A9). The ICAP is a built in hardware macro, which has direct access to the configuration memory and it requires a reconfiguration controller that is built as part of the design to manage bitstream movement between the ICAP macro and the processor. The Hardware Internal

Configuration Access Port (HWICAP)[1] is a reconfiguration controller that contains a complex state machine and a First In First Out (FIFO) buffer designed to access the bitstreams from the configuration memory of the FPGA. The efficiency of these components affects the reconfiguration speed of DCS. The "LUT reconfiguration speed" is defined as the number of Look Up Tables (LUTs) reconfigured per unit of time. Conversely, the "LUT reconfiguration time" is the time taken by the system to reconfigure a single LUT in a design. Investigations have shown that reconfiguration time is a major limiting factor for the DCS implementation on a Xilinx FPGA [48]. The main reason for the slow reconfiguration speed is the complexity of the HWICAP architecture and the lower communication bandwidth between the processor and the ICAP controller, resulting in a data throughput of 19 MBps [12]. We proposed a novel idea for improving the reconfiguration speed of DCS using placement constraints in [49]. However, this improvement comes at the cost of the application's performance (the maximum clock the application design can support); an average of 6% of the application's performance needs to be compromised. This approach may not be suitable if the application's performance is an important metric. Therefore, in order to improve the reconfiguration speed of DCS without affecting the application's performance I propose custom reconfiguration controllers: MiCAP [50] and MiCAP-Pro [51], built on a Xilinx 7 series FPGA (Zynq-SoC).

4.2 Internal Configuration Access Port

The Static Random Access Memory (SRAM) cells of LUTs, Switch blocks, Connection blocks, Block Random Access Memory (BRAM) blocks and Digital Signal Processing (DSP) blocks together form the configuration memory of an FPGA fabric. The Xilinx ICAP primitive provides internal access to the configuration memory of the FPGA. This interface can be used to download configuration data into the configuration memory during run-time. It is also possible to read the configuration data from the configuration memory. The ICAP can also be used for reading the status register of the configuration memory. The structure of the ICAP interface with the FPGA configuration memory is depicted in Figure 4.1.

4.2.1 ICAP architecture

The ICAP primitive contains two separate data ports for reading (O) and writing (I) the data. Each bus supports a data width of 32-bits. It has a clock input (CLK) and an active-low ICAP enable (CSIB) input. The ICAP primitive can support a maximum clock frequency of 100 MHz for a reliable implementation. The CSIB is an active-low, chip enable signal used to turn the ICAP ON/OFF. There is a read/write

[1] HWICAP is an Intellectual Property block provided by Xilinx

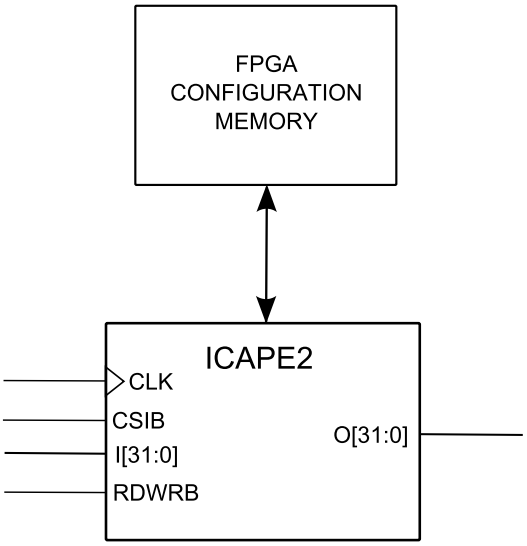


Figure 4.1: ICAP primitive in Zynq-SoC

select input signal (RDWRB) used to select the direction of the data. By setting the "RDWRB" signal to high, the data can be read from the configuration memory and to write the data back to the configuration memory one has to set the signal to low. The data is written at the rising edge of the clock. Therefore, the writing of configuration data can be controlled by either the clock or the CSIB signal. There is no "Busy" signal, in contrast to the ICAP primitive present in the Virtex-5 and Virtex-6. The validity of the read data is checked deterministically [23].

To access the configuration bitstreams of an FPGA, a series of commands has to be written to the ICAP's input for every rising edge of the clock cycle. These commands help the user to orchestrate the ICAP to read the configuration data or write the configuration data to the configuration memory. Therefore, to start the access (either read or write) of the configuration frames, it is mandatory to send the ICAP commands first. Therefore, the access begins with the ICAP write activity. The frame address is placed at a certain location in between these commands to let the ICAP know which frames have to be accessed.

With a clock input of 100 MHz and a data width of 32 bits, the maximum throughput of the ICAP is 400 MBps [52]. However, the HWICAP that encapsulates the ICAP port supports only 19 MBps due to its inefficient architecture that contains a complex state machine and a communication overhead between the ICAP and the processor which is unnecessary for DCS. Therefore, we need a lite-weight controller to improve the reconfiguration and overall performance of DCS.

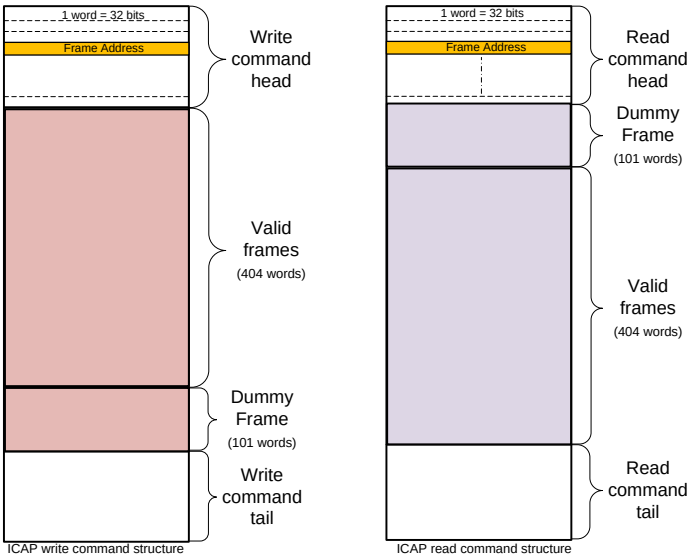


Figure 4.2: ICAP commands

4.2.2 ICAP Commands

To access the configuration memory of an FPGA, a series of commands have to be written to the ICAP's input for every rising edge of the clock cycle. The ICAP read command consists of a read command header that contains the frame address to point to the corresponding frame and a read command tail to desynchronize and safely close the ICAP after reading the bitstreams. Therefore, to read the bitstreams, an ICAP read command header has to be sent to the ICAP and then capture the bitstream data followed by sending the ICAP read command tail to desynchronize the ICAP.

Similarly, the ICAP write command consists of a write command header that contains the frame address and a write command tail to desynchronize the ICAP. To write the configuration data into the configuration memory, an ICAP write command header has to be sent to the ICAP. Next, the ICAP is ready to accept the frames that are to be written into the FPGA configuration memory. Finally, we close the ICAP by sending the write command tail. Figure 4.2 shows a brief overview of the ICAP commands.

4.3 MiCAP

The basic architecture of MiCAP is shown in Figure 4.3. MiCAP consists of 4 major parts: two asynchronous FIFO buffers, an ICAP state machine and the ICAP

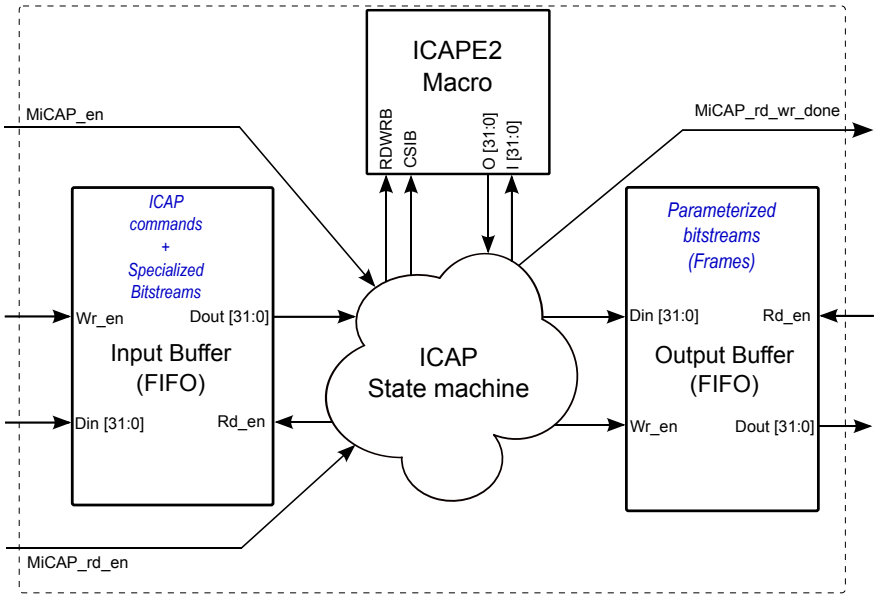


Figure 4.3: MiCAP architecture

primitive. All the elements of MiCAP are synchronized by a common clock with a frequency of 100 MHz.

1. Input Buffer: this is an asynchronous FIFO buffer that holds the ICAP read and write commands along with specialized configuration data which are to be written into the configuration memory. The application software is responsible to store all the configuration data into the input buffer before triggering the write activity of MiCAP. Therefore, the input buffer acts as the configuration data source for MiCAP's write activity.

2. Output Buffer: this is also an asynchronous FIFO buffer that holds the configuration data fetched by the ICAP primitive during the read activity. All the data read from the configuration memory via the ICAP is stored in the output buffer. Therefore, the output buffer acts as a sink to MiCAP's read activity. Once the data is ready, the processor has to read the frames from the output buffer.

3. ICAP primitive: this is a design element that gives access to the configuration data of the FPGA. Using this element the commands and data can be read or written into the FPGA configuration memory. The ICAP primitive architecture of the Zynq-SoC is explained in Section 4.2.

4. ICAP State machine: this is the brain of MiCAP. It contains multiple states that orchestrate MiCAP's read and write activity. The state machine contains 3 major states: a *wait state*, a *read state* and a *write state*. The description of each of the states is as follows.

4.3.1 State machine

a) *Wait state*: in this state, the MiCAP's state machine waits until all the data (frames + ICAP commands) are filled into the input FIFO.

b) *Read state*: MiCAP's read activity is triggered by the processor by setting "MiCAP_en" and "MiCAP_read_en" signals to high. In this state, first the RDWRB signal is set to high and then the ICAP primitive is enabled by setting the CSIB signal to a low value. The read command present in the input buffer is fetched and written to the ICAP's input port. The command is written for every rising edge of the clock. Once the read command is sent, the ICAP starts fetching the configuration data. The frames fetched from the ICAP are written into the output buffer. Once the read activity is completed, the ICAP is disabled by setting the CSIB signal to high. The "MiCAP_rd_wr_done" signal is set to high once MiCAP's read activity is accomplished.

c) *Write state*: MiCAP's write activity is triggered by the processor by setting the "MiCAP_en" signal to high and the "MiCAP_read_en" signal to low. In this state, first the RDWRB signal is set to low and then the ICAP primitive is enabled by setting the CSIB signal to low. The write command is fetched from the input buffer and the command is written to the ICAP's input port. Once the write command is sent, the ICAP believes that next incoming data is the configuration data that has to be written into the configuration memory of the FPGA. Now the state machine reads the data from the input buffer and writes the data into the ICAP input port. The ICAP continues to write the data sent from the input buffer into the configuration memory until the input buffer is empty. The "MiCAP_rd_wr_done" signal is set high once MiCAP's write activity is accomplished.

The MiCAP architecture implemented on the Zynq-SoC platform is depicted in Figure 4.4.

4.3.2 MiCAP with single port RAM

The improved version of MiCAP contains an extra single port RAM that holds the ICAP read and write commands well before the data transaction begins. These commands are non-volatile in contrast to the data in the input buffer, hence MiCAP can make use of these commands multiple times as needed. Therefore, the input

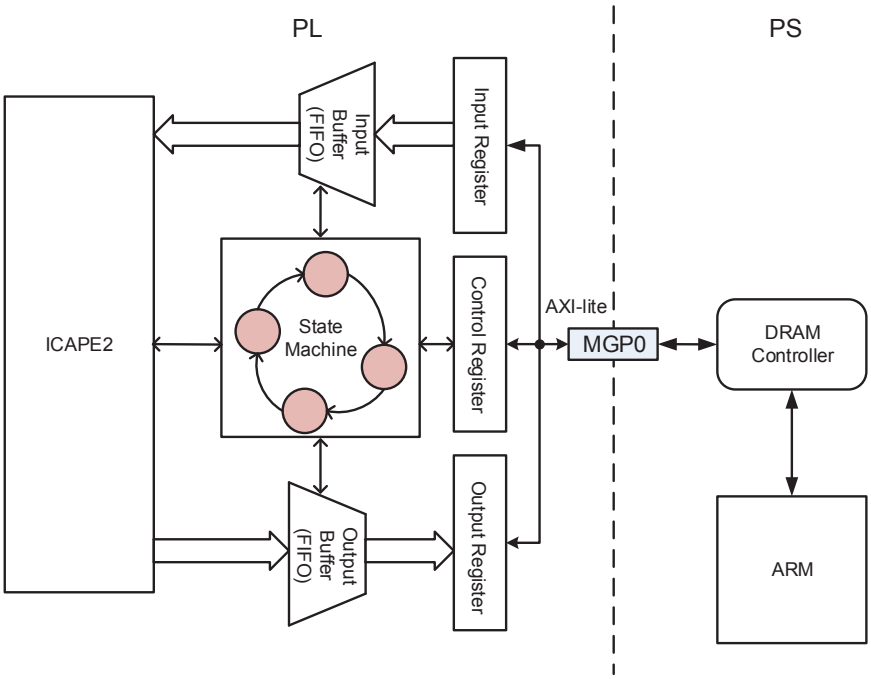


Figure 4.4: MiCAP implementation on the Zynq-SoC

buffer (FIFO) is used to hold only the specialized bitstreams (frames) that replace the stale frames present in the configuration memory. This saves a significant amount of time during the reconfiguration since only the configuration frames are transferred between Processing System (PS) and PL of the Zynq-SoC.

Figure 4.5 shows MiCAP with single port RAM. The state machine handles the multiplexing of data between the frames from the input buffer and the ICAP commands from the single port RAM to establish the proper reconfiguration process. However, this version of MiCAP utilizes more FPGA resources (LUTs and FFs) compared to the basic MiCAP.

The HWICAP and MiCAP use the GP port of the Zynq-SoC to transfer the data between PS and PL and therefore, the controllers suffer from a data-transfer bottleneck during reconfiguration resulting in a reduced throughput by a factor 20× compared to the throughput the ICAP can handle. To overcome the data-transfer bottleneck we propose a DMA-based reconfiguration controller called MiCAP-Pro.

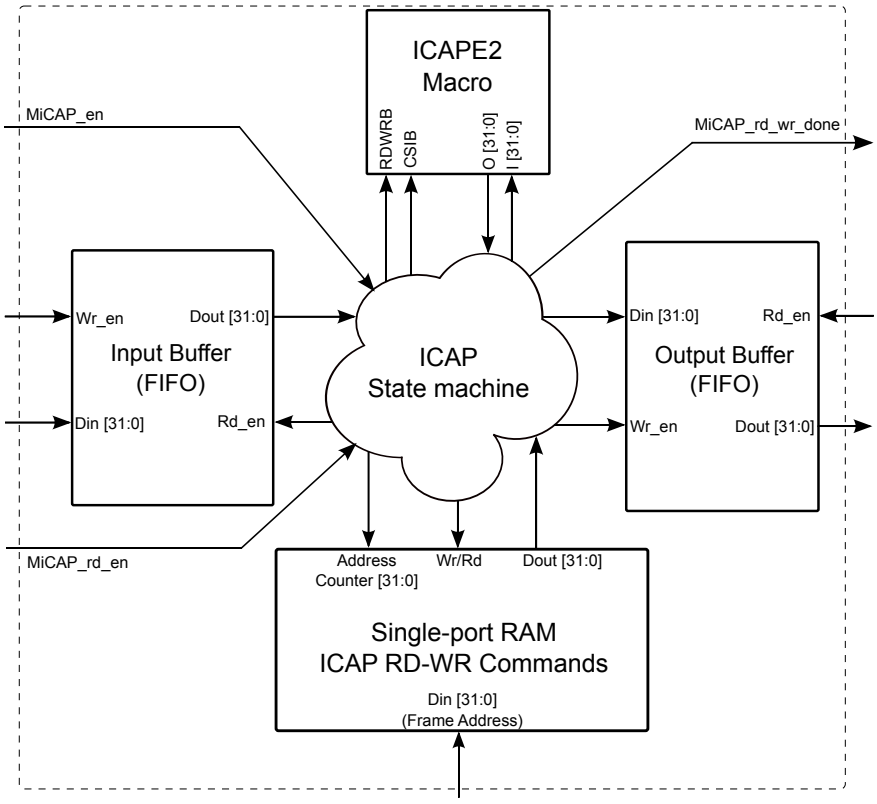


Figure 4.5: MiCAP with single port RAM

4.4 MiCAP-Pro

The implementation of MiCAP on the Zynq-SoC is shown in Figure 4.4. Clearly, MiCAP uses a master general purpose (MGP0) port to transfer the data between PS and PL regions. There are 3 memory mapped registers for the data transfer. The input register is used to send the ICAP commands and the specialized bitstreams to swap the configuration data into the configuration memory through ICAPE2.[2] The command register is used to issue read/write commands to MiCAP and the output register is used to receive the read frames from the configuration memory.

An AXI-lite bus establishes interconnect between MiCAP and the memory mapped registers. The data transfer between MiCAP and the DRAM memory of the PS region is performed via a low communication bandwidth MGP0 port. This was one of the reasons for the slow reconfiguration speed and hence the MGP0 acts as a bottleneck during the data transfer. To eliminate this bottleneck we make

[2]ICAPE2 is the advanced version of the ICAP primitive present in Xilinx 7-series FPGAs

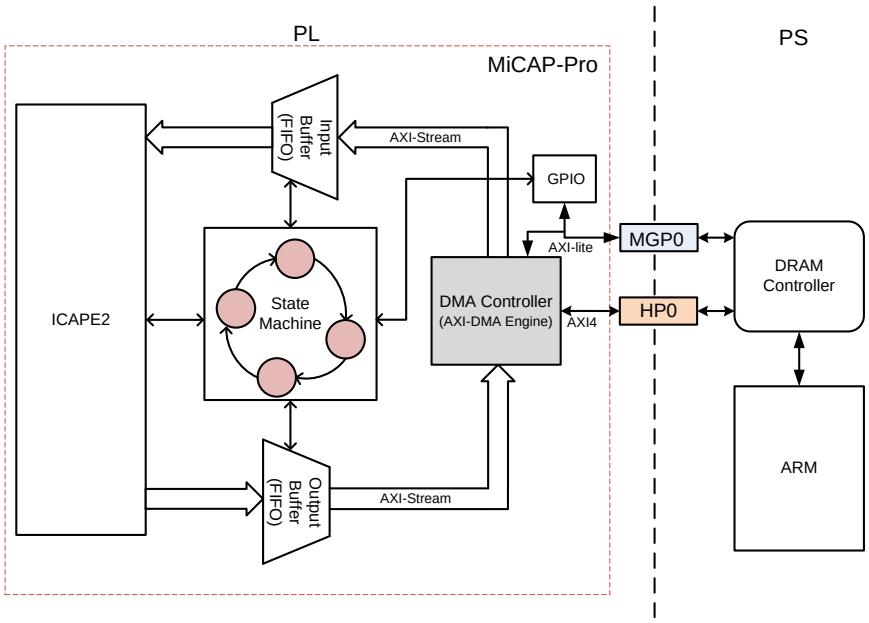


Figure 4.6: MiCAP-Pro implementation on the Zynq-SoC

use of HP ports. MiCAP-Pro is an extended version of MiCAP that makes use of the HP port to establish a high speed data transfer between PS and PL regions during the reconfiguration.

4.4.1 MiCAP-Pro architecture

MiCAP-Pro is an improved version of MiCAP in the sense that it inherits almost all parts of the architecture from MiCAP but overcomes the bottleneck during the configuration data transfer between the processor and the ICAPE2 by making use of the high performance port (HP0) of the Zynq-SoC. Since the HP ports can be accessed only by a master from the PL region, we use a DMA controller that acts as a master to the high performance ports. The data transfer occurs through the HP0 port of the Zynq-SoC, thus establishing a very high speed data transfer that contributes to the faster reconfiguration speed. The architecture of MiCAP-Pro is depicted in Figure 4.6.

4.4.2 AXI DMA Engine

The AXI Direct Memory Access (DMA) provides a high bandwidth data transfer between the DRAM controller present in the PS and the AXI-Stream type peripherals implemented on the Programmable Logic [53]. The initialization, management

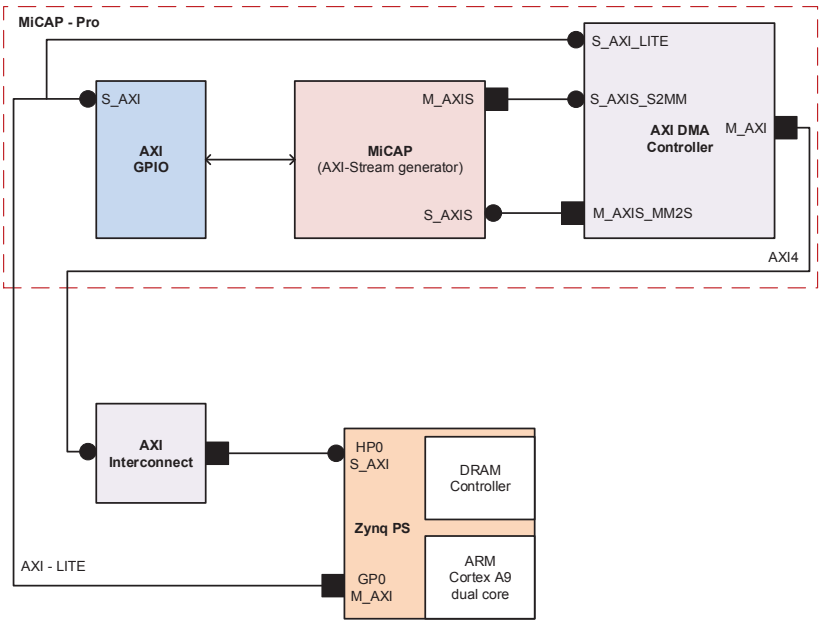


Figure 4.7: MiCAP-Pro interconnections

and status registers of the DMA engine can be accessed using an AXI-lite interface. The DMA comes with an optional scatter-gather capability that can offload the data transaction completely from the processor in a processor based system.

The high speed data movement between the PS and the peripheral target (on the PL) is achieved through a burst-capable AXI4 bus. The DMA supports high speed data transfer between Memory mapped to Stream (MM2S) type and the Stream to Memory mapped (S2MM) type target peripherals. The data transfer is achieved by using full duplex communication and therefore allowing MM2S and the S2MM transfers in parallel. Some of the typical applications such as high speed data transaction between the System and the Ethernet can be achieved by using AXI DMA for efficient data transfer.

4.4.3 MiCAP-Pro interconnections

The interconnections of MiCAP-Pro are shown in Figure 4.7. Clearly, MiCAP-Pro makes use of two interfaces, a general purpose port (GP0) and a high performance port (HP0) of the Zynq-SoC. The control registers of the AXI DMA engine are programmed by the processor through the AXI-lite interface via the GP0 port. The status registers of the DMA can be read with the help of the same interface.

The interface is also used for implementing the handshake signals between the state machine of MiCAP and the processor using I/O pins of the AXI GPIO.

The configuration data that needs to be read from or written into the configuration memory is accessed in the form of a data stream by the DMA controller. Therefore, in this case MiCAP acts as a stream generator. The generated stream is interfaced with the DMA engine using full duplex communication through the AXI-stream bus. Each of the AXI-stream buses has a master (the port of the master is represented as a black square) and a slave (the port of the slave is represented as a black circle). The Master initiates the data transfer while the slave waits for the master's command.

The AXI stream generator encapsulates MiCAP's state machine, the ICAPE2 and the input/output FIFO buffers. The state machine takes care of filling up the configuration bitstreams streaming from the DMA controller into the input buffer before it triggers the ICAPE2 to write the bitstreams (present in the input buffer) into the FPGA configuration memory. The data is transferred from the DMA to the input buffer via the MM2S port. The bitstreams read by the ICAPE2 are initially stored in the output buffer. Once the DMA controller is ready to accept data, the state machine triggers the streaming of data from the output buffer to the DMA controller via the S2MM port.

The bitstream transfers occur between the DMA engine and the HP port through the burst-capable AXI4 bus. The data is transferred via an AXI interconnect [54] that acts as a slave to the DMA engine and a Master to the HP port. Thus the AXI interconnect is used as a data synchronizer. The data is accessed by the DRAM controller via the HP port which is capable of providing high speed data transfer. The configuration data can be temporarily stored in the off chip DRAM memory and the ARM processor can access these data by requesting access to the DRAM controller.

4.5 Results on reconfiguration controllers

In this section, we present the results of the experiments with different reconfiguration controllers. I compare the results and study the overall effect of using custom reconfiguration controllers on a DCS system. We measured the reconfiguration speed of a single TLUT of a parameterized design using soft-timers (AXI Timer v1.03a). We also evaluated the total time to reconfigure all the TLUTs of the parameterized design (total reconfiguration time).

4.5.1 Reconfiguration time

The total reconfiguration time of different parameterized applications implemented with different reconfiguration controllers is tabulated in Table 4.1 and the graph

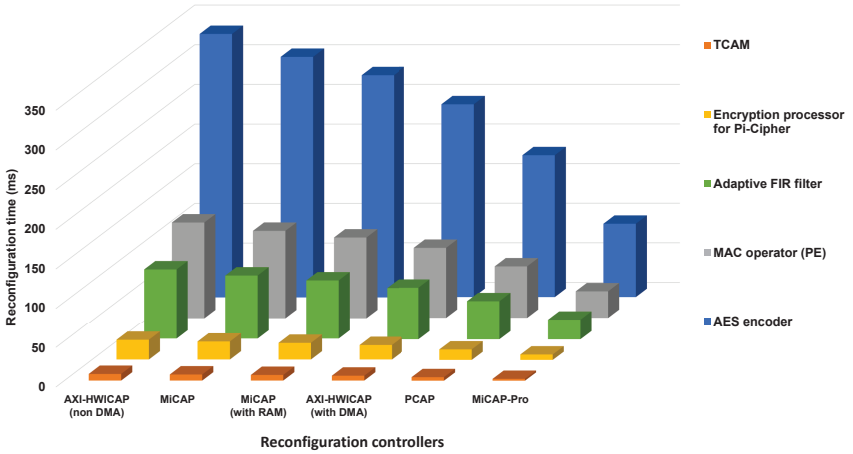


Figure 4.8: Reconfiguration time for parameterized applications with different reconfiguration controllers

that illustrates the comparison is depicted in Figure 4.8. Clearly, the DCS system using MiCAP-Pro has the lowest reconfiguration time. The worst reconfiguration time can be seen for the DCS with AXI-HWICAP. The basic MiCAP and MiCAP with single port RAM show an improvement in the reconfiguration time by $\approx 10\%$ and $\approx 17\%$ respectively compared to the AXI-HWICAP.

MiCAP-Pro's reconfiguration time is drastically improved by a factor of ≈ 3 compared to AXI-HWICAP. There are 3 main reasons for the improvement:

1. Buffers: MiCAP-Pro has I/O buffers whose depth is large enough to hold all the required data during the reconfiguration. Therefore, the data transaction is not stalled during the reconfiguration process in contrast to case of the AXI-HWICAP.

2. Optimized state machine: the state machine of MiCAP-Pro is simple and it does not need logic to handle the reconfiguration stall phenomenon since the I/O buffers never get full.

3. High Performance ports: the data transaction between PS and PL regions occurs through HP ports of the Zynq-SoC. These ports are fast enough to provide high speed communication bandwidth between PS and PL regions.

The measured reconfiguration time for one TLUT is tabulated in Table 4.2. These values are linearly proportionate with the total reconfiguration time listed in Table 4.1.

Table 4.1: Total reconfiguration time (ms) for different parameterized applications with different reconfiguration controllers

Application	**#TLUTs**	**AXI HWICAP (non DMA)**	**MiCAP**	**MiCAP with RAM**	**AXI HWICAP (with DMA)**	**PCAP**	**MiCAP-Pro**
TCAM	36	8.2	7.5	6.9	6.06	4.4	2.3
MAC operator (PE)	526	120.9	110.4	102.04	88.6	65.2	33.7
AES Encoder	1444	332.1	303.2	280.1	243.3	179.05	92.5
Encryption processor for π-Cipher	108	24.8	22.6	20.9	18.1	13.3	6.9
Adaptive FIR filter	384	88.3	80.6	74.4	64.7	47.6	24.1

Table 4.2: Reconfiguration time of a TLUT for different reconfiguration controllers

Reconfiguration Controller	**TLUT Reconfiguration time (μs)**
AXI-HWICAP (non-DMA)	230
MiCAP	210
MiCAP with single port RAM	194
AXI-HWICAP (with DMA)	168.5
PCAP	124
MiCAP-Pro	**64.1**

4.5.2 Reconfiguration controller data throughput

The measured data throughput of each reconfiguration controller is tabulated in Table 4.3. The other reconfiguration controllers used for Partial Reconfiguration such as ZyCAP [55], PCAP [26] and FaRM [56] controllers are also considered for the data throughput comparison. Figure 4.9 shows the comparison of the data throughput between each of the reconfiguration controllers. The controllers are distinguished depending upon the capability to implement a DCS system on the Zynq-SoC platform.

Table 4.3: Data throughput of reconfiguration controllers

Reconfiguration Controller	**Data throughput ($MBps$)**	**Allow DCS ?**
AXI-HWICAP (non-DMA)	19	Yes
AXI-HWICAP (with DMA)	67	Yes
PCAP	128	No
FaRM (only on PLB)	174	No
ZyCAP	382	No
Open-source controller D^{2}PR-EDAC	319.9	No
MiCAP	22	Yes
MiCAP with single port RAM	23	Yes
MiCAP-Pro	**272**	Yes

The ZyCAP has the highest throughput among all of the reconfiguration controllers. The open-source controller with D^{2}PR-EDAC capability has the second highest throughput. However, both controllers lack the configuration data readback capability and hence they are suitable only for Dynamic Partial Reconfiguration (DPR) but not for Dynamic Circuit Specialization (DCS). The configuration

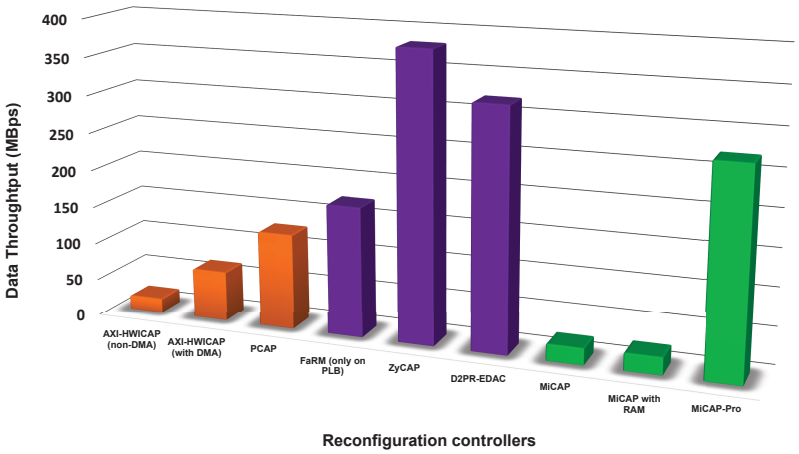


Figure 4.9: Data throughput of different reconfiguration controllers

read-back capability is a very essential feature required for DCS.

MiCAP-Pro has the third highest throughput among all of the reconfiguration controllers but remains the fastest reconfiguration controller that is suitable to implement DCS. The controller has the configuration read-back feature and is designed specifically to implement parameterized designs using DCS. Since the controller has I/O buffers, a significant delay is introduced before streaming the data into the ICAP port and therefore, throughput of MiCAP-Pro is 90 MBps lower compared to the ZyCAP. The buffers are necessary during configuration read-back so that the frames are not lost during the read procedure.

The FaRM controller has the configuration read-back capability but it is supported only on the Processor Local Bus (PLB) [43]. Therefore, the controller cannot be used on the Zynq-SoC unless major modifications are made. However, the throughput of the FaRM controller is not as high (≈ 100 MBps) as the throughput of MiCAP-Pro.

The throughput of the PCAP is less than half of that of MiCAP-Pro. Therefore, the reconfiguration speed using PCAP is lower than the reconfiguration speed using MiCAP-Pro. The estimated reconfiguration time of one TLUT using the PCAP is $\approx 124\ \mu s$.

The rest of the reconfiguration controllers has almost the same throughput as the AXI-HWICAP.

Table 4.4: Resource utilization of the reconfiguration controllers

Reconfiguration Controller	FF	LUTs	BRAMs
AXI-HWICAP (non-DMA)	675	500	1
PCAP	0	0	0
MiCAP	221	290	0
MiCAP with single port RAM	234	330	0
MiCAP-Pro	**2154**	**2032**	**2**

Note : For a reliable implementation, each reconfiguration controller can support up to maximum 100 MHz.

4.5.3 Resource utilization

The FPGA resources utilized by each of the reconfiguration controllers to implement DCS are tabulated in Table 4.4. The PCAP does not utilize any of the PL region (FPGA) resources since the AXI-PCAP bridge is a part of the PS region of the Zynq-SoC [26]. The basic MiCAP utilizes the lowest FPGA resources compared to the rest of the controllers that are implemented on the PL region. The AXI-HWICAP utilizes about twice as much resources than MiCAP and MiCAP with single port RAM.

However, MiCAP-Pro needs about $\approx 10\times$ more resources than MiCAP and ≈ 4 times more than the AXI-HWICAP. The main reason for this very high resource utilization is the usage of the DMA controller and the AXI stream bus to provide the interface with the region via HP ports. Therefore, MiCAP-Pro comes at the cost of extra FPGA resources for high speed reconfiguration for DCS. The trade-off of resources with improved speed will be further explored using the functional density.

4.5.4 Custom reconfiguration controllers and functional density

The effect of using high and low speed reconfiguration controllers on the overall system implemented using DCS can be best explained using the Functional Density explained in Chapter 3, Section 3.4.

The functional density curves for an adaptive FIR filter implemented using DCS with different reconfiguration controllers is shown in Figure 4.10. The x-axis represents the average time (in clock cycles) between two parameter value changes. The Generic implementation (FIR[3] filter implemented without DCS) has no variation in the functional density since it uses a fixed number of FPGA resources. It has the highest functional density if parameter values change frequently. The functional density for DCS approaches is higher than for the generic

[3]The functional density curves for other parameterized applications exhibit similar variations

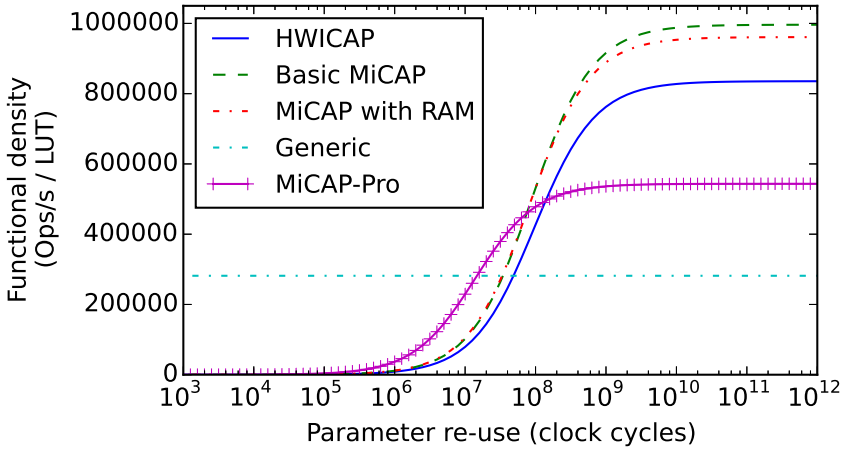


Figure 4.10: Functional Density curves for adaptive FIR filter

implementation as long as the parameters do not change values too frequently. So at the right side of the cross-over point (longer times between parameter changes) the DCS implementation is more beneficial. The curve for DCS implemented with MiCAP-Pro rises well before all other curves because of the drastic improvement in the reconfiguration speed. Thus, it allows the parameter values to change faster at the cost of extra implementation area. The DCS with MiCAP-Pro utilizes more implementation area (LUTs) which is why the magnitude of the functional density is lower for infrequent reconfigurations than with all other implementations except the generic implementation.

The curve for DCS implemented with basic MiCAP rises before the conventional DCS (with HWICAP) because of the improvement in the reconfiguration speed and hence the parameter values are allowed to change faster than before with the same gain in area. Similarly, the functional density curve for the DCS implementation using MiCAP with single port RAM rises just before the functional density curve with basic MiCAP due to the slight improvement in reconfiguration speed. The DCS with basic MiCAP and MiCAP with single port RAM uses less implementation area (less number of LUTs). Therefore, the functional density of each of these is much higher than that of DCS with HWICAP.

In conclusion, the functional density shows that the area-time trade-off of MiCAP-Pro is most beneficial with an average frequency of parameter value changes (between 10^7 and 10^8 clock cycles). For very infrequent parameter changes, the basic MICAP provides more gain as it takes less area and the reconfiguration time overhead has less impact.

Table 4.5: Average Power and Energy results of the reconfiguration controllers

Reconfiguration Controller	**Idle Power** (mW)	**Reconfiguration Power** (mW)	**Energy** (μJ)
AXI-HWICAP(non-DMA)	3.3	25.2	2838.3
MiCAP	0.6	3.9	314.5 (11.1%)
MiCAP-Pro	1.6	30.1	740.1 (26.1%)

Note 1: the reconfiguration controllers have a gated clock and the idle power is measured while the clock is turned off.
Note 2: the value between brackets show the percentage of power compared to the HWICAP.

4.5.5 Power and Energy analysis of the reconfiguration controllers

In this section I present the power and energy analysis of the reconfiguration controllers. I use the same power measurement setup explained in Chapter 3, Section 3.6.1 and the measurement bench shown in Figure 3.11 for the Xilinx ZC702 development board.

The idle and reconfiguration power of each reconfiguration controller is tabulated in Table 4.5. Clearly, the AXI-HWICAP consumes the highest idle power compared to the rest of the controllers. MiCAP is a lightweight controller[4] and therefore, it consumes much lower power compared to the other two controllers. MiCAP-Pro utilizes ≈ 4 times more functional resources than the AXI-HWICAP. Therefore, the power consumption during reconfiguration is higher than the AXI-HWICAP and MiCAP.

The energy consumption values show that MiCAP is 9 times more energy efficient than the AXI-HWICAP due to its lightweight feature. However, MiCAP-Pro's energy consumption is 4 times lower than the AXI-HWICAP due to increased reconfiguration speed and hence it proves to be more energy efficient than the AXI-HWICAP.

The power consumption during frame read, modify and write back activity is depicted in Figure 4.11. Clearly, the variation of power of MiCAP-Pro is moderately higher than the AXI-HWICAP while MiCAP has lowest variation of power during the micro-reconfiguration activity.

Since the reconfiguration time of the custom reconfiguration controllers is lower than for the AXI-HWICAP, the energy variation of MiCAP and MiCAP-Pro is much lower than for the AXI-HWICAP as shown in Figure 4.12. MiCAP proves to be the most energy efficient reconfiguration controller for DCS.

The source code of MiCAP and MiCAP-Pro can be accessed at [57] and [58] respectively.

[4] the data throughput of MiCAP is 17% better than that of the AXI-HWICAP

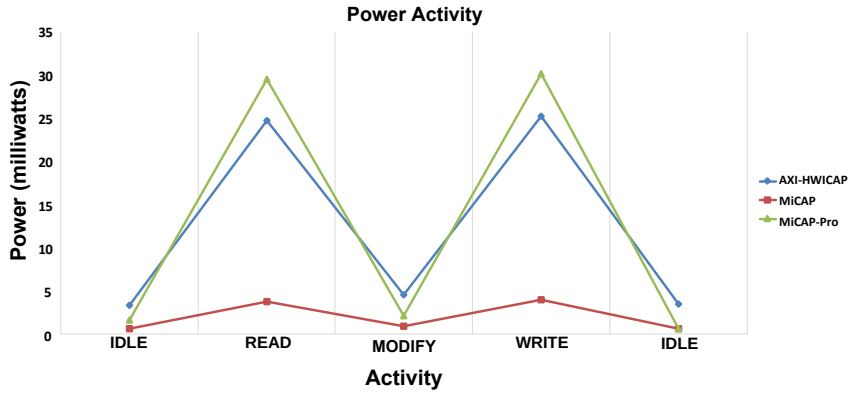


Figure 4.11: Power activity of the reconfiguration controllers

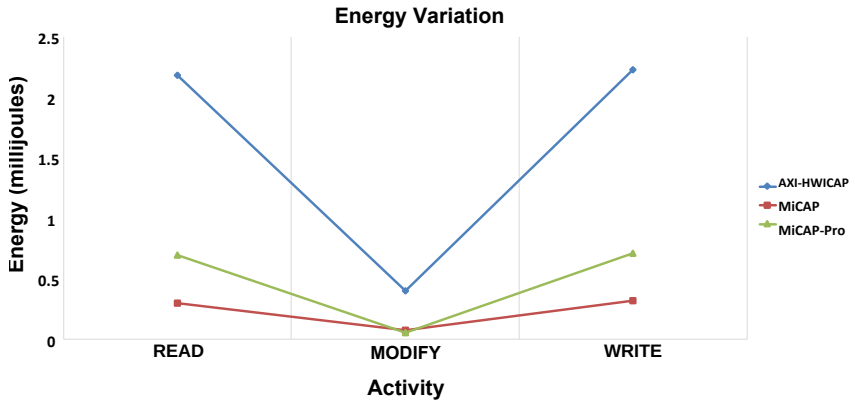


Figure 4.12: Energy variations of the reconfiguration controllers

4.6 Improving reconfiguration speed using placement constraints

The Xilinx HWICAP driver function "XhwIcap_setClb_bits" described in Chapter 3 Section 3.2.2.2 is used to reconfigure the truth table entries of a single LookUp Table (LUT) during run time. However, with existing Xilinx FPGA column based architectures, we propose to reconfigure multiple LUTs at the same time. We do this by using design placement constraints to cluster the bits that have to be changed in the same reconfiguration columns and customizing the reconfiguration driver function: "XhwIcap_setClb_bits". This gives us a significant improvement in reconfiguration speed. However this improvement comes at the cost of a slight reduction in the performance of the design.

4.6.1 Custom reconfiguration drivers

In this section, I propose two different principles to modify the reconfiguration drivers of the corresponding reconfiguration controllers. These modifications optimize the read activity during the micro-reconfiguration.

- MRMW: Multi - Read - Modify - Write: The conventional drivers follow the read-modify-write back principle to reconfigure every TLUT separately. In order to exploit the advantage of the existing frame structure that is imposed by the column based Xilinx FPGA architecture, we propose to modify truth table entries of multiple TLUTs within a single read activity. If multiple TLUTs of a parameterized design are placed in a single column then each of these TLUTs has a certain set of truth table entries that are located in the same frame. However, all 64 entries of a single TLUT are spread over 4 different frames. We have modified the reconfiguration process (into driver MRMW) that takes place in 3 steps:

    1. Read multiple frames: with the help of the frame address, four frames containing all the truth table entries of a column of TLUTs and LUTs are read from the configuration memory. If there are multiple TLUTs placed in a single column, the truth table values of multiple TLUTs are read with a single read activity.
    2. Modify frames: before modification, the function locates the truth table bits of all the TLUTs that are present in the frame. The current truth table entries of these TLUTs are replaced with the specialized truth table bits, which are generated by the SCG. Thus multiple TLUTs are specialized in a single attempt.
    3. Write-back frames: with the help of the same frame address, the modified or specialized truth table values are updated in all the TLUTs of the

column by swapping in multiple frames into the configuration memory of the FPGA. This updates all the truth table entries of multiple TLUTs that are placed in a single column.

Hence for a single read frames activity, multiple TLUTs can be reconfigured and this proves to be efficient since reading and writing back the frames for each TLUT can be avoided in contrast to the case of the conventional driver.

If the number of TLUTs in a parameterized design is higher than what fits in a single CLB column then multiple CLB columns containing multiple TLUTs can be used in order to achieve the gain in reconfiguration speed.

The TLUTs can be forcibly placed in a single column by using design placement constraints. However, the main concern with using the placement constraints is the design performance. Strict placement constraints would lead to hindrance of the design performance. There will be a trade-off between the reconfiguration speed and the design performance which is investigated in Section 4.7.2.

- MROMW: Multi - Read once - Modify - Write: the MRMW reconfiguration driver can be further optimized at the cost of DRAM memory. The memory is used as a cache to store the frames that are read during a reconfiguration. We have optimized the read frame activity for future reconfiguration of the same TLUTs.

  1. Read frames once: with the help of the frame address, four frames containing all the truth table entries of a column of TLUTs and LUTs are read from the configuration memory. If there are multiple TLUTs placed in a single column, the truth table values of multiple TLUTs are read with a single read activity. Once the frames are read, each frame is stored in DRAM memory of the Zynq-SoC. If the processor has to reconfigure the same TLUTs at a later time, it can directly access the frames from the DRAM memory instead of requesting the same frames from the configuration memory via the ICAP. Since the data access from the DRAM memory is faster than the configuration memory, the read frame activity can be bypassed for the future reconfigurations of the same TLUTs.

  The rest of the reconfiguration steps: multi-modify and write-back frames remain unchanged. However, the bitstream's cache is updated for every write-back activity in order to keep the cached bitstream consistent with the actual configuration of the FPGA.

Table 4.6: Dimensions for the Placement Constraints

	16-tap FIR	**32-tap FIR**	**64-tap FIR**
Number of TLUTs to be clustered	384	768	1536
Zynq-SoC	50 × 5	50 × 11	50 × 14
Virtex-5	20 × 13	20 × 27	20 × 38

Note: Above dimensions are in the form of Length × Width of the CLB columns.

4.6.2 Placement constraints to improve reconfiguration speed

The main aim of using placement constraints is to force multiple TLUTs to cluster all their truth table entries in a minimal number of frames. The placement constraints are used to restrict where the design's logic is placed. It forces the placer to use a certain area of the FPGA. We have described the correlation between the CLB columns and the frame structure in Chapter 2 Section 2.1.3. Our approach is to force more TLUTs to be placed in a single CLB column so that their truth table entries can be reconfigured with a minimal number of frame accesses.

We have used the "AREA_GROUP" constraint [59]. This constraint allows us to specify that certain parts of the design can only be placed in a pre-determined rectangular region of the FPGA's CLBs. To determine the exact size of this rectangular region the maximum length of the CLB column and minimum width of the CLB rows have to be considered. The maximum length of the CLB column is equal to its height (50 for the Zynq-SoC) in a given clock region and it ensures that more TLUTs can fit the specified area, while the minimum CLB rows ensures that we use the minimal number of CLB columns possible.

The exact area constraint differs for both targeted FPGAs. We first used the constraint to place the TLUTs in an exact minimum number of CLB columns determined by the number of LUTs present in it. For example, in the Zynq-SoC each column has 200 LUTs. Therefore to place the 64-tap FIR filter (1536 TLUTs), it is sufficient to use 8 columns. However with 8 columns, the router was not able to route the design. Hence we increased the width of the rectangular area by increasing the number of columns until the router was able to route the whole design. The width of the rectangular area in terms of CLB columns for different configurations of the FIR filter is tabulated in Table 4.6.

For a 64-tap FIR filter, the average number of TLUTs clustered in a single CLB column of the Zynq-SoC is 110 which is 52% of the total LUTs available in a single CLB column and there are a maximum of 156 TLUTs clustered in a single column which is 75%, remaining LUTs are not a part of the reconfiguration process and hence they are used for the non-reconfigurable parts of the problem. Table 4.7 shows the percentage of TLUTs clustered.

Table 4.7: TLUTs cluster rate of 64-tap FIR filter in a single CLB column

	Zynq-SoC	
	Average	**Maximum**
Clustered TLUTs	52%	75%
Remaining LUTs	48%	25%

Table 4.8: FIR filter configurations

Taps	**Multipliers**	**TLUTs**
16	32	384
32	64	768
64	128	1536

4.7 Results on custom reconfiguration drivers

In this section, I present the results of our experiments on custom reconfiguration drivers: MRMW and MROMW. Table 4.9 shows the reconfiguration time distribution of a TLUT using three different reconfiguration controllers. Clearly, MiCAP-Pro is the fastest reconfiguration controller between all three controllers. In order to evaluate the effect of using custom reconfiguration drivers on the three controllers, we consider the total reconfiguration time (time taken to reconfigure all the TLUTs of the DCS system).

There were 3 different experiments conducted on each of the reconfiguration controllers:

1. Experiments with MRMW reconfiguration drivers and without placement constraints.
2. Experiments with MRMW reconfiguration drivers and with placement constraints.
3. Experiments with MROMW reconfiguration drivers and with placement constraints.

4.7.1 Experiments with MRMW reconfiguration drivers and without placement constraints

In this experiment, we have not used placement constraints and hence the TLUTs were automatically placed by the placer without constraints from the user. The placer tool had full freedom to choose its own place for the TLUTs in different CLB columns.

Table 4.9: Reconfiguration time distribution of a single TLUT

Reconfiguration controller	**Micro-reconfiguration task**	**Time** (μs)	TLUT **Reconfiguration time** (μs)
HWICAP	Read Frames	111.5	234
	Boolean Evaluate and Modify	18	
	Write back Frame	100.5	
MiCAP	Read Frames	97	210
	Boolean Evaluate and Modify	18	
	Write back Frame	95	
MiCAP-Pro	Read Frames	23	64.1
	Boolean Evaluate and Modify	18	
	Write back Frame	23.1	

Table 4.10: CLB columns - TLUTs placed without placement constraints

	16-taps FIR	**32-taps FIR**	**64-taps FIR**
No. of TLUTs to be clustered	384	768	1536
CLB Columns	25	42	50

Table 4.10 shows the number of columns in which the TLUTs were placed by the placer without any placement constraints. Further investigations have shown that there were multiple TLUTs placed for a given CLB column and therefore, we can still use the principle of modifying multiple TLUTs for a single read activity

The TLUTs of the parameterized FIR filter design were reconfigured with 3 different reconfiguration controllers. We used custom reconfiguration drivers of MRMW. The corresponding time required to reconfigure all the TLUTs of the parameterized design and the Improvement Factor (IF) is tabulated in Table 4.11.

Table 4.11: Total Reconfiguration time without placement constraints

FIR filter taps	**TLUTs**	**Reconfiguration controller**	**Total reconfiguration time (ms)**	**Improvement factor**
16	384	HWICAP	88.3/ 7.4	12
		MiCAP	80.6 / 6.9	12
		MiCAP-Pro	24.6 / 3.3	8
32	768	HWICAP	176.6 / 13.1	13
		MiCAP	161.2 / 12.2	13
		MiCAP-Pro	49.2 / 6.2	8
64	1536	HWICAP	353.2 / 18.4	19
		MiCAP	322.4 / 17.4	19
		MiCAP-Pro	98.4 / 12.1	8

Note: above timing values are in the form of normal reconfiguration drivers / custom MRMW reconfiguration drivers.

Table 4.12: Total Reconfiguration time with placement constraints

FIR filter taps	TLUTs	Reconfiguration controller	Total reconfiguration time (ms)	Improvement factor
16	384	HWICAP	88.3/ 4.4	20
		MiCAP	80.6 / 4.3	19
		MiCAP-Pro	24.6 / 3.1	8
32	768	HWICAP	176.6 / 9	20
		MiCAP	161.2 / 8.7	19
		MiCAP-Pro	49.2 / 6	8
64	1536	HWICAP	353.2 / 16.4	22
		MiCAP	322.4 / 16.1	20
		MiCAP-Pro	98.4 / 9.6	10

Note: above timing values are in the form of normal reconfiguration drivers / custom MRMW reconfiguration drivers.

Clearly, there was a drastic reduction in the reconfiguration time compared to the standard reconfiguration drivers. The reconfiguration speed was improved drastically, at least by a factor of 12 for the HWICAP and MiCAP. Similarly, the improvement in reconfiguration speed by a factor of ≈ 8 was observed for MiCAP-Pro. This improvement was achieved since we overcome the reading of the same frames that contain configuration of multiple TLUTs. The data transfers between PS and PL regions of the Zynq-SoC are the major bottleneck for the HWICAP and MiCAP. Therefore, bypassing the frame read activities in the driver contributes a lot to the reconfiguration speed and is the major reason for the improvement in the reconfiguration speed. Since we did not use any placement constraints, the overall performance of the DCS system remains unchanged.

4.7.2 Experiments with MRMW reconfiguration drivers and with placement constraints

In this experiment, we force the placer tool to place the maximum possible number of TLUTs in an exact minimum number of CLB columns by using "AREA_GROUP" placement constraints. Table 4.6 shows the minimum CLB columns in which the TLUTs were placed. The parameterized design was reconfigured using MRMW drivers using three different controllers. The total reconfiguration time is tabulated in Table 4.12.

Clearly, the reconfiguration speed was even further improved by at least a factor ≈ 20 for the HWICAP and MiCAP. The reconfiguration speed was improved at least by a factor ≈ 8 for MiCAP-Pro. The improvement is due to the placement of TLUTs in a reduced number of CLB columns compared to the previous experiment.

With the help of the placement constraints, the truth table entries of multiple TLUTs were clustered in a single CLB column. Therefore, this method gives an

Table 4.13: Maximum clock frequency the design can support on the Zynq-SoC

	16-taps FIR	**32-taps FIR**	**64-taps FIR**
No. of TLUTs	384	768	1536
Clock frequency in MHz	108.6 / 102.8	108.6 / 102.2	108.6 / 101.2

Note: above clock frequency values are in the form of without placement constraints / with placement constraints.

advantage of modifying more TLUTs for a single frame read activity. The MRMW driver exploits the advantage and reconfigures multiple TLUTs thereby reducing the reconfiguration time.

The improvement in the reconfiguration speed comes at the cost of a reduction in the design performance. Introducing the placement constraints causes the design to have a longer critical path then the conventional implementation. This causes a decrease in the maximum clock frequency the design can support as observed in Table 4.13. Clearly, an increase in the number of TLUTs decreases the design performance. The overall average deterioration in design performance is about 6 MHz (or a deterioration of $\approx 6\%$). A similar experiment on clustering the dynamic bits for multi-mode circuits [60] (design with different configurations) for an adaptive FIR filter is presented in [61]. A maximum of 30% increase in wire length is observed.

4.7.3 Experiments with MROMW reconfiguration drivers and with placement constraints

In this experiment, we used a custom reconfiguration driver of MROMW. Introducing the placement constraints reduces the number of CLB columns in which the TLUTs are placed. When using a custom driver of MROMW, the frames that contain TLUT truth table entries are stored in the DRAM of the Zynq-SoC after they are read during the reconfiguration of the TLUTs for the first time. The DRAM acts as a cache so that we can reuse the truth table entries for reconfiguring the same TLUTs in future requests. Therefore, we bypass the frame read activity and hence the reconfiguration time is reduced. Table 4.14 shows the reconfiguration time of the DCS system after using the MROMW reconfiguration driver. Clearly, we observe an improvement in reconfiguration speed by 12% compared to the MRMW driver. However, this small improvement comes at the cost of memory that is used to store the frames for reconfiguring the TLUTs. The DRAM should store at least 404 words (1 word = 32 bits) of the frame data to reconfigure multiple TLUTs present in a single CLB column.

We limit the use of the MROMW driver to the experiments with placement constraints only. This is because the number of frames that contain truth table entries of TLUTs is small compared to the number of frames without placement

Table 4.14: Reconfiguration time using MROMW driver

FIR filter taps	TLUTs	Reconfiguration controller	Total reconfiguration time (ms)
16	384	HWICAP	4.4 / 3.8
		MiCAP	4.3 / 3.8
		MiCAP-Pro	3.1 / 2.8
32	768	HWICAP	9 / 7.7
		MiCAP	8.7 / 7.7
		MiCAP-Pro	6 / 5.4
64	1536	HWICAP	16.4 / 14.7
		MiCAP	16.1 / 14.7
		MiCAP-Pro	9.6 / 9

Note: Above values are in the form of custom MRMW drivers/ MROMW drivers.

constraints and therefore it is worth to store minimum possible frames rather than storing the frames that contain TLUTs which are wide spread across the multiple clock regions of the FPGA.

For a 64-taps parameterized FIR filter, in order to store all the frames (that contain truth table entries of 1536 TLUTs) in the DRAM memory, we need a memory space of 5656 words ($14 \times 404 = 5656$) or $\approx$ 23 KB in the DRAM memory. The comparison of the reconfiguration time of the parametrized FIR filter with 64-taps implemented using DCS using different reconfiguration controllers is depicted in Figure 4.13. The naming conventions used in the following figures is described in Table 4.15.

Clearly, the TLUTs of the parameterized design are reconfigured efficiently with less overhead of reconfiguration time using custom reconfiguration controllers when used along with custom MRMW and MROMW reconfiguration drivers.

4.7.4 Functional density curves

The functional density curve was plotted against the rate of change of the input parameters for the adaptive FIR filter with three different configurations: 16, 32, and 64 taps. We have plotted the functional density curve for each reconfiguration controller and observed the variation in functional density of the DCS system after using standard, custom MRMW without placement constraints, MRMW and custom MROMW drivers with placement constraints.

The functional density curves for the HWICAP, MiCAP and MiCAP-Pro are depicted in Figure 4.14, Figure 4.15 and Figure 4.16 respectively. The naming conventions for the reconfiguration controllers are listed in Table 4.15. The x-axis represents the average time (in clock cycles) between two parameter value changes. We observe a similar behavior of functional density curves in the DCS

Table 4.15: Naming convention for reconfiguration controllers and their definitions

Reconfiguration controllers	**Definition**
(Reconf_controller)1	DCS system with (Reconf_controller) and standard reconfiguration driver. (read, single modify, write)
(Reconf_controller)2	DCS system with (Reconf_controller) with custom MRMW reconfiguration driver and without placement constraints.
(Reconf_controller)3	DCS system with (Reconf_controller) with custom MRMW reconfiguration driver and with placement constraints.
(Reconf_controller)4	DCS system with (Reconf_controller) with custom MROMW reconfiguration driver and with placement constraints.

Note: (Reconf_controller) can be HWICAP, MiCAP and MiCAP-Pro in the above naming convention.

systems implemented using three different reconfiguration controllers.

The functional density for the DCS with custom MRMW driver (without placement constraints), rises well before the functional density of the DCS that uses the standard reconfiguration driver, introducing the placement constraints for MRMW and MROMW custom drivers improves the reconfiguration speed furthermore and hence the corresponding functional density curves rise earlier compared to the functional density curve with standard reconfiguration drivers. This shows that improving the reconfiguration speed allows the parameters to change faster with the same gain in area compared to DCS whose reconfiguration speed is slow.

However, since the design performance is slightly reduced due to placement constraints, the magnitude of the functional density curves is relatively lower compared to the DCS without placement constraints forming the main trade-off. The HWICAP and MiCAP have similar functional density curves (except MiCAP has a higher magnitude of functional density) since they have approximately equal throughput [62]. Using the custom reconfiguration drivers improves the reconfiguration speed drastically. However, the functional density curves for MiCAP-Pro only show a small improvement in reconfiguration speed after using MRMW and MROMW reconfiguration drivers. Since the data throughput of MiCAP-Pro is very high, the effect of using custom reconfiguration drivers to improve the reconfiguration speed is relatively lower. The impact of using placement constraints can be also seen in the functional density curves of MiCAP-Pro.

Commercial applications such as TCAM used for packet classification in network routers [63] can benefit from DCS. A content of the memory is an infre-

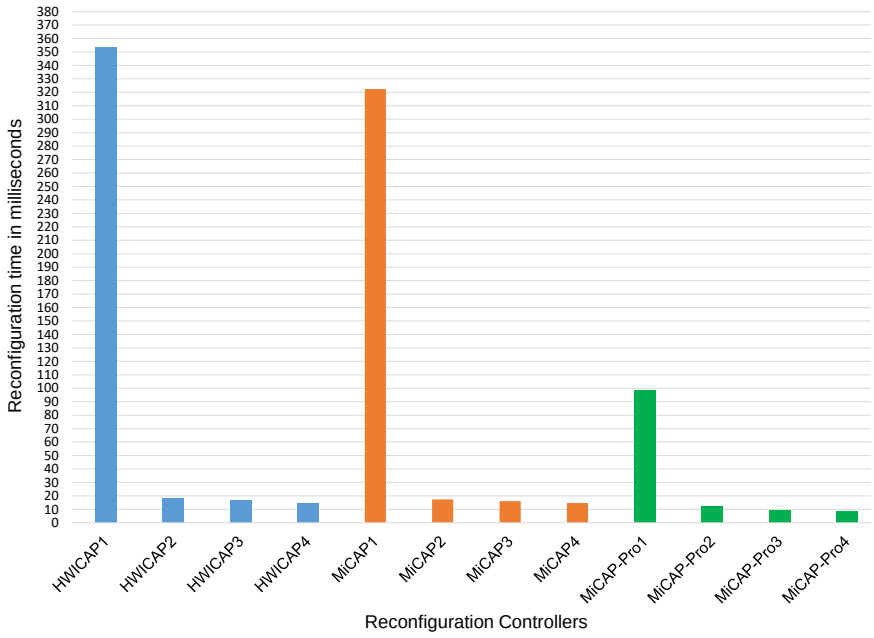


Figure 4.13: Reconfiguration time comparison between standard reconfiguration driver and custom reconfiguration drivers

Note: the naming conventions are explained in Table 4.15

quently changing input value and therefore, can be used as a parameter input. However, in the network routing if the content of the TCAM has to be updated then the reconfiguration speed plays an important role. If the reconfiguration speed is too slow then it affects the network router's performance. The parameterized TCAMs can benefit from our proposed methods and overcome the barrier of the reconfiguration time without affecting the router's performance.

Until now I have presented my work on the fine-grained level of FPGA. In the next chapter, I evaluate the effect of custom reconfiguration controllers and custom reconfiguration drivers on coarse-grained reconfigurable arrays realized on the FPGAs.

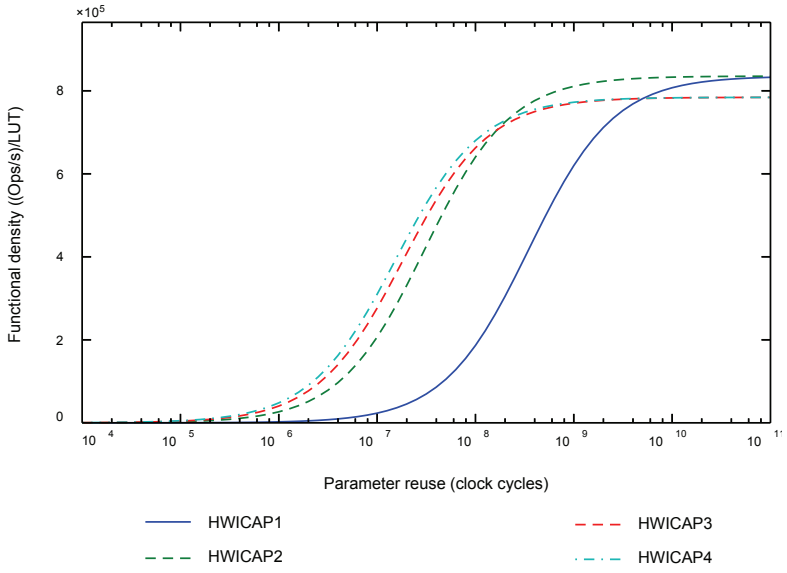


Figure 4.14: Functional Density curves for HWICAP with different reconfiguration drivers

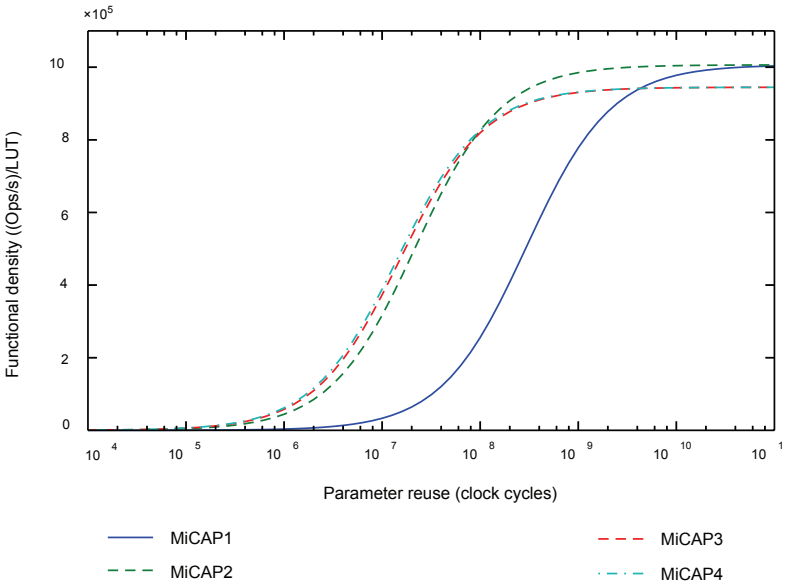


Figure 4.15: Functional Density curves for MiCAP with different reconfiguration drivers

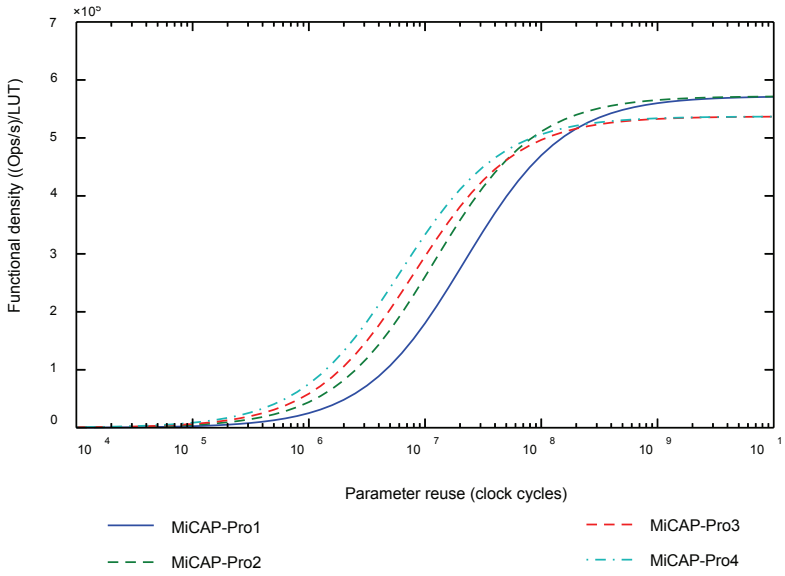


Figure 4.16: Functional Density curves for MiCAP-Pro with different reconfiguration drivers

5

DCS for FPGA Overlay architectures

A software programmer with limited knowledge of hardware will encounter various hurdles to develop applications that run on FPGAs. Indeed, software programming is a very different experience than realizing the applications on an FPGA architecture. This usually results in lower design productivity for software engineers and creates a huge gap between what resources the application requires and what resources the FPGA fabric actually provides when the same application is implemented optimally. This design gap can be alleviated by bringing the FPGA implementation closer to a programming model through the use of FPGA overlays. FPGA overlays are larger functions that are implemented efficiently on the FPGA fabric and are programmable through software programming methods. In this chapter, I present basics of FPGA overlays and their characteristics. The role of DCS in FPGA overlays is an important aspect of my research, and hence an efficient implementation for FPGA overlays (using DCS) is presented as well.

5.1 Introduction to Overlays

An FPGA overlay is a virtual architecture that overlays on top of the physical FPGA fabric [64]. Adding reconfigurability enables customization of the applications and supports more than one application. The overlay can then be a virtual FPGA or a computing architecture called Virtual Coarse-Grained Reconfigurable Array (VCGRA) specifically designed to realize an application at hand. The applications are described using high-level languages that can be easily compiled and

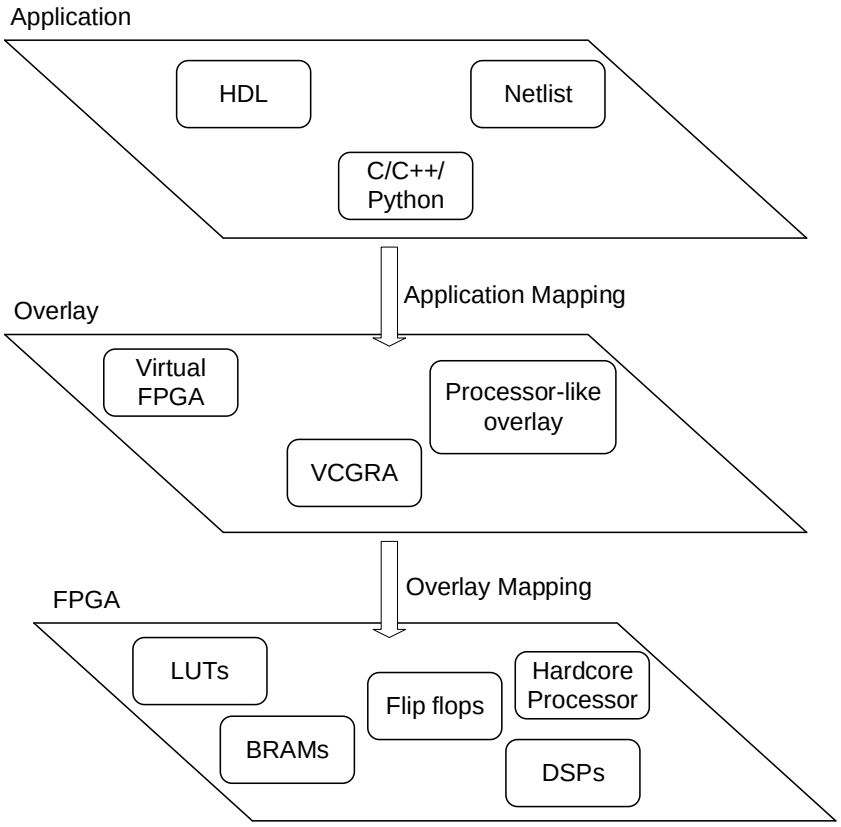


Figure 5.1: An overlay architecture for FPGA application development [64]

mapped on to the virtual overlay layer without knowing the details of the low-level FPGA hardware (Figure 5.1).

5.1.1 Types of Overlays

The FPGA overlays can be grouped into three types: Virtual FPGAs, Virtual Coarse-Grained Reconfigurable Arrays (VCGRAs) and Processor-like overlay.

1. **Virtual FPGAs** are built virtually or physically on top of the off-the-shelf FPGA fabrics. The virtual FPGA overlays have different configuration and features than the typical FPGA device. Therefore, having a virtual FPGA layer over an FPGA fabric improves the application portability and compatibility. The virtual architectures proposed in [65] [66] [67] and [68] are examples for virtual FPGAs.

2. **VCGRAs** are the Coarse-Grained Reconfigurable Arrays (CGRAs) enable ease of programmability and result in low development costs. They enable the ease of use specifically in reconfigurable computing applications. The smaller cost of compilation and reduced reconfiguration overhead enables them to become attractive platforms for accelerating high-performance computing applications such as image processing. The CGRAs are ASICs and therefore, expensive to produce for low-volume products. However, FPGAs are relatively cheaper for low volume products but they are not so easily programmable. The combination of the best of both worlds results in implementing a Virtual Coarse-Grained Reconfigurable Array (VCGRA) on top of the fine-grained FPGA fabric. This coarsens the granularity of an FPGA from its physical fine-grained configurable fabric (LUTs, FFs, BRAMs, DSPs, etc.). VCGRAs provide a trade-off between performance of the hardware and the flexibility of a software program for compute intensive parts of the applications. Therefore, VCGRAs are suitable to accelerate compute intensive kernels. The overlays presented in [34] [69] [70] are the examples of VCGRAs. There also exist an example for overlay on an overlay (superimposed virtual architecture) called SICTA. This is mainly used for online debugging of VCGRA and supports on-demand three level fault-mitigation [71].

3. **Processor-like overlays** are processor-like intermediate layers built on top of the FPGA fine-grained FPGA fabric. The compatibility and usability of processor-like overlays are developed in the user's perspective. Therefore, customized soft-processors with user defined ISA fall under this category. These overlays provide a high degree of control and extensive data parallelism to make them suitable for data acceleration. The fine-grained vector processors presented in [72] [73] [74] [75] are examples of processor-like overlay architectures.

5.1.2 Benefits of Overlays

The overlays bridge the gap between the applications and the FPGA fabric. The virtualization inherits many benefits to the software programmer from a software programming model and hence software programmers can expect similar benefits as with CPU virtualization such as compatibility, portability and isolation. Apart from these benefits employing FPGA overlays has proven to offer high speed compilation thereby improving the designer's productivity with better debugging support [64].

My research contributions are situated in the area of VCGRAs and therefore, in the following I will emphasize more on VCGRAs. To understand VCGRAs in detail, I first describe about CGRAs in the following section.

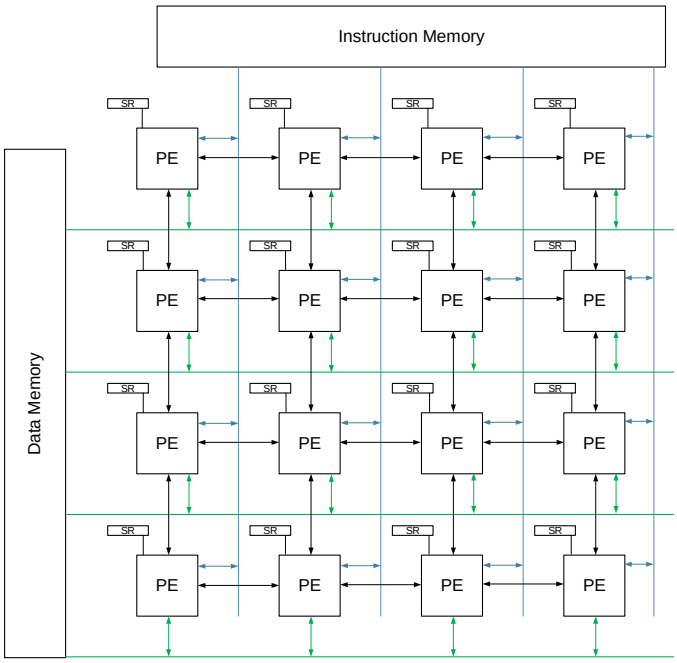


Figure 5.2: A CGRA architecture

5.2 Coarse-Grained Reconfigurable Arrays (CGRAs)

The FPGAs provide high flexibility but they suffer from low efficiency (compared to ASICs) due to fine bit-level granularity reconfiguration that results in longer reconfiguration penalty. The CGRAs serve as the promising alternatives between FPGAs and ASICs.

As the name suggests, CGRAs comprise of coarse-grained processing elements (PEs) or functional units (FUs) connected via mesh-like interconnect as shown in Figure 5.2. Each PE is associated with a settings register (SR) (also called control register) that decides the functionality[1] of the PE and can be programmed on a per cycle basis. The PEs are capable of performing complex functions and hence they are more powerful than the LUTs of an FPGA.

[1]Each PE is multiplexed with different functionality, the select lines that decide the function of each PE are connected to the settings register. Therefore, the settings register decides the function of each PE.

5.3 Virtual Coarse-Grained Reconfigurable Arrays (VCGRAs)

The programming of the FPGAs (called configuration) is usually done starting from the RTL that describes a circuit in the lower abstraction level (gate level) and therefore the compilation time is high, thus resulting in a slow design cycle. However, this limitation is not present for processing units (CPUs, GPUs and DSPs) since they can be easily programmed at a higher abstraction level and hence they have very low development cost and shorter time-to-market [76]. Although a designer can choose between Processors and ASICs for his design implementation, processors have high flexibility but the performance is often too low and especially the performance per Watt. The ASICs cost too much for low-volume products. FPGAs provide high flexibility and low-cost (price) targeting low-volume products, but they suffer from a longer development cycle which becomes inevitable for the designer.

To overcome such limitations for the FPGAs, a CGRA can be realized on an FPGA. This type of implementation is called Virtual Coarse-Grained Reconfigurable Array (VCGRA). The programming model for the VCGRAs is different by the fact that the code can be written on a higher abstraction level. This reduces the compilation time by several orders of magnitude as compared to the fine-grained FPGA, thus VCGRAs act as intermediate virtual fabrics [67] to curb the development costs. Figure 5.3 shows a fragment of a VCGRA. The architecture consists of groups of coarse-grained processing elements connected using virtual connection blocks and switch blocks forming a communication network (inter-connect). The processing elements are powerful and more complex than the LUT of an FPGA. The complexity of the processing elements can range from a simple ALU to a fully capable RISC processor.

Each PE has a settings register used to configure a function of the PE. With the proper connection settings (configured in the settings register of the VSB - Virtual Switch Block), every application that uses these PEs can be implemented. The settings registers are realized on FFs and are updated using a dedicated bus that enables us to program the settings of the PEs and VSBs.

5.3.1 Conventional VCGRA tool flow

The tool flow to implement a conventional VCGRA[2] is depicted in Figure 5.4. The tool flow is obtained by combining the standard FPGA tool flow (left hand side) with the VCGRA tool flow (right hand side).

The right hand side of the tool flow describes the mapping of an application on a given (V)CGRA architecture. This is also called Spatial programming.

[2] A standard CGRA implemented on an FPGA is called conventional VCGRA.

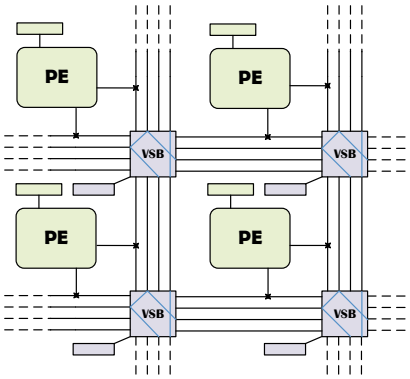


Figure 5.3: A fragment of VCGRA grid with Processing Elements (PEs), Virtual Switch Blocks (VSB) and corresponding settings registers (rectangles)

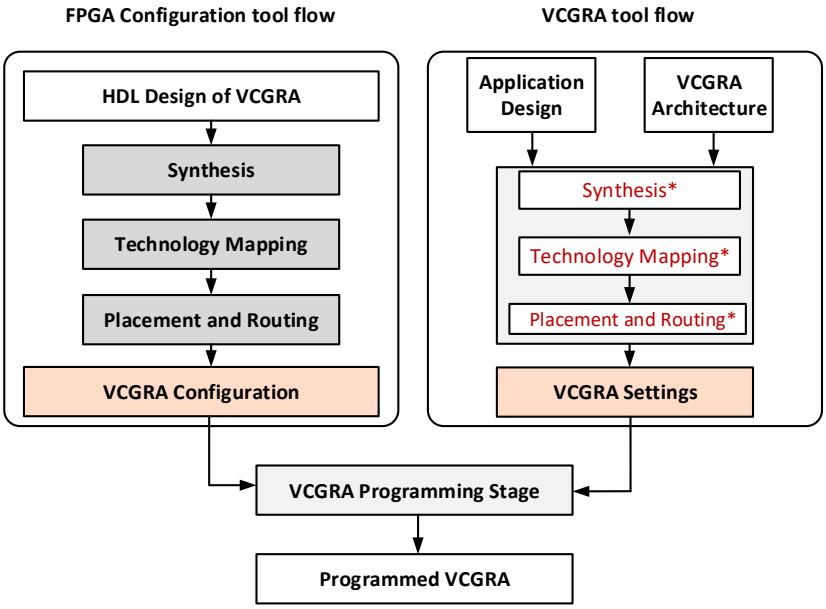


Figure 5.4: Tool flow for CGRA implemented on an FPGA (conventional VCGRA)

*Note: * indicates steps considering PEs (and VSBs) as a basic programmable component*

There are different approaches to solve the CGRA mapping problem described in [77] [78] [79]. Since the application has to be mapped on the coarser PEs,

the compilation time is much shorter than the mapping of an application on the fine-grained bit-level. The VCGRA tool flow generates the settings for the PEs and VSBs as a result. In conventional VCGRAs, these settings are loaded (programmed) into the settings register using a dedicated bus.

The higher level VCGRA tool flow that produces these VCGRA settings consists of a synthesis and a mapping tool in which the textual description of the application design is parsed and converted into a netlist of Processing Elements (PEs). Next, we perform placement of the synthesized netlist of PEs on to the virtual PEs of the VCGRA architecture. The tool flow has to take care that all the interconnection links of the VCGRA are implemented on the VCGRA architecture's communication network. We make use of a router to establish optimal connections between the placed elements of the VCGRA architecture. The Place and Route (P&R) result determines what the functionality of each PE is and how the communication network is exactly used and that is reflected in the VCGRA settings.

Because the basic programmable element in the VCGRA tool flow is a PE, the tools (synthesis, mapper, place and route) need considerably less complexity and time to generate the settings values than the standard FPGA compilation would. This is because the higher abstraction level reduces the problem size and therefore the tools are faster.[3] If the application design specification changes with the same VCGRA platform then we can generate the settings values much faster than processing the new design with the standard FPGA tool flow.

As will be explained further, the settings registers, PEs and VSBs can be optimized using the parameterized configuration tool flow that will result in a VCGRA implementation using DCS. The level of parameterization within the PEs leads to a classification of the VCGRA (implemented using DCS) into two types:

1. Partially parameterized VCGRA: in this type of VCGRA, the settings register controls the programming of a PE , just as in the conventional VCGRA. The PE contains an input (non parameterized) connected to the settings register. The functionality of the PE is decided by the different combinations of data present in the settings register. This is accomplished by multiplexing different circuits within a PE. The multiplexers (intra-connects) within a PE are not parameterized. However, for efficient implementation, the settings registers themselves are implemented on LUTs. In the parameterized configuration approach these LUTs are implemented using TLUTs, thus we can get rid of the low speed bus needed to update the LUT memory values for the settings register. Therefore, to update the settings register (implemented using TLUTs) we reconfigure the truth table entries of the TLUTs using one step micro-reconfiguration. Hence to reprogram a PE, it is sufficient to

[3] In my experiments, this is done using an ad-hoc tool but this can easily be generalized in the future.

have a single level micro-reconfiguration i.e, changing the contents of the settings register. For the VSBs the settings registers are optimized in the same way as for the PE. However, with the help of TCONMAP mapper we are able to map the virtual channels on to the TCONs. Hence all the virtual interconnects that use LUTs (as in the case of conventional VCGRAs) are optimized.

2. Fully parameterized VCGRA: in this type of VCGRA, not only the settings register but also the intra-connects of a PE are parameterized. The parameterization is achieved by expressing the intra-connects of a PE as Boolean functions of parameters present in the settings registers. Therefore, there exist TLUTs and TCONs within a PE and to change the functionality we need two levels of micro-reconfiguration: one to change the contents of the settings register and the second to change the configuration of the intra-connects of the PE based on the values in the settings register.

5.3.2 Partially parameterized VCGRA tool flow

Figure 5.5 explains the tool flow for the partially parameterized VCGRA design. The VCGRA tool flow makes use of a VCGRA architecture that defines the granularity as well as the possible functionality of the PEs and describes the possible ways the PEs are interconnected. The PEs are implemented using the standard FPGA configuration tool flow (left hand side of Figure 5.5), hence no optimization is expected. However, the settings registers and inter-connect VSBs are parameterized using the parameterized reconfiguration tool flow and is explained in Section 5.3.3.

In the VCGRA tool flow (right hand side of Figure 5.5) is essentially same as the VCGRA tool flow explained in the sub section 5.3.1, the user will determine the VCGRA settings that will program the required programmable elements of the partially parameterized VCGRA to realize the desired application. The programming is accomplished via one step micro-reconfiguration of the VCGRA settings registers with the VCGRA settings as parameters. The left hand side of the tool flow is responsible for generating VCGRA structure in such a way that it results in partially parameterized VCGRA configuration.

In the partially parameterized configuration tool flow, only the settings register of each PE and VSB is parameterized (this is shown in the middle part of the tool flow Figure 5.5). This will result in realizing settings register on TLUTs. Therefore, once the specialized configuration is generated only one step micro-reconfiguration is necessary to update the settings registers of PEs and VSBs.

Authors in [80] have shown that partially parameterized VCGRA implementations use LUTs of the FPGA to realize reconfigurable virtual switch blocks, virtual registers and other virtual components. With the help of parameterized configu-

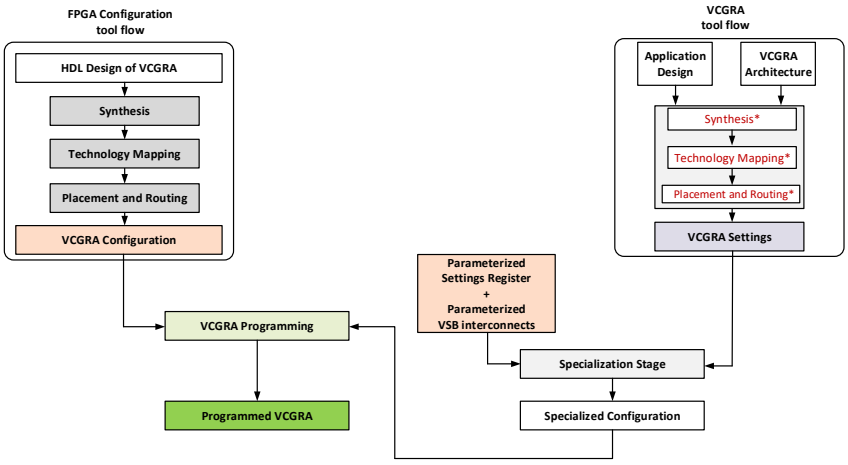


Figure 5.5: Implementation of applications on a partially parameterized VCGRA using the parameterized configuration tool flow

*Note: * indicates steps considering PEs (and VCs) as a basic programmable component*

ration, the authors were able to map the virtual components to the lower level physical resources (settings registers were mapped onto configuration memory and VSBs were mapped onto switch blocks) of an FPGA. This saves a significant amount of functional resources (LUTs) of an FPGA.

In the next section, I will recap the tool flow for generating parameterized configurations that was explained in Chapter 3, Section 3.2.

5.3.3 Tool flow for parameterized configuration

The VCGRA tool flow builds on top of the parameterized reconfiguration tool flow, which provides a parameterized version of the settings registers and the interconnect VSBs (left side of Figure 5.5). The tool flow for generating parameterized configuration was explained in Chapter 3, Figure 3.2. I will recap briefly the two staged tool flow here.

There are two stages: a *generic* stage and a *specialization* stage. In the *generic* stage, a HDL design with parameterized inputs (annotated with "`-PARAM`") describes the application. The application is processed to yield a Template Configuration (TC) and a Partial Parameterized Configuration (PPC). A detailed explanation on each step of the generic stage is explained in [48].

The final output of the generic stage is the TC and PPC. The TC contains static bits (0's and 1's) which are the non-reconfigurable parts of the implementation.

The PPC contains multi-port Boolean functions of the parameter inputs. In order to generate specialized bitstreams, the PPC has to go through the specialization stage.

Upon a value change in parameter inputs, the configuration bits of the TLUTs and TCONs are reconfigured with specialized bits that are thus generated after evaluation of the Boolean functions for a specific set of parameter values.

In the *specialization* stage, for every change in parameter values, the Boolean functions are evaluated by a Specialized Configuration Generator (SCG) to generate specialized bitstreams. The SCG takes a specific parameter value and evaluates the Boolean functions to produce specialized bits. The SCG can be implemented on an embedded processor (PowerPC, ARM or MicroBlaze) present in the FPGA. With the help of a configuration interface such as HWICAP or MiCAP [50] [62], the FPGA is reconfigured with the specialized bitstream.

This means that in the parameterized VCGRA approach, the settings registers are mapped onto the configuration memory of the FPGA which is in contrast with conventional implementations that map the registers to the flip-flops of a logic cell present in the FPGA. Therefore, a significant amount of flip-flops can be saved.

As long as we are only reconfiguring LUTs, we are able to parameterize LUTs on commercial Xilinx FPGAs. However, parameterization of other primitives such as routing switches can be done only on a hypothetical FPGA but not on the commercial FPGAs (because we do not have access to the low level routing reconfiguration infrastructure). Hence, the tool flow explained above is used to generate the parameterized configuration for a hypothetical FPGA.

5.3.4 Fully parameterized VCGRA tool flow

The work presented in the previous section was limited to parameterize the registers of the PEs and inter-connect (VSBs) of the VCGRA only, thus resulting in a partially parameterized VCGRA. In this section, my main contribution is to parameterize not only the PEs and VSBs, but also the connections within each PE (intra-connect) using Tunable Connections (TCONs) [81]. This is accomplished by using TCONMAP mapper [30]. The parameterized intra-connections can be automatically mapped to the lower level physical connection blocks and switch blocks of the FPGAs, thus avoiding the use of LUTs to implement the intra-connect. Therefore, we save an additional amount of physical LUT resources of the FPGA. However, this will add an extra micro-reconfiguration step that is needed to reconfigure intra-connects of each PE as shown in Figure 5.6. Thus, the fully parameterized VCGRA can be programmed using two steps micro-reconfiguration.

1. Micro-reconfiguration of a settings register: the settings register of each PE can act as an input or a parameter for each PE and hence it is an inevitable configurable part of the PE. Since the settings register content is part of the

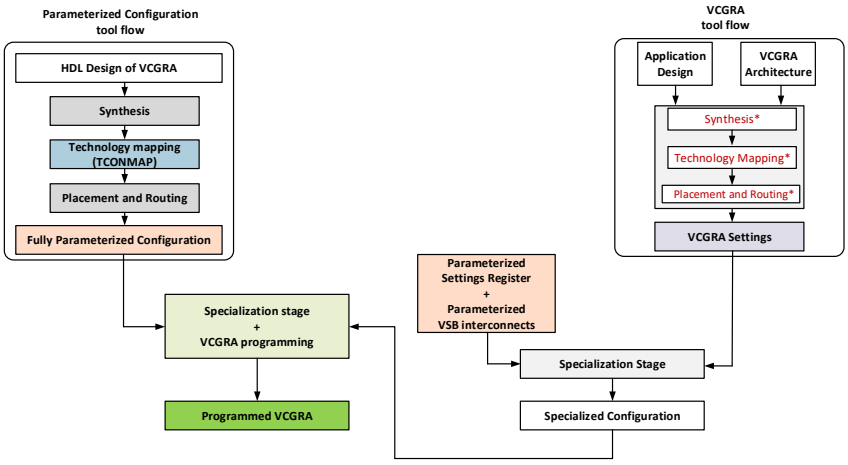


Figure 5.6: Implementation of applications on a fully parameterized VCGRA using the parameterized configuration tool flow

*Note: * indicates steps considering PEs (and VSBs) as a basic programmable component*

FPGA configuration, we have to micro-reconfigure the frames that hold the LUT for storing the contents of the settings register. Therefore, to change only the input of a PE, we micro-reconfigure the settings register only.

2. Micro-reconfigurations of the TLUTs/TCONs of each PE: a fully parameterized VCGRA contains PEs with TLUTs and TCONs (Figure 5.7) that needs to be micro-reconfigured with the VCGRA settings.To change the functionality (intra-connects) of a PE we need to micro-reconfigure TLUTs and TCONs of each PE.

To make it simple we combine the two levels of micro-reconfiguration and present the whole reconfiguration time in our results. However, I will present the difference between the two levels of micro-reconfiguration using a functional density curves in Section 5.4.4.

5.3.5 Limitation of parameterized VCGRAs

In a conventional VCGRA implementation, the settings registers are updated using a dedicated bus. However, in the parameterized (both partially and fully) VCGRA implementation, the settings registers of each PE and the routing switches are updated by reconfiguring each frame of the FPGA that contains setting bits of the VCGRA. This is usually accomplished by read-modify and write back frames of the FPGA (*micro-reconfiguration* [48]).

For a VCGRA application that contains dynamic Network-On-Chips or PEs that require cycle-by-cycle context switching, we cannot afford the cost of reconfiguration time so often and therefore, such applications may not be suitable to be handled by a parameterized VCGRA.

5.3.6 Advantages of parameterized VCGRAs

In the case of much less frequent reconfiguration needs, the parameterized reconfiguration reduces the overhead of the conventional VCGRA as follows:

- The settings registers of the VCGRA are implemented on the TLUTs. Therefore, the settings data (which is a part of the truth table entries of the TLUTs) are mapped on the configuration memory and therefore, the need of a dedicated bus to update the settings register is avoided.
- The PEs of the VCGRA are optimized by symbolic constant propagation that is integrated within the parameterized configuration tool flow.
- Each VCGRA intra- and inter-connection is mapped onto lower level reconfigurable routing switches (TCONs). Therefore, we reduce the utilization of the LUTs for implementing the connection network.

Even with the limitations of VCGRAs we believe that a large number of VCGRA implementations [67] [82] [83] [84] in which cycle-by-cycle context switching is not needed, can benefit from the above advantages of the partially and fully parameterized VCGRA implementation. Since my contributions in this chapter are mainly on fully parameterized VCGRAs, in the following sections I will present the fully parameterized VCGRA architectures that are suitable to implement a high-performance image processing application (explained in Section 5.4.1).

5.4 Fully Parameterized MAC VCGRA grid

In the previous work on parameterized VCGRA implementations, authors of [80] parameterized the LUTs and inter-connects of the VCGRA. They were able to save 50% of the LUT resources. However, they did not parameterize the intra-connect of the VCGRA (connections within a single PE) since they had no automatic technology mapper to generate TCONs for the intra-connections. In order to overcome this limitation we use an automatic technology mapper called TCONMAP [81] that can generate TCONs and TLUTs simultaneously for a given parameterized application.

In Figure 5.7, a fully parameterized PE is depicted. Each PE in the VCGRA grid contains Tunable LookUp Tables (TLUTs) optimized to implement the required functions of the PE. The optimization is achieved by the method for optimization for constant parameters as a result of parameterization of the LUTs. In

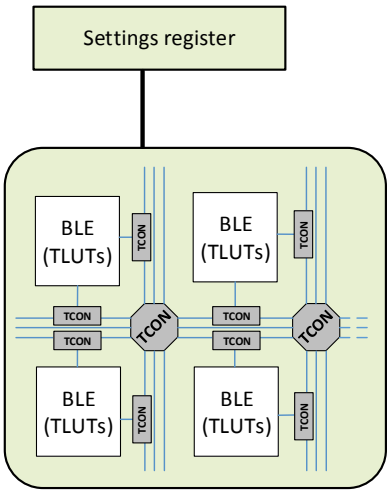


Figure 5.7: A fully parameterized Processing Element (PE) containing Tunable LUTs (TLUTs) and Tunable Connections (TCONs) within a single PE

this approach, instead of implementing the parameter inputs of the application as regular inputs, they are implemented as constants and the functions of each TLUT are specialized for these constants. For every change in parameter input values, the function of the TLUT is re-optimized for new constant values by reconfiguring the configuration of the TLUT. A group of TLUTs form a Basic Logic Element (BLE) of the PE.

The BLEs of the PE are connected using virtual routing switches (connection blocks and switch blocks) within the PE that form an intra-connect. The PE also contains a virtual routing network composed of wires that is responsible of carrying required signals between the BLEs. A virtual routing switch consists of connection multiplexers (Figure 5.8) with configuration memory. The routing switch connects the wires between BLEs within the routing network depending on the configuration values stored in the configuration memory of the switch and therefore providing an opportunity to parameterize the routing network.

TCONMAP replaces the virtual routing switches with the TCONs. A TCON consists of configuration memory that can be reconfigured depending on the parameter inputs. Therefore, a connection between two BLEs can be made or broken depending on the values of the parameter inputs of a VCGRA application. Further, with the help of TPLACE and TROUTE, these connections can be placed and routed on to the lower-level physical routing switches thereby reducing the PE intra-connect overhead on the physical LUTs of the FPGA. With the help of TCONs, a significant reduction in routing resource consumption (at least by 40%)

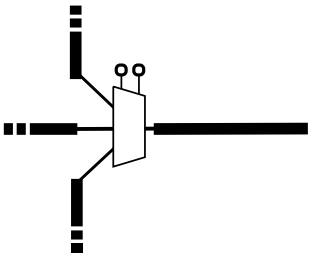


Figure 5.8: A connection multiplexer with configuration memory shown in circles

has been observed in the experiments of [85]. We aim at similar improvements by using the TCON concept on a VCGRA implementation of a retinal vessel segmentation application.

5.4.1 Retinal Vessel Segmentation Application

In this section, we present an HPC application that is used to investigate the benefits of the fully parameterized VCGRA approach. We have designed the PEs of the VCGRA based on the HPC application. Only those parts of the application that will be implemented on the reconfigurable logic (hardware modules) for the performance acceleration will be used for the VCGRA implementation.

In computer vision, segmentation refers to the process of partitioning a digital image in multiple segments in order to extract prominent features and locate objects and/or boundaries. The particular application of interest - retinal vessel segmentation - refers to the extraction of the vessel structure from the background in fundus images. Vessel segmentation enables the extraction of morphological attributes of retinal blood vessels, such as length, width and branching pattern, that assist the diagnosis, screening, treatment and evaluation of various cardiovascular and ophthalmologic diseases such as diabetes, hypertension, arteriosclerosis and choroidal neovascularization.

The Retinal Vessel Segmentation application that we have implemented is based on the concept of matched filters [86] and is presented in Figure 5.9. From an initial 2D input retinal image (RGB image), the green channel is retained as it contains most of the information. A preprocessing step is then applied in order to provide a more suitable and clear image for the main filtering operations. The preprocessing involves histogram equalization, optic disc removal and outer region removal.

The resulting image goes through a denoising function by means of a Gaussian filter to reduce the effect of high frequency noise (applying two sets of coefficients of size 5×5 and 9×9 respectively). The main vessel detection function that follows, involves filtering and thresholding the denoised image. Since the cross-section

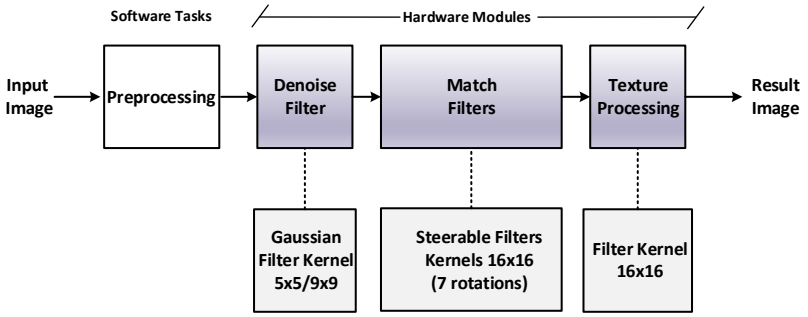


Figure 5.9: High level presentation of the processing steps for the retinal vessel segmentation application

of a vessel can be modeled as a Gaussian function, a series of Gaussian-shaped filters can be used to "match" the vessels for detection. Steerable filters are used (in the current implementation, seven directions are considered) to separate the pixels with the strongest responses (7 different sets of 16×16 coefficients). The problem with this approach is that not only vessels but also non-vessel edges can be identified in the response image. To minimize this effect, a third processing step that involves texture filtering is applied so as to retain in the final image only lines of certain thickness and above. The vessels filtering is also applied in the form of a modified filter applied many times in the image with different sizes depending on the desired filtering effect (e.g 5×5, 9×9, 16×16). In general the number of filters applied in the pipeline is a tunable parameter which depends on the quality of the images and/or the imaging technology that is used.

Figure 5.9 presents an overview of the application. It should be noted that the preprocessing steps are implemented in software, while all filtering operations are implemented as hardware modules. All hardware modules employ the same interfaces and are virtually the same in principle: they all share the same core architecture and what changes is size and coefficients of the filter kernels. The orientation of the filter is defined from the coefficients itself.

5.4.2 VCGRA for the HPC application

The filters (hardware modules) of the HPC application described above need to be accelerated by realizing the filter actions on the reconfigurable logic. We use our fully parameterized VCGRA approach to implement the filters on an FPGA. We used a floating point Multiply-Accumulate (MAC) operator as a processing element (Figure 5.10). We have used the "FloPoCo" [87] floating point library to build the floating point addition and multiplication and thus, we use the FloPoCo

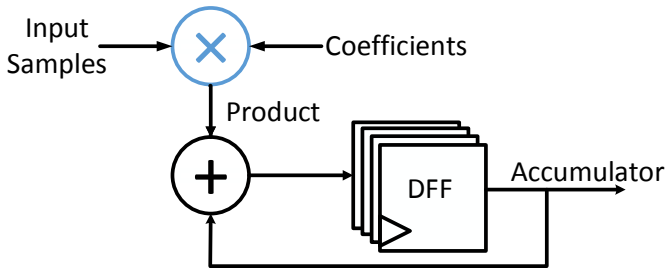


Figure 5.10: Floating Point MAC Operator for Filter Applications

floating point format with a 6-bit exponent and a 26-bit mantissa. We have not used any dedicated multipliers or adders while generating the floating point operators using the "FloPoCo" library.

In the MAC operation, the image samples are multiplied with the filter coefficients. Later, they are added to the previously accumulated values after the multiplication. The coefficients of the filter determine the filter configurations such as the noise level of the denoise filter of the vessel segmentation application.

The floating point multiplication is parameterized with the coefficient as a parameter input. The value of the coefficient input changes infrequently. For each infrequent change in the coefficient value, a specialized bitstream is generated and the multiplication is reconfigured accordingly. The settings register for each MAC operator (PE) holds an integer value for the counter that decides the number of iterations the MAC operation should perform with a fixed coefficient value. Therefore, in order to change the filter coefficients and counter values, each PE (MAC operator) needs to be reconfigured.

5.4.3 Results on MAC grid

The Processing Element (PE) of the filter application (MAC operator) was described using VHDL with annotated parameter inputs ("`-PARAM`"). The annotation helps to differentiate between the regular inputs and the parameterized inputs. With the help of Quartus II (v10.0), the PE was synthesized and later subjected to logic optimization by using the ABC tool [88]. We used the TCONMAP mapper [81] to generate TLUTs and TCONs.

The LUT resource utilization of a single PE is tabulated in Table 5.1. The values are compared with the conventional VCGRA. Clearly, the total number of 4-input LUTs utilized by the PE with our VCGRA approach shows a significant reduction by $\approx 30\%$. We also observe a difference in the logic depth level by 3 and hence it contributes to the improvement in the performance of the PE.

All the TCONs (568) can be implemented on the physical switch blocks and

Table 5.1: Resource utilization and P&R results of a PE

VCGRA	LUTs (of which TLUTs)	TCONs	Logic Depth level	WL	mCW
Conventional	2522(0)	0	36	27242	10
Fully Parameterized	1802(526)	568	33	16824	10

Note : For the fully parameterized implementation the 526 TLUTs are included in the 1802 LUTs.

connection blocks, instead of LUTs, thus saving a significant amount of LUT resources of the FPGA. In the conventional VCGRA implementation, the connections made by these TCONs are realized on the LUTs which is an overhead of ≈31% of the total LUTs of the parameterized VCGRA.

The synthesized PE was subjected to the TPaR Place and Route (P&R) tool [85]. We used *4LUT_sanitized* FPGA architecture from VPR [89] to perform the P&R. The results of P&R for a single PE (MAC operator) are tabulated in Table 5.1. Clearly, the proposed method (fully parameterized VCGRA) has the total wire length (WL) decreased by ≈ 31% as compared to the conventional VCGRA implementation, thus saving a significant amount of routing wire resources of the FPGA.

The experiments presented in [80] [85] show an increase in the minimum channel width (mCW) when using TCONs. However, our results show that the minimum routing channel width of both implementations are the same. We observe no overhead on the minimum channel width of the FPGA after using TCONs for the inter- and intra-connections of the VCGRA.

A fully parameterized 4×4 VCGRA grid

The resource utilized by the grid contains 16 PEs and 9 VSBs, each of them has a settings register and therefore, the conventional VCGRA would consume twenty five 32-bit registers. In the conventional implementation, these registers are realized using the FPGA's logic-cell flip-flops. However, with the parameterized VCGRA tool flow we map them to the configuration memory of the FPGA and hence we reduce the flip-flops utilization to zero.

Also, in the conventional VCGRA implementation the routing switches (connection blocks + switch blocks) that are needed to realize a 4×4 VCGRA grid is 41 (9 VSBs and 32 Virtual Connection blocks) and again these would have to be realized on the LUTs of the FPGA. However, with our fully parameterized VCGRA implementation we can target physical routing resources and thereby reduce the functional resource utilization (LUTs) to zero.

With the use of parameterized VCGRA configurations, a significant reduction in FPGA resource utilization is observed. However, this gain does not come for free. There is a reconfiguration overhead consisting of reconfiguration time, Boolean function evaluation time and the PPC memory size [48]. The estimated reconfiguration time depending on the number of TLUTs and TCONs for one PE is 251 ms. The reconfiguration speed can be improved using the techniques described in [49].

In the vessel segmentation application, the coefficients of the Gaussian filter and the texture processing filter can be configured by the user but such change is infrequent. Therefore, the reconfiguration time cost for these two filters is minimal. For example, 1000 images (of the same size) can be denoised and texture processed at the reconfiguration time cost of 251 ms per PE per 1000 images. Therefore, the two filters benefit from the parameterized reconfiguration technique along with the advantage of improved programmability in the VCGRA tool flow.

However, the matched filter needs to be reconfigured with 7 different sets of coefficients (which is not an infrequent change in parameter values) for a single image and hence the filter is reconfigured 7 times and incurs the expensive reconfiguration time cost of 1.8 s per PE per image. The number of parameter changes cannot at all be called infrequent, hence the matched filter does not benefit from parameterized reconfiguration. However, it can still benefit from fast compilation from the VCGRA tool flow since the mapping of filter on to the CGRAs requires shorter compilation time compared to the standard FPGA implementations.

One way to overcome this limitation is to use 7 copies of the matched filter configured with the required 7 different sets of coefficients and pipeline them. This approach increases resource utilization cost by 7× the single matched filter and hence there exists a trade-off between reconfiguration time cost and resource utilization.

The reconfiguration time can be reduced by a factor up to 38 using the techniques explained in Chapter 4.

5.4.4 Functional density curves of parameterized VCGRAs

In this subsection I present the functional density curves for partially and fully parameterized VCGRA of the MAC grid used to realize the HPC application. The functional density curve is plotted against the rate of change of the input parameters. We plot the functional density of the VCGRA implementations in three different forms:

1. Conventional VCGRA (VCGRA_Conv): A Virtual Coarse-Grained Reconfigurable Array implemented using classic way without using parameterized configuration technique.

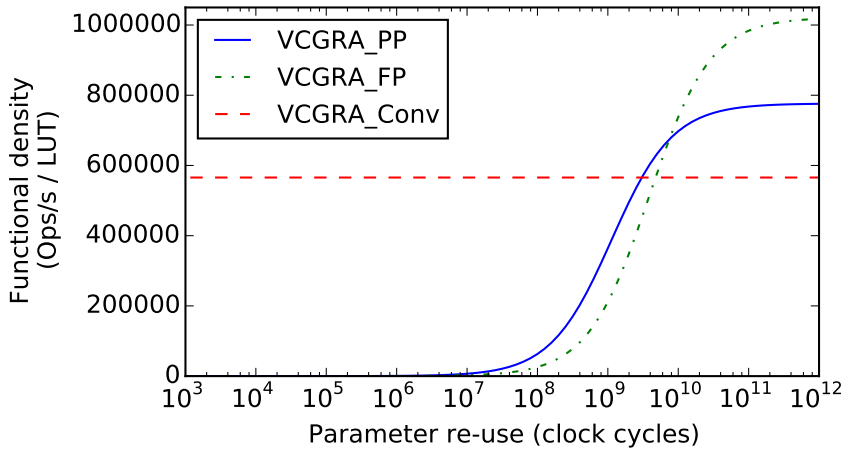


Figure 5.11: Functional density curves for a partially parameterized, fully parameterized and a conventional VCGRA implementation

2. Partially parameterized VCGRA (VCGRA_PP): In a partially parameterized VCGRA only settings register and inter-connect VSBs are parameterized and hence it requires only one step micro-reconfiguration.

3. Fully parameterized VCGRA (VCGRA_FP): In a fully parameterized VCGRA not only the settings register and inter-connect VSBs but the intra-connects of each PE are parameterized and hence it requires two steps micro-reconfiguration.

Figure 5.11 shows the functional density curves for a partially parameterized, fully parameterized and a conventional VCGRA implementation. In the conventional VCGRA implementation there is no reconfiguration involved and hence the design utilizes a constant resources. This kind of implementation is good if the settings of a PE has to be changed frequently. The fully parameterized VCGRA enables the designer to reconfigure both the settings register and the TLUTs/TCONs within each PE. Therefore, the reconfiguration time overhead is higher than the partially parameterized VCGRA (where only settings registers are reconfigured). However, since the resource utilization is relatively low for the fully parameterized VCGRA, the magnitude of the functional density is higher than the partially parameterized VCGRA.

5.5 The heterogeneous VCGRA grid: Pixie

In this section, I propose a general purpose heterogeneous (fully parameterized) VCGRA grid suitable to implement digital image processing filters and other math operations. The PEs of the proposed VCGRA grid are identical to each other. However, the PEs can be configured to perform different arithmetic operations such as Add, Sub, Mul, Div, etc. and hence they emulate heterogeneity in their functions.

In [80] a specific VCGRA for regular expression matching is introduced, while in the previous section I described a floating point MAC operator, which is specialized for image processing tasks. I describe a more generic VCGRA, which includes basic processing elements and flexible virtual Channels (VC). In contrast to the MAC-version described earlier, users are not restricted to MAC-operations and different application data-flow graphs can easily be mapped onto the VCGRA. Thus, it is not necessary to massively modify the application for acceleration. Having a more general VCGRA offers another level of abstraction where the user can evaluate the suitability of a VCGRA for different kinds of applications. In addition, as all processing elements are working in parallel, the processing of the data can be pipelined as well as the proposed grid can be optimized for a specific application class, which results in higher throughput. Our design is currently focused on task graph representations of data-flow-oriented applications. However, we plan to extend the functionality of our hardware to also support designs which contain more control-flow oriented code. For this reason the extension of the graph representation with control flow operations is planned.

The methodology is programming language independent, because currently the tool chain's input is the data-flow graph of an application. Nodes of a graph represent the processing element functions, while edges show the dependencies and the data flow between the processing elements. Currently, the implementation supports arithmetic operations (addition, subtraction, multiplication, division) as well as comparison (greater than, equal to). In addition, a PE also has modes for buffering a value and support for an idle state. Buffering is necessary for resolving data dependencies between node inputs from different levels. Other operations modulo and multiply and accumulate (MAC) are also supported.

The grid of processing elements is organized in levels, whereby two levels are divided by one intermediate virtual channel (VC). This kind of design was chosen to enable the use of pipelining in the architecture. Every level of processing elements works as a pipeline stage. A specific kind of a VC is used as memory interface. It connects the grid to a microprocessor, which controls the VCGRA execution using any desired communication interface. The distribution of incoming data to the first row of processing elements is controlled by an external configuration signal. A synchronous start signal enables the execution in the first level

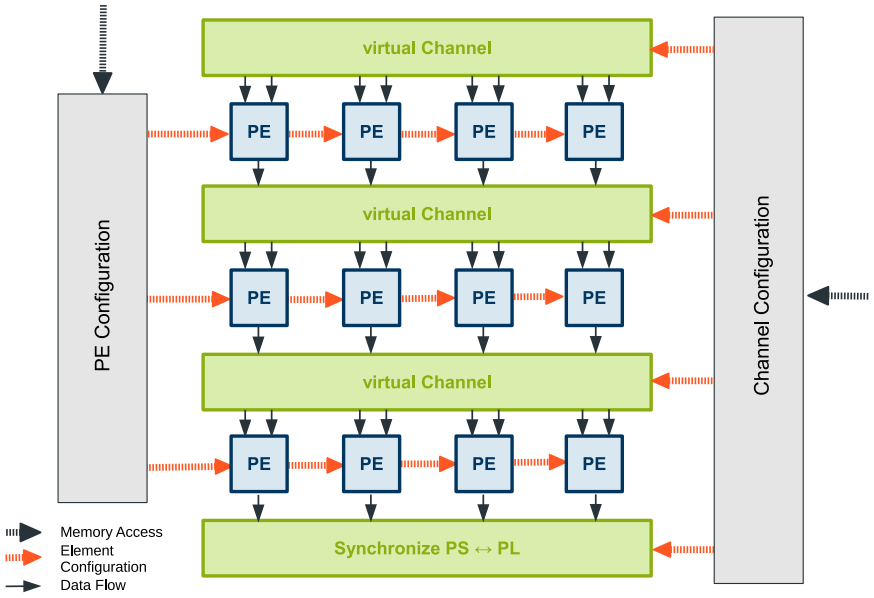


Figure 5.12: An overview of a design for a parameterized VCGRA

of processing elements, when all incoming data dependencies are fulfilled. The start signal is to be controlled from outside the VCGRA, the other levels of PEs are synchronized with their predecessors within the array. No additional control from a microcontroller is necessary. When the input data has been processed by the VCGRA, the processing system is notified to fetch the output data. The operation of the processing elements as well as the routing within the VC is realized by reconfiguration using the TLUT/TCON tool flow. The example in Figure 5.12 also shows opportunities for acceleration. If the grid is big enough, multiple instances of the same graph can be implemented.

We created a tool that eases the task of designing VCGRAs with different shapes. In addition to the rectangular style, where every row contains the same number of PEs we support an arbitrary number of inputs and outputs at a VC which leads to application specific grid designs if necessary. Automatic generation of these grids for a specific application class is currently work in progress. The functionality of the processing elements is extendable. For instance, we also experimented with PEs enabling floating point operations for addition and multiplication. The currently used PE structure has been optimized when compared to the MAC-operator described in the previous section.

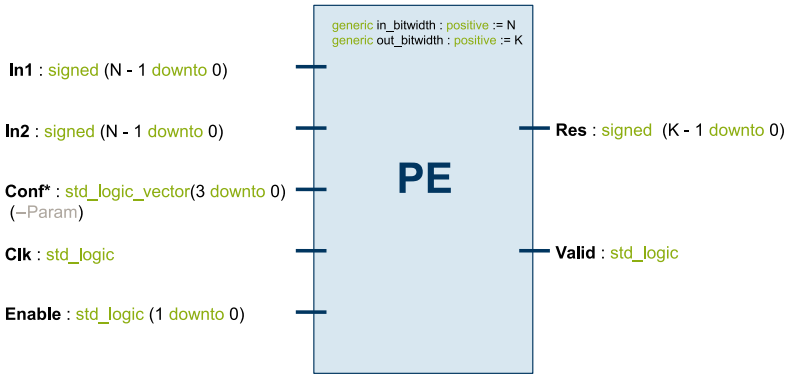


Figure 5.13: Schematic of a Processing Element

5.5.1 A fully Parameterized Processing Element (PE)

The PE (Figure 5.13) is designed as a finite state machine containing three states: *AWAIT_DATA*. *PROCESS_DATA*, *VALID_DATA*. Normally, it performs an operation on it's two inputs, which is set by a parameterized configuration input. The result is saved in an output buffer and is set to be valid for one cycle. To synchronize the inputs of a PE the two inputs have to be enabled. This is done by using the valid signal from a previous PE in an upper level. However, incoming values are buffered in every clock cycle. The current grid with its PEs is designed to support pipelined data flow applications. Therefore, no temporary results are saved within a processing element. We experiment with a MAC operation and design an element with a buffer to save the accumulated result, but we do not support graph mapping for that operation yet. The data bitwidths of input and output are configurable. However, the bitwidths of the two input values of a PE have to be equal. As shown in Figure 5.14, the adjustment of the bitwidth is done within the virtual channels.

If a PE is used as a buffer, the previous VC links the same data to both inputs of a PE. Thus, both inputs are enabled by the same valid signal and the data is copied to the output of the PE. If a processing element is unused in a configuration it is configured with *NONE*. A VC can bind arbitrary data to a PE's inputs. With the *NONE* configuration, the PE does not generate any output or change the valid signal to synchronize a successor.

The intra-connects of the PE are also parameterized and therefore, the reconfigurable connections within the PE are also mapped on the tunable connections (TCONs) using the TCONMAP mapper [81].

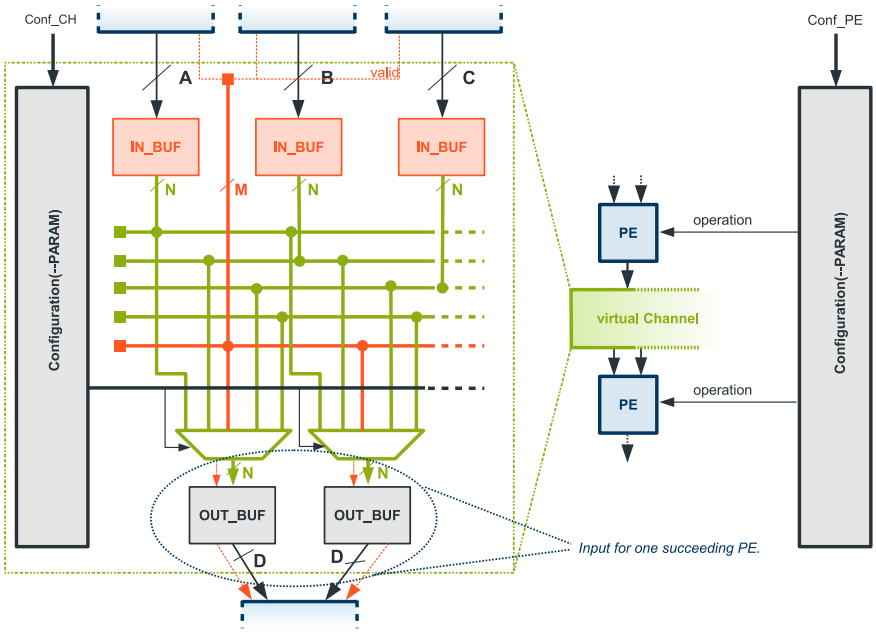


Figure 5.14: Schematic of a Virtual Channel

5.5.2 Fully Parameterized Virtual Channel (VC)

The architecture of a VC is shown in Figure 5.14. The implementation currently needs a lot of routing resources (specifically connection multiplexers). However, as the design is specially suited to be implemented using the TLUT/TCON tool flow the huge amount of multiplexers and connections which are dependent on a parameterized input are expected to need a significantly reduced amount of resources in the implementation, compared to an implementation using vendor tools.

One multiplexer per output is used to connect one specified input with the configured output. The select-input line of a multiplexer handles the specialization and is set as a parameter for the TLUT/TCON tool chain. This allows the TCON tool flow to distinguish which connections are used mutually exclusive in time. As a result, these routing resources can be shared within the FPGA.

All inputs of the predecessors of a channel are buffered at the input of the channel. The valid signals of all previous PEs are collected. Every input of a succeeding PE has a multiplexer with as many inputs as predecessors of the channel and gets a signal vector of all validating signals. Depending on the configuration, the output multiplexer routes the data value and the corresponding validating signal to an output buffer of the channel. A channel input can be routed to several channel outputs. As symbolized with the different letters at the connections, the

channel supports different bitwidths for data paths. The internal bitwidth is set to the biggest data input, which can occur within a configuration,

$$N = \max\{A, B, C, D, \cdots\} \tag{5.1}$$

while the bitwidth of the validation signal vector depends directly on the number of predecessors.

$$M = \#\text{predecessors} \tag{5.2}$$

The bitwidth of the internal channel connections is known a priori during the analysis of the task graph and is currently not changeable. Moreover, the size of a multiplexer and its bitwidth (bw) of a configuration word depend on the number of inputs or predecessors and are also fixed during system generation.

$$\text{bw} = \lceil \log_2 \{\#\text{predecessors}\} \rceil \tag{5.3}$$

Nevertheless, the usage of the TCON tool flow shows promising results, which are described in more detail in the results Section 5.5.5.

5.5.3 Building a VCGRA

The PE and VC are the basic elements of a VCGRA that provide flexibility regarding their functionality and data bitwidths. Describing the whole VCGRA grid in VHDL is a time consuming task. Therefore we developed a tool that automatically creates the VHDL top-level description of a VCGRA from a description of the hardware structure. The only inputs needed are the number of input elements from memory and the structure of the grid. The grid's structure is described by the number of processing elements in each level of the architecture and the elements' input and output bitwidths. All other parameters (e.g. for the channels) are automatically derived from the mentioned input data. The tool's output is VHDL code defining the hardware structure of the grid.

5.5.4 Edge Detection

For demonstration purposes we implemented the Sobel edge detection kernel on the proposed VCGRA. The Sobel filter is used for edge detection. An algorithm for the Sobel edge detection filter is shown in Algorithm 1.

The setpoint of the Sobel kernel is set to the midpoint of the mask. Every pixel of an image is convolved with the kernel. The result of the convolution is saved at the current position of the filter mask's setpoint in the image. A task graph representation of the algorithm is shown in Figure 5.15. It shows the kernel code of the innermost loop. Blue nodes are pixel values, which lay underneath the kernel mask; red nodes are the corresponding filter coefficients. A gray node symbolizes an operation and is mapped to a corresponding PE. The edges are managed by

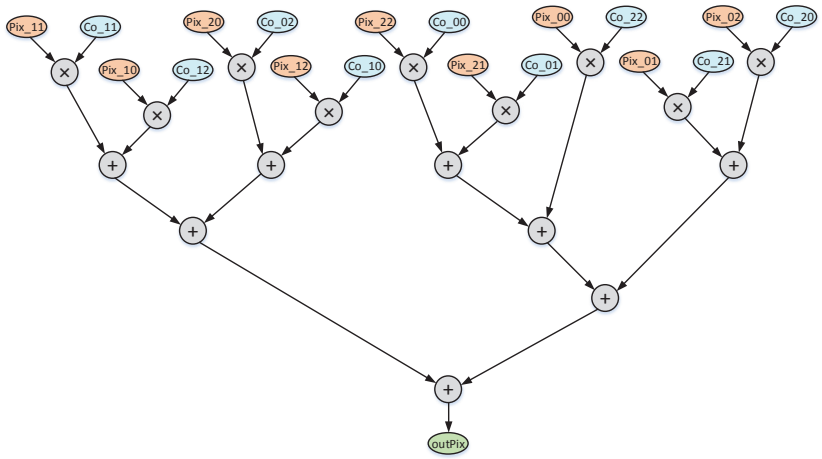


Figure 5.15: Task graph representation of a 3 × 3 filter mask

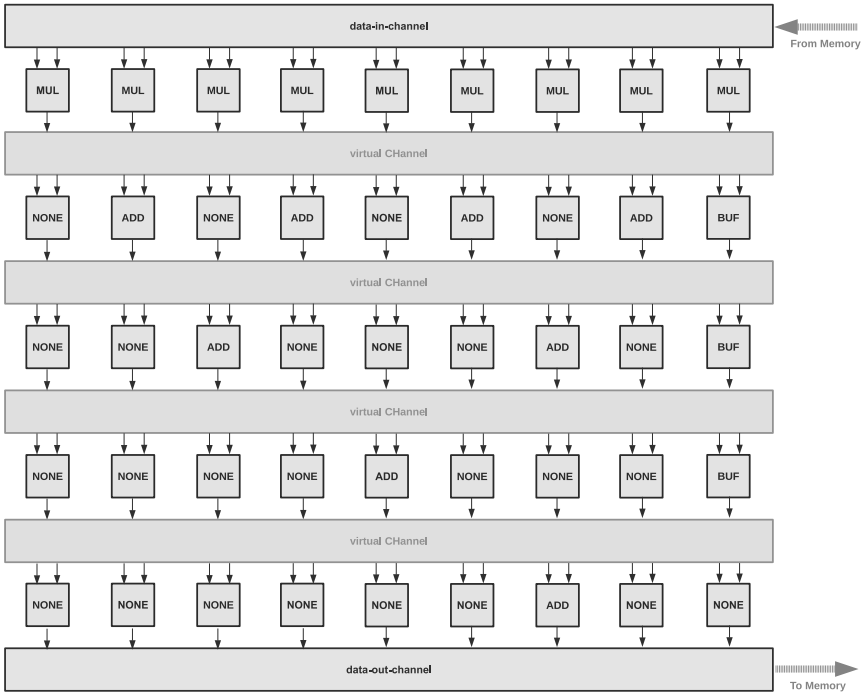


Figure 5.16: VCGRA grid for the Sober edge detection filter

Algorithm 1 Edge Detection

```
1: procedure SOBEL(image)                                  ▷ grayscale image
2:     center ← 0, 0                                       ▷ setpoint of kernel
3:     for all pixel in image do
4:         pos ← pixel_coordinates                         ▷ pixel position in image
5:         sum ← 0
6:         for j ← −1, 1 do
7:             for i ← −1, 1 do
8:                 temp ← sobel[center + j][center + i]
9:                 ×pixel[pos − j][pos − i]
10:                sum ← sum + temp
11:            end for
12:        end for
13:        image[pos] ← sum
14:    end for
15: end procedure
```

the configuration of the virtual channels. At least, the green node symbolizes a result of the convolution for a single pixel value. We used a small image processing kernel for demonstration, because a task graph becomes very large for bigger masks. However, we are also able to implement bigger kernels on a VCGRA. The weighted pixel value of the multiplication on the right border of the array is buffered in every stage of the array until it is used in the last addition. The size of an array is arbitrary. For demonstration we choose an array which is as big as needed to implement all levels of the task graph. However, it is also possible to choose bigger or smaller arrays. For bigger arrays with more stages than necessary, an output value has to be buffered in every stage until it reaches the data output channel at the bottom. Bypassing of levels of the array is not supported.

The VCGRA grid for the Sobel edge detection application is depicted in Figure 5.16. The grid consists of 4 VCs and 45 PEs. For the simple implementation on a hypothetical FPGA, we have considered to design the rectangular VCGRA grid and hence we observe that the majority of the PEs are configured with the *NONE* operation. However, this could be optimized by designing an inverted triangular grid.

5.5.5 Results on Pixie

The VCGRA components described in the previous section were synthesized and were subjected to the TPaR Place and Route (P&R) tool [85]. The P&R was performed using the *4LUT_sanitized* FPGA architecture from VPR [89]. The results of P&R for the VCGRA and its components are explained in the following subsections.

Table 5.2: Resource utilization and P&R results

	LUTs (of which TLUTs)	TCONs	Logic Depth level	WL	mCW
VC Conventional	176(0)	0	2	3186	7
VC Fully Parameterized	32(0)	72	1	782	4
PE Conventional	408(0)	0	47	3832	8
PE Fully Parameterized	387(32)	22	47	3769	8
PE_FP Conventional	2191(0)	0	47	23388	10
PE_FP Fully Parameterized	1668(584)	798	47	17676	10
Grid Conventional	17066(0)	0	155	176200	14
Grid Fully Parameterized	16099(976)	561	153	169560	12

5.5.5.1 Virtual Channel (VC)

The Virtual Channel is described in VHDL. The channel has parameterized connection multiplexers whose select lines are the parameter inputs. With the help of the TCONMAP mapper we were able to map the VC on the TCONs. Therefore, the major part of the VC does not need LUTs to make it reconfigurable.

The P&R results of the VC implementation are tabulated in Table 5.2. From the top two lines of this table we observe that 82% of the logic is mapped on the reconfigurable physical switches (TCONs) instead of the physical LUTs and multiplexers (as in the conventional implementation). We also observe a significant decrease of 76% in wire length (WL) between conventional and parameterized implementation due to the fact that the minimum channel width (mCW) is reduced by 42%. This optimization can be achieved at the cost of a reconfiguration time of 4.6 ms (not shown in the table).

5.5.5.2 Processing Element (PE)

We have designed a Processing Element that comes with two different versions: a fixed point PE and a floating point PE. The P&R results of the fixed point and floating point PE are tabulated in Table 5.2.

The logic resources (LUTs) used by the fixed point PE are optimized by 5% and we also observe a difference in wire length by 2%. This optimization can be achieved by investing a reconfiguration time costs of 3.4 ms. The PE contains 13%

of its resources (TLUTs + TCONs) that are responsible for the reconfigurability of the processing element.

The floating point PE was built using an open source floating point library called "FloPoCo" [87]. We used the FloPoCo[4] floating point format with a 6-bit exponent and a 26-bit mantissa. We have not used any dedicated multipliers or adders while generating the floating point operators using the "FloPoCo" library. The floating point PE implementation was optimized by 24% and a decrease in wire length by 25% is also observed. This optimization can be achieved at the cost of a reconfiguration time of 88.5 ms. There is no difference in logic depth level and minimum channel width in both types of PEs. The PE contains 82% of its resources (TLUTs +TCONs) that are responsible for the reconfigurable part of the processing element. The proposed floating point PE consumes 13% less resources compared to a MAC operator presented in Section 5.4.3.

5.5.5.3 A fully parameterized 4×4 heterogeneous VCGRA grid

A fully parameterized 4×4 VCGRA grid was implemented using fixed point PEs and VCs. The P&R results are tabulated in Table 5.2. The logic resources of the whole grid are optimized by 6% and the wire length is reduced by 4% due to a reduction in logic depth level by 2 and in minimum channel width by 2 as well. This optimization can be achieved at the cost of a reconfiguration time of 98.5 ms.

5.5.5.4 Sobel filter

To implement the Sobel filter we need 45 PEs and 4 VCs. To reconfigure all the processing elements and virtual channels it costs 156 ms and 18.4 ms of reconfiguration time respectively.

5.5.5.5 Compilation time

The time taken to map the Sobel edge detection application is less than one second. The time taken to compile the hardware description of the VCGRA grid into bitstreams is approx. 1200 seconds. In the conventional implementation for every new image processing application, the development time would cost more than 1200 seconds. However, with the VCGRA approach, the total time to set up a new image processing application is very minimal since it costs only the mapping time and the total reconfiguration time.

In conclusion the proposed grid can be used as overlay architecture on a low cost FPGA platform that does not consist of hard coded primitives such as DSP blocks. We built the Sobel edge detection filter and the results show a promis-

[4] Another open source floating point library variant VFloat [90] [91] can be used.

ing improvement in the compilation times and thus bridging the gap between the application and the FPGA fabric.

In the next chapter, I introduce a custom FPGA configuration memory that overcomes the dependency on the ICAP port for reconfiguration and helps in drastic improvement of the reconfiguration speed resulting in ultra-fast micro-reconfiguration. For obvious reasons, the proposed custom configuration memory results are based on simulations only.

6

Custom FPGA configuration memory architecture for ultra-fast reconfiguration

This chapter presents custom FPGA configuration memory architecture to drastically improve reconfiguration speed. The chapters 2 and 3 presents the importance of run-time reconfiguration technique in FPGAs. It offers design flexibility under low-cost silicon area and power budgets, at the cost of reconfiguration overhead. The reconfiguration time overhead produced by the conventional configuration ports (such as ICAP) is too high for the reconfiguration technology to be embraced as a standard. Furthermore, the current FPGA configuration memory architecture restricts the access of configuration data to the frame level; this significantly delays the reconfiguration process. The work presented in this chapter explores the design space of the configuration memory architecture that fits the design of large FPGA's and is suitable to accomplish needs for ultra-fast reconfiguration. Therefore, the proposed method could be a stepping stone for next generation FPGA configuration memory architectures. Our simulation results show a reconfiguration speed gain of a factor of at least 1000 for substantially big parameterized applications that comes with the cost of extra auxiliary hardware used on top of the column based FPGA architecture.

6.1 Auxiliary hardware for the custom FPGA architecture

The conventional run-time reconfiguration is too slow due to the sequential access of configuration data (frames) via the ICAP port. To overcome this problem a parallel reconfiguration memory architecture has to be considered which leads us to the design of a custom FPGA configuration memory. The parallel reconfiguration memory architecture helps to overcome the dependency on ICAP port by providing the required configuration data to each CLB columns independently. To establish such a structure we need auxiliary hardware on top of the current column-based FPGA architecture.

The auxiliary hardware that we used to design a custom FPGA configuration memory architecture consists of a polymorphic register file (PRF).

6.1.1 Polymorphic Register File

A PRF is a novel register file organization (Figure 6.1) that is capable of dynamically creating a variable number of two-dimensional registers of arbitrary sizes [92]. A detailed architecture of a PRF memory is described in the following.

- **Address Generator Unit**: starting from upper left coordinates of the block being accessed and the access type (e.g., row), the Address Generation Unit (AGU) computes the individual coordinates of all PRF elements which are accessed.
- **The Intra-module Address Function**: the Addressing Function **A** computes the intra-module address using the individual coordinates of all the elements which are being accessed. The coordinates are computed by the AGU.
- **Module Assignment Function**: the Module Assignment Function (MAF) **M** computes the index of the corresponding memory module starting from the logical address which is being accessed. The index is then used by the crossbar to rearrange the data items.
- **Memory banks**: the memory banks store the actual data of the parallel memory. Each memory bank is assumed to be capable of producing one data item per clock cycle.
- **Crossbar**: the crossbar rearranges the inputs according to the select signals. The select signals are computed using the Module Assignment Function, and specify the position of each output of the crossbar.

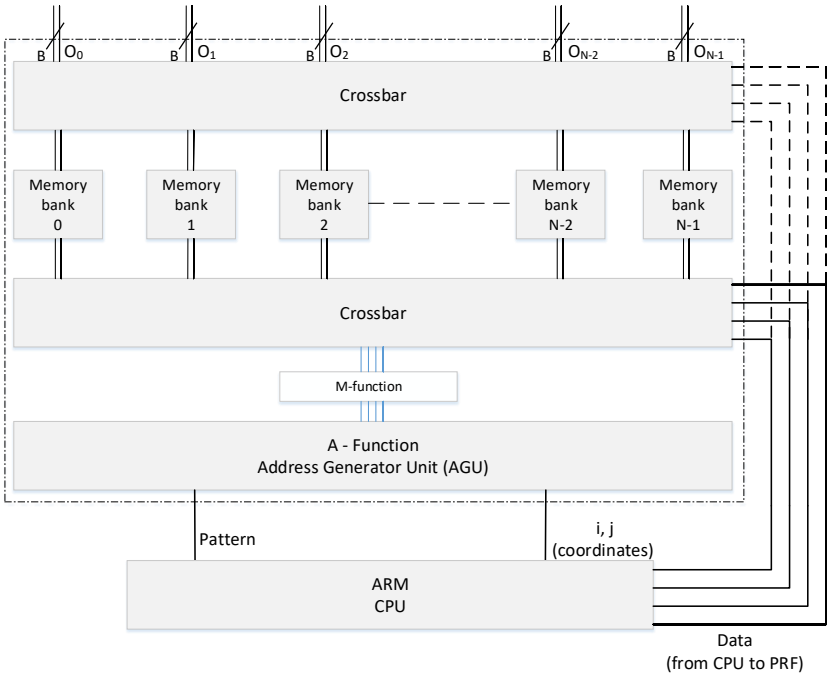


Figure 6.1: Polymorphic Register File Architecture

We adapt the proposed PRF model to our requirements by removing the "Address Generation Unit", "M-function unit" and a second stage crossbar of the PRF. This results in a simple parallel memory architecture that contains a crossbar and parallel memory banks (hence increases the memory bandwidth) and is connected to the embedded processor.

6.1.2 Network-on-Chip

The crossbar in the PRF is a bulk element that consumes most of the silicon area. To overcome this limitation one has to consider alternatives to the crossbar unit that is responsible for data distribution. The network-on-chip is the one of the alternatives suitable to integrate within the PRF.

The current trends in technology scaling have enabled the integration of hundreds of *Intellectual Property (IP) cores*[1] in a single chip. However, the conventional bus-based and ad-hoc interconnects cannot efficiently handle the heavy communication demands required by such complex systems. As a result, *on-chip*

[1] An IP core is a pre-designed module that fulfills certain task(s). It can be a Processing Element (PE), embedded memory block, custom logic, I/O device, etc.

interconnection networks have emerged as a promising solution to connect various micro-architecture IP cores.

A *Network-on-Chip (NoC)* is a shared and distributed interconnection network of programmable routers (switches) connected by links integrated onto a single chip. Using a NoC, the communication between the IP cores is realized by generating and forwarding packets through a network infrastructure. Thus, the bandwidth can be shared and used more efficiently as opposed to the other communication solutions [93] [94] [95].

Since a general System-on-Chip (SoC) platform is to be used for many different applications, the NoC should be able to support a wide range of bandwidth and Quality-of-Service (QoS) requirements [96]. One way to provide such flexibility is to design a large generic NoC with an over-engineered technology [97]. However, NoCs devised on static configurations will rapidly become so complex that designing them will be prohibitively inefficient. As a result, the design methodology of large-scale NoCs need to be fundamentally extended in order to develop systems which are able to continually adapt to changes and tune themselves based on the underlying dynamic environment. This can be best pursued by integrating reconfiguration techniques to NoCs in order to automatically orchestrate the network activities so that the performance can be effectively maintained. In fact, the next generation of NoCs are envisioned to be dynamically (run-time) reconfigurable. The reconfigurable NoCs represent a new sets of benefits in terms of area overhead, performance, power consumption, fault tolerance, and QoS compared to the previous generation [97]. NoC-based reconfiguration is most often of coarser granularity compared with the FPGAs. More precisely, the reconfigurable resources in NoCs are the routers and communication links rather than wires [98]. Although the reconfiguration can be performed at different levels, the incorporation of reconfiguration into a NoC design mainly revolves around two intertwined concepts: the routing method, and the network architecture. Finding techniques to employ the routing configuration without deadlocks is a challenging task.

The design of a NoC begins with the specification of performance requirements and cost constraints. These criteria then drive the design choices, such as topology, flow control mechanism, and routing strategy for a particular network. Finally, the performance of the network needs to be tested and evaluated through simulations in order to guide the initial decisions for the network design aspects [93]. In the following sub sections, a brief introduction on the network simulator and its configuration parameters is presented followed by a brief discussion the evaluation criteria.

6.1.2.1 Network Simulator

BookSim 2.0 interconnection network simulator [99] [100] [93] is used to conduct the performance assessments in the current study. BookSim is a cycle-accurate

network simulator designed as a companion to [93]. BookSim models the network at the flit-level and supports multiple topologies and routing algorithms due to its modular design. Router pipeline delays and wire latencies for transmitting the packets are also modeled in BookSim. It is written in C++ and is freely available at [100].

BookSim runs a simulation in three phases: *warm-up*, *measurement*, and *drain* in order to measure the steady-state of the network. First, the simulator is warmed up for N_1 cycles to bring the network to equilibrium. During the warm-up phase, packets are not timed or counted because the network is not stabilized yet. Once warm-up is complete, BookSim runs N_2 measurement cycles during which packets entering the source queue are tagged with their start times. Finally, the drain phase should be run long enough for all of the measurement packets to reach their destination.[2] It is important to mention that the latency measures are computed from the start and finish times of all measurement packets, either they have arrived at the destination during the measurement phase or during the drain phase. Although the packets generated during the warm-up and drain phases are ignored, they affect the measurement by providing the background traffic and interacting with the measurement packets [93].

6.1.2.2 Configuration Parameters

Modeling a NoC requires the specification of a large set of configuration parameters, such as topology, routing algorithm, network size, buffer size, packet size, number of Virtual Channels (VCs)[3] per physical channel, etc. Some of these parameters [101] are briefly explained as follows:

1. **Topology**: the first step in the design of NoCs is selecting a topology that optimizes the throughput, latency, and cost given the application demands and communication constraints. A network is composed of a set of shared routers and channels. The connection pattern of these routers and channels defines the topology of the network which is usually modeled by a graph. In fact, the topology of a NoC is analogous to a roadmap, such that the channels (like roads) carry the packets (like cars) from the source to the destination address [93]. The selection of the network topology has a significant impact on the overall performance, area, and power consumption [102]. This is due to the fact that the topology affects the number of *hops*[4], and thereby, the latency and energy consumption in the network. Moreover, the implementation complexity depends heavily on the topology.

[2] If a network is subject to starvation, the drain phase may never complete [93].

[3] VCs presented in this chapter are w.r.t NoC but not VCGRAs.

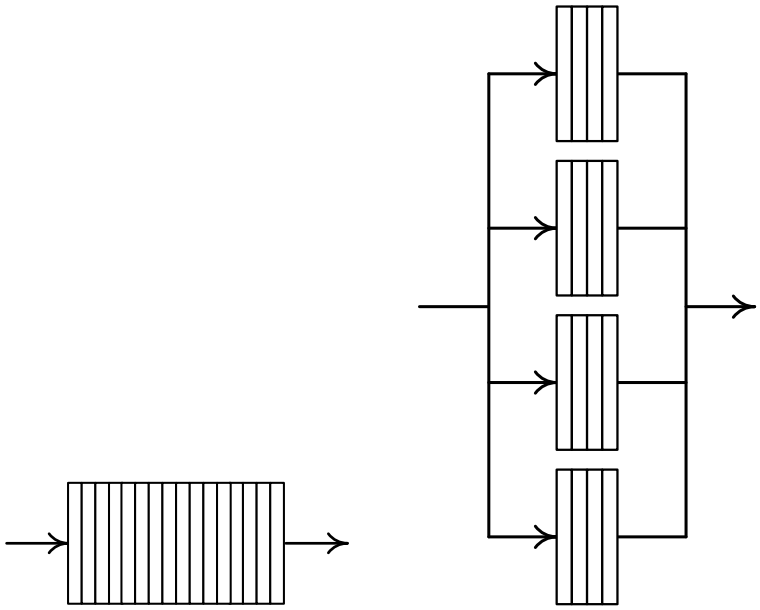
[4] The number of hops indicates the number of channels visited across the path from the source to the destination [93].

2. **Routing Strategy**: once a topology is chosen, there may exist several paths (sequences of nodes and channels) that a message can take through the network to reach its destination. The routing strategy determines which of the possible paths the message has to take. A good routing decision tends to minimize the length of the path while balancing the load placed on the shared resources of the network. The latency of the message is affected by the length of the path which is usually referred to as the number of hops [93].

3. **Packet Size**: in the *Wormhole switching* technique [93], a packet is decomposed into small units called *FLITs (FLow control digIT)* which are then routed consecutively through the network. As a result, a flit is the basic unit of bandwidth and storage allocation in the wormhole flow control mechanism. The position of a flit in a packet determines whether it is a *head flit*, *body flit*, or *tail flit*. A head flit is the first flit of the packet and carries the packet's routing information, namely the destination address. The head flit is followed by zero or more body flits containing the actual payload of the data, and ends with the tail flit. Unlike packets, body and tail flits have no routing or sequencing information and thus must follow the head flit along its route and remain in order. More precisely, once the head flit has been accepted by a channel, the remaining flits must be accepted before the flits of any other packet can be accepted. As a packet traverses a network, the head flit allocates the channel and buffers, and the tail flit deallocates them [93, 94].

4. **Virtual Channels (VCs)**: networks are composed of two types of resources: communication channels and buffers. Typically, a single buffer is associated with each channel. Buffers are commonly operated as FIFO queues, as shown in Fig. 6.2a. In wormhole-switched networks, when a head flit arrives, a buffer will be assigned to the incoming packet, and is reserved until the tail flit is transmitted [93]. This problem of idling channels due to resource coupling can be overcome by exploiting Virtual Channels (VCs). Note that incorporating additional physical wires is a very expensive and inefficient solution. A VC consists of a buffer that can hold one or more flits of a packet. Each group of VCs shares the bandwidth of a physical communication channel. However, each VC requires its own queue. Virtual Channels decouple the allocation of buffers from the allocation of channels by providing multiple buffers[5] for each channel in the network. In fact, each VC operates as if it was using a distinct physical channel. Thus, by splitting a single buffer storage with a 16-flit queue into four VCs with 4-flit queues as depicted in Fig. 6.2b, virtually four paths are provided for the packets to be routed [103]. The buffers in each lane can be allocated independently of

[5] Adding VCs to a network is analogous to adding lanes to a street [93].



(a) Without VCs *(b) With VCs*

Figure 6.2: Organization of buffers in a network [93]

the buffers in any other lane. Hence, a blocked message holds only a single lane idle and can be passed using any of the remaining lanes [93] [104] [94].

6.1.2.3 Evaluation Criteria

There are different metrics to analyze the performance of a particular NoC. In the absence of faults, the most important evaluation criteria is the *latency* [94] [93]. Moreover, NoCs must operate within tight *power* and *area* budgets. On the other hand, supporting high-performance on-chip communication can be realized by exploiting power- and area-hungry network resources (such as wide interconnects and their associated wide FIFOs, switches equipped with adaptive routing techniques, etc.). As a result, NoC design is usually characterized by a power-area-performance trade-off [105] [106]. The three metrics that we use as a base for evaluation criteria is explained as follows:

1. **Latency**: is defined as the time[6] elapsed from the initiation of the message at the source node until the tail of the message is received at the destination node. If the study only considers the network hardware, the initiation of the message refers to the time when the message header is injected into the

[6] In BookSim, the simulator's clock cycle is the unit of measurement.

network at the source node. In our work, the time spent by the message in the waiting queue before injection at the source node is also included in the latency. This queuing time is usually negligible unless the network is close to its saturation point [94].

The *header latency*, T_H, is the time required for the header of the message to traverse the network. As shown in equation (6.1), the header latency is the sum of router delay, d_R, and wire delay, d_W, at each hop, multiplied by the hop count, H [107]:

$$T_H = (d_R + d_W)H \tag{6.1}$$

The *serialization latency*, T_S, is the time required for the message to cross the channel. Given the message length, L, and the channel width, W, the serialization latency can be expressed as [107]:

$$T_S = \frac{L}{W} \tag{6.2}$$

The *zero-load* assumption is that a packet never contends for network resources with other packets. Under this assumption, the zero-load latency, T_Z, is calculated as [93]:

$$T_Z = T_H + T_S \tag{6.3}$$

The zero-load latency gives a lower bound on the average latency of a message through the network by ignoring the latency caused by contention of the packets over shared resources [93]. However, measuring the latency through simulations equips us with the real latency that the packets experience throughout the network, including the contention latency.

2. **Power**: the Orion 3.0 [108] power library was integrated in BookSim to obtain the power consumption results. The power dissipation of the network (including the communication channels, input buffers, router control logic, and output control modules) were calculated using the ITRS 32nm technology provided by CACTI [109] with the supply voltage of 0.9 V. The leakage power is included for channels, buffers, and switches.

3. **Area**: the imposed area overhead of a router is of a great concern because of the leverage that it brings to the implementation cost and power dissipation. To assess the area overhead, the router architectures were modeled in Orion 3.0 using the technology parameters from the 32-nm 0.9 V ITRS-HP process provided by CACTI [109]. As can be seen in Table 6.1, the data width is fixed at 64 or 32 bits (flit size), and each input channel has a buffer size of 1 or 2 flits, respectively. Four primary components of area overhead including the input buffers, crossbar switch fabric and arbiter, routing unit, and the communication channels are accounted for in our area model.

6.2 Proposed FPGA Architecture

The proposed architecture can be constructed by using the auxiliary hardware discussed in Section 6.1. In our design we propose two possible solutions (architecture topologies): a crossbar-based parallel memory and a NoC-based parallel memory. However, our investigations revealed that the crossbar-based parallel memory turns out to be infeasible due to over utilization of hardware resources.

The parallel memory boosts the data transfer to the SRAM cells of the LUTs (configuration) during the reconfiguration. The proposed hardware can be integrated into the prevailing column-based FPGA configuration memory to form a custom FPGA architecture that can facilitate ultra-fast reconfiguration and thus reduces the reconfiguration time overhead.

The proposed custom architecture contains a bus called Configuration Access Bus (CAB). The bus contains configuration data (64 bit) lines, LUT address lines (M) and a single line that can decide on a read or write operation of the LUT SRAM cells. All the LUTs in the CLB column have access to the CAB. The LUT address lines are used to choose one LUT at a time for a given clock cycle during the reconfiguration. Hence all the LUTs in a CLB column are (re)configured serially.

6.2.1 Crossbar-based parallel memory

The crossbar-based parallel memory used for (re)configuring the FPGA is depicted in Figure 6.3 where M is the number of LUTs to be addressed, and N is the number of CLB columns. The Polymorphic Register File described in the previous section is adapted to facilitate the data access in parallel to all the CLB columns during (re)configuration.

The memory banks store the configuration data well before the reconfiguration process begins. The stale data in the SRAM cells of the LUTs are replaced with the new configuration data stored in the memory banks. The data consist of a LUT address, the read/write selection bit for scheduling followed by the configuration data bits. The embedded processor (ARM Cortex-A9) is responsible to schedule the configuration data into the memory banks in such a way that during every clock cycle of the reconfiguration process the configuration data belonging to the different CLB columns are made accessible to the different CABs. Therefore, multiple LUTs located in different CLB columns can be reconfigured in a single clock cycle.

In the adapted parallel memory, each CAB has access to a single memory bank only. The crossbar is used to multiplex the data present in the memory banks so that all CABs can access every memory bank. This ensures that if one memory bank is full, the processor can fill the configuration data into another memory

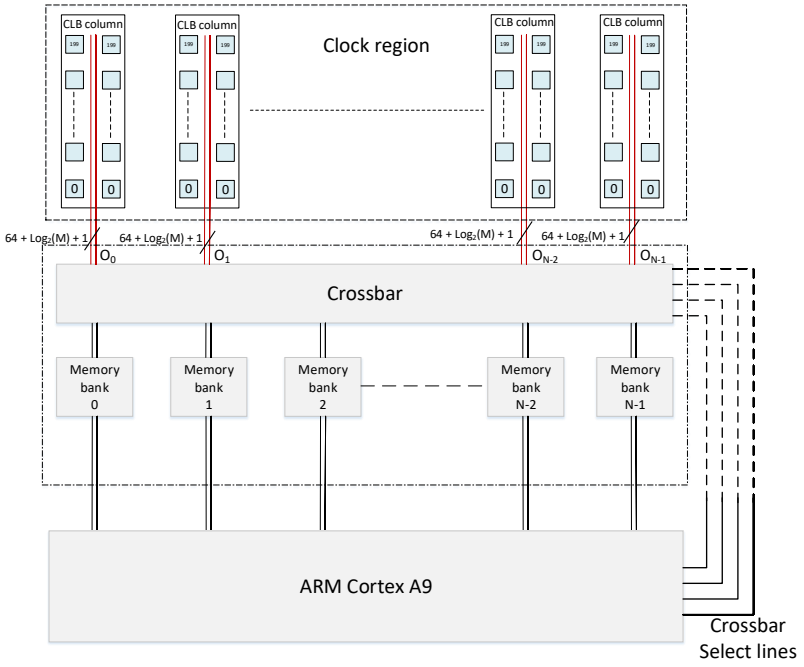


Figure 6.3: A crossbar-based parallel memory for custom FPGA configuration memory

bank and configures the select lines of the crossbar so that the memory bank can be accessed by the appropriate CAB.

With this architecture, we envision that the parallel memory with a crossbar is not a suitable solution for the following two reasons:

1. For an efficient and faster reconfiguration, it is necessary to maintain the number of memory banks proportional to the number of CLB columns (the lower the number of memory banks is, the more the data are congested in the parallel memory). Therefore, the crossbar-based parallel memory architecture becomes infeasible for larger numbers of CLB columns.

2. The processor has to program the crossbar (select lines of multi-stage multiplexers) for every new configuration data being pushed into the memory banks.

To overcome this issue we propose a NoC-based parallel memory.

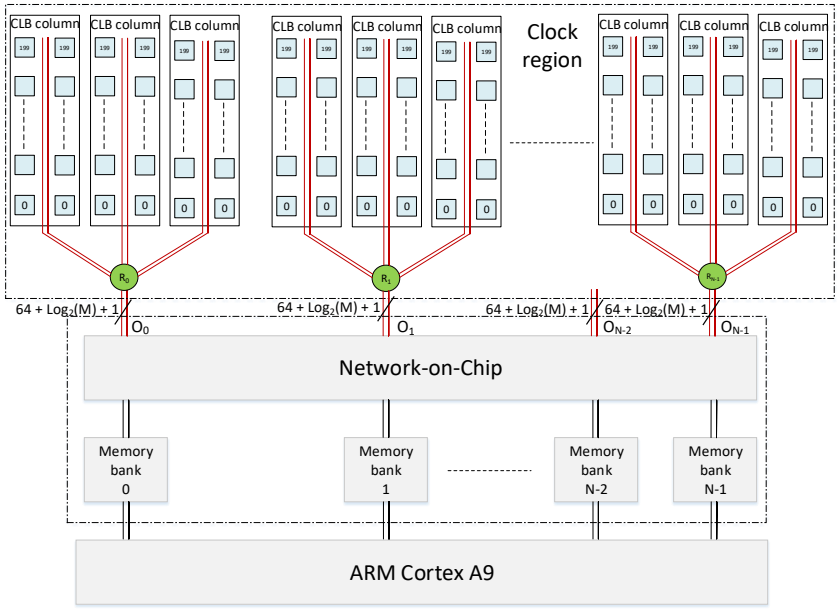


Figure 6.4: A NoC-based parallel memory for custom FPGA configuration memory

6.2.2 NoC-based parallel memory

In the proposed architecture, we replace the crossbar of the adapted parallel memory with a NoC. The NoC has routers connected to multiple CABs. As shown in Figure 6.4, for simplicity we connect 3 CABs to each NoC router thus reducing resources. The main reason for using a NoC instead of the crossbar is to decentralize the control of the configuration data after storing them in the memory banks. The decentralization helps in improving the data transfer speed as the configuration data can be transferred to the CABs in parallel.

The first step in the design of NoCs is selecting a topology that optimizes the throughput, latency, and cost given the application demands and constraints [93]. We have explored two different NoC topologies to implement the proposed architecture.

6.2.3 Butterfly NoC

A *butterfly* or k-ary n-fly is a well-known NoC topology consisting of k^n nodes and n stages of crossbar routers. There are two groups of routers in a butterfly: external and intermediate. The IP cores are connected to the external routers, while the intermediate routers just handle the packet switching and cannot be exploited

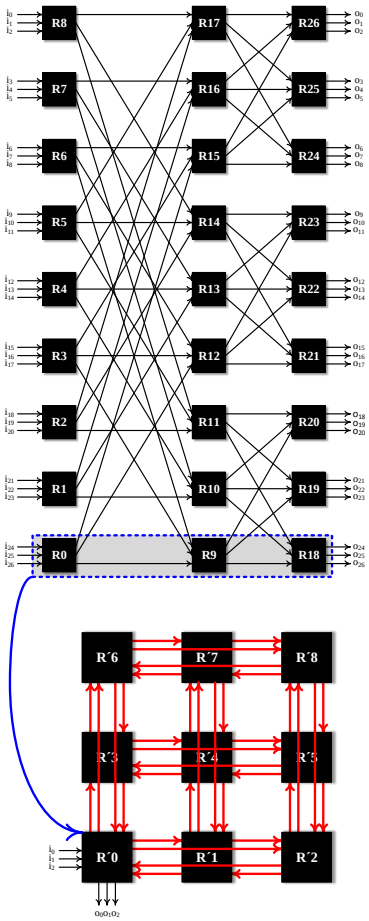


Figure 6.5: Block diagram of the 3-ary 3-fly (top), and the corresponding 3-ary 3-flat (bottom) NoC topology

as the source/destination [93].

In order to implement the proposed NoC-based parallel memory architecture, we have designed a 3-ary 3-fly network as shown in Figure 6.5. The NoC routers are responsible for transferring the configuration data to the appropriate CABs. As can be seen in Figure 6.4, each external router in the last stage is connected to 3 CLB columns (i.e. each output port to one CLB column). The configuration data is serially transferred to the CABs once it arrives at the destination output port.

We have employed the *Destination-Tag (DT)* routing algorithm [93] for the butterfly network in which the destination address is used to select the output port at each stage of the network.

6.2.4 Flattened Butterfly NoC

The *flattened butterfly* topology is derived by combining (or flattening) the routers in each row of a conventional butterfly topology while preserving the inter-router connections. Therefore, the flattened butterfly is similar to a generalized hypercube network. However, the concentration in the flattened butterfly significantly reduces the wiring complexity of the topology, allowing it to scale more efficiently [110].

In our work, we collapsed the 3-stage radix-3 butterfly network of Figure 6.5 to construct the flattened butterfly illustrated in the bottom of the same figure. The resulting flattened butterfly has nine radix-7 routers. Among the seven router ports, four are used for inter-router connections: two for the connections in dimension X and two for the connections in dimension Y. Those connections are marked with red arrows in the figure. The remaining three ports are used for the IP cores attached to each router since the routers have a concentration factor of 3. Those ports are shown for router 0 in the figure. Note that the input/output ports of the remaining routers are not depicted in the figure for simplicity.

Routing in a flattened butterfly requires a hop from an IP core to its local router, zero or more inter-router hops, and a final hop from a router to the destination IP core [111]. Both minimal and non-minimal routing algorithms can be used for the flattened butterfly topology [110].

We evaluated the proposed flattened butterfly NoC using the following three routing algorithms:

1. *Minimal Adaptive (MIN AD)* routing algorithm: in the MIN AD routing method, the XY or YX minimal routing direction is chosen adaptively such that the packet is forwarded to the channel with the shortest queue [111]. At least two Virtual Channels (VCs) [93] should be used in MIN AD to prevent deadlock.

2. *Valiant's (VAL)* routing algorithm [112]: the main objective of the VAL non-minimal oblivious routing method is to balance the traffic load across the network by converting any traffic pattern into two phases of random traffic. First, a random intermediate node b is picked and the packets are routed minimally from source to b. Then, the packets are routed minimally from b to the destination. This approach is able to perfectly balance the load on average, but at the cost of doubling the worst-case hop count. Any minimal routing algorithm can be used for each phase. In this work, we have employed the Dimension-Order Routing (DOR)[7] for both phases. Two VCs, one for each phase, are required to avoid deadlock in this algorithm [111].

3. *Universal Globally-Adaptive Load-balanced (UGAL)* routing algorithm [113]:

[7] DOR routes the packets by crossing dimensions in an increasing order, nullifying the offset in one dimension before routing in the next one.

UGAL is a non-minimal adaptive routing method which chooses between MIN AD and VAL on a packet-by-packet basis to minimize the estimated delay for each packet. The delay is estimated based on the product of queue length and hop count. For benign traffic patterns and at low loads, UGAL routes the traffic minimally matching the performance of MIN AD. However, for adversarial patterns at high loads, the traffic is routed non-minimally matching the performance of VAL [111].

The NoC topology makes use of the address bits present in the data packet to multiplex different CABs to the different memory banks. Therefore, the embedded processor needs not participate during reconfiguration once the configuration data are stored in the memory banks. Thus, we save significant amount of CPU clock cycles.

6.2.5 Significance of the proposed architecture

1. The proposed architecture enables parallel reconfiguration of the LUTs located in different CLB columns which is in contrast with ICAP based reconfiguration. To reconfigure the SRAM cells of the interconnection we must make sure parallel reconfiguration does not cause short circuits that may lead to a DRC error. Therefore, as of now we consider to reset (reconfigure with bit 0) the existing configuration of the switch blocks and the connection blocks before reconfiguring the interconnection configuration.

2. The configuration data access at the frame level is completely eliminated. The proposed architecture enables the processor to access the configuration data in chunks as small as configuration bits of a single LUT. Therefore, if it is required to reconfigure a single LUT then there is no need to access a complete frame of 101 words.

3. The LUTs located in the same CLB column are reconfigured serially. However, reconfiguring each LUT does not consume much time and hence the serial reconfiguration of multiple LUTs in the same CLB column is still faster compared to the classic ICAP based reconfiguration.

4. In case of Dynamic Circuit Specialization, micro-reconfiguration enables the processor to reconfigure a single LUT by accessing configuration in terms of frames in three steps: read frames, modify and write-back frames. However, with the help of the proposed architecture the processor can accomplish micro-reconfiguration in a single step by writing the configuration data in a LUT directly. The specialized configuration data for each LUT is generated after evaluating the Boolean functions. This evaluation is done for the entire content of the truth table bits of a LUT and hence the read-back

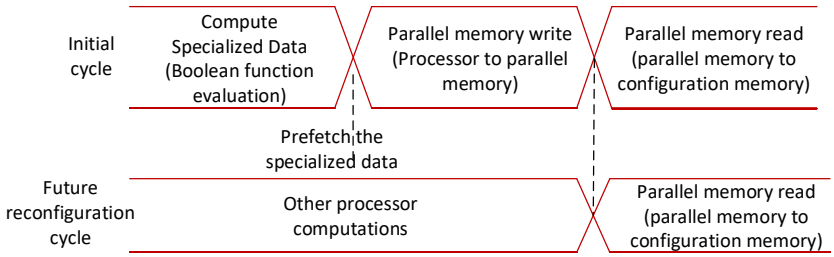


Figure 6.6: Specialized data prefetch cycle

frames and modify frames step are completely eliminated. The reconfiguration time overhead reduces by at least by a factor of two.

5. The proposed architecture enables the processor to evaluate the Boolean functions (assuming parameter values are available), keep the specialized configuration data ready, and let the parallel memory prefetch the specialized data (assuming the memory banks are deep enough to hold multiple specialized configuration data), all while the current reconfiguration cycle is still under progress. Hence the processor need not wait for the current reconfiguration process to end and then start the Boolean evaluation for the next cycle of reconfiguration. Once the prefetch is done, the CPU cycles can be used for other useful computations (Figure 6.6).

6. In case of modular region based reconfiguration, the configuration data is written into the LUTs using the same approach as described above. The only gain we can expect is high speed reconfiguration due to parallel reconfiguration.

6.3 Results

The following describes the estimated hardware costs for the proposed architecture and the simulated results obtained after running the experiments with the set of parameterized applications.

6.3.1 Estimated hardware cost

The proposed configuration memory architecture comes at the cost of hardware costs incurred by the auxiliary hardware. The costs for the new configuration memory infrastructure mainly include:

1. Configuration Access Bus: for a clock region X0Y0 of the Zynq-SoC (consists of 25 CLB columns and each column consists of 50 CLBs), the proposed architecture was configured to consist of 80 wires dedicated for reconfiguration in the routing channel. These lines are located in the routing channel along with the interconnect lines. Supposing that 300 wires are present in the Stratix IV architecture [114], an overhead of 27% can be a reasonable estimate of wire overhead. We can overcome this overhead by making use of existing wires in the channel and multiplexing their functionality for reconfiguration and for application execution.

2. Memory banks: Each memory bank corresponds to one Block RAM of a commercial FPGA. Considering the Artix-7 FPGA, each memory bank can hold upto 36 kB of configuration data. With the proposed NoC-based architecture we need 8 memory banks for the clock region X0Y0 of the Zynq-SoC.

3. Circuits and wires: an additional circuit is needed in each slice of a CLB that can access the CAB to read/write the configuration data from the LUT SRAM cells. This part of the work is planned as a part of the future work.

4. Network-on-Chip: the performance of the proposed NoC architectures were evaluated using the BookSim 2.0 interconnection network simulator [99]. The Orion 3.0 [108] which is an enhanced NoC power and area simulator was also integrated in BookSim to evaluate the hardware overhead of the proposed schemes. The butterfly and flattened butterfly topologies are supported by Orion models.

The configuration parameters and corresponding results for both 3-ary 3-fly and 3-ary 3-flat networks are listed in Table 6.1. A packet can be decomposed into several fixed length contiguous units called *FLITs*[8] *(FLow control digIT)* that are transmitted from the IP cores to the network and then routed consecutively through the network [93]. The packet latency is defined as the number of cycles spent between the generation of the message at the source until the tail flit reaches the destination [115]. To ensure a fair comparison, the routers in all of the configurations were equipped with equal entries for the buffer queues. The router pipeline consists of three cycles for routing computation, switch allocation, and switch traversal. The router pipeline for VC-based methods requires an additional cycle for VC allocation. The inter-router link traversal is fixed to one cycle for all configurations. Moreover, the adopted traffic pattern exposes the inherent communication behavior of the applications.

First, the simulator was warmed up for 100,000 cycles to be stabilized and then the results were averaged over the next 1,000,000 cycles. The results from the

[8]Flit is the basic unit of bandwidth and storage allocation in the network.

Table 6.1: NoC configuration parameters and results

Network Topology	**Configuration parameters**					**Results**		
	Channel width (bit)	**Packet size (flit)**	**Number of VCs**	**Buffer size (flit)**	**Routing algorithm**	**Latency (cycle)**	**Power (Watt)**	**Area (mm^2)**
3-ary 3-fly	64	2	2	1	DT	268	0.78	0.064
	32	3	2	2	DT	209	0.38	0.027
3-ary 3-flat	64	2	2	1	MIN AD	438	1.05	0.093
	64	2	2	1	VAL	872	1.05	0.093
	64	2	2	1	UGAL	619	1.05	0.093
	32	3	2	2	MIN AD	338	0.49	0.036
	32	3	2	2	VAL	653	0.49	0.036
	32	3	2	2	UGAL	457	0.49	0.036

warm-up period are ignored because the network is not stabilized yet. Moreover, all of the values reported in this section are obtained by averaging the results from 10 distinct simulation samples to ensure a fair comparison between the proposed architectures.

For power and area of the interconnect, we used technology parameters from the 32 nm 0.9 V ITRS-HP process provided by CACTI [109]. The area model accounts for four primary components of area overhead: input buffers, switch fabric, communication channels, and output buffers (Table 6.1).

6.3.2 Reconfiguration simulation results

We have created a reconfiguration time estimator tool based upon the proposed architecture. The tool takes the location of each LUT (that needs to be reconfigured) as the input and gives the estimation of the reconfiguration time. We have used five parameterized applications as a benchmark to evaluate the improvement in the micro-reconfiguration speed with the proposed architecture. The benchmark includes parameterized applications explained in chapter 3, Section 3.3 along with the parameterized threshold module used for the edge detection explained as follows.

Threshold for edge detection: an edge detection algorithm implemented in hardware consists of a module that can be configured to adjust a threshold level. Every pixel is compared with the threshold level to detect the edge of an image. Since the threshold level value is an infrequently changing parameter, a total of 33 TLUTs were required to store the threshold value for an image edge detection implementation. For every change in threshold values those 33 TLUTs are micro-reconfigured.

Each of the benchmark applications was implemented on the Zynq-SoC without any user constraints so that the placer tool can place the design on TLUTs efficiently. The locations of all TLUTs were recorded and annotated to our reconfiguration time estimator.

The NoC latency is also included in the reconfiguration time estimator so that we can estimate the total time taken during the reconfiguration process. We consider only the relevant NoC topology and their latency to evaluate the reconfiguration time. As of now the estimated area and power only for the NoC (with different topologies) can be obtained from the NoC simulator.

The reconfiguration time estimate based on the proposed NoC-based parallel memory architecture is tabulated in Table 6.2. Clearly, the parameterized applications (FIR and MAC operator) that have a huge number of TLUTs can benefit more from the proposed architecture. The reconfiguration speed is improved by at least 1000×. This gain primarily comes from the parallel reconfiguration, as the TLUTs are sparsely spread over different CLB columns. For smaller parameter-

Table 6.2: Reconfiguration time comparison

Parameterized Benchmarks	**NoC Network Topology**	**NoC Latency** (cycles)	**# TLUTs**	**Reconfiguration time** (cycles)	**Conventional Reconfiguration time** (cycles)	**Improvement factor**
FIR	3-ary 3-fly	268	384	1298	2461440	1896
	3-ary 3-flat	872	384	1902	2461440	1294
MAC operator	3-ary 3-fly	268	468	2958	2999880	1014
	3-ary 3-flat	872	468	3562	2999880	842
ROM	3-ary 3-fly	268	4	308	25640	83
	3-ary 3-flat	872	4	912	25640	28
TCAM	3-ary 3-fly	268	41	678	262810	388
	3-ary 3-flat	872	41	1282	262810	205
Threshold module	3-ary 3-fly	268	33	598	211530	354
	3-ary 3-flat	872	33	1202	211530	176

ized applications, the reconfiguration speed gain is relatively smaller due to the fact that some TLUTs are located in the same CLB column and hence they don't benefit from the parallel reconfiguration.

In the worst case, suppose for an application whose reconfigurable part is mapped on to a single CLB column over all the LUTs (400) present in a CLB column, the reconfiguration has to occur sequentially. In that case our estimated gain could reach up to 160× compared to the fastest reconfiguration method used in the micro-reconfiguration.

7

Conclusions and Future work

Dynamic Circuit Specialization (DCS) is an optimization technique used for implementing a parameterized application on an FPGA. The application is said to be parameterized when some of its inputs, called parameters, are infrequently changing compared to the other inputs. Instead of implementing these parameter inputs as regular inputs, in the DCS approach these inputs are implemented as constants and the design is optimized for these constants. When the parameter values change, the design is re-optimized for the new constant values by micro-reconfiguring the FPGA. Therefore, the goal of DCS is to use simpler and optimized circuits instead of a bulky generic version of the same circuit.

Micro-reconfiguration is a fine-grained form of partial reconfiguration tailored to accomplish DCS. One has to consider the overheads of the micro-reconfiguration while considering DCS implementations. The standard method of implementing DCS incurs undesirable overheads and therefore, DCS suffers from diminishing effects.

In order to conclude this dissertation I recapitulate the goal of the thesis:

A detailed study of overheads of DCS and providing suitable solutions with appropriate custom FPGA structures is the primary goal of the dissertation. I also suggest different improvements to the FPGA configuration memory architecture. After offering the custom FPGA structures, I investigated the role of DCS on FPGA overlays, and the use of custom FPGA structures that help to reduce the overheads of DCS on FPGA overlays. By doing so, I hope I am able to convince the developer to use DCS (which now comes with minimal overheads) in real-world applications.

7.1 Conclusions

In this section, I present the overall conclusions of the research focusing on how my goals are achieved for a given set of problems.

7.1.1 Overheads and custom FPGA structures

In this dissertation, I presented four major overheads of the micro-reconfiguration: Boolean evaluation time, PPC memory size, reconfiguration time, and the static and dynamic power consumption. The Boolean evaluation time depends on the capability of the SCG. A stack machine-based custom SCG was already proposed in the previous work in [32]. The efficient configuration representation can lead to a reduction in PPC memory size which is already proposed in [32]. However, the reconfiguration time is the major overhead of all, and it influences the energy consumed during micro-reconfiguration.

In Chapter 3, I investigated the impact of the evolution in (Xilinx) FPGA architectures on DCS implementations. The FPGA architecture for various platforms plays a significant role in the micro-reconfiguration, and hence it influences the corresponding overheads. The trend in the evolution of the Xilinx FPGA architecture shows that there has been a minimal emphasis on the reconfiguration technology. This could prove a major problem for implementing an HPC application that requires reconfiguration as an intrinsic feature.

To improve the reconfiguration speed with existing FPGA architectures, in Chapter 4 I have proposed custom reconfiguration controllers specifically designed to implement DCS. I have also introduced two different reconfiguration drivers: MRMW and MROMW that accelerate the reconfiguration speed of the micro-reconfiguration controller. When used all together, we achieve an improvement factor of 40. The custom controllers are power and energy efficient by factor of four compared to the standard reconfiguration controller (HWICAP) IP provided by Xilinx. The effect of the improvement in reconfiguration speed was explained using functional density curves.

7.1.2 FPGA overlays and DCS

Conventional CGRAs suffer from high reconfiguration overhead and has no parameterization options. A strategy to implement efficient VCGRAs on FPGAs is proposed in Chapter 5. The use of DCS on VCGRAs is studied on two VCGRA grids: MAC grid and Pixie. These two grids are used to realize a real-world HPC image processing application called Retinal Vessel Segmentation. In this dissertation I have described two different variants of parameterized VCGRA implementations:

1. Partially parameterized VCGRA: A partially parameterized VCGRA implementation in which only the settings registers are optimized and realized on TLUTs.

2. Fully parameterized VCGRA: A fully parameterized VCGRA implementation contains PE and virtual interconnects that are optimized by parameterizing the inter and intra-connects of each PEs and VSBs.

The implementation of MAC grid VCGRA in three styles was studied using a functional density curve.

7.1.3 Custom FPGA configuration memory

One of the main reasons for the reconfiguration time overhead of the micro-reconfiguration is the frame-level access of the FPGA configuration. To overcome the frame-level access, I have proposed a parallel memory structure for each clock region in Chapter 6. The parallel memory contains a configuration distribution network that comes with two variants: cross-bar based and NoC-based.

The NoC-based parallel memory helps to decentralize the distribution of the configuration data to different CLB columns. Our simulation results show the proposed configuration memory structure can influence to drastically improve the reconfiguration speed at least by a factor of 1000. However, the hardware resource consumption for this structure is too high, but future work may be conducted to optimize the hardware resources.

7.2 Future work

I conclude this dissertation with possible scope for the future research on DCS and FPGA overlays.

7.2.1 Secured DCS for space applications

A system implemented with the state of the art DCS lacks two important features: security and robustness. Thus, the current DCS may not be suitable for space applications. In this context, I suggest for the future work to investigate all the security vulnerabilities of a conventional DCS system and address them with appropriate solutions. A custom reconfiguration controller with a capability to secure the frames using cryptography-based solutions for secured reconfiguration is essential. The controller should be equipped with a closed loop feedback system to establish robust reconfiguration that assures any perturbations during reconfiguration can be handled efficiently.

The configuration bitstreams of a Static Random Access Memory (SRAM) based Field Programmable Gate Array (FPGA) are prone to security risks if the FPGA is used in non-encrypted mode [116] [117]. The risks could be bitstream tampering, bitstream cloning, bitstream reverse engineering, etc. However, commercial FPGA vendors such as Xilinx provide an option to use the FPGA in encrypted mode to secure a 7 Series FPGA bitstream. In this mode, a standard 256-bit Advanced Encryption Standard (AES) [35] cryptographic system is used for encryption and decryption of the FPGA bitstream [118]. The 7-series FPGA AES system consists of software-based encryption. The encryption is performed with a 256-bit encryption key. The encryption key can be programmed by the user and then stored in a dedicated RAM memory, in a battery backed up RAM (BBRAM) or in a eFUSE memory.

The encrypted bitstream has to be decrypted before it is programmed on to the configuration memory and hence on-chip AES decryption logic is used before the programming of the bitstreams on to the configuration memory. The on-chip decryption logic can be utilized decrypting the FPGA bitstream only.

Using the FPGA in encryption mode for implementing secured DCS is practically impossible due to the limitations imposed by the Xilinx FPGAs that it does not support encrypted configuration frame read back. The configuration read back is a required step to specialize the configuration frames during run-time. To overcome this limitation we propose a custom high-speed reconfiguration controller that supports encrypted configuration read back.

In many cases, the parameterized applications have to be robust enough to handle faults that can occur during reconfiguration. The current DCS does not have fault-tolerant feature. Therefore, a robust reconfiguration method along with a secured custom reconfiguration controller is needed to implement robust and secured DCS systems.

7.2.2 Floating point overlay library

The Floating-point operator implemented on FPGAs proves to be an efficient accelerator for image processing applications. There are many strategies to automatically tune the precision of the floating operator for a given application [119] [120]. Therefore, I suggest an overlay library for floating point operations with automatic tuning algorithms. The advantages of the VCGRA tool flow on reducing the development costs can be used for realizing the floating point accelerators for different precisions.

Bibliography

[1] G.E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 5, pp. 33–35, Sept 2006.

[2] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[3] T. Mitra, "Heterogeneous Multi-core Architectures," *IPSJ Transactions on System LSI Design Methodology*, vol. 8, pp. 51–62, Aug 2015.

[4] M. Pricopi, T.S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 15:1–15:10.

[5] M. Pricopi and T. Mitra, "Bahurupi: A polymorphic heterogeneous multi-core architecture," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 22:1–22:21, Jan. 2012.

[6] L. Hansen, "Unleash the Unparalleled Power and Flexibility of Zynq UltraScale+ MPSoCs," *White Paper: Zynq UltraScale+ MPSoCs*, 2016, accessed: 2017-03-21.

[Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp470-ultrascale-plus-power-flexibility.pdf

[7] D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer Science & Business Media, 2012.

[8] 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 *2006 International Conference on Field Programmable Logic and Applications*, Aug 2006, pp. 1–6.

[9] M. Huebner, T. Becker, and J. Becker, "Real-time lut-based network topologies for dynamic and partial fpga self-reconfiguration," in *Proceedings of the 17th Symposium on Integrated Circuits and System Design*, ser. SBCCI '04. New York, NY, USA: ACM, 2004, pp. 28–32.

[10] Xilinx UG892, "Vivado Design Suite User Guide," accessed: 2017-04-13.

[Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_3/ug892-vivado-design-flows-overview.pdf

[11] Xilinx UG702, "Partial Reconfiguration User Guide," accessed: 2017-08-08.

[Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf

[12] Xilinx DS817, "Xilinx LogiCORE IP AXI HWICAP (v2.02.a)," accessed: 2017-04-26.

[Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v2_02_a/ds817_axi_hwicap.pdf

[13] K. Bruneel, W. Heirman, and D. Stroobandt, "Dynamic Data Folding with Parameterizable Configurations," *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 4, 2011.

[14] C.B. Ciobanu, A.L. Varbanescu, D. Pnevmatikatos, G. Charitopoulos, X. Niu, W. Luk, M.D. Santambrogio, D. Sciuto, M.A. Kadi, M. Huebner, T. Becker, G. Gaydadjiev, A. Brokalakis, A. Nikitakis, A.J.W. Thom, E. Vansteenkiste, and D. Stroobandt, "EXTRA: Towards an Efficient Open Platform for Reconfigurable High Performance Computing," in *2015 IEEE 18th International Conference on Computational Science and Engineering*, Oct 2015, pp. 339–342.

[15] D. Stroobandt, A.L. Varbanescu, C.B. Ciobanu, M.A. 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 *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, June 2016, pp. 1–7.

[16] P. Lysaght and W. Rosenstiel, *New algorithms, architectures and applications for reconfigurable computing*. Springer, 2005.

[17] K. Heyse, "Improving the Gain and Reducing the overhead of Dynamic Circuit Specialization and Microreconfiguration," Ph.D. dissertation, Ghent University, 2015.

[18] P.B. Minev and V.S. Kukenska, "The Virtex-5 Routing and Logic Architecture," *Annual Journal of Electronics*, vol. 3, pp. 107–110, 2009.

[19] Xilinx UG474, "7 Series FPGAs Configurable Logic Block," accessed: 2017-04-12.

[Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[20] Xilinx DS180, "7 Series FPGAs Data Sheet: Overview (v2.4)," accessed: 2017-04-12.

[Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[21] Xilinx DS890, "UltraScale Architecture and Product Data Sheet: Overview (v2.11)," accessed: 2017-04-12.

[Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[22] Xilinx DS190, "Zynq-7000 All Programmable SoC Overview (v1.10)," accessed: 2017-04-12.

[Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[23] Xilinx UG470, "7 Series FPGAs Configuration User Guide," accessed: 2017-04-10.

[Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf

[24] Xilinx UG191, "Virtex-5 FPGA Configuration User Guide," accessed: 2017-04-10.

[Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[25] Xilinx DS817, "Xilinx LogiCORE IP AXI HWICAP (v2.03.a)," accessed: 2017-04-19.

[Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v2_03_a/ds817_axi_hwicap.pdf

[26] C. Kohn, "Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices (XAPP1159)," 2013, accessed: 2017-04-21.

[Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1159-partial-reconfig-hw-accelerator-zynq-7000.pdf

[27] L. Hellerman, "A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 3, pp. 198–223, June 1963.

[28] P.W. Foulk, "Data-folding in SRAM configurable FPGAs," in *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, Apr 1993, pp. 163–171.

[29] M.J. Wirthlin and B.L. Hutchings, "Improving functional density through run-time constant propagation," in *Proceedings of the 1997 ACM Fifth International Symposium on Field-programmable Gate Arrays*, ser. FPGA '97. New York, NY, USA: ACM, 1997, pp. 86–92.

[30] K. Heyse, B. Al Farisi, K. Bruneel, and D. Stroobandt, "TCONMAP: Technology Mapping for Parameterised FPGA Configurations," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 4, pp. 48:1–48:27, Sep. 2015.

[31] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 964–969.

[32] F. Mostafa Mohamed Ahmed Abouelella, K. Bruneel, and D. Stroobandt, "Efficiently generating FPGA configurations through a stack machine," in *Field Programmable Logic and Applications, 20th International conference, Abstracts*, Milano, Italy, 2010.

[33] H. Gazit, "Ternary content-addressable memory," Jun. 5 2012, US Patent 8,195,873.

[Online]. Available: https://www.google.com/patents/US8195873

[34] 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 *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE xplore, 2016, pp. 265–270.

[35] F.I. Processing and S.P. 197, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," 2001, accessed: 2017-04-11.

[Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[36] T. Davidson, F. Mostafa Mohamed Ahmed Abouelella, K. Bruneel, and D. Stroobandt, "Dynamic Circuit Specialization for key-based encryption algorithms and DNA alignment," *INTERNATIONAL JOURNAL OF RECONFIGURABLE COMPUTING*, vol. 2012, p. 13, 2012.

[37] M. El-Hadedy, H. Mihajloska, D. Gligoroski, A. Kulkarni, D. Stroobandt, and K. Skadron, "A 16-bit Reconfigurable encryption processor for Pi-Cipher," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. Chicago, USA: IEEE xplore, 2016, pp. 162–171.

[38] A. DeHon, "Reconfigurable Architectures for General-Purpose Computing," Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, USA, Tech. Rep., 1996.

[39] Xilinx UG011, "PowerPC Processor Reference Guide (v1.3)," accessed: 2017-04-11.

[Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug011.pdf

[40] Xilinx DS100, "Virtex-5 Family Overview (v5.1)," accessed: 2017-04-11.

[Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[41] Xilinx UG012, "Virtex-II Pro and Virtex-II Pro X FPGA User Guide(v4.2)," accessed: 2017-04-11.

[Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug012.pdf

[42] Xilinx UG761, "AXI Reference Guide (v13.1)," accessed: 2017-04-11.

[Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[43] Xilinx DS531, "LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a)," accessed: 2017-04-11.

[Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf

[44] Xilinx DS402, "LogiCORE IP On-Chip Peripheral Bus V2.0 with OPB Arbiter (v1.00d)," accessed: 2017-04-11.

[Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/opb_v20.pdf

[45] K. Bruneel, K. Heyse, and D. Stroobandt, "The TLUT tool flow," 2011.

[Online]. Available: https://github.com/UGent-HES/tlut_flow

[46] V. Weaver and S. McKee, "Code density concerns for new architectures," in *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, Oct 2009, pp. 459–464.

[47] R. Bonamy, D. Chillet, S. Bilavarn, and O. Sentieys, "Power Consumption Model for Partial and Dynamic Reconfiguration," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, Dec 2012, pp. 1–8.

[48] A. Kulkarni, K. Heyse, T. Davidson, and D. Stroobandt, "Performance Evaluation of Dynamic Circuit Specialization on Xilinx FPGAs," in *FPGAworld Conference 2014, Proceedings*. Association for Computing Machinery, 2014, pp. 1–6.

[49] A. Kulkarni, T. Davidson, K. Heyse, and D. Stroobandt, "Improving reconfiguration speed for Dynamic Circuit Specialization using Placement Constraints," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–6.

[50] A. Kulkarni, V. Kizheppatt, and D. Stroobandt, "MiCAP: A custom Reconfiguration Controller for Dynamic Circuit Specialization," in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, Dec 2015, pp. 1–6.

[51] A. Kulkarni and D. Stroobandt, "MiCAP-Pro: a high speed custom reconfiguration controller for Dynamic Circuit Specialization," *Design Automation for Embedded Systems*, vol. 20, no. 4, pp. 341–359, 2016.

[52] S. Hansen, D. Koch, and J. Torresen, "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 174–180.

[53] Xilinx PG021, "LogiCORE IP AXI DMA (v6.03a)," accessed: 2017-04-11. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v6_03_a/pg021_axi_dma.pdf

[54] Xilinx PG059, "LogiCORE IP AXI Interconnect (v2.1)," accessed: 2017-04-11. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[55] K. Vipin and S. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 41–44, Sept 2014.

[56] F. Duhem, F. Muller, and P. Lorenzini, *FaRM: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 253–260.

[57] A. Kulkarni and D. Stroobandt, "MiCAP: Micro-reconfigurable Configuration Access Port," 2015.

[Online]. Available: https://github.com/UGent-HES/MiCAP

[58] A. Kulkarni and D. Stroobandt , "MiCAP-Pro: Pro version of Micro-reconfigurable Configuration Access Port," 2015.

[Online]. Available: https://github.com/UGent-HES/MiCAP-Pro

[59] Xilinx UG625, "Constraints Guide (v 13.4)," accessed: 2017-04-20.

[Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/cgd.pdf

[60] B. Al Farisi, K. Bruneel, J.M.P. Cardoso, and D. Stroobandt, "An automatic tool flow for the combined implementation of multi-mode circuits," in *Proceedings - Design, Automation, and Test in Europe Conference and Exhibition*, 2013, pp. 821–826.

[61] B. Al Farisi, "Techniques for Low-Overhead Dynamic Partial Reonfiguration of FPGAs," Ph.D. dissertation, Ghent University, 2015.

[62] A. Kulkarni and D. Stroobandt, "How to efficiently reconfigure Tunable LookUp Tables for Dynamic Circuit Specialization," *INTERNATIONAL JOURNAL OF RECONFIGURABLE COMPUTING*, vol. 2016, pp. 1–11, 2016.

[63] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, March 2006.

[64] H.K.H. So and C. Liu, "FPGA overlays," in *FPGAs for Software Programmers*. Springer International Publishing, 2016, pp. 285–305.

[65] R. Lysecky, K. Miller, F. Vahid, and K. Vissers, "Firm-core Virtual FPGA for Just-in-Time FPGA Compilation (Abstract Only)," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 271–271.

[66] D. Grant, C. Wang, and G.G. Lemieux, "A CAD Framework for Malibu: An FPGA with Time-multiplexed Coarse-grained Elements," in

Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 123–132.

[Online]. Available: http://doi.acm.org/10.1145/1950413.1950441

[67] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, Oct 2010, pp. 13–22.

[68] D. Koch, C. Beckhoff, and G.G.F. Lemieux, "An efficient FPGA overlay for portable custom instruction set extensions," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–8.

[69] A. Kulkarni, D. Stroobandt, A. Werner, F. Fricke, and M. Huebner, "Pixie: A heterogeneous Virtual Coarse-Grained Reconfigurable Array for high performance image processing applications," in *3rd International Workshop on Overlay Architectures for FPGAs (OLAF2017)*, 2017, pp. OLAF/2017/01:1–OLAF/2017/01:6.

[70] M. Al Kadi and M. Huebner, "Integer Computations with Soft GPGPU on FPGAs," in *International Conference on Field-Programmable Technology (FPT '16)*, 2016.

[71] A. Kourfali, A. Kulkarni, and D. Stroobandt, "SICTA: A Superimposed In-Circuit Fault Tolerant Architecture for SRAM-based FPGAs," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2017, pp. 1–4.

[72] P. Yiannacouras, J.G. Steffan, and J. Rose, "Fine-grain Performance Scaling of Soft Vector Processors," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 97–106.

[73] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, Aug 2011, pp. 1–5.

[74] A.K. Jain, X. Li, P. Singhai, D.L. Maskell, and S.A. Fahmy, "DeCO: A DSP Block Based FPGA Accelerator Overlay with Low Overhead Interconnect," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 1–8.

[75] A.K. Jain, S.A. Fahmy, and D.L. Maskell, "Efficient overlay architecture based on DSP blocks," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 25–28.

[76] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker, "A Heterogeneous Multicore System on Chip with Run-time Reconfigurable Virtual FPGA Architecture," in *Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 143–149.

[77] M.A.A. Tuhin and T.S. Norvell, "Compiling parallel applications to coarse-grained reconfigurable architectures," in *2008 Canadian Conference on Electrical and Computer Engineering*, May 2008, pp. 001 723–001 728.

[78] R. Gnanaolivu, T.S. Norvell, and R. Venkatesan, "Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization," in *2010 International Conference of Soft Computing and Pattern Recognition*, Dec 2010, pp. 145–151.

[79] J.A. Brenner, S.P. Fekete, and J.C. van der Veen, "A minimization version of a directed subgraph homeomorphism problem," *Mathematical Methods of Operations Research*, vol. 69, no. 2, pp. 281–296, 2009.

[Online]. Available: http://dx.doi.org/10.1007/s00186-008-0259-0

[80] K. Heyse, T. Davidson, E. Vansteenkiste, K. Bruneel, and D. Stroobandt, "Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAs," in *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications*. Piscataway, NJ, USA: IEEE, 2013, pp. 1–8.

[81] K. Heyse, B. Al Farisi, K. Bruneel, and D. Stroobandt, "TCONMAP: Technology Mapping for Parameterised FPGA Configurations," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 4, pp. 48:1–48:27, Sep. 2015.

[82] J. Divyasree, H. Rajashekar, and K. Varghese, "Dynamically reconfigurable regular expression matching architecture," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, July 2008, pp. 120–125.

[83] L. Sekanina, "Virtual Reconfigurable Circuits for Real-world Applications of Evolvable Hardware," in *Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ser. ICES'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 186–197.

[84] T. Miyoshi, H. Kawashima, Y. Terada, and T. Yoshinaga, "A Coarse Grain Reconfigurable Processor Architecture for Stream Processing Engine," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 490–495.

[85] E. Vansteenkiste, B. Al Farisi, K. Bruneel, and D. Stroobandt, "TPaR: Place and Route Tools for the Dynamic Reconfiguration of the FPGA's Interconnect Network," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 3, pp. 370–383, March 2014.

[86] S. Chaudhuri, S. Chatterjee, N. Katz, M. Nelson, and M. Goldbaum, "Detection of blood vessels in retinal images using two-dimensional matched filters," *IEEE Transactions on medical imaging*, vol. 8, no. 3, pp. 263–269, 1989.

[87] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *Design Test of Computers, IEEE*, vol. 28, no. 4, pp. 18–27, July 2011.

[88] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-strength Verification Tool," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 24–40.

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

[90] X. Wang and M. Leeser, "VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 3, pp. 16:1–16:34, Sep. 2010.

[91] X. Fang and M. Leeser, "Open-Source Variable-Precision Floating-Point Library for Major Commercial FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, pp. 20:1–20:17, Jul. 2016.

[92] C. Ciobanu, G. Kuzmanov, A. Ramirez, and G. Gaydadjiev, "A polymorphic register file architecture," *5th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, pp. 245–248, 2009.

[93] W.J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, USA: Morgan Kaufmann Publishers, 2004.

[94] J. Duato, S. Yalamanchili, and L.M. Ni, *Interconnection Networks: An Engineering Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[95] M. Daneshtalab and M. Palesi, Eds., *Routing Algorithms in Networks-on-Chip*. New York, NY, USA: Springer Publishing Company, Inc., 2014.

[96] M.B. Stensgaard and J. SparsÃÿ, "ReNoC: A Network-on-Chip Architecture with Reconfigurable Topology," in *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*, April 2008, pp. 55–64.

[97] J.S. Shen and P.A. Hsuing, "Dynamic Reconfigurable Network-on-chip Design: Innovations for Computational Processing and Communication," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 6, pp. 42–43, Nov. 2013, reviewer-Schaefer, Robert.

[98] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip," *ACM Comput. Surv.*, vol. 38, no. 1, Jun. 2006.

[99] N. Jiang, D.U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, J. Kim, and W.J. Dally, "A detailed and flexible cycle-accurate Network-on-Chip simulator," in *Proc. IEEE Int. Symp. Perf. Analysis Syst. Software (ISPASS)*, 2013, pp. 86–96.

[100] N. Jiang, G. Michelogiannakis, D. Becker, B. Towles, and W.J. Dally, *BookSim 2.0 User's Guide*, Stanford University.

[101] P. Bahrebar, "Adaptive routing methods for on-chip interconnection networks," Ph.D. dissertation, Ghent University, Belgium, 2017.

[102] U.Y. Ogras and R. Marculescu, "Energy- and performance-driven NoC communication architecture synthesis using a decomposition approach," in *Proc. ACM/IEEE Design Automat. Test in Europe Conf. (DATE)*, 2005, pp. 352–357.

[103] A. Agarwal, C. Iskander, and R. Shankar, "Survey of Network on Chip (NoC) architectures & contributions," *Journal of Eng., Comput. and Arch.*, vol. 3, no. 1, 2009.

[104] L.M. Ni and P.K. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, 1993.

[105] L. Benini and D. Bertozzi, "Network-on-Chip architectures and design methods," *IEE-Proc. Comput. Dig. Techn.*, vol. 152, no. 2, pp. 261–272, 2005.

[106] P.V. Gratz, "Network-on-Chip implementation and performance improvement through workload characterization and congestion awareness," Ph.D. dissertation, The University of Texas at Austin, USA, 2008.

[107] B. Grot, J. Hestness, S.W. Keckler, and O. Mutlu, "Express cube topologies for on-chip interconnects," in *Proc. IEEE Int. Symp. High Perf. Comp. Arch. (HPCA)*, 2008, pp. 163–174.

[108] A.B. Kahng, L. Bin, and S. Nath, "ORION 3.0: A comprehensive NoC router estimation tool," *IEEE Embedded Syst. Letters*, vol. 7, no. 2, pp. 41–45, 2015.

[109] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi, "CACTI 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[110] J. Kim, J. Balfour, and W.J. Dally, "Flattened butterfly topology for on-chip networks," *IEEE Comp. Arch. Letters*, vol. 6, no. 2, pp. 37–40, 2007.

[111] J. Kim, W.J. Dally, and D. Abts, "Flattened butterfly: A cost-efficient topology for high-radix networks," in *Proc. Int. Symp. Comput. Arch. (ISCA)*, 2007, pp. 126–137.

[112] L.G. Valiant, "A scheme for fast parallel communication," *SIAM Journal on Comput.*, vol. 11, no. 2, pp. 350–361, 1982.

[113] A. Singh, "Load-balanced routing in interconnection networks," Ph.D. dissertation, Stanford University, USA, 2005.

[114] K.E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–8.

[115] É. Cota, A. de Morais Amory, and M. Soares Lubaszewski, *Reliability, Availability and Serviceability of Networks-on-Chip*. Springer Science & Business Media, 2012.

[116] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, "On the Vulnerability of FPGA Bitstream Encryption Against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 111–124.

[117] H. Kashyap and R. Chaves, "Compact and On-the-Fly Secure Dynamic Reconfiguration for Volatile FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 2, pp. 11:1–11:22, Jan. 2016.

[118] Xilinx XAPP1239, "Using Encryption to Secure a 7-Series FPGA Bitstream (v1.0)," 2015, accessed: 2017-05-02.

[Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1239-fpga-bitstream-encryption.pdf

[119] C. Rubio-González, C. Nguyen, H.D. Nguyen, J. Demmel, W. Kahan, K. Sen, D.H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–12.

[120] P. Panchekha, A. Sanchez-Stern, J.R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," *SIGPLAN Not.*, vol. 50, no. 6, pp. 1–11, Jun. 2015.