Syed Mohammad Asad Hassan Jafri

Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures

Turku Centre *for* Computer Science

TUCS Dissertations
No 191, January 2015

Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures

Syed Mohammad Asad Hassan Jafri

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in Auditorium Beta on January 28, 2015, at 12 noon.

University of Turku
Department of Information Technology
FI-20014 Turku
Finland 2014

Supervisors

Assoc. Juha Plosila
Department of Information Technology
University of Turku
Finland

Prof. Ahmed Hemani
Department of Electronic Systems
Royal Institute of Technology
Sweden

Prof. Hannu Tenhunen
Department of Information Technology
University of Turku
Finland

Reviewers

Assoc. Prof. Magnus Jahre
Department of Computer and Information Science
Norwegian University of Science and Technology
Norway

Assoc. Prof. Tulika Mitra
School of Computing
National University of Singapore
Singapore

Opponent

Prof. Jari Nurmi
Department of Electronics and Communications Engineering
Tampere University of Technology
Finland

ISBN 978-952-12-3164-3
ISSN 1239-1883

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin Originality Check service.

Abstract

This thesis presents a novel design paradigm, called Virtual Runtime Application Partitions (VRAP), to judiciously utilize the on-chip resources. As the dark silicon era approaches, where the power considerations will allow only a fraction chip to be powered on, judicious resource management will become a key consideration in future designs. Most of the works on resource management treat only the physical components (i.e. computation, communication, and memory blocks) as resources and manipulate the component to application mapping to optimize various parameters (e.g. energy efficiency). To further enhance the optimization potential, in addition to the physical resources we propose to manipulate abstract resources (i.e. voltage/frequency operating point, the fault-tolerance strength, the degree of parallelism, and the configuration architecture). The proposed framework (i.e. VRAP) encapsulates methods, algorithms, and hardware blocks to provide each application with the abstract resources tailored to its needs. To test the efficacy of this concept, we have developed three distinct self adaptive environments: (i) Private Operating Environment (POE), (ii) Private Reliability Environment (PRE), and (iii) Private Configuration Environment (PCE) that collectively ensure that each application meets its deadlines using minimal platform resources. In this work several novel architectural enhancements, algorithms and policies are presented to realize the virtual runtime application partitions efficiently. Considering the future design trends, we have chosen Coarse Grained Reconfigurable Architectures (CGRAs) and Network on Chips (NoCs) to test the feasibility of our approach. Specifically, we have chosen Dynamically Reconfigurable Resource Array (DRRA) and McNoC as the representative CGRA and NoC platforms. The proposed techniques are compared and evaluated using a variety of quantitative experiments. Synthesis and simulation results demonstrate VRAP significantly enhances the energy and power efficiency compared to state of the art.

Acknowledgments

The research work presented in this thesis has been carried out in the department of Information Technology, University of Turku with close collaboration with Electronic Systems department Royal institute of Technology (KTH) from September 2010 to October 2014. This work would not have been possible in four years without the support of many people. First of all, I would like to express my deepest gratitude to my supervisors, Prof. Ahmed Hemani, Assoc. Prof. Juha Plosila, Assoc. Prof. Kolin Paul and Prof. Hannu Tenhunen, for their excellent guidance, patience, and support. In addition, I would also like to specially thank Prof. Stanislaw Piestrak, for continuous guidance and support during the course of my thesis. I would like to acknowledge the support of my loving wife, Hira, and wonderful sons (Ailee and Jari). Without their love, encouragement, and patience, I would not be able to spend nights in the university to conduct the demanding research. I would like to show my gratitude to the PhD researchers Adeel Tajammul, Omer Malik, Liang Guang, Ali Shami, and Nasim Farahini, who have supported me throughout my research. It gives me great pleasure to acknowledge Assoc. Prof. Magnus Jahre and Assoc. Prof. Tulika Mitra for the detailed reviews and the constructive comments on the manuscript. I thank Prof. Jari Nurmi for agreeing to be my opponent. I greatly appreciate the financial support for my doctoral studies from the Higher Education Commission of Pakistan, Turku Centre of Computer Science (TUCS), Nokia foundation, Ulla Tuomisen saatio and University foundation of Turku. Finally, I would like to thank my parents for their constant love, support, and prayers and dedicate this thesis to them. Turku, October 2014 Syed Mohammad Asad Hassan Jafri

Contents

List of Figures

List of Tables

List of Abbreviations

ACC	Application Configuration Controller
ACMS	Autonomous Configuration Mode Selection
AHB	Amba High Performance Bus
ALU	Arithmetic and Logic Unit
APVFS	Autonomous Parallelism, Voltage, and Frequency Selecti
BB	Basic Block
BIC	Bus based Indirect Configuration
BIST	Built In Self Test
BLS	BLind Scrubber
CC	Configuration Controller
CDMA	Code Division Multiple Access
CGIR	Compact Generic Intermediate Representation
CGRA	Coarse Grained Reconfigurable Architecture
CGU	Clock Generation Unit
CP	Control Packets
DC	Direct Configuration
DED	Double Error Detection
DFMC	Direct Feed and Multicast Controller
DIC	Distributed Indirect Configuration
DME	Data Management Engine
DMR	Double Modular Redundancy
DP	Data Packets
DPM	Dynamic Power Management
DPU	Data Path Unit
DRIC	Dynamically Reconfigurable Isolation Cell
DRRA	Dynamically Reconfigurable Resource Array
DSM	Distributed Shared Memory
DWC	Duplication With Comparison
Dynamic Voltage and Frequency Scaling	DVFS
ECC	Error Correcting Codes
EDC	Error Detecting Codes
EIS	Error Invoked Scrubber
FA	Fault tolerance scheme Adaptive
FFT	Fast Fourier Transform

FIR	Finite Impulse Response filter
FPGA	Field Programmable Gate Array
FT agent	Fault Tolernace agent
GCM	Global Configuration Memory
GMP	PCE Generation and Management Packet
GRLS	Globally Ratio Synchronous Locally Synchronous
HBUS	Horizontal BUS
HI	Hierarchical Index
IAPF	Inter Packet Fault tolerance
IPF	Intra Packet Fault Tolerance
LCC	Local Configuration Controller
MFD	Memory Feed Distributed
MFMC	Memory Feed Multi Cast
MFS	Memory Feed Sequential
MLFC	Memory Load and distributed Feed Controller
MM	Matrix Multiplication
MN	Main Node
Mod	Modulo
MP	Maping Pointer
MTTF	Mean Time To Failure
NoC	Network on Chip
Par	Parallel
Parpar	Partially Parallel
PCC	Platform Configuration Controller
PCE	Private Configuration Enviornment
PE	Processing Element
PLA	Programmable Logic Array
PM agent	Power Management agent
PMU	Power Management Unit
POE	Private Operating Enviornment
PRE	Private Reliability Enviornment
PREX	Private Execution Environments
RAG	Resource Allocation Graph
RAM	Random Access Memory
RBS	ReadBack Scrubber
Reg-file	Register file
RowMultiC	Row Multi-Casting
RTM	Runtime Resource Manager
SB	circuit switched Switch Box
SEC	Single Error Correction
Ser	Serial

SEU	Single Event Upsets
TMR	Tripple Modular Redundancy
VA	Voltage Adaptive
VBUS	Vertical BUS
VCU	Voltage Control Unit
VI	Vertical Index
WLAN	Wireless LAN
VRAP	Virtual Runtime Adaptive Partitions

Chapter 1

Introduction

1.1 Trends and developments

In this section, this thesis will explain various trends and challenges faced by the digital design industry that prompted this thesis. Since commercial production of integrated circuits started in the early 1960s, the increasing speed and performance requirements of the applications have driven the designers to manufacture increasingly smaller transistors. The smaller transistors enhance performance by allowing to embed additional silicon on a chip and increase the operating frequency. As shown in Table 1.1, the reduction in transistor sizes has followed *More's law*, which predicts that on-chip transistor density doubles every 18 to 24 months.

Table 1.1: Circuit size from 1963-2010 [2].

Year	Integration Level	Transistor Count
1963	Small Scale Integration (SSI)	< 100
1970	Medium Scale Integration (MSI)	100-300
1975	Large Scale Integration (LSI)	300-30000
1980	Very Large Scale Integration	(VLSI) 30000-1 million
1990	Ultra Large Scale Integration	(ULSI) $>$ 1 million
2010	Giga Scale Integration (GSI)	$>$ 1 billion

1.1.1 Power wall

With the arrival of 3D integration, the Moore's law continues to offer exponential increases in transistor count per unit area [120]. However, the *power wall* limits the maximum allowable transistor frequency. The issue of power wall arises because the power consumed by a chip operating at voltage V and frequency F is given by $Power = QFCV^2$. Where C and Q are respec-

tively the capacitance and the activity factor. The formula simply states that an increase in voltage increases the power consumption (and therefore the activity factor) exponentially. Since the maximum allowable frequency is dependent on the operating voltage, high frequency chips require expensive cooling methods. Therefore, to meet the performance requirements, the industry opted parallelism instead of increasing the chip frequency. This trend can be seen in the processor generation shown in Table 1.2. It can be seen from the table that the processor speeds increased till approximately 3GHz but after that the industry has started to focus on exploiting parallelism to enhance performance.

Table 1.2: Processor frequencies different generations [52].

Year	Model	Process	Clock	Transistor Count
1993	Pentium	0.8um	66 MHz	3.1 million
1995	Pentium Pro	0.6um	200 MHz	5.5 million
1997	Pentium II	0.35um	300 MHz	7.5 million
1999	Pentium III	0.25um	600 MHz	9.5 million
2000	Pentium IV	0.18um	2 GHz	42 million
2005	Pentium D	90nm	3.2 GHz	230 million
2007	Core 2 Duo	65nm	2.33 GHz	410 million
2008	Core 2 Quad	45nm	2.83 GHz	820 million
2010	Six-Core Core i7-970	32nm	3.2 GHz	1170 million
2011	10-Core Xeon	32nm	2.4 GHz	2600 million

1.1.2 Utilization wall and Dark silicon

Utilization wall [120] is a recent concept in digital design industry that limits the usable transistors on chip. It states that even with constant voltage and frequency a dense chip will consume additional power (i.e. even at same voltage, a 20 nm chip will consume more power than a 65 nm chip). As a consequence, in future designs the power and thermal limits will allow only a portion chip to operate full throttle (voltage and energy). To understand this problem, consider Table 1.3 [120]. The table shows how transistor properties change with each process generation, where S is the scaling factor. e.g. for shifting from a $45nm$ to a $32nm$ process generation, $S = 45/32 = 1.4$. The table distinguishes the factors that governed the transistor properties before and after 2005. In pre 2005 era (also called Dennard scaling era), it was possible to simultaneously scale the threshold and the supply voltage. In this era the transistor properties were governed by *Dennards Scaling* which implies that power consumption is proportional to the area of a transistor. In the post 2005 period (post Dennard scaling era), the

threshold or supply voltage could no longer be easily scaled without causing either exponential increases in leakage or transistor delay [120]. The table shows that as the number of transistors increases by S^2, their frequency increases by S, and their capacitance Q decreases by $1/S$. The Dennard/post Dennard Scaling eras differ in supply voltage V_{DD} scaling (under Dennard scaling, V_{DD} goes down by $1/S$, but in the post Dennard scaling era, V_{DD} remains fixed because the threshold voltage V_t cannot be scaled). When scaling down to the next process generation, the change in a design power (δ P) is given by $\delta P = \delta QFCV_{DD}$ (with additional squaring for the V_{DD} term). Therefore, while the Dennard scaling promised constant power when migrating between process generations, since 2005 power increases by S^2. For future designs it is predicted that heat dissipation (resulting from additional power) will be significant to burn the device [90]. It is predicted that in future the power and thermal limits will allow only a portion of the chip to remain operational, leaving a significant fraction left unpowered, or dark. This phenomenon known as dark silicon. As a consequence of dark silicon, with every process generation, the amount of usable transistors will decrease. To deal with the dark silicon era, architectural, algorithmic, and technological solutions are needed to efficiently utilize the on-chip resources.

Table 1.3: The origin of utilization wall [120].

Transistor property	Dennard Scaling era	Post Dennard scaling era
δ Density	S^2	S^2
δ Frequency	$\approx S$	$\approx S$
δ Capacitance	$1/S$	$1/S$
δ V_{DD}^2	$1/S^2$	≈ 1
δ Power=δ $QFCV_{DD}^2$	1	S^2

1.1.3 Fault-tolerance becoming critical

Every new process generation is marked by smaller feature size, lower node capacitance, higher operating frequency, and low voltage. These properties enhance performance, lower the power consumption, and allow to make smaller embedded chips. However, these properties affect the noise margins and amplify susceptibility to faults. It is therefore predicted that the number of on-chip faults will increase as technology scales further into the nano-scale regime, making fault tolerance a critical challenge of future designs [17, 99, 53].

1.2 Problem statement

From the discussion above, three conclusions can be drawn: (i) the power wall has forced the industry to opt for parallelism (since parallelism allows to perform the same task at lower frequency/voltage), (ii) the utilization wall makes dark silicon a critical consideration for future designs, necessitating the use of efficient runtime power management techniques and customizable hardware, and (iii) the small feature sizes has made variability an essential consideration for contemporary digital designs. All these trends make efficient resource management an essential challenge. The future platforms will host multiple applications with arbitrary communication/computation patterns, power budgets, reliability requirements, and performance deadlines. For these scenarios, compile time static decisions are sub-optimal and undesirable. Unlike the classic resource managers [94], that handled only physical component (like memory and computational units), the next generation resource managers should also manipulate additional performance/cost metrics like reconfiguration, reliability, voltage, and frequency) to get the maximum chip performance. To solve this challenge requires a framework based on theoretical foundations. The framework should simultaneously address the algorithms, the architecture, and the implementation issues for simultaneously managing the physical and abstract on-chip components. This thesis presents a systematic approach to design next generation resource managers. The approach is called Virtual Runtime Adaptive Partitions (VRAP). The proposed approach (i.e. VRAP) is based on virtualization and it provides a framework that allows each application to enjoy the operating point, reliability, and configuration infrastructure tailored to its needs.

1.3 Background

Efficient resource management (to optimize e.g. power, resource utilization) in the prevailing research trends (dark silicon era, fault-tolerance considerations, platforms hosting multiple applications), necessitates the use of a resource manager that can not only dynamically allocate and reclaim physical but also manipulate the performance and cost metrics such as voltage, frequency, reliability, and configuration architecture. To achieve these goals, the resource manager should provide various services such as configuration optimization, power optimization, and adaptive fault-tolerance. Existing works deal with these goals and services separately. Figure 1.1 highlights the various components of a resource manager and the implementation alternatives chosen by the researchers.

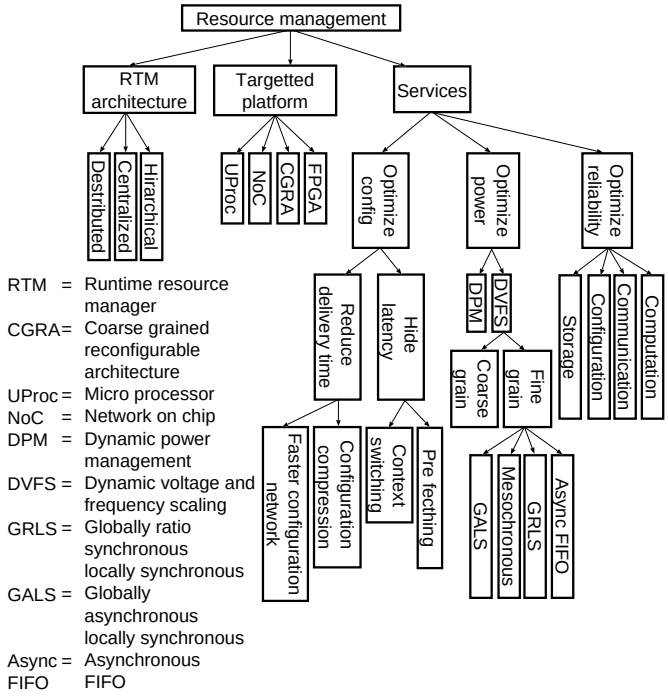

Figure 1.1: Resource management taxonomy

1.3.1 Services

In this section, a briefly explanation of various services provided by the proposed resource managers will be presented. Our discussion will cover three categories: (i) power optimization, (ii) configuration optimization, and (iii) reliability optimization.

Power optimization

Power optimization constitute techniques directly targeted towards reducing energy/power. Broadly, the power optimization techniques can be classified as dynamic voltage and scaling (DVFS) and dynamic power management (DPM) [15]. DVFS exploits the fact that voltage and frequency have conflicting impact on the power consumption. It scales the voltage and frequency to meet the application requirements. DVFS reduces dynamic power. Most recent surveys on DVFS can be found in [69, 19]. DPM switches off the part of the device that is free. It reduces the static power consumption. Depending on the granularity of power management, DVFS can range from coarse-grained to fine-grained. *Coarse-grained DVFS*, scales the operating point of entire platform for the application needing maximum performance. *Fine-grained DVFS* offers better energy efficiency by allowing to modify the frequency/voltage of each resource separately [70]. However,

its realization is strongly challenged by factors such as voltage switching and synchronization overheads [33].

Reliability optimization

Fault tolerance will be an essential feature in future designs. However, in a platform that hosts multiple applications, each application can potentially have different reliability requirements (e.g. control information in many DSP applications require higher reliability than the data streams). In addition, the reliability needs of an application can also vary depending on the operating conditions (e.g. temperature, noise, and voltage etc.). Providing maximum (worst case) protection to all applications imposes high area and energy penalty. To cater this problem, *flexible reliability schemes* have been proposed [6, 5, 55], which reduce the fault-tolerance overhead by providing only the needed protection for each application. The flexible reliability schemes vary greatly depending on the component to protect (computation, communication and/or storage). Most of the existing research (on flexible reliability) that protects the computation, only support shifting between different levels of *modular redundancy*. In modular redundancy, an entire unit (e.g. ALU) is replicated, making it an expensive technique that costs at least twice energy and area overhead compared to the unprotected chip. To protect the communication and the memories, in addition to modular redundancy, the adaptive reliability schemes also employ low cost error detecting codes (EDCs) [55].

Configuration optimization

In modern platforms, the concurrency and communication patterns among applications is arbitrary. Some applications enjoy dedicated resources and do not require further reconfiguration. While other applications, share the same resources in a time-multiplexed manner, and thus require frequent reconfigurations. Additionally, a smart power management system might dynamically serialize/parallelize an application, to enhance energy efficiency by lowering the voltage/frequency operating point. This requires a reconfiguration architecture that is geared to dynamically and with agility reconfigure arbitrary partitions of a fabric instance. To address these requirements, concepts like configuration pre-fetching [35, 109, 88], context switching, configuration compression [34, 51, 45], and faster reconfiguration networks [129, 128, 58] have been proposed. While these techniques do solve the problem, they come at a considerable cost (i.e. they improve the agility at cost of space and vice-versa). An even bigger problem is that, they address the reconfiguration requirements of only a certain category of applications/algorithms.

1.3.2 Control Architecture

Control architecture is responsible for monitoring and managing various components of a device (a CGRA or NoC our case). The resource mangers with centralized control architecture enjoy high efficiency since they monitor the entire platform centrally and make decisions accordingly [26, 113]. However, the centralized managers suffer from a single point of failure, larger volume of monitoring (resources and resource states), and central point of communication (between the manager and the hosted resources)and therefore are not scalable. To make the control architecture scalable, distributed control architectures were proposed [4]. The distributed controllers monitor only a part of device. They assume that by optimizing each portion of the device separately, the entire platform will be optimized. However, in this approach (also termed as greedy approach) the efficiency badly suffers, since the distributed units are unaware of the platform state. As a trade off between scalability (provided by the distributed resource managers by reducing the communication hot spots) and efficiency (provided by the centralized resource managers due to the availability of system level information), the recent works propose on hierarchical control architectures [27]. In these architectures, the basic control is distributed but the distributed blocks are also allowed to communicate with each other. The coordination allows them to optimize even at system level.

1.3.3 Implementation Platforms

The ASIC or fully customized designs are extremely efficient in terms of area, energy and power. However, the entire design flow is costly in terms of time design time, effort, and manufacturing cost. Furthermore, since ASICs are usually designed to support only a single application under specific conditions, a separate ASIC is needed for every application hosted by a chip. Software approach allows to use the same processor for implementing any function using the load store architecture, and thereby reduce design time and design effort. However, the load store architecture is slow since it does not allow to create specialized data paths provided by the ASIC implementation. To tackle these problems, the digital design industry has taken two paths: (i) increase the ASIC flexibility and (ii) increase the processor performance.

Increasing ASIC flexibility

The increase in the ASIC flexibility was achieved by devices such as Programmable Logic Arrays (PLAs), Field Programmable Gate Arrays (FPGAs), and Coarse Grained Reconfigurable Architectures (CGRAs). PLAs were first devices that introduced flexibility in ASICs. They allowed to

implement any logic function using configurable *AND* planes linked to programmable *OR* gate planes. However, once configured, they could not be reprogrammed. To tackle this problem, the SRAM based FPGAs (with virtually infinite reconfiguration cycles) were introduced. To realize a logic function the FPGAs store its implementation in a look up table. The look up table based implementation is costly in terms of configuration memory, area, power and energy consumption. Initially, the FPGAs were solely used for prototyping. Since the last decade, fueled by the demands of high performance of multimedia and telecommunication applications, coupled demand for low non recurring engineering and time to market FPGAs are now increasingly used to implement actual designs. However, since FPGAs are slower than ASICs they fail to meet the high performance requirements of modern applications. To meet the high performance requirements the idea of coarse grained reconfigurable architectures was proposed [132]. CGRAs enhance silicon and power efficiency by implementing commonly used processing elements (e.g. ALUs, multipliers, FFTs etc.) in hardware.

Increasing processor performance

To enhance the processor performance, the initial approach was to increase its clock speed. However, as explained in Section 1.1.1, due to the power wall, the computing industry took an irreversible transition towards parallelism since 2005. As a result, today the performance is achieved by integrating a number of smaller processors.

On the basis of the architectural characteristics, Figure 1.2 [132] depicts various platforms. The figure shows that both the approaches are slowly coming closer together. For performance improvements in software implementations, single core powerful processor has given way to simpler many processor systems. To enhance the flexibility in hardware solutions the PLAs gave way to FPGAs. To enhance the performance, the coarse grained architectures (as an alternative to FPGAs), have been a subject of intensive research since the last decade [112]. Major FPGAs manufacturers (Xilinx and Altera) already integrate many coarse-grained components (like DSPs) in their devices. It is expected the performance requirements will derive the industry to devote a significant percentage of device area for coarse grained components. Considering these trends, CGRAs and network on chips (NoCs) have been chosen, as candidate platforms, to test the efficacy of the proposed VRAP framework.

1.4 Objectives and methods

To cope with the current and future design challenges, this thesis presents a novel design paradigm called Virtual Runtime Adaptive Partitions (VRAP).

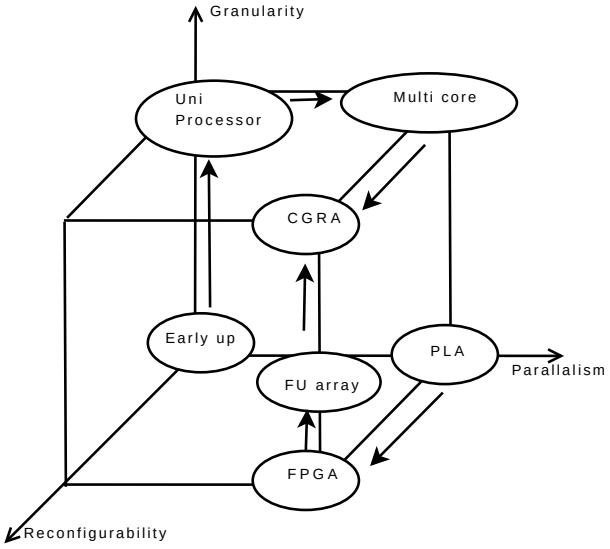

Figure 1.2: Direction for future platforms

Figure 1.3 illustrates goals, resources (both abstract and physical), and services needed to realize the next generation resource managers. The main goal of our methodology is to meet the application requirements (i.e. deadlines, reliability, power budget), on a flexible platform, with the overheads close to its customized implementation. The proposed resource management paradigm incorporates algorithms, hardware, and the architectural locks/switches to provide each application with only the resources essential to meet it deadlines and minimize energy.

A generic approach to realize the proposed architecture is shown in Figure 1.4. To make the problem manageable, this thesis have divided the framework into three phases: (i) private configuration environments (PCE), (ii) private reliability environments (PRE), and (iii) private operating environments (POE). PCE deals with the hardware/software necessary to implement a configurable reconfiguration architecture. PRE investigates the architectural switches needed to realize adaptive reliability. POE explores the architecture needed to manipulate the voltage and frequency for reducing the power consumption. It should be noted that the three environments chosen for this thesis are for proof of concept. Additional optimization criteria e.g. private thermal environments can also be merged in the VRAP framework.

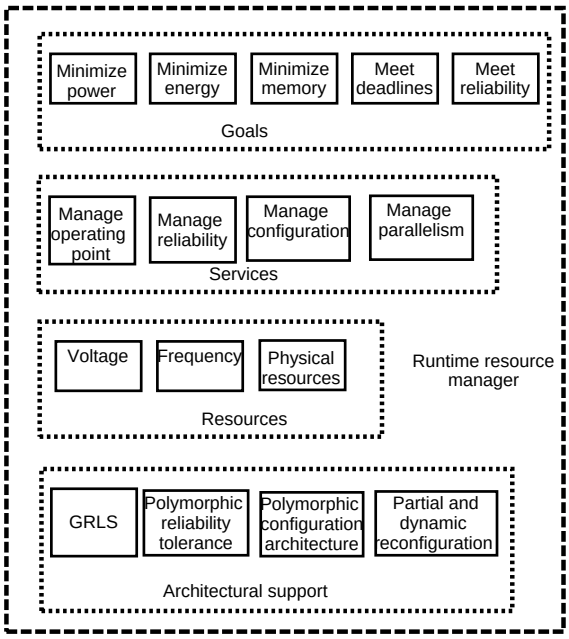

Figure 1.3: Goals, resources, services and architectural support for resource management

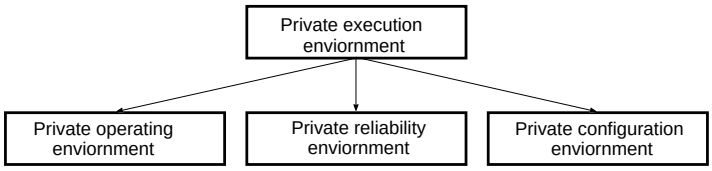

Figure 1.4: Private configuration environments approach

1.5 Contributions

Since the VRAP framework is implemented in three stages, the contributions will also be described in three parts.

1.5.1 Private Configuration Environments

This thesis proposes private configuration environments for CGRAs. The configuration infrastructure is developed in two stages: (i) an efficient and agile configuration architecture development and (ii) the enhancement of scratchpad memory to implement Private Configuration Environments (PCE).

To design efficient and agile configuration mechanism, the thesis combines LZSS compression with *RowMultiC* that minimizes the configware transfers to DRRA configuration memory. The obtained results, using a few applications, suggest that the proposed method has a negligible penalty in terms of area (1.2%), while provides a significant reduction in the configuration cycles (up to 78%) and energy (up to 94%) required to configure DRRA. To further reduce the configuration cycles, this thesis also presents a technique to compactly represent multiple bitstreams, corresponding to different application implementations (with different degree of parallelism). The compact representation is unraveled at runtime. The simulation results, using FFT with three versions (with different degree of parallelism), revealed that the CGIR saves an additional 18% memory for 2 versions and 33% memory for 3 versions.

After developing the reconfiguration mechanism, the thesis also presents an on-demand reconfiguration. On-demand reconfiguration relies on a morphable data/configuration memory, supplemented by morphable hardware. By configuring the memory and the hardware, the proposed architecture realizes four configuration modes: (i) direct feed, (ii) direct feed multi-cast, (iii) direct feed distributed, and (iv) multi context. The obtained results suggest that significant reduction in memory requirements (up to 58 %) can be achieved by employing the proposed morphable architecture. Synthesis results confirm a negligible penalty (3 % area and 4 % power) compared to a DRRA cell.

1.5.2 Private Reliability Environments

The thesis proposes private reliability environments for both CGRAs and NoCs. For CGRAs, this thesis presents an adaptive fault-tolerance mechanism to provides the on-demand reliability to multiple applications. To provide on-demand fault-tolerance, the reliability requirements of an application are assessed upon its entry. Depending on the assessed requirements, one of the five fault-tolerance levels are provided: (i) no fault-tolerance, (ii) temporary fault detection, (iii) temporary/permanent fault detection, (iv)

temporary fault detection and correction, or (v) temporary/permanent fault detection and correction. In addition to modular redundancy (employed in the state-of-the-art CGRAs offering flexible reliability levels), this thesis presents the architectural enhancements needed to realize sub-modular, residue mod 3 redundancy. The residue mod 3 coding allows to reduce the overhead of the self-checking and fault-tolerant versions by 57% and 7%, respectively. The polymorphic fault-tolerant architecture is complemented with a morphable scrubbing technique to prevent fault accumulation. The obtained results suggest that the on-demand fault-tolerance can reduce energy consumption up to 107%, compared to the highest degree of available fault-tolerance (for an application needing no fault-tolerance).

For NoCs, this thesis presents an adaptive fault tolerance mechanism, capable of providing the on-demand protection to multiple traffic classes. On-demand fault tolerance is attained by passing each packet through a two layer, low cost, class identification circuitry. Upon identification, the packet is provided one of the four fault tolerance levels: (i) no fault tolerance, (ii) end to end DEDSEC, (iii) per hop DEDSEC, or (iv) per hop DEDSEC with permanent fault detection and recovery. The obtained results suggest that the on-demand fault tolerance incurs a negligible penalty in terms of area (up to 5.3%) compared to the fault tolerance circuitry, and provides a significant reduction in energy (up to 95%), compared to state of the art.

1.5.3 Private Operating Environments

Private operating environments are presented for both CGRA and NoC. In CGRA domain, this thesis presents the architecture and implementation of energy aware CGRAs. The proposed architecture promises better area and power efficiency, by employing Dynamically Reconfigurable Isolation Cells (DRIC)s and Autonomous Parallelism Voltage and Frequency Selection algorithm (APVFS). The DRICs utilize reconfiguration to eliminate the need for most of the dedicated hardware, required for synchronization, in traditional DVFS techniques. APVFS ensures high energy efficiency by dynamically selecting the application version which requires the minimum frequency/voltage to meet the deadline on available resources. Simulation results using representative applications (Matrix multiplication, FIR, and FFT) showed up to 23% and 51% reduction in power and energy, respectively, compared to traditional designs. Synthesis results have confirmed significant reduction in DVFS overheads compared to state of the art DVFS methods.

In NoC domain, this thesis presents the design and implementation of a generic agent-based scalable self-adaptive NoC architecture to reduce power. The system employs dual-level agents with SW/HW co-design and synthesis. The system agent is implemented in software, with high-level instructions

tailored to issue adaptive operations. The effectiveness and the scalability of the system architecture is demonstrated using best-effort dynamic power management, using distributed DVFS. The experiments revealed that the adaptive power management saved up to 33% energy and up to 36% power. The hardware overhead of each local agent is only 4 % of a router area.

1.6 Research publications and contributions

Overall, the thesis has resulted in 22 accepted peer-reviewed international publications (4 ISI-Indexed journals and 18 conference papers). In addition, 2 ISI-Indexed Journal and 2 conference papers are submitted for review.

This monograph is based on the following publications.

Accepted Journal Publications

1. Syed M. A. H. Jafri, Liang Guang, Ahmed Hemani, Kolin Paul, Juha Plosila, Hannu Tenhunen: Energy-aware fault-tolerant NoCs addressing multiple traffic classes, in Microprocessors and Microsystems- Embedded Hardware Design. 2013. In press. doi:dx.doi.org/10.1016/j.micpro.2013.04.005.
   Authors Contribution The author proposed the idea to provide different reliability level to different traffic classes while using the hierarchical agent based framework developed by Liang. Compared to the conference version of this paper, the author also designed an inter-agent communication protocol. The Author performed all the experiments and also wrote most of the manuscript. The other authors provided guidance and supervision.

2. Syed M. A. H. Jafri, Stanislaw Piestrak, Oliver Sentieys, and Sebestien Pillement: Design of Coarse Grained Reconfigurable Architecture DART with Online Error Detection, in Microprocessors and Microsystems-Embedded Hardware Design. 2013. In press. doi:dx.doi.org/10.1016/j.micpro.2013.12.004.
   Authors Contribution The author designed and evaluated the residue mod 3 for the CGRA DART, while Prof. Stanislaw came up with the idea to protect DART using residue mod 3. In addition, he also suggested pipelining to eliminate the timing overheads incurred by the conference version of this paper.

3. Syed M. A. H. Jafri, Stanislaw Piestrak, Kolin Paul, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Private reliability environments for efficient fault-tolerance in CGRAs, Springer Design Automation for Embedded Systems. 2013. In press.
   Authors Contribution The author proposed and designed an adaptive version of Residue mod 3 to provide efficient fault-tolerance for

mixed criticality application in CGRAs. On addition, the author presented the architectural modifications needed to realize adaptive scrubbing in CGRAs. Prof. Stanislaw and Assoc. Prof. Kolin provided the essential related work in the field. The other coauthors provided supervision and helped in the manuscript preparation.

4. Nasim Farahini, Ahmed Hemani, Hasan Sohofi, Syed M. A. H. Jafri, Muhammad Adeel Tajammul, Kolin Paul: Parallel Distributed Scalable Address Generation Scheme for a Coarse Grain ReconïňĄgurable Computation and Storage Fabric, Submitted to Microprocessors and Microsystems- Embedded Hardware Design (Accepted)
Authors Contribution The author evaluated the effect of various compression methods on the hardware presented by Nasim.

Accepted Conference Publications

5. Syed M. A. H. Jafri, Guillermo Serrano Leon, Masoud Daneshtalab, Ahmed Hemani, Kolin Paul, Juha Plosila, Hannu Tenhunen: Transformation Based Parallelism for low power CGRAs, Field programmable logic (FPL) 2014 (Accepted). **Authors Contribution** The author proposed the idea to provide hardware transformation based parallelism, rather than storing multiple versions. Bachelor student Guillermo, wrote VHDL code of the transformer and performed the experiments. The other authors provided guidance and supervision.

6. Syed M. A. H. Jafri, Masoud Daneshtalib, Muhammad Adeel Tajammul, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Cascaded compression architecture for efficient configuration in CGRAs, International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2014 (Accepted). **Authors Contribution** The author, and Prof. Hemani proposed the idea to combine various compression techniques into a single architecture. The author wrote most of the paper and performed most of the experiments. Adeel mapped various versions of FFT to conduct the experiments.

7. Syed M. A. H. Jafri, , Guillermo Serrano Leon, Junaid Iqbal, Masoud Daneshtalab, Ahmed Hemani, Kolin Paul, Juha Plosila, Hannu Tenhunen: RuRot: Run-time Rotatable-expandable Partitions for Efficient Mapping in CGRAs International Conference on Embedded Computer Systems: Architecture, Modeling and Simulations (SAMOS) 2014 (Accepted). **Authors Contribution** The author proposed the idea to provide hardware based dynamic remapping in CGRAs and wrote most of the paper. Bachelor student Guillermo, wrote VHDL code of the mapper and performed the experiments.

8. Syed M. A. H. Jafri, Masoud Daneshtalab, Muhammad Adeel Tajammul, Kolin Paul, Ahmed Hemani, Peeter, Ellervee, Juha Plosila, Hannu Tenhunen: Morphable compression architecture for efficient configuration in CGRAs, Euromicro conference on Digital System Design (DSD) 2014 (Accepted). **Authors Contribution** The author, and Prof. Hemani proposed the idea to cascade various compression techniques. The author wrote most of the paper and performed most of the experiments. Adeel mapped various versions of FFT to conduct the experiments.

9. Syed M. A. H. Jafri, Stanislaw Piestrak , Kolin Paul, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Implementation and evaluation of configuration scrubbing on CGRAS: A case study, in Proc. International symposium on System on Chip , pp 1-8 Oct 2013.
Authors Contribution The author designed and evaluated architecture for configuration scrubbing the CGRA DRRA. Prof. Stanislaw and Prof. Kolin provided essential related work in this field.

10. Syed M. A. H. Jafri, Stanislaw Piestrak, , Kolin Paul, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Energy-Aware Fault-Tolerant CGRAs Addressing Application with Different Reliability Needs, in Proc. Euromicro Conference on Digital System Design (DSD), pp 525-534, Sept 2013.
Authors Contribution The author designed and evaluated the residue mod 3 for the CGRA DRRA. In addition the author also proposed a method to adapt the reliability level provided by the system at runtime. The remaining co-authors provided essential guidance.

11. Syed M. A. H. Jafri, Muhammad Adeel Tajammul, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Energy aware task parallelism for efficient dynamic voltage and frequency scaling in CGRAs, in Proc. International Conference on Embedded Computer Systems: Architecture, Modeling and Simulations (SAMOS), 2013, 104-112.
Authors Contribution The author designed and evaluated runtime task parallelism on CGRA DRRA. Adeel programmed the applications to conduct the experiments. The remaining coauthors provided essential guidance.

12. Muhammad Adeel Tajammul, Syed M. A. H. Jafri, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Private configuration environments for efficient configuration in CGRAs, in Proc. Application Specific Systems Architectures and Processors (ASAP), 2013, 227-236.
Authors Contribution The author designed the architecture to configure DRRA from the Global configuration memory (that can be con-

nected externally to DRRA) while Adeel designed framework to configure the DRRA using internal scratchpad memory. The author also wrote most of the manuscript and performed the simulations.

13. Syed M. A. H. Jafri, Ozan Ozbak, Ahmed Hemani, Nasim Farahini, Kolin Paul, Juha Plosila, Hannu Tenhunen: Energy-Aware CGRAs using Dynamically Re-configurable isolation Cells, in Proc. International Symposium for Quality and Design (ISQED), 2013, 104-111
Authors Contribution The author designed the architecture to implement DVFS in CGRA DRRA. The implementation was done by Ozan. The idea to use Dynamically Re-configurable isolation Cells was provided by Prof. Hemani. The manuscript was also written by the Author.

14. Syed M. A. H. Jafri, Liang Guang, Ahmed Hemani, Kolin Paul, Juha Plosila, Hannu Tenhunen: Energy-aware fault-tolerant NoCs addressing multiple traffic classes, in Proc. Euromicro Conference on Digital System Design (DSD), 2012, 242-249.
Authors Contribution The author proposed the idea to provide different reliability level to different traffic classes while using the hierarchical agent based framework developed by Liang. The Author performed all the experiments and also wrote most of the manuscript. The other authors provided guidance and supervision.

15. Syed M. A. H. Jafri, Liang Guang, Axel Jantsch, Kolin Paul, Ahmed Hemani, Hannu Tenhunen: Self-adaptive Noc Power Management with Dual-level Agents - Architecture and Implementation. Proc. Pervasive and Embedded Computing and Communication Systems (PECCS), 2012, pp 450-458.
Authors Contribution The author implemented and evaluated most of the agent based framework proposed by Liang on McNoC platform. Liang wrote most of the manuscript.

16. Syed M. A. H. Jafri, Ahmed Hemani, Kolin Paul, Juha Plosila, Hannu Tenhunen: Compact generic intermediate representation (CGIR) to enable late binding in coarse grained reconfigurable architectures. In Proc. International Conference on Field Programmable Technology (FPT), 2011: 1-6.
Authors Contribution The author designed and implemented the compression algorithm to compactly represent the configware for DRRA. Assoc. Prof. Kolin and Prof. Hemani provided the main idea. The author wrote most of the manuscript.

17. Syed M. A. H. Jafri, Ahmed Hemani, Kolin Paul, Juha Plosila, Hannu Tenhunen: Compression Based Efficient and Agile Configuration Mech-

anism for Coarse Grained Reconfigurable Architectures. In Proc. International Symposium on parallel and distributed processing workshops (IPDPSW), 2011: 290-293.
Authors Contribution The author designed and implemented the configuration mechanism for DRRA. Assoc. Kolin and Prof. Hemani provided the idea of using LEON3 for the work. The author also wrote most of the manuscript.

18. Syed M. A. H. Jafri, Stanislaw J. Piestrak, Olivier Sentieys, SÃľbastien Pillement: Design of a fault-tolerant coarse-grained reconfigurable architectures: A case study. in Proc. International Symposium for Quality and Design (ISQED), 2010: 845-852.
Authors Contribution The author designed and evaluated the residue modulus 3 for the CGRA DART, while Prof. Stanislaw came up with the idea to protect DART using residue mod 3 codes.

Accepted papers not included in this thesis

19. Liang Guang, Syed M. A. H. Jafri , Tony Yang, Juha Plosila and Hannu Tenhunen: Embedding Fault-Tolerance with Dual-Level Agents in Many-Core Systems, in Proc. Workshop on Manufacturable and Dependable Multicore Architectures at Nano Scale (MEDIAN 2012).

20. Liang Guang, Syed M. A. H. Jafri, Bo Yang, Juha Plosila Hannu Tenhunen: Hierarchical Supporting Structure for Dynamic Organization in Many-Core Computing Systems. Proc. Pervasive and Embedded Computing and Communication Systems (PECCS) , pp.252-261, 2013

21. Syed M. A. H. Jafri, Tuan Nguyen, Masoud Daneshtalab, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: NeuroCGRA: A CGRA with support for neural networks Accepted for Publication in Proc. Dynamically Reconfigurable Network on Chip (DrNoC) 2014.

22. Hassan Anwar, Syed M. A. H. Jafri, Masoud Daneshtalab, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Exploring Neural Networks on CGRAs. Accepted for Publication in Proc. MES 2014

Submitted Journal Publications

23. Syed M. A. H. Jafri, Muhammad Adeel Tajammul, Ahmed Hemani, Juha Plosila, Hannu Tenhunen: Morphable conïĭňĄguration architecture to address multiple reconïĭňĄguration needs. IEEE Transaction on VLSI

24. Syed M. A. H. Jafri, Ozan Ozbak, Ahmed Hemani, Nasim Farahini, Kolin Paul, Juha Plosila, Hannu Tenhunen: Architecture and Implementation of Dynamic Parallelism, Voltage, and Frequency Scaling

(PVFS) on CGRAs. Submitted to ACM Journal of Emerging Technologies.

1.7 Thesis Navigation

Fig. 1.5 shows the thesis navigation. The figure contains core technical areas, chapters, proposed schemes, constituents, and publications together. The purpose of this figure is to aid the reader in understanding this thesis pictorially.

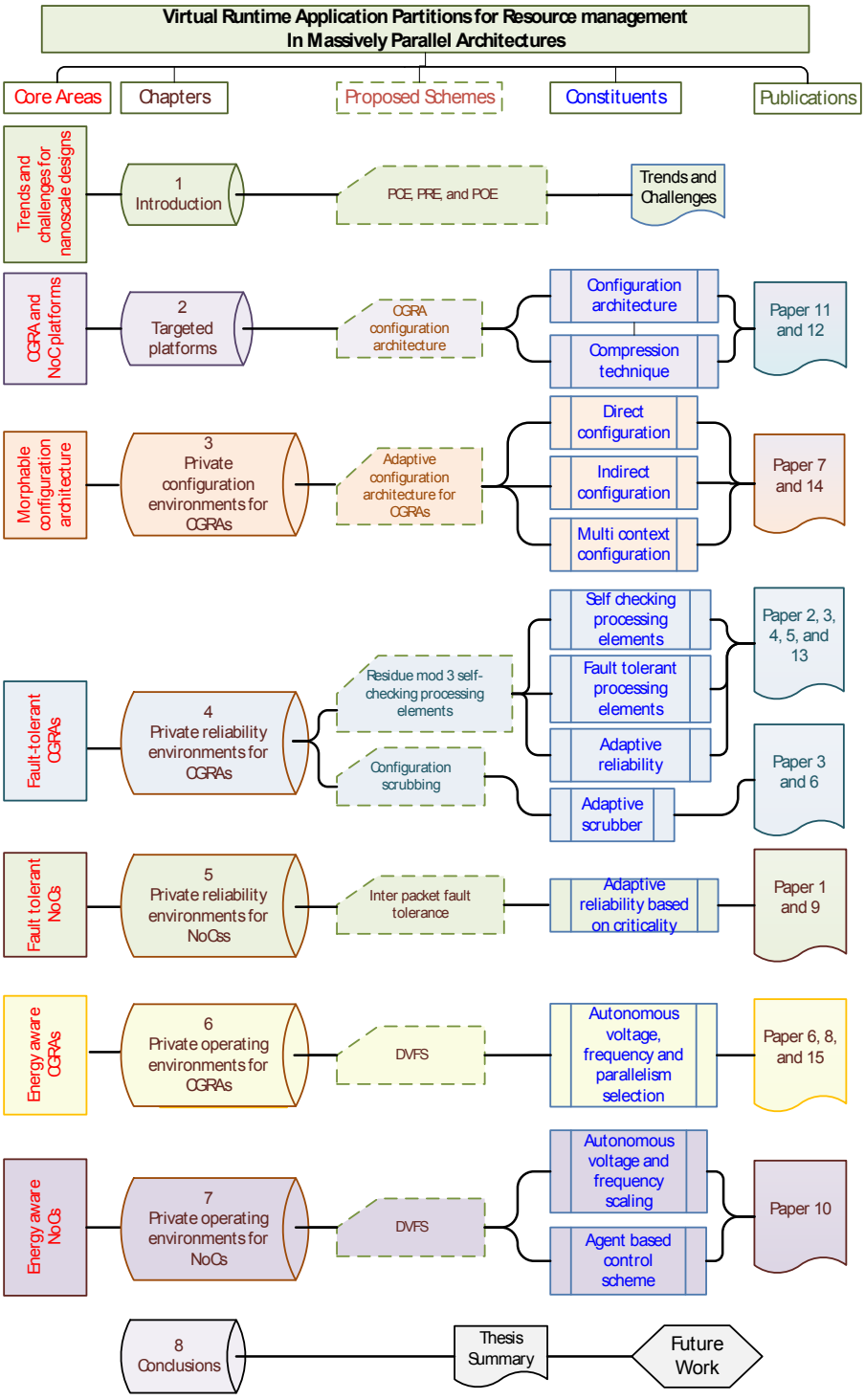

Figure 1.5: Navigation of the thesis

Chapter 2

Targeted platforms

In this chapter, will explain the experimental platforms used to test the efficacy of PREX framework. For this purpose, we have chosen Coarse Grained Reconfigurable Architectures (CGRAs) and Network on Chips (NoCs). The motivation for choosing the CGRAs and NoCs has already been given in Chapter 1.

2.1 DRRA before our innovations

Unlike FPGAs, contemporary CGRAs vary greatly in their architecture (i.e. computation, communication and storage), configuration scheme, energy consumption, performance, and reliability. Therefore, in the absence of a standard CGRA we had to isolate a platform to test the validity of our framework. For this thesis, we have chose Dynamically Reconfigurable Resource Array (DRRA) [111] due to three reasons: (i) we had available complete information about its architecture (from the RTL and the physical design), so that we could implement the proposed architectural modifications easily; (ii) DRRA has a grid based architecture, which is the most dominant design style for CGRAs, it therefore allowed us to compare our work with other CGRAs; and (iii) we had available a library for commonly used DSP function (containing FFTs, FIRs), allowing us to quickly map DSP applications and perform cost/benefit analysis of the proposed techniques on real world applications. DRRA is a dynamically reconfigurable coarse-grained architecture developed at KTH [110]. In this section, we will explain the DRRA architecture before our enhancements.

As depicted in Figure 2.1, DRRA is composed of two main components: (i) DRRA computation layer and (ii) DRRA storage layer (DiMArch). In Table 2.1, the functionality of these components is listed. DRRA computation layer performs the computations. DiMArch is a distributed memory fabric template that complements DRRA with a scalable memory archi-

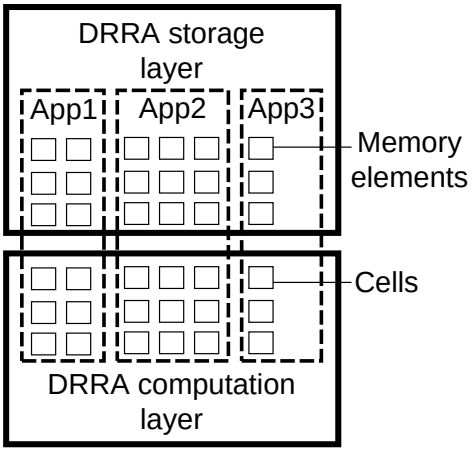

Figure 2.1: Different applications executing in its private environment

tecture. DRRA can host multiple applications, simultaneously. For each application, a separate partition can be created in the DRRA storage and computation layers.

Table 2.1: Local controller functionality

Component	Functionality
DRRA computation layer	Perform computations
DRRA storage layer	Store data for computations

2.1.1 DRRA computation layer

The computation layer of DRRA is shown in Figure 2.2. DRRA computational layer is divided into four components: (i) register files (Reg-files), (ii) morphable Data Path Units (DPUs), (iii) circuit switched interconnects , and (iv) sequencers organized in rows and columns. The register files store data for the DPUs that perform computations. Each register file contains two ports (port A and port B). Circuit switched interconnects provide interconnectivity between the different components of DRRA (DPUs, circuit switched interconnects, reg-files and sequencers). The sequencers hold the configware which corresponds to the configuration of the components (reg-files, DPUs, and circuit switched interconnects). Each sequencer stores up to 64 35-bit instructions and can configure elements the in same row and column as the sequencer itself. The configware loaded in the sequencers contains sequence of configurations required to perform an operation. To understand the process of configuration, consider for example that we want to add the contents of reg-file 0 ($row = 0, column = 0$) to the contents of

reg-file 1 ($row = 0, column = 1$), using DPU 1 ($row = 0, column = 1$) , and store the result to register file 2 ($row = 0, column = 2$). To configure the DRRA for this operation, 3 sequencers are required: (i) sequencer 0 containing one instruction to configure register file 0 (ii) sequencer 1 containing three instructions to configure reg-file 1, MDPU 1, and circuit switched interconnect 1 (iii) sequencer 2 containing two instructions to configure reg-file 2 and circuit switched interconnect 2. It should be noted that this example was just for illustrative purposes, we could have performed the same operation using only one sequencer by loading the inputs from different ports of same register file and then storing the result to the same register file.

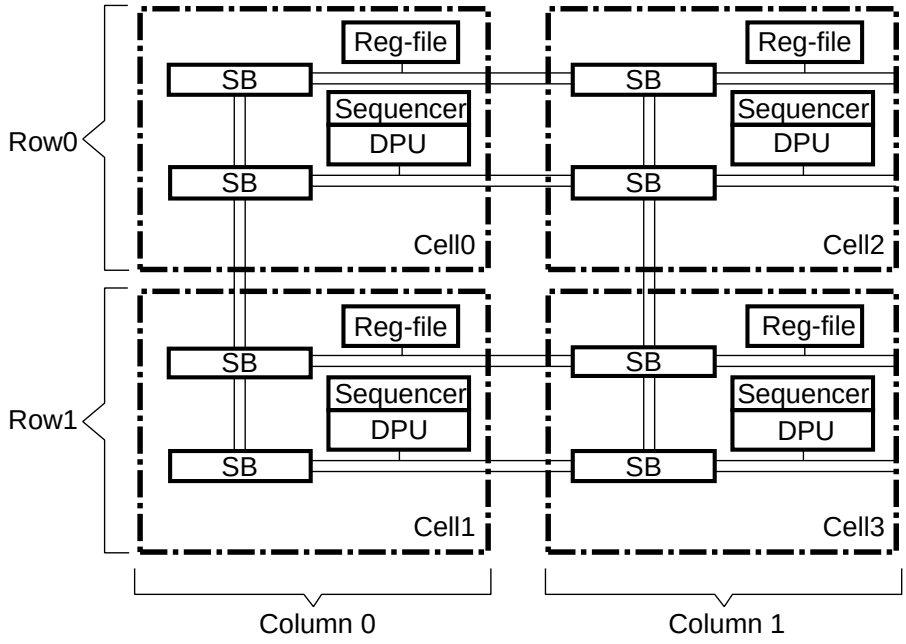

Figure 2.2: Computational layer of DRRA

2.1.2 DRRA Storage layer (DiMArch)

DiMArch is a distributed memory template that complements DRRA with a scalable memory architecture. Its distributed nature allows a high speed data access to the DRRA computational layer [118, 86]. DRRA was designed to host multiple applications with potentially different memory to computational ratio. To efficiently utilize the memory resources, DiMArch dynamically creates a separate memory partition for each application [118]. As shown in Fig. 2.3, DiMArch is a 2-dimensional array of *memory tiles*. Depending on their function, the tiles are classified into two types: (i) Configuration Tile (ConTile) and (ii) Storage Tile (STile). The memory tiles present in the row, adjacent to the DRRA computation layer, are called

ConTiles. The ConTiles manage all data transfers and contain five components: (i) SRAM, to store data for computational layer, (ii) an address generator to provide data from appropriate addresses, (iii) a crossbar, to handle data transfers between tiles, (iv) an Instruction Switch (iSwitch), to handle the transfer of control instructions between tiles [117], and (v) a DiMArch sequencer, to store the sequence in which data will be transferred to the DRRA computational layer. The memory tiles present in rows, non-adjacent to the DRRA computational layer, are called STiles. They are mainly meant for data storage and therefore do not contain the DiMArch sequencer.

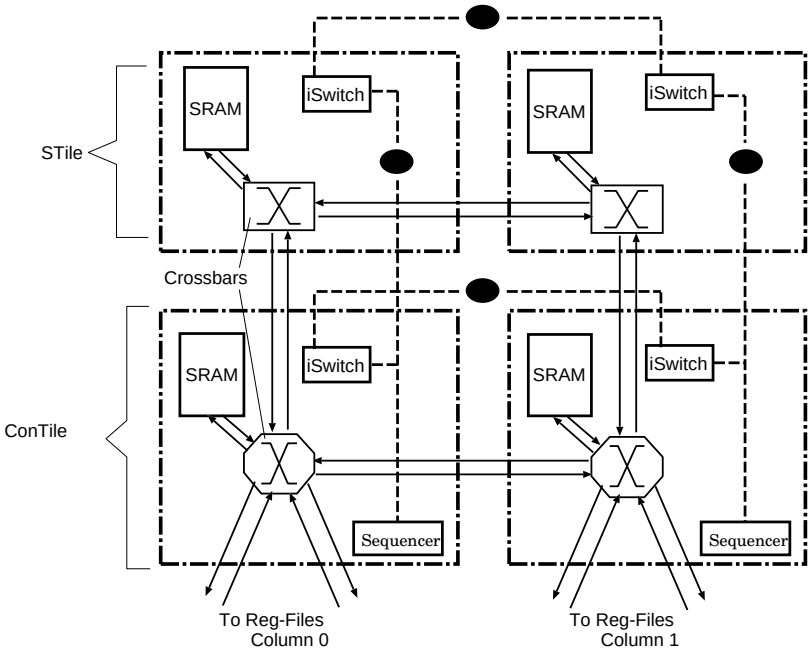

Figure 2.3: DRRA storage Layer

2.1.3 DRRA programming flow

Figure 2.4 depicts the programming flow of DRRA [58]. The configware (binary) for commonly used DSP functions (FFT, FIR filter e.t.c.) is written either in VESYLA (HLS tool for DRRA) and stored in a library. To map an application, its (simulink type) representation is fed to the compiler. The compiler, based on the available functions (present in library) constructs the binary for the complete application (e.g. WLAN).

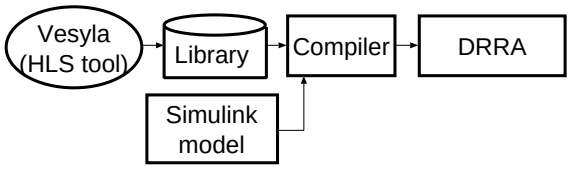

Figure 2.4: DRRA programming flow

2.2 Control and configuration backbone integration

When our thesis started, DRRA lacked a runtime reconfiguration and control mechanism. The tests were performed by manually feeding each sequencer with the machine code. To manage delivery of configware from the on chip memory to the sequencers in DRRA, we integrated a LEON 3 processor, as shown in Figure 2.5. The processor was connected to AHB bus, inspired from the architectures presented in [11, 43, 71]. The choice of using LEON 3 connected to AHB bus was dictated by the ease of implementation, power, and flexibility offered by this architecture. It should however be noted, that this architecture can be improved significantly by using direct memory access (DMA) and an advanced bus like AXI, but implementation of such an architecture is beyond the scope of this thesis. In our architecture, the LEON 3 processor delivers the configuration bitstream from the memory to the DRRA. The loader acts as an interface between the AHB bus and the DRRA fabric.

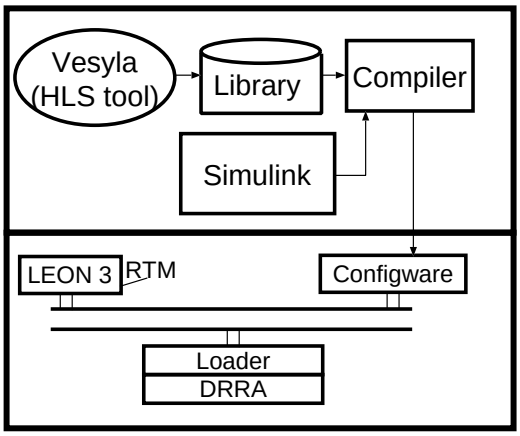

Figure 2.5: DRRA control and configuration using LEON 3

To configure DRRA efficiently, we have employed a multi-casting. Multicasting allows compression and ability to configure multiple components simultaneously [58]. To utilize these benefits, we modified the DRRA ad-

dressing scheme. In particular, we employed RowMultiC originally proposed in [123]. In the original DRRA addressing scheme, n bits required to program m components were $n = \lceil log(m)/log(2) \rceil$ bits. Each sequencer was assigned a unique identity ranging from 0 to $2^n - 1$. The address decoding was achieved by comparing the incoming address with the assigned identity. In multicasting, as shown in Figure 2.6, each sequencer is assigned a unique ID on the basis of its row and column number. Hence, the generated address contains 2 parts. The first part contains $r = number\ of\ rows$ and the second part contains c bits where $c = number\ of\ columns$. Hence, the overhead of implementing this scheme is $overhead_{total} = (r + c) - n$ bits. To address a sequencer, 1 is placed in the column and the row bits of the address. Multiple sequencers can be addressed by placing multiple 1s in the row or column positions. For decoding, the incoming address is compared with the assigned row and column number. If the corresponding row and column number of the sequencer is 1, then the device is programmed.

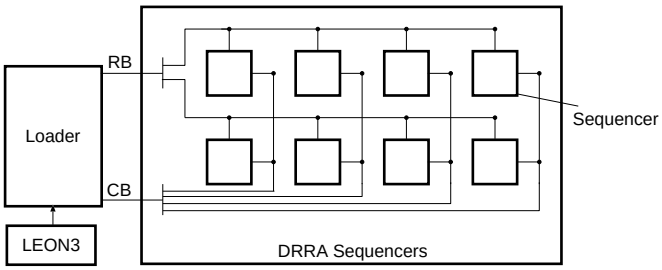

Figure 2.6: Multicasting architecture

2.3 Compact Generic intermediate representation to support runtime parallelism

One of the contributions of this thesis is to integrate runtime parallelism with conventional power management techniques. The runtime parallelism allows to make aggressive power management decisions and therefore enhance energy efficiency. Consider for example that N components execute a task in T seconds consuming E energy. For a perfectly paraleizable task $N * C$ components can perform the same task in T/C seconds. To reduce the energy, the voltage and frequency can be scaled down to reduce the energy efficiency while still meeting the deadlines. The support for runtime parallelism, was provided by using two phase method initially proposed in [128]. This two phase method has two phases: (i) offline and (ii) online. In the offline phase, different versions of each application, with different levels of parallelism are stored. At runtime, the most efficient version is mapped to the system. The two phase approach [128], however suffered

from prohibitive configuration memory requirements arising from the need to store multiple versions. Unfortunately, the need for extra memory increases linearly with the versions. To cater this problem, we presented a compression method, called *Compact Generic Intermediate Representation (CGIR)*. CGIR, instead of storing a separate binary for each version, stores a compact, unique, and customizable representation. To formalize the potential savings of our method, consider for example that, $A(i)$, bits are needed to map the i^{th} implementation of application A. Total bits needed to represent configware for each application, C_A, in two phase approach is given by equation

$$C_A = \sum_{i=1}^{v} A(i), \tag{2.1}$$

where v is the number of versions. Total bits needed to represent configware for each application, C_{AC}, in CGIR based approach is given by equation

$$C_{AC} = A(i_{max}) + \sum_{i=1}^{v} seq(i), \tag{2.2}$$

where $A(i_{max})$ is the version with maximum storage requirement and $seq(i)$ represents the sequences stored for each version. It was shown in [57] that $seq(i)$ represents only a small part (17% to 32%) of total implementation giving considerable overall savings when multiple versions are stored. In this section we will describe the method to develop CGIR from raw configware (hard binaries of different versions).

2.3.1 FFT example

To illustrate the self similarities among different versions, we have chosen 16-point DIT radix 2 FFT algorithm. For achieving various versions, we have used pipelined (cascaded) approach [54]. We have implemented 3 versions of FFT with one, two and four butterflies respectively. In Figure 2.7, we show the mapping of a complex FFT butterfly on DRRA. Each butterfly requires 4 DPUs and 4 reg-files. reg-file 0 and reg-file 2 hold the real and complex bitstreams, respectively. Twiddle factors are pre-stored in reg-file 1. DPU 0 and DPU 2 consume data from reg-file 0, reg-file 1, and reg-file 2 and feed the outputs to DPU 1 and DPU 3. DPU 1 and DPU 3 utilize this data along with the delayed version of input bitstream (stored in reg-file 3) and twiddle factors (stored in reg-file 2) to produce the final outputs.

A fully serial version (SV) containing a single butterfly is shown in Figure 2.8. The solid boxes indicate the sequencer numbers. The numbers in parentheses indicate row and column numbers, respectively. The dotted boxes containing four solid boxes constitute the butterfly shown in Figure

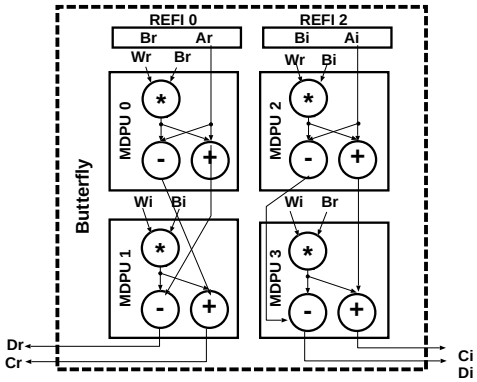

Figure 2.7: Mapping of each butterfly on DRRA fabric

2.7. A fully serial version, requires six sequencers (4 to configure the butterfly and 2 for storing the intermediate results). A partially parallel FFT version (PPV), is shown in Figure 2.9. It requires twelve sequencers. Eight sequencers store configware of MDPU, reg-file, and switch box for implementing the 2 butterflies and four additional sequencers are needed to store configware for reg-files, which hold intermediate results. A fully pipelined (cascaded) FFT version (PV), using 4 butterflies is shown in Figure 2.10. It requires 16 sequencers to store configware of MDPU, reg-file, and switch box for implementing the 4 butterflies.

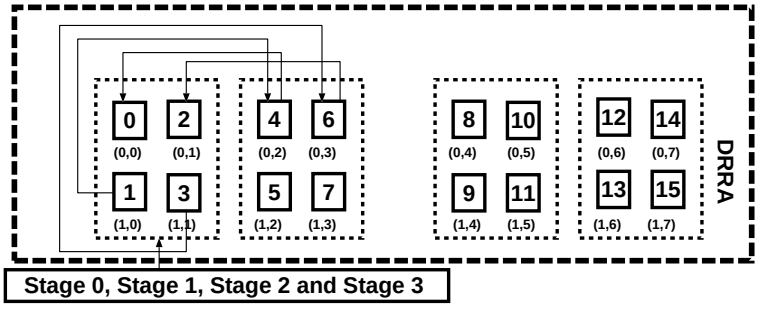

Figure 2.8: Fully serial FFT mapping on DRRA cells

2.3.2 Compact Generic Intermediate Representation

In this section we will describe the method to develop CGIR from raw configware (hard binaries of different versions).

Basic Block

To exploit the regularities among different versions, we introduce the terminology of *Basic Block* (BB). A BB is a piece of configware that performs

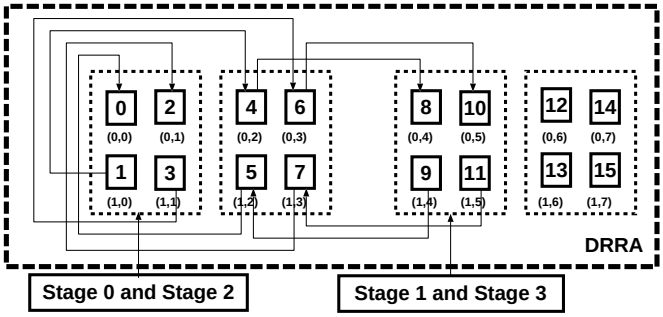

Figure 2.9: Serial parallel FFT mapping on DRRA cells

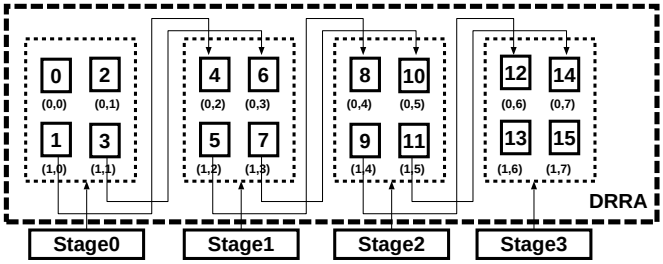

Figure 2.10: Fully parallel FFT mapping on DRRA cells

identical functions in all versions. A completely sequential implementation contains only a single BB. The number of BBs in a version depends on its level of parallelism. In the FFT example SV will have only one BB (implementing a single butterfly), while a PV will have 4 BBs (implementing 4 butterflies). A complete CGIR consists of BBs, interconnections between them, and some additional code for synchronization.

Effects of Parallelism on Basic Block Configware

Although, each BB is functionally identical, variations in configware of two BBs occur when parallelism is exploited using data parallelism or pipelining. For data parallelism, differences in configware arise due to the difference in the physical placement of BBs. For functional parallelism, the differences in configware occur from the differences in both the physical placement and the delay when each BB receives data.

For identical functions, the DPU instructions remain same regardless of the location of the BB on DRRA. Reg-file instructions are also location invariant, however, if dependencies exist like in the case of pipelining, each instruction has a different delay. Switch box instructions are sensitive to location. Simply put, reg-file instructions are delay sensitive, Switch box instructions are location sensitive, and DPU instructions are both delay and location insensitive. Therefore, the DPU instructions remain same in all

versions, the reg-file instructions in different versions differ only in delay, and the switch box instructions in various versions differ only in placement. Hence, instead of storing all the instructions, we store only delays for reg-file instruction and only placement information for switch box instructions.

Extra Code for Communication and Synchronization between BBs

Most of the compression possibilities arise from extracting regularities among BBs. However, some additional code is required for connecting and synchronizing these BBs. For simplicity, we have decided not to compress this part and store it as a hard binary.

CGIR Generation

In Figure 2.11, we have shown how the CGIR based representation is stored and unraveled. The extra code for communication and synchronization is stored as hard binary (it is not transformed). All the DPU instructions are also stored as hard binaries. However, since the DPU instructions are the same in all versions, compression is achieved by storing DPU instructions for a single BB. To create different versions, the same code is sent to different sequencers. In addition, if a version contains multiple BBs, configware for only one BB needs to be stored, and its copy is sent to different sequencers to achieve parallelism (we call it *internal compression*). The reg-file and switch box instructions for each BB are stored as intermediate representations. For reg-file instructions, the delay field is represented by a variable. A set of delay values for each version is stored separately. For switch box instructions, the location information is stored in two fields: (i) Hierarchical Index (HI) and (ii) Vertical Index (VI). Hence, the HI and VI fields are represented by a variable. A set of values for each version is stored separately (this storage is shown at the bottom of Figure 2.11). An extra bit (EB) is used to indicate whether an instruction is a hard binary or an intermediate representation. $EB = 0$, indicates that the word is a hard binary. $EB = 1$, indicates that the word is an intermediate representation. The method for unraveling this code will be explained in next Section 2.3.3.

2.3.3 The Two Phase Method

Before explaining how the CGIR is unraveled at runtime, we will describe the changes in two phase method (shown previously in Figure 2.4).

Programming Flow

Inspired from [128], we have designed two phase method for optimal version selection. Figure 2.12 illustrates the details of our method. The configware

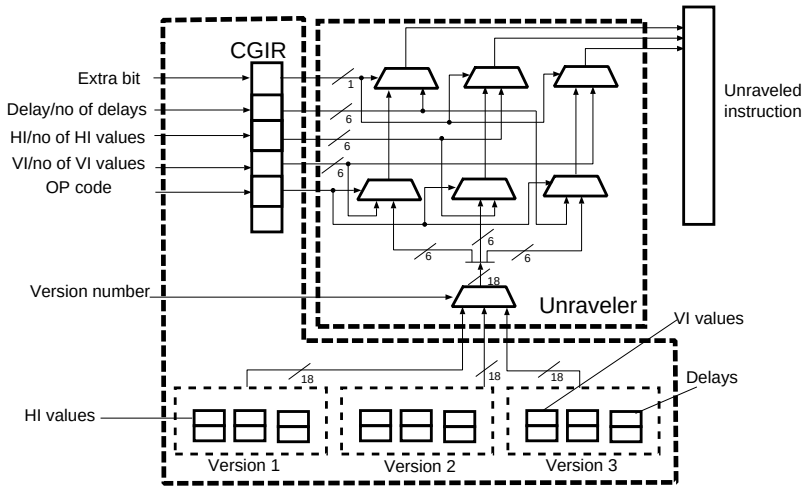

Figure 2.11: Runtime Extraction of CGIR

for commonly used DSP functions (FFT, FIR filter e.t.c.) is written either in VESYLA (HLS tool for DRRA) or *MANAS* (which is the assembler for DRRA) and stored in an offline library. The library, thus created, is profiled with throughputs and energy consumptions of each implementation. When an application is to be loaded, an offline compiler isolates the versions which meet the deadlines of the application and sends them to RTM, as CGIRs. Each CGIR compactly represents multiple versions. The RTM unravels the CGIR by selecting the most optimal version (in terms of power consumption, memory utilization etc.), considering the available resources.

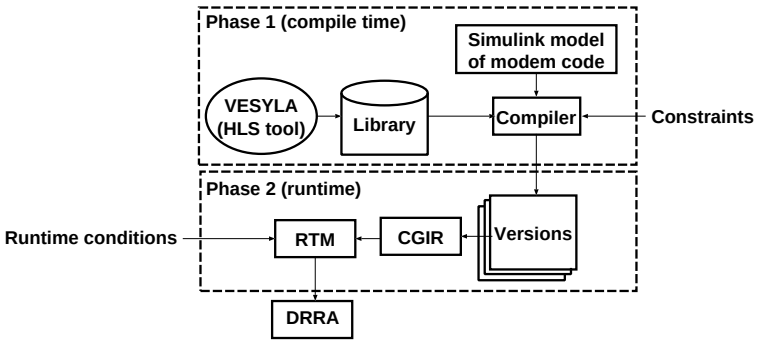

Figure 2.12: DRRA programming flow

Runtime Unraveling

The runtime unraveling can either be performed by the in software or hardware. Due to the ease of implementation (considering the complex problems tackled), for most part of the thesis we have employed software based unrav-

eling. In this technique, the processor analyzes each configuration instruction before feeding it to DRRA. If the instruction represents a hard binary, it is fed directly to DRRA. If its a soft primitive, then its unraveled and the unraveled instruction is sent to DRRA. For the sake of completion, we will also show how the soft binary can be quickly unraveled using a hardware based solution. Figure 2.11 shows the circuitry for unraveling the CGIR in hardware. EB is analyzed to determine whether an incoming instruction represents a hard binary or an intermediate representation. Upon detection of an intermediate representation, set of sequences to be replaced in the intermediate representation are extracted from CGIR depending on the version to be configured. Finally, from the OP code, it is determined whether the incoming sequence indicates delay for reg-file instructions or HI values and VI values for switch box instructions. Once the delay or HI and VI fields have been inserted, the instruction is sent to the sequencer.

2.4 Network on Chip

In this thesis, we have chosen McNoC to test the effectiveness of our method. McNoC is a packet switched network on chip platform, which uses regular mesh topology [23]. We chose McNoC due to the following reasons: (i) we had available full RTL code allowing us to make architectural modifications easily; (ii) McNoC had in built power management system which allowed us to test the effect of power management on a NoC; (iii) McNoC is a very well documented platform with over 100 publications, and (iv) the architecture of McNoC is very similar to the contemporary academic and industrial platforms allowing us to extend the framework to other architectures. The overall architecture of McNoC is shown in Figure 2.13. Broadly, McNoC can be divided into two different components: (i) network on chip and (ii) power management infrastructure.

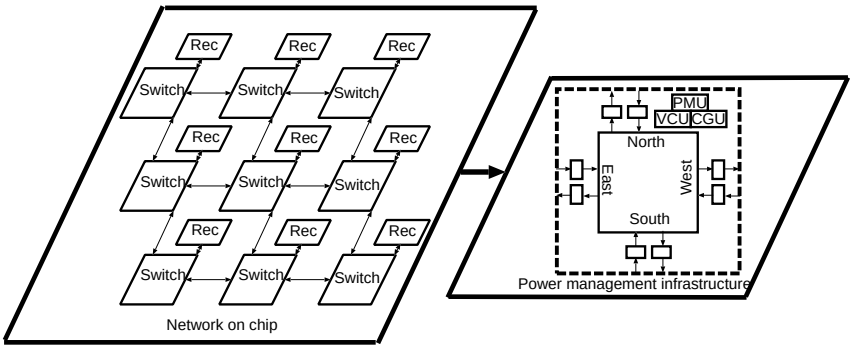

Figure 2.13: McNoC architecture

2.4.1 Nostrum

McNoC uses the Nostrum network-on-chip as communication backbone [93, 97, 83]. It uses regular mesh topology with each node comprising of a resource (rec in Figure 2.13) and a switch. Every resource contains a LEON3 processor, a memory and a *Data Management Engine* (DME) [25, 1]. DME acts as a memory management unit and interfaces the node with the network. Nostrum uses buffer-less switches which provide *hot potato* X-Y routing [37]. In this routing strategy, as long as there are no contentions, packets are routed normally using dimension order routing. If multiple packets contend for a link, the packet with most hop-count is given priority and the rest are randomly misrouted to another free link. The main benefit of using hot potato routing is that it allows to use buffers-less routers. The buffer-less routers have significantly small energy and area costs (compared to buffered routers) and are a subject of intensive research for low power NoCs [89, 46, 67, 73]. Since we target low power NoCs, nostrum provided us with a perfect platform. For buffered NoCs, the relative overhead of implementing the proposed framework (compared to a router) is expected to be significantly smaller thereby enhancing the feasibility of our approach. The methods, presented in this thesis can easily be extended to accommodate buffered in routers.

2.4.2 Power management infrastructure

A power management system has been built on top of Nostrum by introducing a *Globally Ratio Synchronous Locally Synchronous (GRLS)* wrapper around every node [22, 21]. The wrapper is used to ensure safe communication between nodes and to enable Dynamic Voltage and Frequency Scaling (DVFS). The access point to provide the power services is given by the Power Management Unit (PMU), which uses Voltage Control Unit (VCU) and Clock Generation Unit (CGU) to control the voltage and the clock frequency, respectively, in each node. A detailed description of GRLS is beyond the scope of this thesis and for details, an interested reader can refer to [23].

2.5 Experimental Methodology

To access the efficacy of the presented work, the author will implemented various components of VARP framework on DRRA or McNoC in the proceeding chapters. To estimate additional overheads, the synthesis results will be done using 65 nanometer technology at 400 MHz frequency, using Synopsys design compiler (unless otherwise stated). Most of the algorithms will be implemented on the LEON3 processor.

2.6 Summary

In this chapter, the architectural details of the CGRA and the NoC platforms, used in this thesis, were presented. To evaluate VRAP on a CGRA, DRRA was chosen. However, before this thesis DRRA lacked a configuration backbone essential to evaluate the VRAP. Therefore, before implementing the core contributions of thesis thesis, i.e. PCE, PRE, and POE, we enhanced DRRA with a smart and efficient configuration mechanism. To evaluate VRAP on a NoC, we have chosen McNoC. McNoC is RTL based cycle accurate simulator. It already contained a comprehensive Data Management Engine (DME) complemented by a power management infrastructure, that allowed to implement POE and PRE on the existing McNoC platform.

Chapter 3

Private Configuration Environments for CGRAs

3.1 Introduction

In this chapter, we will present a polymorphic configuration architecture, that can be tailored to efficiently support reconfiguration needs of the applications at runtime. Today, CGRAs host multiple applications, running simultaneously on a single platform. To enhance power and area efficiency they exploit late binding and time sharing. These features require frequent reconfigurations, making reconfiguration time a bottleneck for time critical applications. Existing solutions to this problem either employ powerful configuration architectures or hide configuration latency (using configuration caching). However, both these methods incur significant costs when designed for worst-case reconfiguration needs. As an alternative to worst-case dedicated configuration mechanism, we exploit reconfiguration to provide each application its Private Configuration Environment (PCE). PCE relies on a morphable configuration infrastructure, a distributed memory sub-system, and a set of PCE controllers. The PCE controllers customize the morphable configuration infrastructure and reserve portion of the a distributed memory sub-system, to act as a context memory for each application, separately. Thereby, each application enjoys its own configuration environment which is optimal in terms of configuration speed, memory requirements and energy.

Specifically, we deal with the case when a CGRA fabric instance hosts multiple applications, running concurrently (in space and/or time), and each application has different reconfiguration requirements. Some applications enjoy dedicated CGRA resources and do not require further reconfiguration. While other applications, share the same CGRA resources in a time-multiplexed manner, and thus require frequent reconfigurations. Additionally, a smart power management system might dynamically serial-

ize/parallelize an application, to enhance energy efficiency by lowering the voltage/frequency operating point. This requires a reconfiguration architecture that is geared to dynamically and with agility reconfigure arbitrary partitions of the CGRA fabric instance. To address these requirements, concepts like configuration caching [35], [109], [88], configuration compression [34], [51], [45], and indirect reconfiguration [129], [128], [58] have been proposed. While these techniques do solve the problem, they come at a considerable cost (i.e. they improve the agility at cost of space and vice-versa). Moreover, they address the reconfiguration requirements of only a certain category of applications/algorithms; when a different category of application is instantiated, either the resources are under-utilized or the reconfiguration speed suffers. In this chapter, we propose a configurable reconfiguration architecture, that allows different partitions of CGRA fabric instances to have a reconfiguration infrastructure that is adapted to its needs. In essence, we are proposing second order reconfigurability; reconfiguring the reconfiguration infrastructure to match the application needs. In particular, we distinguish between four reconfiguration architectures: (i) Direct Configuration (DC), (ii) Distributed Indirect Configuration (DIC), (iii) Bus-based Indirect Configuration (BIC) and (iv) multi-context configuration, as shown in Fig. 3.1 (a). Each of these architectures incur different costs (in terms of reconfiguration time, configuration memory, and energy). The DC requires the least memory (and hence power/energy) but is too slow to support applications needing time sharing and late binding ([58], [44], Section 3.7). The DIC offers high speed reconfiguration at the cost of additional memory/power. The BIC allows to compress data resulting in reduced memory requirements (see [123] and Section 3.7), compared to distributed configuration infrastructure, at the cost of performance. The multi-context architecture offers high frequency reconfiguration at the cost of high memory.

To efficiently utilize the silicon and energy resources we present a morphable architecture, that can dynamically morph into DC, DIC, BIC, or multi-context. As shown in Fig. 3.1 (b), the proposed scheme relies on a reconfigurable infrastructure (hardware) supported by a morphable scratch pad memory. The polymorphic infrastructure can be tailored to realize either direct, bus based or distributed communication. The morphable memory can morph into data memory, single context configuration memory, or multi-context configuration memory. Each application can have its own customized reconfiguration architecture (infrastructure and memory), which we call Private Configuration Environment (PCE). The proposed scheme is generic and in principle applicable to all grid based CGRAs with a scratch pad data memory [43], [103], [80]. To report concrete results, we have chosen DRRA [111] as a representative CGRA. Simulating practical applications (WLAN and Matrix Multiplication) show that our solution can save up to

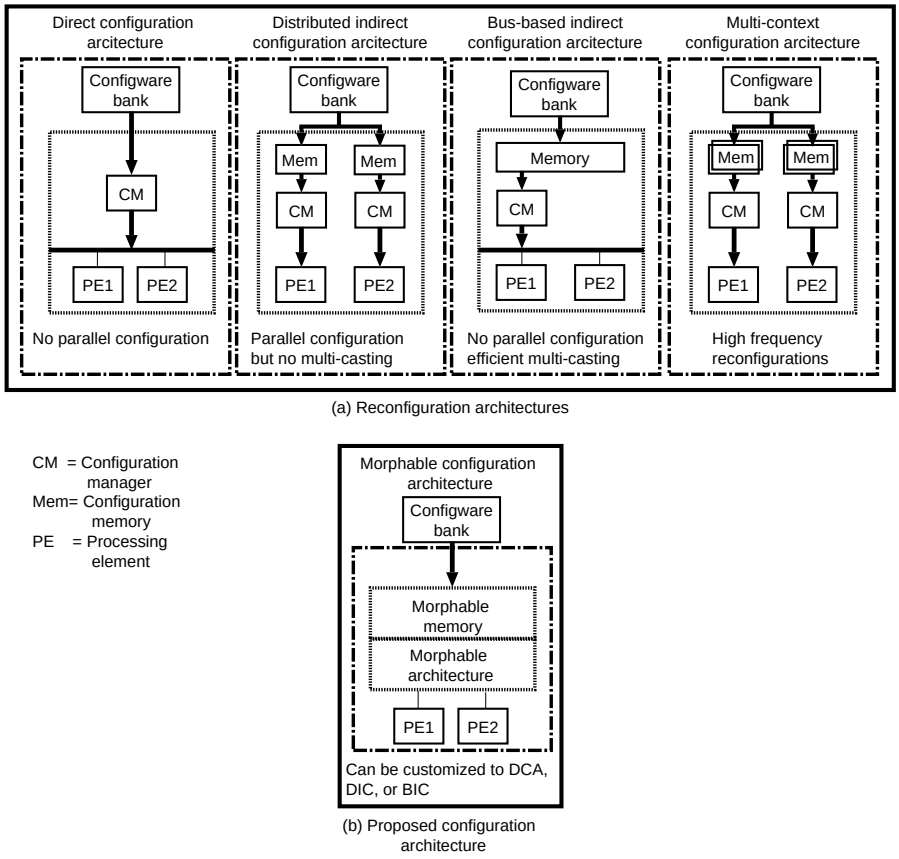

Figure 3.1: Motivation for Private Configuration Environments (PCE)

58 % memory (compared to the worst case), by changing the configuration modes. Synthesis results confirm that implementing the proposed technique incurs negligible overheads (3 % area and 4 % power).
This work had five major contributions:

1. we propose a morphable configuration infrastructure which can be tailored to match the application configuration needs, thereby promising significant reductions in memory and energy consumption;

2. we exploit existing data memory to mimic configuration caching and context switching, thereby eliminating the need for dedicated contexts;

3. we propose an Autonomous Configuration Mode Selection (ACMS) algorithm that based on the reconfiguration deadlines and available memory selects a configuration mode that consumes the least energy;

4. we present a 3-tier hierarchical configuration control and management layer, to realize the above concepts in a scalable manner (Section 3.4); and

5. we formalize (Section 3.6) and analyze (Section 3.7) potential benefits and drawbacks of using the morphable reconfiguration architecture.

3.2 Related Work

A configuration architecture is composed of two main elements: (i) *configuration delivery mechanism* and (ii) *internal configuration memory*. The configuration delivery mechanism transfers the configware, that determines the system functionality, from an external storage device to the internal configuration memory. Therefore, as shown in Fig. 3.2, the techniques that enhance the configuration efficiency, either reduce the configuration delivery time and/or optimize the internal configuration memory. In this section, we will review the most prominent work from both areas that is relevant to our approach.

Traditionally, reconfigurable architectures were provided with only one configuration memory and the configware was loaded in daisy chained fashion [96]. DeHon [35] analyzed the benefits of hiding configuration latency by employing multi-context reconfiguration memories. To allow fast reconfiguration, DAPDNA-2 [109] and FE-GA employ four, DRP-1 [88] and STP 16 employ 16, and ADRES employs 32 contexts in their architectures. However, the redundant context memory is both area and power hungry. As a result of redundant contexts, the configuration memory consumes 50% and 40% of area in ADRES and MuCCRA [7], respectively [8]. Additionally, the configuration caching consumes prohibitive dynamic (due context switching) and static (due to additional memory) power. As a part of solution

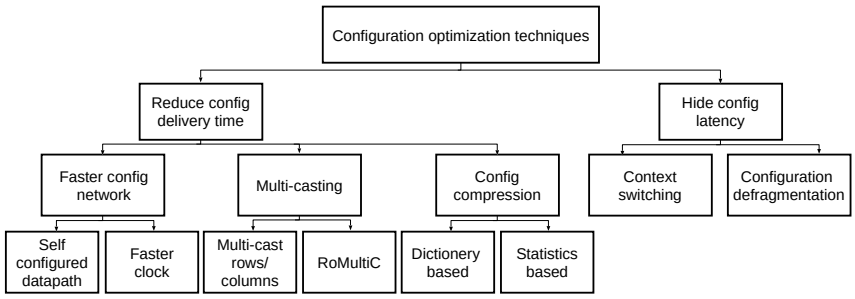

Figure 3.2: Classification of methodologies to optimize configuration

to this problem, Compton [30] presented a method for configuration data de-fragmentation. The proposed method reduces the unusable areas in configuration memory, created during reconfiguration. Thereby, it enhances the configuration memory utilization. All the research, that attempt to enhance the configuration efficiency, employ dedicated contexts regardless of the configuration requirements. As an alternative, we suggest using the contexts with configurable size by exploiting the scratch pad memory to mimic the functionality of multiple contexts.

Existing research that reduces the configware delivery time employs configuration compression, multi-casting or a faster configuration network. Configuration compression utilizes regularity in data to minimize the size of configuration bitstream. Multicasting reduces the configuration latency by configuring multiple PEs, simultaneously. Morphosys [43], reduces the configuration cycles by allowing all the PEs in a row/column to be configured in a single cycle. However, since the entire row/column has to be configured with same data, it incurs significant hardware overheads if different elements in a row/column perform different functions. The hardware wastage in Mosphosys was considerably reduced by the RowMultiC, presented in [123]. This technique, uses two wires, indicating column and row respectively, connected to each cell of a CGRA. The cells which have one set in both column and row wire are configured with the same data in a single cycle. This scheme was later employed by [7] and [72] to optimize their configuration architectures. The multi-casting technique in this thesis is also inspired from RowMultiC. We enhance its effectiveness by suggesting how it can be scaled. SmartCell [80] employs both multi-casting and context switching to reduce the excessive configuration time. We employed a combination of RowMultiC and dictionary based compression to enhance configuration efficiency [57]. Sano and Amano [108] proposed an adaptive configuration technique to dynamically increase the configuration bandwidth. The proposed approach combines the configuration bus with the computation network, at runtime. When high speed configuration is needed, the network otherwise used for computation is stalled and used to reduce configuration time. Furthermore,

since the configuration is not as complex as the computations, they suggest to use a faster network for configuration.

3.3 Private Configuration Environments (PCE)

The reconfigurable fabric DRRA efficiently hosts multiple applications by dynamically creating a separate partition in its computation and memory layers. However, before our enhancements, all applications were provided a dedicated serial bus based configuration mechanism. Since different applications can also have different reconfiguration requirements, we have upgraded the DRRA computation and storage layer to implement a morphable reconfiguration architecture. Thus each application on the DRRA fabric can have a configuration scheme tailored to its needs, called Private Configuration Environment (PCE). The proposed scheme relies on a morphable storage layer and a reconfigurable infrastructure. To realize a morphable storage layer, DiMArch, that previously served as data memory to the DRRA computational fabric, can now morph into context memory for different configurations. The configuration infrastructure is made morphable by embedding a set of controllers to handle data transfers. The details of the morphable memory and reconfigurable infrastructure will be given in Sections 3.3.4 and 3.3.5, respectively.

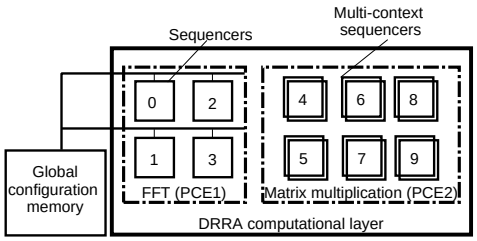

Figure 3.3: Logical view of private configuration environment

To clearly illustrate the concept of PCE, consider the case of a DRRA instance, shown in Fig. 3.3, that simultaneously hosts FFT and Matrix Multiplication (MM). It is assumed that MM needs fast and frequent reconfigurations (using multiple contexts) while FFT once configured needs no further reconfiguration. Providing FFT with fast multi-context configuration architecture would be a waste of area and energy. The proposed methodology promises reductions in these overheads by morphing into PCE1 and PCE2 for FFT and MM, respectively. Where PCE1 provides simple direct loading from memory and PCE2 provides a fast multi-context reconfiguration architecture.

3.3.1 PCE Configuration modes

To achieve different levels of performance and memory consumption, the proposed architecture can morph into four configuration modes: Direct Feed (DF), Memory Feed Multi-Cast (MFMC), Memory Feed Distributed (MFD), and multi-context. In Table 3.1, we briefly estimate the costs and benefits of these modes. The estimates will be formally evaluated in Section 3.6 and actual figures will be reported in Section 3.7. In Direct Feed mode (DF), the configuration bitstream is directly fed to the DRRA sequencers from the global configuration memory (see Fig. 2.5). This method requires high reconfiguration time, due to additional latency of moving configware from global configuration memory via AHB bus to the loader. The DF mode incurs low memory costs since it requires no intermediate storage. The Memory Feed Multi-Cast (MFMC), the Memory Feed Distributed (MFD), and the multi-context modes copy the configware transparently to DiMArch, before transferring it to DRRA sequencers. Thereby, they reduce the configuration latency (global configuration memory to the loader) at the cost of additional intermediate memory. In memory feed multi-cast mode, the configware is directly fed to the DRRA sequencers from DiMArch. This mode offers code compression by storing identical configuration words only once [123], [7]. The memory feed distributed mode feeds the configware from the DiMArch to multiple sequencers (belonging to different columns), simultaneously. Thereby the MFD mode reduces configuration time. It requires additional memory, since same configuration words need to be stored in multiple locations. The multi-context mode, stores multiple configurations of same application in different memory banks of DiMArch. This mode allows high speed of operation (same as MFD) and high frequency context switching.

Table 3.1: Configuration modes

Configuration mode	Configuration infrastructure	Configuration time	Configuration memory	Targeted domain
Direct Feed (DF)	Bus based sequential and multi-cast configware transfers from global memory	High	Low	Applications needing no reconfiguration
Memory Feed Multi-Cast (MFMC)	Bus based sequential and multi-cast configware transfers from DiMArch	Medium	Medium	Applications using late binding
Memory Feed Distributed (MFD)	Distributed sequential transfers from DiMArch	Low	High	Applications using late binding
Multi-context	Multiple parts of DiMArch act as multiple contexts	Low	Highest	Applications using time sharing

To further illustrate the need for different configuration modes, consider

for example that a platform hosts Wireless LAN (WLAN). Given that abundant resources are available and no further reconfigurations are needed, the direct feed mode (with minimum memory requirements) will be the most efficient configuration technique. If the WLAN application can be parallelized/serialized (e.g. to enhance its energy efficiency [60]) the system requires some initial reconfigurations to stabilize. To meet the reconfiguration needs of this system either memory feed multi-cast or memory feed distributed modes would be feasible. Finally, if the platform has limited resources and the WLAN is time multiplexed with MPEG4 decoder. To meet the deadlines, this system will require fast and frequent reconfigurations that can be only provided by multi-context configuration mode.

3.3.2 PCE life time

The proposed scheme provides multiple applications with the configuration architectures, tailored to their needs, called Private Configuration Environments (PCE). A PCE has the life time equal to the application, for which it is created. Before an application enters a platform, its PCE is created and required resources reserved. During execution, the PCE manages the context switches and configware delivery. After the application exits the platform, the PCE is terminated and the reserved resources released. Broadly, the life time of a Private Configuration Environment (PCE) can be divided into six stages: (i) memory banks in DiMArch (data memory) are reserved to act as configuration memory, (ii) the application configware is stored in the reserved memory banks, (iii) the context switching and data transfer instructions are sent to the DiMArch sequencers (Section 3.3.4), (iv) configuration infrastructure is morphed to mimic the configuration mode (Section 3.3.5), (v) the application starts executing with the data transfers and context switches managed by the DiMArch sequencer (Section 3.3.4), and (vi) the PCE is terminated once application leaves the platform.

3.3.3 PCE Generation and Management Packet (GMP)

To realize the six stages of PCE, discussed in Section 3.3.2, additional information is stored with the configware of each application. This additional information identifies the peculiarities of a PCE (e.g. configuration mode and contexts). Fig. 3.4 (a) shows the original Application ConfigWare (ACW) along with the appended PCE information, collectively called PCE *Generation and Management Packet (GMP)*. The GMP packet contains four types of instructions: (i) PCE Context Length (PCL) instructions, (ii) Application ConfigWare (ACW), (iii) Data sequencing instructions (Dseq), and (iv) Context sequencing instructions (Cseq). The PCL instructions are loaded first from the global configware memory to a DiMArch sequencer (see Sec-

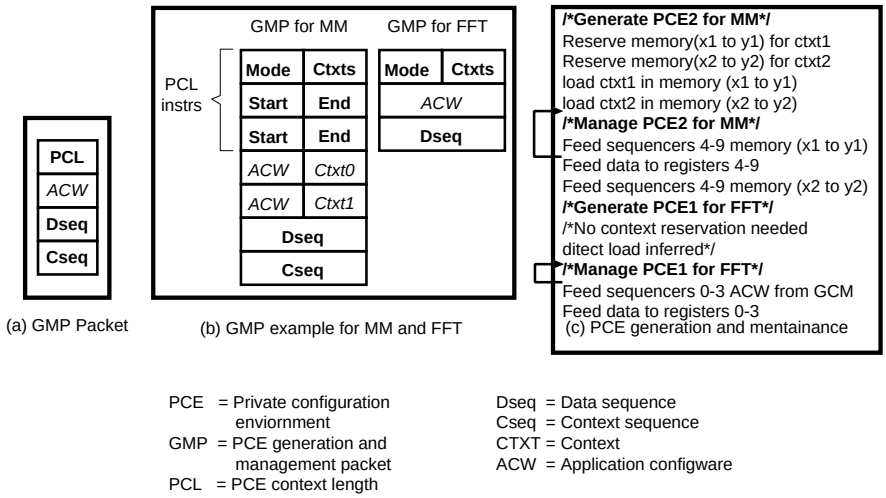

Figure 3.4: PCE generation and management

tion 2.1.2). Depending on the PCL instructions, the DiMArch sequencer either creates or manages a PCE. The Dseq instructions identify the locations and order, to transfer data for computation. The Cseq instructions dictate the locations and order in which context switches should be made. To illustrate how a PCE is generated and managed (using the GMP), we reuse the example of FFT and matrix multiplication, discussed earlier in this section. Remember that FFT and matrix multiplication use direct feed and multi-context configuration modes, respectively. Therefore, as shown in Fig. 3.4 (b), matrix multiplication and FFT have different GMP packets . For FFT, the PCL field contains only one instructions indicating that FFT will use direct feed mode (needing no context reservation). For matrix multiplication, the PCL field contains three instructions. The first instruction indicates that multi-context configuration mode with two contexts should be reserved. The other two instructions identify the start and end addresses of DiMArch memory banks to be reserved for each context. After the memory banks are reserved, the application configware is sent to the reserved DiMArch memory banks. Finally, the Dseq and Cseq instructions are copied to the DiMArch sequencers to manage data transfers and context switching, respectively. Fig. 3.4 (c) depicts how the DiMArch sequencer decodes the packet to generate and manage the PCEs.

3.3.4 Morphable Configuration Memory

Before our modifications, all the configware was stored in a global configuration memory (see Fig. 2.5), before its transfer to the relevant sequencers

[58]. But as discussed later in sections 3.6 and 3.7, the cost of programming the sequencer from the global configware memory is too high to support late binding or time sharing (provided by MFMC, MFD, and multi-context modes). To allow fast and frequent reconfigurations, we have extended the functionality of the existing distributed-data-memory, DiMArch (described in Section 2.1.2), to store configware as well. To efficiently support variable memory to computation ratios of different applications, DiMArch can be dynamically divided into multiple partitions, by the software [118]. Each partition can be viewed as a local memory for the application. Each partition can be subdivided into two parts: (i) configware partition and (ii) data partition. Before an application is mapped, a request to reserve a memory/configware partition, is sent to DiMArch. Based on the request, DiMArch creates memory/configware partitions of appropriate sizes. The configware partition can be further morphed into three states: (i) centralized single context, (ii) distributed single context, and (iii) distributed multi context. The centralized single context state assumes that DiMArch is connected to a bus based configuration infrastructure and outputs data sequentially. The distributed single context state considers that DiMArch is supported by a distributed configuration architecture. In this configuration mode, DiMArch copies data in multiple memory banks, from where it can be transferred to the DRRA sequencers, in parallel. In distributed multi-context state, DiMArch stores different chunks of a configware in multiple memory banks to perform high frequency context switching. To realize different configuration states, DiMArch sequencers (Section 2.1.2) are employed. The DiMArch sequencers are implemented as simple state machines that control and manage each partition. The state machines determine when data or configware is sent to the reg-files or sequencers, respectively. For further information on DiMArch sequencer, we refer to [118].

3.3.5 Morphable Configuration Infrastructure

Fig. 3.5 depicts a high-level overview of DRRA configuration infrastructure. In this section, we will explain how each configuration mode is realized using this hardware. The intelligence for morphing the configuration architecture resides in a Configuration Controller (CC). To allow scalability, the CC is implemented hierarchically in three layers, as explained later in Section 3.4. For the direct feed mode, the CC performs three steps: (i) it loads the configware from the configware bank to the Horizontal Bus (HBUS), (ii) it asserts the sequencer addresses directly to DRRA using RowMultiC (see [123] and Section 3.4.2), and (iii) it directs the Local Configuration Controller (LCC), present with each column of the DRRA, to copy the data from the HBUS to vertical bus VBUS , effectively broadcasting data to all sequencers. To support memory feed distributed, memory feed multi-

cast, and multi-context modes, the configware is first loaded to DiMArch by the DiMArch sequencers using the DiMArch network shown in Fig. 2.3. In distributed mode, the configware from the DiMArch memory bank is transferred to the MLFCs. The MLFCs depending on the address transfer the configware to the sequencers. In MCMF mode, the configware is placed by MLFCs on its VBUS, and simultaneously the multi-cast addresses are sent to the CC.

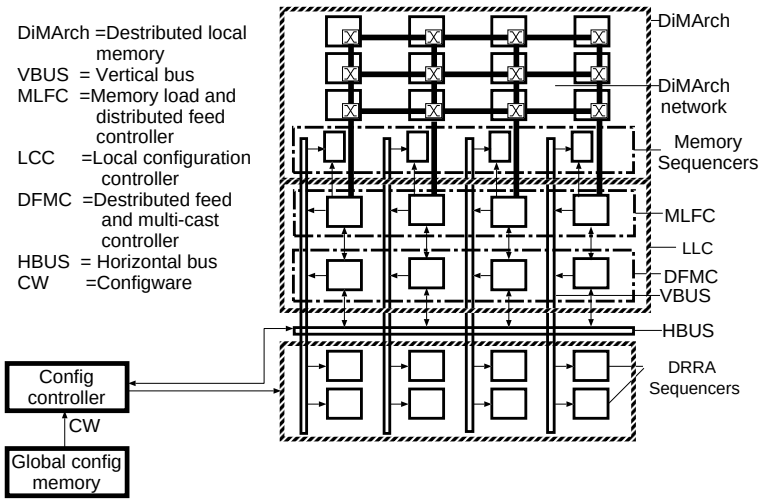

Figure 3.5: Private Configuration Environment (PCE) infrastructure

3.4 Hierarchical configuration backbone

To ensure scalability, we have implemented the proposed configuration architecture using three hierarchical layers of *configuration controllers*: (i) *local controller*, (ii) *application controller*, and (iii) *platform controller*. A logical view of these controllers is shown in Fig. 3.6. A single platform controller manages all the Private Configuration Environments (PCEs) of the platform. It is connected to the loader and receives the PCE generation and management packets (see Section 3.3.2) from the global configuration bank. Depending on the contents of the packet and the available free resources the platform agent sends the PCE generation and management packet to one of the application controllers. Each application controller creates and manages a PCE, by interacting with local controllers. A set of local controllers coordinate with each other and the application controller to realize one of the configuration modes. The configuration controllers are implemented as simple state machines, which depending on the chosen configuration mode, direct the configuration words towards appropriate path. For a given application, the application and local controllers can operate in either direct

feed, memory feed distributed, or memory feed multi-cast mode (see Section 3.3.1). Before shifting to memory feed distributed or memory feed multi-cast mode, the controllers first load DiMArch in an additional mode, called memory load mode. The multi context mode is realized by the morphable DiMArch memory and hence the controllers are oblivious to it.

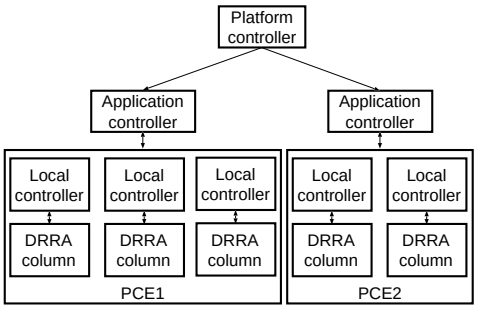

Figure 3.6: Hierarchical configuration control layer

3.4.1 Local controller

A separate controller, called local controller, is embedded with each column of DRRA. To generate and manage a private configuration environment, a set of local controllers work in harmony. The basic functionality of the local controller is shown in Table 3.2. To efficiently realize the functionality we have implemented the local controllers in two parts: (i) multi-cast controller and (ii) DiMArch controller.

Table 3.2: Local controller functionality

Configuration mode	Functionality
Direct feed	Copy configware from horizontal bus to vertical bus to vertical bus
Memory load	Copy configware from horizontal bus to DiMArch sequencer
Memory Feed Distributed (MFD)	Copy configware from DiMArch sequencers to vertical buses
Memory Feed Multi-Cast (MFMC)	(i) One of the local controllers copies configware from the DiMArch sequencer to horizontal bus (ii) All local controllers copy the configware from horizontal bus to the vertical bus

Multi-cast controller

As depicted in Fig. 3.7 (a), the multi-cast controller is connected to horizontal bus, vertical bus, and DiMArch controller interface. To determine its operating mode, the multi-cast controller continuously snoops for valid

data, on the horizontal bus and the DiMArch controller interface. Upon detecting valid data on the horizontal bus, it morphs to either direct feed or memory load mode.In direct feed mode it copies the configware from the horizontal bus to the vertical bus, thereby broadcasting the configware to all sequencers. In memory load mode the configware is sent to the DiMArch controller interface. If the multi-cast controller detects valid data on its DiMArch controller interface it copies the configware and multi-cast addresses to the vertical bus and the horizontal bus, respectively. The application configuration controller later use these addresses to enable the appropriate sequencers, as will be explained in Section 3.4.2.

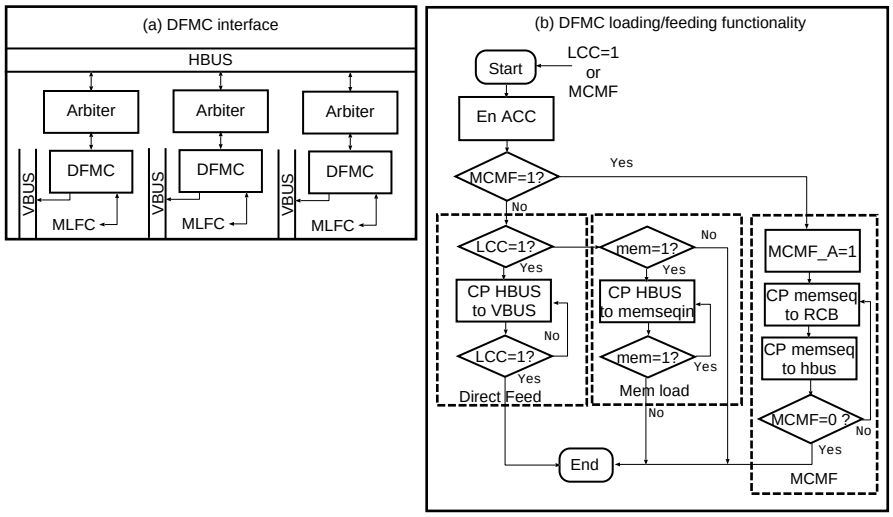

Figure 3.7: Direct Feed and Multi-Cast controller (DFMC)

DiMArch controller

The functionality of DiMArch controller is depicted in Fig. 3.8. Each DiMArch controller is connected to the memory sequencer, multi-cast controller, and vertical bus. To determine its operating mode, the DiMArch controller monitors the memory sequencer and the multi-cast controller interface. On detecting valid data on its multi-cast controller interface, it morphs to memory load mode. In memory load mode, the DiMArch controller copies the configware on the multi-cast controller interface to the vertical bus and signals the memory sequencers to load the data in reserved memory (See Section 3.3.2). If the DiMArch sequencer finds data on its interface with memory sequencer, it morphs to memory feed distributed or memory feed multi-cast mode.In memory feed multi-cast mode and the configware is sent to the multi-cast controller. In memory feed distributed mode the configware is copied to the vertical bus, and the addresses sent to the

multi-cast controller.

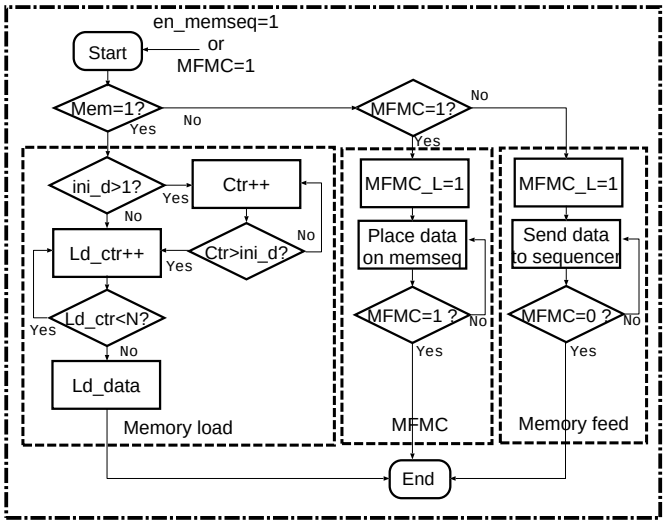

Figure 3.8: Memory Load and distributed Feed Controller (MLFC)

3.4.2 Application controller

An application controller is embedded with a set of local controllers to generate and manage a private execution environments. The architecture and functionality of the application controller is depicted in figures 3.9 and 3.10, respectively. The application controller is connected to platform controller, horizontal bus, row bus, and column bus. The total number of rows and columns in a private configuration environment is a design time decision. Each wire RB_i and CB_j is connected to the all the sequencers in i^{th} row and j^{th} column, respectively. During operation, the application controller snoops, for valid data, on platform controller interface and horizontal bus. If it detects valid data on platform controller interface, it morphs to either direct feed or memory load mode. In direct feed mode the application controller performs two tasks: (i) it asserts the row bus and column bus addresses and (ii) it copies the data to horizontal bus. In memory load mode the configware from platform controller is sent directly to the horizontal bus. If application controller finds valid data on horizontal bus, memory feed multi-cast mode is inferred and row bus/column bus addresses are extracted from the horizontal bus.

3.4.3 Platform controller

The platform controller is the general manager responsible for dynamically generating all the private configuration environments in a platform. A logi-

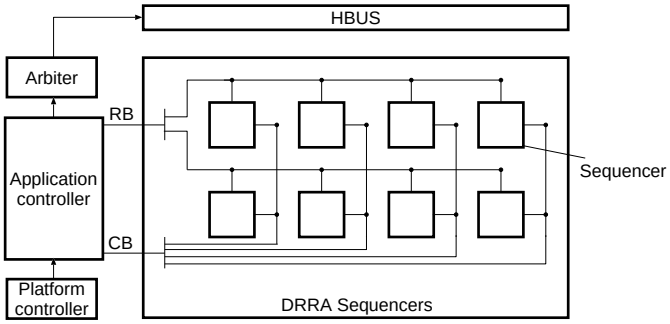

Figure 3.9: Application controller architecture

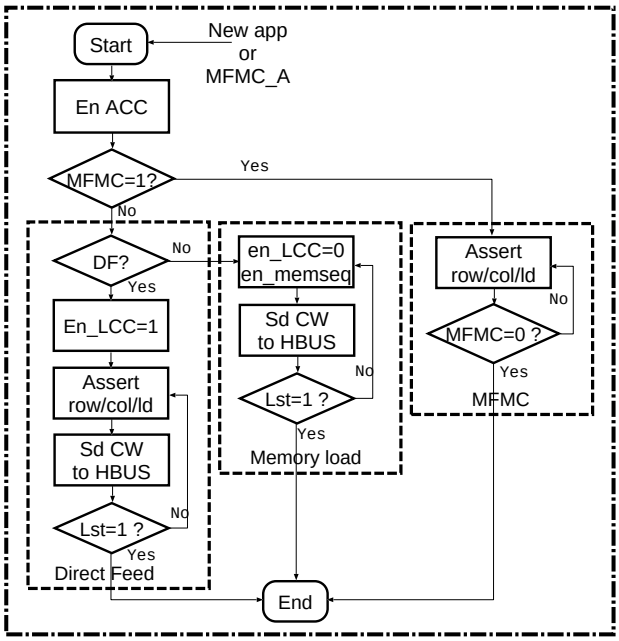

Figure 3.10: Application controller functionality

cal representation of the Platform Controller (PCC) is depicted in Fig. 3.11. The platform controller based on the sequencers to be programmed, identifies the candidate application controller. It is mainly intended to ensure scalability of the proposed bus based morphable architecture. For small projects (applications hosting only a single applications) the platform controller can be completely removed from the system. For mid to large sized projects the platform controller can be controlled by a dedicated thread in the LEON3 processor (2.5). In this thesis, we mainly target future platforms hosting multiple applications simultaneously and therefore, we control and manage the platform controller, by software.

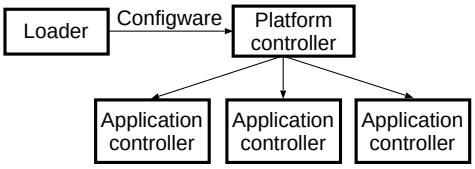

Figure 3.11: Platform controller logical/functional representation

3.5 Application mapping protocol

In this section, we will explain how the polymorphic reconfiguration architecture is customized for each application.

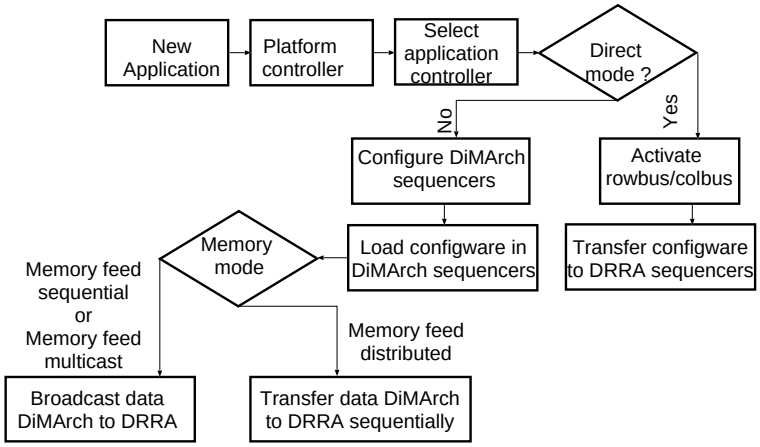

Figure 3.12: Configuration protocol

3.5.1 Configware datapath setup

Fig. 3.12 depicts how the datapath for the configware of an application is setup. Before mapping an application to the reconfigurable fabric, the

platform controller determines the application controller where the configware should be sent. The algorithm to determine an appropriate application controller is beyond the scope of this thesis. Some details about similar algorithms can be found in [40]. Upon reception of configware from the platform controller, the application controller checks the mode field in the received configware (see Fig. 3.4). If the field indicates direct load, the configware is loaded directly to DRRA sequencers. To load the configware, the row and column lines are first asserted (to activate the destination sequencers) and then the configware is broadcast to the horizontal and vertical buses (HBUS and VBUS). If the mode field indicates indirect loading, the configware is sent to the DiMArch memory. It should be noted that before loading the actual application configware, the DiMArch sequencers are programmed. The configured DiMArch sequencers first load the configware to the memory and then feed it to the DRRA sequencers at selected times. For multi-context memory feed modes, multiple copies of configware are stored. In the memory feed direct mode, only a single copy of configware is stored, while for the memory feed distributed mode configware is stored in multiple memory banks.

3.5.2 Autonomous configuration mode selection

Up till now we assumed that the configuration mode for each application is determined by the programmer at compile time. To autonomously select the configuration mode, we have implemented a simple algorithm, called *Autonomous Configuration Mode Selection algorithm (ACMS)*, on the platform controller (note that the platform controller itself is realized on LEON3 processor as a software). ACMS based on the available resources and application deadlines selects the configuration mode needing the least memory at runtime. The algorithm is depicted in Fig. 3.13. To illustrate the motivation for using this algorithm, consider for example that an application A requests CGRA resources. Given the availability of resources, the application will be mapped to the platform. However, if the sufficient resources are not available the platform controller will call the ACMS algorithm, that will attempt to time multiplex the application with an existing application. To time-multiplex multiple applications, ACMS finds a mapped application, B, with a slack larger than the deadline of application A. If such an application is found, A and B are multiplexed and the mode fields of both applications are modified. The current version of ACMS only make dynamic mode selection decisions if the resources consumed by the mapped application (application B in this example) are greater than or equal to the resources needed by the application to be mapped (application A in this example). The algorithm presently works only if both the mapped and to be mapped applications are in memory feed modes. The dynamic change from direct feed to memory

feed mode will require further architectural modification and which are not covered in this thesis.

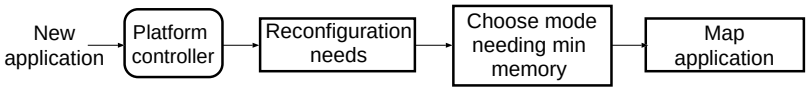

Figure 3.13: Autonomous Configuration Mode Selection algorithm (ACMS)

3.6 Formal evaluation of configuration modes

In this section, we will formally analyze the performance and memory requirements of each configuration mode.

3.6.1 Performance

The time, t_s, needed to configure an application, A_i, in direct feed mode (sequentially) is given by:

$$t_s = \sum_{i=0}^{seq} \sum_{j=0}^{Wi} (T(CW_{(i,j)}) + L_s), \tag{3.1}$$

where seq, Wi, L_s, and $T(CW_{(i,j)})$ denote the total sequencers to be fed, the total configware words in each sequencer, the latency from the global configuration memory to the HBUS, and the time needed for loading a configware word from VBUS to the sequencers. In the proposed configuration scheme, $T(CW_{(i,j)})$, remains constant, since it is achieved via broadcast. Therefore, Equation 3.1 can be written as:

$$t_s = \sum_{i=0}^{seq} \sum_{j=0}^{Wi} (T(CW) + L_s), \tag{3.2}$$

The time needed for direct feed in employing multi-casting, t_{md}, is given by:

$$t_{md} = t_s - (\sum_{l=0}^{MC} T_{cw} * G(l) - 1 + L_s), \tag{3.3}$$

The time needed for memory feed distributed mode, t_d, is given by:

$$t_d = max(t_{seq}(i)), \tag{3.4}$$

where $t_{seq}(i)$ is the time needed to feed configware to the i^{th} sequencer and is given by:

$$t_{seq}(i) = \sum_{j=0}^{Wi} T_{cw} + L_m. \tag{3.5}$$

Where L_m is the latency for feeding data from the DiMArch and given by:

$$L_m = D + 1, \tag{3.6}$$

where D is the distance of memory bank from MLFC. Therefore, provided that the configware is in DiMArch, distributed mode promises significant reductions in reconfiguration time, compared to direct mode. The time needed for memory feed multi-cast mode, t_m, is given by:

$$t_m = (\sum_{i=0}^{seq} \sum_{j=0}^{Wi} T(CW_{(i,j)}) + L_m) - (\sum_{l=0}^{MC} T_{cw} * G(l) - 1 + L_m), \tag{3.7}$$

where MC and $G(l)$ denote the words which can be multi-cast and the group of sequencers to configuration word l can be broadcast. It will be shown later in Section 3.7 that $L_s >> L_m$. Finally, the time needed to switch a context in multi context mode is same as that of the memory feed distributed mode. The multi-context mode is useful for time sharing when applications need to shift their contexts frequently. Therefore, from the equations 3.2, 3.3, 3.4, and 3.7 it can be concluded that $t_s > t_{md} > t_m > t_d$.

3.6.2 Memory requirements

Configuration memory, CM_s, needed to configure an application, A_i, in direct feed mode (sequentially) is given by:

$$CM_s = \sum_{i=0}^{seq} W_i * l_{CW}, \tag{3.8}$$

where seq, W_i, and l_{CW} denote the sequencers, the configuration words in the i^{th} sequencer, and the length of a configuration word. The configuration memory, CM_{mc}, required for direct feed, by employing multi-casting, is given by:

$$CM_{mc} = CM_s - (\sum_{l=0}^{MC} l_{cw} * seq(l) - 1), \tag{3.9}$$

where MC and $seq(l)$ denote the words which can be multi-cast and the number of sequencers to which word l can be broadcast. Equations 3.8 and 3.9 clearly indicate that multi-cast feeding requires lesser memory. Configuration memory, CM_{MFD}, needed for distributed mode, is given by:

$$CM_{MFD} = 2 * CM_s, \tag{3.10}$$

It should be noted that CM_s and $ctxt * CM_s$ bits will be needed in the global configuration memory Global Configuration Memory (GCM) and the

DiMArch, respectively. Configuration memory needed for multi-cast memory feed, CM_{MFMC}, is given by:

$$CM_{MFMC} = (ctxt + 1) * CM_{mc}. \tag{3.11}$$

The configuration memory, CM_{cs}, required for multi context mode, is given by:

$$CM_{cs} = CM_{MFD} * ctxts, \tag{3.12}$$

where $ctxt$ denotes the number of contexts reserved. From equations 3.8, 3.9, 3.10, 3.11, and 3.12 it is obvious that the memory requirements of $C_{cs} > CM_{MFD} > CM_{MFMC} > CM_s > CM_{ms}$.

3.6.3 Energy consumption

In this section, to visualize the effect of configuration mode on configuration energy consumption, we will present a very simplistic energy model. The actual energy estimates, using Synopsys Design Compiler will be reported in Section 3.7. Configuration energy, E_s, needed to configure an application, A_i, in direct feed mode (sequentially) is given by:

$$E_s = \sum_{i=0}^{seq} W_i * E_{CW}, \tag{3.13}$$

where seq, W_i, and E_{CW} denote the sequencers, the configuration words in the i^{th} sequencer, and the and the energy required to transport a configuration word to from memory the sequencer.

The configuration energy, E_{mc}, required for direct feed, by employing multi-casting, is given by:

$$E_{mc} = E_s - (\sum_{l=0}^{MC} E_{G2B}(l) * seq(l) - 1), \tag{3.14}$$

where MC and $seq(l)$ denote the words which can be multi-cast and the number of sequencers to which word l can be broadcast. E_{G2B} is the energy needed to transport a word from the global configuration memory to the HBUS. Equations 3.13 and 3.14 indicate that multi-cast feeding requires $\sum_{l=0}^{MC} E_{G2B}(l) * seq(l) - 1$ lesser than the direct feed mode. Configuration energy, E_{MFMC}, needed for multi-cast memory feed mode is given by:

$$E_{MFMC} = E_{mc} + Recof * E_{mem}, \tag{3.15}$$

where $Recof$ are the total number of reconfigurations and E_{mem} is the reconfiguration energy to feed from the memory. The configuration energy, E_{cs}, required for multi context mode, is given by:

$$E_{cs} = E_{mc} + \sum_{i=0}^{Recof} E_{mem}(i), \tag{3.16}$$

where $E_{mem}(i)$ denotes the energy needed to feed the DRRA from the i^{th} context. It is assumed that different contexts will be placed very close together making $E_{cs} \approx E_{MFMC}$. From equations 3.13, 3.14, 3.15, and 3.16 it is obvious that the energy requirements of $E_s > E_{ms} > E_{MFMC} \approx E_{cs}$.

3.7 Results

In this section, we will perform cost benefit analysis of the proposed approach.

3.7.1 Configuration time and Memory requirements of various configuration modes

To analyze the configuration time and memory requirements of various configuration modes, on real application, we mapped six representative applications/algorithms on the DRRA: (i) Fast Fourier Transform (FFT), (ii) Matrix Multiplication (MM), (iii) Finite Impulse response Filter (FIR), and (iv) wireless LAN transmitter (WLAN), 2-D convolution, and block interleaver). The motivation for choosing FFT, FIR, MM, 2-D convolution, and block interleaver is their wide spread in DSP application. WLAN was selected to analyze the benefits on a real complete application. For the FFT and MM multiple versions with different levels of parallelism (serial, partially parallel (par par), and fully parallel) were simulated. Each application was configured using the three configuration modes, shown in Section 3.3: (i) Direct Feed (DF), (ii) Memory Feed Multi-Cast (MFMC), and (iii) Memory Feed Distributed (MFD). In addition, we also simulated the configuration time and memory, with no multi-casting support. Therefore, two additional modes: Direct Feed Sequential (DFS) and modes Memory Feed Sequential (MFS) were created. Table 3.3 shows the time needed to configure the applications. It is clearly seen that the direct feed modes have require significantly large configuration time compared to the memory feed modes due to large configuration latency. Hence, justifying the assumption made in Section 3.6.1 ($L_s >> L_m$). Table 3.4 compares the configuration time of Memory Feed Distributed (MFD) and Memory Feed Multi-Cast (MFMC) modes. It is seen that, for the tested applications, the MFD mode promises a considerable reduction in configuration time (from 35 % to 80 %) compared to the multi-cast mode. The reason is that, the MFD mode feeds the configuration words in parallel, while the MFMC mode offers parallel feeding only when identical words are fed to multiple sequencers. Table 3.4 and Fig. 3.14, show the memory requirements for the Direct Feed (DF), Memory Feed Distributed (MFD) and Memory Feed Multi-Cast (MFMC) modes. It can be seen that direct feed mode requires significantly lesser memory compared to the memory feed modes (MFD and MCMF) because it does not

require additional copies of configware in DiMArch. The reason for better memory efficiency of Memory Feed Multi-Cast (MFMC) mode, compared to Memory Feed Distributed (MFD) mode is that, the MFMC mode stores identical configuration words only once. The memory requirements of the MFD and MFMC modes are identical only when all the configware words are different (e.g. in case of FIR and MM serial). It should be noted that all the for the multi-context mode, the memory requirements will be a multiple of MFD mode. The reconfiguration time will remain the same as the MFD mode.

Table 3.3: Reconfiguration cycles needed in different configuration modes

Application	Configuration mode				
	DF (Cycles)	DFS (Cycles)	MFS (Cycles)	Multicast (Cycles)	Distributed (Cycles)
FFT64 serial	5577	3120	143	52	80
FFT64 par par	7137	5655	183	29	145
FFT2048	25077	19500	643	63	500
MM serial	819	819	21	13	21
MM par par	1677	1326	43	13	34
MM parallel	2535	1716	65	13	44
FIR	507	546	13	5	14
WLAN	8892	6435	228	52	165
2D convolution	4056	4056	104	75	75
Block interleaver	1872	1872	48	8	8

Table 3.4: Reduction in configuration cycles distributed vs multi-cast

Application	Configuration mode		
	MFD (cycles)	MFMC (cycles)	Reduction %
FFT64 serial	52	80	35
FFT64 par par	29	145	80
FFT2048	63	500	87
MM serial	13	21	38
MM par par	13	34	62
MM parallel	13	44	70
FIR	5	14	64
WLAN	52	165	68

Table 3.5: Memory requirements for different configuration modes

Application	Configuration mode		
	MFMC (bits)	MFD (bits)	DF (bits)
FFT64 serial	5760	10296	2880
FFT64 par par	10440	13176	5220
FFT2048	36000	46296	18000
MM serial	1512	1512	756
MM par par	2448	3096	1224
MM parallel	3168	4680	1584
FIR	1008	936	504
WLAN	11880	16416	5940
2D convolution	7488	7488	3744
Block interleaver	3456	3456	1728

3.7.2 Overhead analysis

To estimate additional overhead incurred by the local, application, and platform controllers, we synthesized the DRRA fabric with PCE infrastructure. Area and power requirements of each component is shown in Table 3.6 and Fig. 3.15. The LCC and ACC arbiter were found to be most costly, consuming power (64 %) and area (39 %). LCC consumes high power since it is active in all configuration modes. Overall, the results confirm that the morphable reconfiguration architecture incurs negligible additional overheads (3 % area and 4 %power). To support RowMultiC (Section 3.4.2), an additional wire is added to every row and column of DRRA. Every cell is connected to the row and the column wire, traversing the cell (see Fig. 3.9). Thereby, each cell requires only 2 wire bus for its addressing. This overhead is significantly smaller compared to the wiring overhead of traditional addressing strategy, i.e. $nlog2$. Where n is the total cells present in the system. The latency for direct loading (from SRAM to DRRA sequencers via AHB bus) and memory loading (from DiMArch to DRRA sequencers via memory sequencers) is 39 and 6 cycles respectively. For memory loading, once the pipeline is filled a configuration word can be sent every cycle.

Table 3.6: Area and power consumption of different components of PCE

	ACC	ACC-arbiter	LCC	LCC-arbiter	DRRA cell
Power μW	13.67	25.1	130.19	33.29	5029
Area μm^2	488	1247	580	890	85679

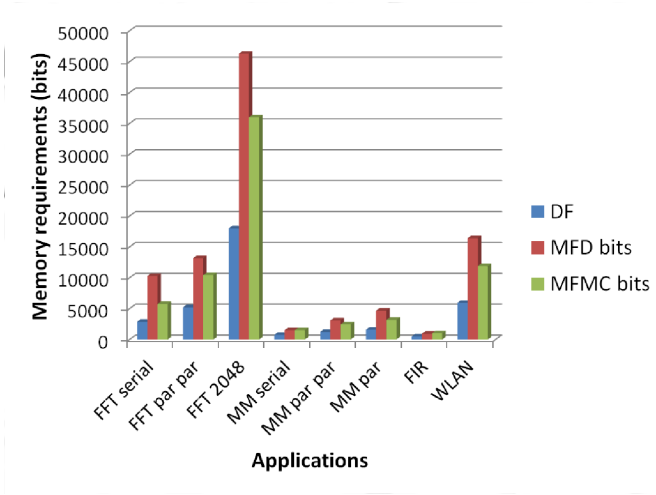

Figure 3.14: Configuration memory requirements for various configuration modes

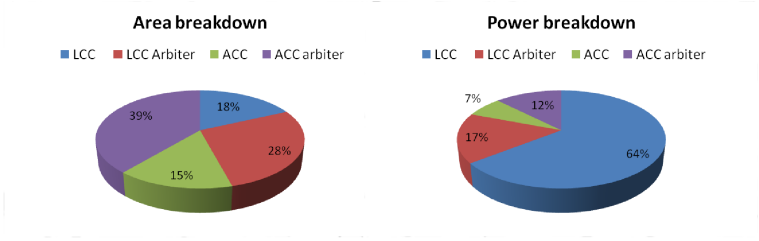

Figure 3.15: Area and power breakdown of various PCE components

3.7.3 PCE benefits in late binding and configuration caching

To demonstrate the benefits of our scheme, we have used *autonomous parallelism, voltage, and frequency selection algorithm (APVFS)*, presented in [60]. The APVFS algorithm stores multiple versions of each application, with different degree of parallelism. High energy efficiency is achieved by dynamically choosing the version that requires the least voltage/frequency, to meet the deadlines on available resources. To ensure low configuration time, the algorithm stores multiple versions in spare contexts. For our experiments, we use WLAN and Matrix Multiplication (MM). WLAN requires a stream to be processed in $4\mu sec$. Additionally, we assume that the application allows to buffer a single stream during reconfiguration stall. For MM, we assumed a synthetic deadline of $1msec$. Additionally, we assume that the applications allows to buffer a single stream during reconfiguration. Using these constraints on DRRA operating at 400 MHz frequency, the WLAN and MM are allowed to stall for 1.6 K and 400 K cycles, respectively. Fig. 3.16 shows the reconfiguration stalls, using different configuration modes. It can be seen that the desired configuration constraints for WLAN and MM are met by MFMC (requiring 11880 bits) and DF (requiring 1584 bits) modes, respectively (see Table 3.5). A traditional worst case architecture (using MFD mode) would require 32832 bits see Table 3.5. Therefore, even for this small example (using a single context), our architecture promises 58 % savings of configuration memory.

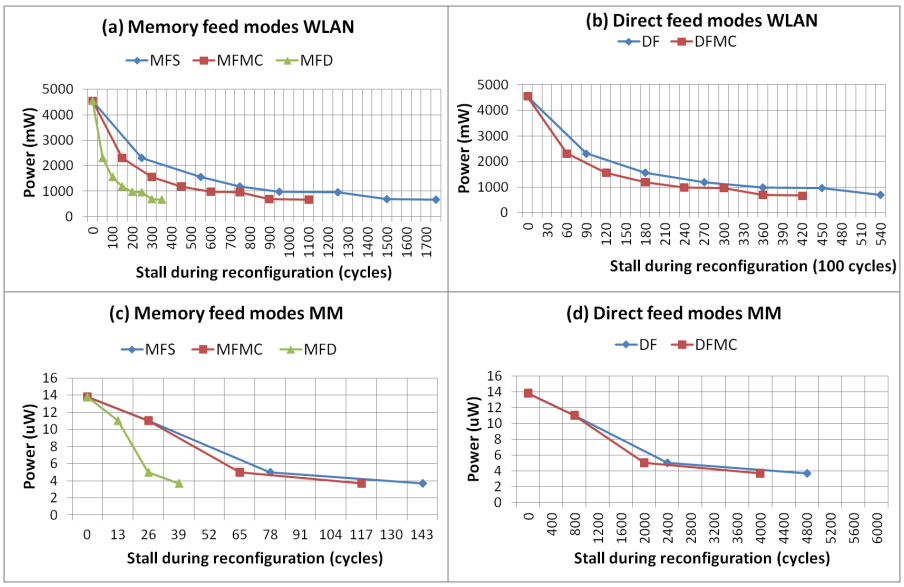

Figure 3.16: Stalls when applying late binding to WLAN and matrix multiplication

3.7.4 PCE in presence of compression algorithms

To reduce the configuration memory, DRRA supports two configuration compression schemes: (i) *loop preservation* and (ii) *Compact Generic Intermediate Representation (CGIR) based compression*.

Loop preservation saves memory by delaying the loop unrolling until the configware reaches the sequencer. Once the configware reaches the DRRA sequencer, an embedded hardware unit unrolls the loops and maps the instructions to the DRRA sequencers. It has been shown that the approach can save on average 55% configuration memory [91]. To evaluate the impact of loop preservation on configuration mode, we mapped six algorithms/applications (64 point FFT, 2048 point FFT, 2D convolution, matrix multiplication, and block interleaver) on DRRA fabric. The configuration cycles and the memory requirements of each configuration mode is shown in Table 3.7. The multi-casting modes are not shown since they are not supported in presence loop preservation. It can be seen that while overall data cycles and memory for all the applications reduces significantly, the difference in configuration modes remain constant.

Table 3.7: Reconfiguration cycles needed in different configuration modes with loop preservation

Application	Configuration mode		
	Direct feed (Cycles)	Memory feed (Cycles)	Memory feed distributed (Cycles)
FFT64 serial	2262	58	19
FFT2048	5460	140	23
2D conv	546	14	9
Matrix mult serial	468	12	12
Block interleaver	1872	48	8

Compact Generic Intermediate Representation (CGIR) is mainly intended to compress configware when multiple versions of an application (with different levels of parallelism) are stored. Storing multiple versions allows to enhance energy efficiency by dynamically parallelizing/serializing an application. Details about how energy efficiency is enhanced by using multiple versions can be found in [60][57]. CGIR compresses data by storing configware for only a single version. The rest of the versions are stored as differences from the original version. The decompression is performed in software by a LEON3 processor. Therefore, in the memory feed modes (MFD and MFMC) configware cannot be stored as a CGIR. To consider the impact of CGIR on different reconfiguration modes, we mapped IFFT (used in WLAN transmitter) with multiple versions on DRRA. The results

Table 3.8: Reconfiguration memory needed for different configuration modes with loop preservation

Application	Configuration mode	
	Direct feed (Bits)	Memory feed (Bits)
FFT64	2556	5112
FFT2048	5040	10080
2D conv	504	1008
Matrix mult	532	864
Block interleaver	1728	3456

Table 3.9: Configuration memory requirements for different versions of IFFT

Versions	No Compression		CGIR	
	DF (bits)	MFD (bits)	DF(bits)	MFD (bits)
1	4050	8100	4121	8100
2	8240	16480	5077	16480
3	12290	24580	6033	24580
4	16340	32680	6989	32680
5	20390	40780	7945	40870

are shown in Table 3.9 and depicted in Fig. 3.17. It can be clearly seen that after the CGIR based compression the difference between the memory requirements of direct feed and memory feed distributed modes increase significantly. The reason for the increase is that the decompression of CGIR into hard binary requires a processor, which is not available in the memory feed modes. From these results it is obvious that the CGIR based compression aggravates the need for proper mode selection.

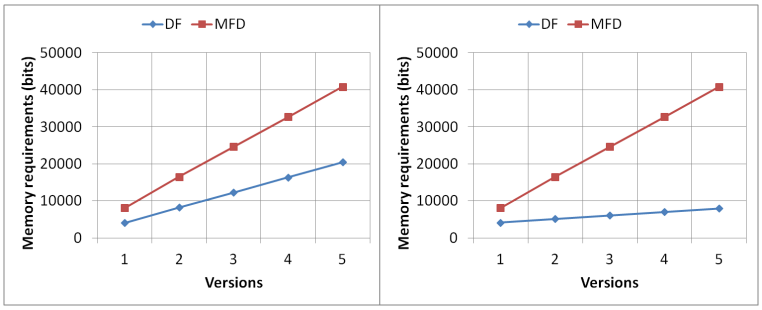

Figure 3.17: Effect of compression on IFFT

3.8 Summary

In this chapter, we have presented a morphable architecture, to provide the on-demand reconfiguration infrastructure to each application, hosted by a CGRA. On-demand reconfiguration was attained by using a morphable data/configuration memory supplemented by morphable hardware. By configuring the memory and the hardware, four configuration modes were realized: (i) direct feed, (ii) direct feed multi-cast, (iii) direct feed distributed, and (iv) multi context. To manage the process in a scalable fashion, a three-tier control backbone, was introduced. It was responsible for customizing the configuration infrastructure upon arrival of a new application. The obtained results suggest that significant reduction in memory requirements (up to 58 %) can be achieved by employing the proposed morphable architecture. Synthesis results confirm a negligible penalty (3 % area and 4 % power) compared to a DRRA cell. Future research on PCEs will involve development of a comprehensive reconfiguration mode selection algorithm. The algorithm, along with memory, will also take into account thermal and energy considerations for optimal mode selection. Additionally, we also plan to test the feasibility of other compression techniques (such as run length encoding and Hoffman encoding) on various reconfiguration modes.

Chapter 4

Private Reliability Environments for CGRAs

4.1 Introduction

With the progress in the processing technology, the size of semiconductor devices is shrinking rapidly, which offers many advantages like low power consumption, low manufacturing costs, and ability to make hand held devices. However, shrinking feature sizes and decreasing node capacitance, the increase of the operating frequency, and the power supply reduction affect the noise margins and amplify susceptibility to faults. It is therefore predicted that the number of on-chip faults will increase as technology scales further into the nano-scale regime, making fault-tolerance an essential feature of future designs [17]. In particular, bit-flips in storage elements called *Single Event Upsets* (SEUs), most often caused by cosmic radiation, are of major concern [63]. In this chapter, we will first present our work on developing private reliability environments for CGRAs followed by PREs for NoCs (in the next chapter).

4.1.1 Private reliability environments for computation, communication, and memory

The superior performance of CGRAs (compared to FPGAs) combined with the increasing importance of fault tolerance has lead the researchers have to develop CGRAs with reliability considerations [5, 56, 55, 6]. Novel CGRAs host multiple applications simultaneously on a single platform. Each application can potentially have different reliability requirements (e.g., a car braking system requires very high reliability while a video streaming can be accommodated on a less reliable platform). In addition, the reliability needs of an application can also vary depending on the operating conditions (e.g.

temperature, noise, voltage, etc.). Providing maximum (worst case) protection to all applications imposes high area and energy penalty. To cater this problem, recently, *flexible reliability schemes* have been proposed [6] [5] [55], which reduce the fault-tolerance overhead by providing only the needed protection for each application. Since the flexible reliability schemes provide each application with the fault-tolerance infrastructure tailored to its need, in this thesis we call them *Private Reliability Environments* (PREs). The existing architectures that offer flexible reliability, only allow to shift between different levels of *modular redundancy*. In modular redundancy, an entire replaceable unit (i.e. a module) is replicated, making it an expensive technique resulting in at least twice energy and area overhead. As an alternative to expensive modular redundancy, we propose a flexible fault-tolerant architecture that, besides modular redundancy allows to use low-cost protection based on Error Detecting Codes (EDCs) [61]. Compared to previously proposed flexible reliability schemes, that protect CGRAs against the same class of faults (e.g. SEUs), the proposed scheme (using EDCs) not only protects data memory, computations, and communications, but also offers significant reduction of energy consumption. In particular, we chose residue modulo (mod) 3 codes, because they have been known as one of the least costly methods which can be used to protect against undetected errors simultaneously in the computations, the data memory, and the communications [56][82]. Depending on the strength of the fault-tolerance approaches used (which imply different energy overhead), the proposed technique offers five different dynamically configurable reliability levels. Our solution relies on an agent based control layer and a reconfigurable fault-tolerance data path. The control layer identifies the application reliability needs and configures the data path to provide the needed reliability.

4.1.2 Private reliability environments for configuration memory

To protect the configuration memory we have used configuration scrubbing. The motivation for using configuration scrubbing in CGRAs is that the modern CGRAs enhance the silicon and power efficiency by hosting multiple applications, running concurrently in space and/or time. Some applications enjoy dedicated CGRA resources and do not require further reconfiguration, whereas some other applications share the same CGRA resources in a time-multiplexed manner, and thus require frequent reconfigurations. Additionally, some CGRAs [112] also support smart power management systems that can serialize/parallelize an application to enhance energy efficiency by lowering the voltage/frequency operating point. To address these requirements multiple copies of the configware are stored and techniques like configuration caching [109][88] and indirect reconfiguration [58][119] are

employed to configure/reconfigure applications. While these techniques do solve the problem, they impose high overheads in terms of configuration memory. Therefore, in many recently proposed CGRAs the configuration memory consumes significant percentage of the overall device area (50% in ADRES [124], 40% in MuCCRA [7], 30% in DRRA [112]). The large configuration memories make configuration scrubbing an interesting technique even for CGRAs. However, to the best of our knowledge, before our thesis the research on configuration scrubbing dealt only with FPGAs without any reference to CGRAs.

4.1.3 Motivational example

As a concrete motivational example (for private reliability environments) consider Fig. 4.1 which depicts a scenario in which a CGRA simultaneously hosts a car braking system and a DSP application (e.g. for video streaming). The dotted boxes indicate the resources occupied by each application. Obviously, the car braking system requires the highest reliability level, because each computation should be correct, on time, and cannot be dropped. We assume that Triple Modular Redundancy (TMR) provides the needed reliability. The computations for the DSP application can be classified into *critical/less-critical computations*, depending on their contributions towards the overall output quality (say, in terms of peak-signal-to-noise-ratio) [10]. While each critical computation is important and needs very high reliability (ensured e.g. by TMR), the less critical computations can be dropped if an error is detected (making a self-checking unit protected using EDCs sufficient). The static fault-tolerant architecture (Fig. 4.1b)) will waste energy because it will provide redundant *modules* for both applications (here, a module is a basic block that typically consists of an ALU, registers and a switch). A number of these modules are combined to realize a complete CGRA. The adaptive modular fault-tolerance (Fig. 4.1c)) enhances the fault-tolerance strength only at the modular level; it allows to increase energy efficiency, by providing separate redundancy for each application. The additional dotted line isolates the resources occupied by the critical (employing TMR) and the less critical (employing duplication with comparison (DWC)) parts of the DSP application. Our solution (Fig. 4.1d)) provides architectural support to allow shifting redundancy even at the sub-modular level.

The proposed scheme is generic and in principle applicable to all grid based CGRAs [43][103]. To obtain some realistic results, we have chosen a Dynamically Reconfigurable Resource Array (DRRA) [111], as a representative CGRA. Simulating practical applications (Fast Fourier Transform (FFT), matrix multiplication, and Finite Input Response (FIR) filter) shows that our solution provides flexible protection, with energy overhead ranging

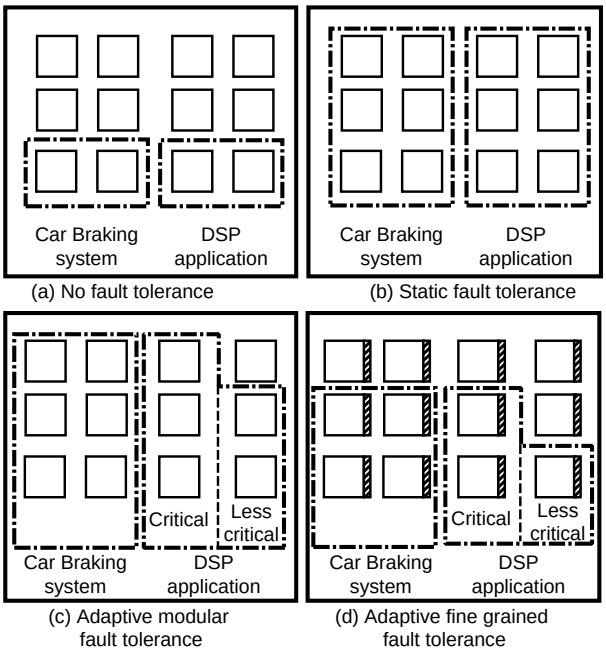

Figure 4.1: Comparison of different fault-tolerance architectures.

from 3.125% to 107% for self-checking to fault-tolerant versions, respectively. Synthesis results confirm that sub-modular redundancy significantly reduces the area overhead (59.1% and 7.1% for self-checking and fault-tolerant versions, respectively), compared to the-state-of-the art adaptive reliability methods.

4.2 Related Work

Since the last decade, fault-tolerance has been a subject of extensive research [78]. In this section, we will review only the most prominent works in adaptive fault-tolerance which are the most relevant to our approach.

4.2.1 Flexible reliability

Much of the work dealing with flexible fault-tolerance attempts to protect the communication system (especially in packet switched network-on-chips). Worm et al. [125] proposed a technique to scale supply voltage depending on observed error patterns. Assuming that the voltage level directly affects reliability, they suggested that a smaller voltage would be sufficient for transmission in less noisy execution conditions, thus increasing/decreasing the voltage depending on the noise level. This work was later used in [126] to propose a self-calibrating on-chip link, where the proposed architecture

achieves high-performance and low-power consumption by dynamically adjusting the operating frequency and voltage swing. Error detection was combined with retransmission to ensure reliability. Li et al. [79] showed that retaining the voltage and changing the fault-tolerance scheme provides a better improvement in reliability per unit increase in energy consumption. Based on their findings, they presented a system capable of dynamically monitoring noise and shifting amongst three fault-tolerance levels of different intensity (Triple ERror detection (TER), Double Error Detection (DED), and parity). The idea behind their strategy is to monitor the dynamic variations in noise behavior and to use the least powerful (and hence the most energy efficient) error protection scheme required to maintain the error rates below a pre-set threshold. Rossi et al. [105] included end-to-end fault-tolerance on specific parts of the Network-on-Chip (NoC) packet to minimize energy and timing overhead. A method for adapting error detection and correction capabilities at run-time, by dynamically shifting between codes of different strengths, was presented in [130] to tolerate temporary faults. The latter work was improved to handle both permanent and temporary faults [100]. The proposed scheme combines Error Correcting Codes (ECC), interleaving, and infrequently used spare wires to tolerate faults. Unfortunately, only a few works present attempts to provide adaptive fault-tolerance to protect computations in CGRAs. Alnajjar et al. [5], [6] proposed a coarse-grained dynamically reconfigurable architecture with flexible reliability to protect both computations and the configuration. The presented architecture offers flexible reliability level by allowing to dynamically shift between Double Modular Redundancy (DMR) and Triple Modular Redundancy (TMR). To reduce the overheads of this method, we presented an architecture to allow flexible reliability even at sub modular level [61].

4.2.2 Scrubbing

Various surveys and classifications of configuration scrubbing in FPGAs, can be found in the existing literature [12][48][75][49]. The scrubbing techniques can be classified on the basis of methodology (intelligence), architecture (location of the scrubber), and system level considerations (reliability offered and power consumed) [49]. On the basis of intelligence the scrubber can be either *blind*, *readback scrubber* or *error invoked scrubber* [48][49].. The blind scrubber scrubs the configuration memory after selected intervals [12], [48]. The readback scrubber first reads the configware form configuration memory and writes to the configuration memory only upon error detection [12], [48]. The error invoked scrubber reduces power consumption and time to recover from faults by combining high level error detection and correction techniques with the configuration scrubbing [13][49]. The scrubbing circuitry of the error invoked scrubber scrubs part of the system in error

upon error detection [81][18]. Depending on the scrubber's location, the configuration memory can be scrubbed internally or externally. In *internal scrubbing* the scrubbing hardware resides inside the reconfigurable device, whereas in *external scrubbing* the scrubbing circuitry is present outside the reconfigurable device [87][64]. On the basis of system level considerations the scrubbing techniques can be classified on the basis of the reliability they provide to a system. One of the adequate reliability measures of a system is the *Mean Time To Failure (MTTF)* which depends primarily on the scrubbing frequency. To calculate the scrubbing frequency, various models have been presented. For instance, Ju-Yueh Lee et al. [75] developed a model to quantify the effect of scrubbing rate on reliability using the stochastic system vulnerability factor of configuration memory bits. They also proposed a *heterogeneous scrubber* which scrubs different parts of the device at different rates depending on their effect on MTTF. In addition, the Markov model [114] and soft error benchmarking [115] (for caches) can also be extended for the configuration memories and used to determine the scrubbing rates. Most of the existing work on configuration scrubbing deals with FPGAs. In this thesis, we implemented and evaluated the efficacy of configuration scrubbing even on CGRAs.

4.2.3 Summary and contributions

The related work reveals that ECCs are mostly used to protect only the interconnects. Existing adaptive fault-tolerance techniques either employ expensive modular redundancy or leave the configuration memory unprotected. Our approach allows to shift reliability on a fine-granular level (using residue mod 3 codes). We will show that this seemingly small change in granularity would significantly enhance the energy efficiency of the whole system (see Section 4.8). In addition, we propose a unique framework for protecting the memory, computation, communication, and configuration against permanent and temporary faults.

By introducing the private reliability environment we made four major contributions:

1. We proposes a morphable fault-tolerance architecture that can be dynamically tailored to match the reliability needs of any application hosted by the CGRA. Compared to the state-of-the-art adaptive architectures that support adaptivity at modular level, our technique also incorporates Error Detecting Codes (EDCs). Thereby, it not only simultaneously protects data memory, computations and communications, but also promises a significant reduction in energy consumption;

2. We present an architecture for low-cost implementation of various configuration scrubbing techniques on a CGRA (so far implemented

only on FPGAs). These schemes on one hand allow to evaluate the overheads and performance of configuration scrubbing techniques and on the other allow to provide the scrubbing technique that optimally matches the scrubbing requirements of an application;

3. We introduce fault-tolerance agents (FTagents) that allow to adapt autonomously between different reliability levels, at run-time; and

4. We present an enabling control and management backbone that provides a foundation for the above concept by configuring the FTagents to meet varying reliability requirements.

4.3 System Overview

In this thesis, we have chosen the DRRA to test the effectiveness of our method. The block level explanation of DRRA computational layer, memory layer, and programming flow has already been presented in Chapter 2. Because we are looking at the DRRA structure specifically from the point of incorporating in it fine-grained (sub-modular) fault-tolerance, we will present a detailed description of its DPU. The DRRA architecture contains a DPU in every cell, in which the computations are performed. As shown in Fig. 4.2, each DPU has three adders, a multiplier, and a subtractor connected by a series of multiplexers. In addition, it contains saturation and Code Division Multiple Access (CDMA) logic to support an industrial application (with Huawei) [92][112]. It has a total of five inputs, four outputs, and a configuration signal CFG (not shown) to realize different DSP functions detailed in Table 4.1. For a detailed discussion and motivation for using the DRRA architecture, an interested reader can refer to [92][112][36].

4.4 Fault Model and Infrastructure

In this thesis, we present a configurable framework to protect the data path against temporary and permanent faults that cause single bit errors. To address the temporary faults, we consider the *Single Event Upsets* (SEUs) which are bit-flips in storage elements, most often caused by cosmic neutron strikes. The motivation for choosing SEUs is that they constitute a major percentage of all faults in modern VLSI digital circuits [63]. An SEU can lead to erroneous data by flipping a bit in the storage elements (data memory, configuration memory, or the registers in computation and communication blocks). The proposed architecture handles single bit errors in computation, communication, and data memory. To protect the configuration memory, a scrubbing scheme similar to [47] can be used. In addition to SEUs, our architecture also handles permanent faults causing single bit data errors.

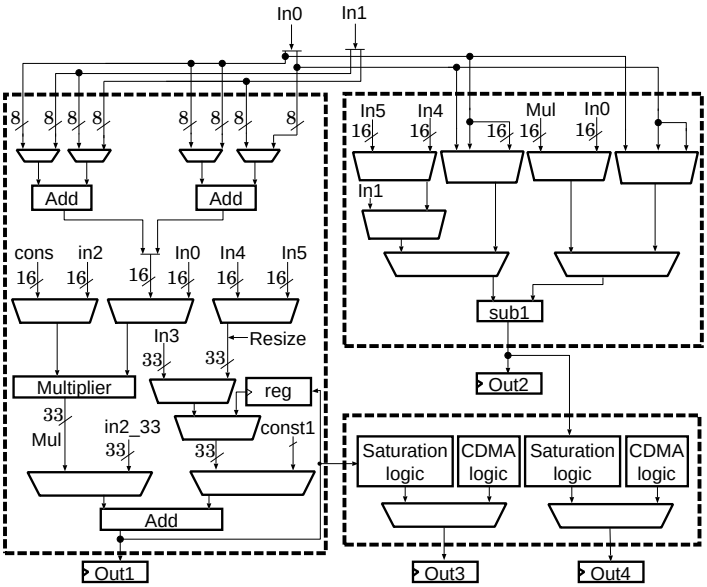

Figure 4.2: DRRA Data Path Unit (DPU).

Table 4.1: DPU functionality.

CFG	Functionality
007	Symmetric serial FIR filter with internal accumulation
003	Symmetric FIR MAC with external accumulation
006	Asymmetric FIR MAC with internal accumulation
002	Asymmetric FIR MAC with external accumulation
20A	FFT butterfly
000	Simple multiplication, two input add
200	Two input subtractor OVSF and scrambler code generator
050	Initialize scrambler registers
030	Shift scrambler registers and code generator
080	Vector rotator MAC
101	Complex number multiplication

OVSF = Orthogonal Variable Spreading Factor
MAC = Multiplier-Accumulator

4.4.1 Residue Mod 3 Codes and Related Circuitry

To handle single bit temporary errors, we use Error Detecting Codes (EDCs). The reason for using them is that modular redundancy approaches, like Duplication With Comparison (DWC) and TMR, not only require prohibitive (over twice and thrice) overhead, but also leave the data memory unprotected (unless the memory system is duplicated, triplicated or uses a sepa-

rate EDC). EDCs like parity checking or arithmetic residue modulo (mod) A codes (A odd integer) require less hardware overhead and can be used to protect memory, computations, and communications. Although simple parity code requires just one additional bit, it incurs excessive overhead (70–90% area [82]) to protect the arithmetic circuitry (adders, multipliers, and subtractors). Therefore, we employ the residue mod 3 codes that detect all arithmetic errors that do not accumulate to a multiple of 3 (hence, all single bit errors), while incurring smaller overhead [56][82].

Working Principle

Fig. 4.3 shows the general scheme of a self-checking arithmetic circuit protected using the residue code mod A. The circuit works as follows. Two operands X and Y along with their check parts mod A, $|X|_A$ and $|Y|_A$, arrive at the inputs of the circuit. The same arithmetic operation $* \in (+, \times, -)$ is executed separately and in parallel on input operands and their check parts, to produce the result $Z = X * Y$ and the check part of the result $|Z|_A = ||X|_A * |Y|_A|_A$. From the result Z, the check part $|Z|_A^*$ is generated independently and used as the reference value for the comparator. Any fault in the arithmetic circuit or the residue generator mod A may influence only the value of $|Z|_A^*$. Similarly, any fault in the arithmetic circuit mod A may influence only the value of $|Z|_A$. Therefore, assuming that no single fault in any of three blocks (arithmetic circuit, arithmetic circuit mod A, and residue generator mod A) produces an error whose arithmetic value is a multiple of A, any such an error would result in a disagreement $|Z|_A \neq |Z|_A^*$, indicated by the comparator.

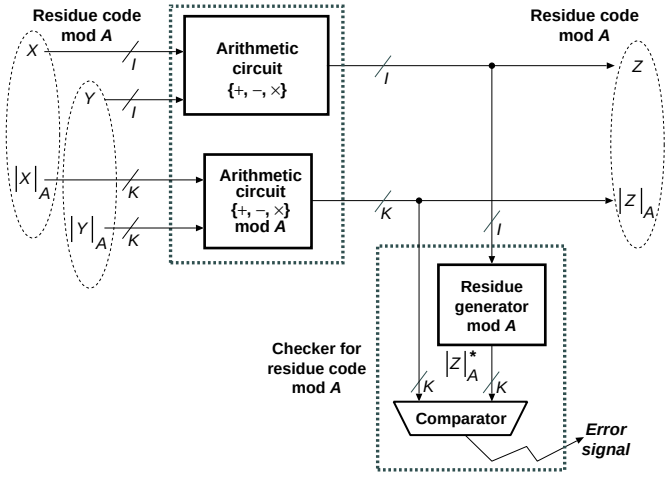

Figure 4.3: Working principle of residue mod 3.

Implementation

The basic arithmetic blocks needed to realize residue mod 3 checking in the DRRA architecture are shown in Fig. 4.4. Fig. 4.4a) shows the logic scheme of the adder/subtractor mod 3, where $|X|_3 = |x_1x_0|$ and $|Y|_3 = |y_1y_0|$ respectively denote the residue mod 3 check parts of the operands X and Y, $S = (s_1s_0)$ represents the check part of the result, and FA denotes a full-adder. Figs. 4.4b) and 4.4c) show the logic schemes of the multiplier residue mod 3 and the 8-input generator of residue mod 3. Assuming that $X = (x_{n-1}, ..., x_1, x_0)$ is an operand to be protected against errors, the residue mod 3 generator calculates $R = (R_1R_0)$, which is the remainder of the integer division of X by 3. For this thesis, we have employed the efficient residue generators from [98]. Residues for bigger numbers can be calculated easily by first calculating the residue mod 3 of each part separately and then adding them mod 3.

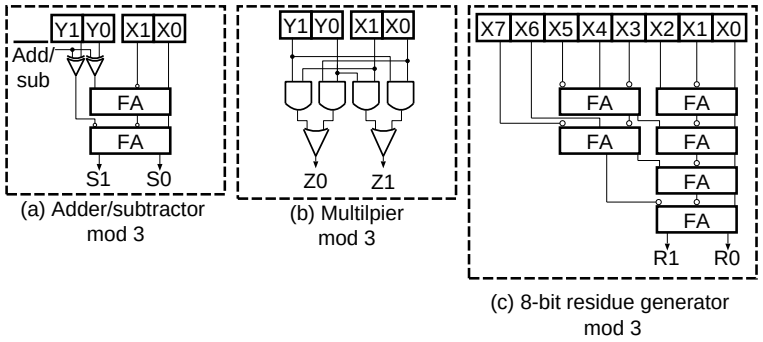

Figure 4.4: Residue adder/subtractor, multiplier, and generator mod 3.

4.4.2 Self-Checking DPU

To protect computations against errors resulting from temporary faults, we have modified the data paths of the four outputs: Out1, Out2, Out3, and Out4 (cf. Fig. 4.2). The data paths of Out1 and Out2 are mainly composed of arithmetic circuits $(\pm, \times)$ whose functioning can be efficiently verified by their residue mod 3 equivalents working in parallel (as shown in Fig. 4.4). Fig. 4.5 illustrates the circuits used to generate Cp1 and Cp2—the check parts for the data paths of Out1 and Out2, respectively. It will be shown later in Section 4.8 that the residue mod 3 equivalents require significantly smaller overhead, compared to corresponding arithmetic blocks they protect. The data paths with outputs Out3 and Out4 contain logic blocks (CDMA and saturation). To verify these outputs, we simply duplicate their data paths, using an additional hardware block called Cp3-4. Henceforth, all the check parts will be jointly referred to as replicas.

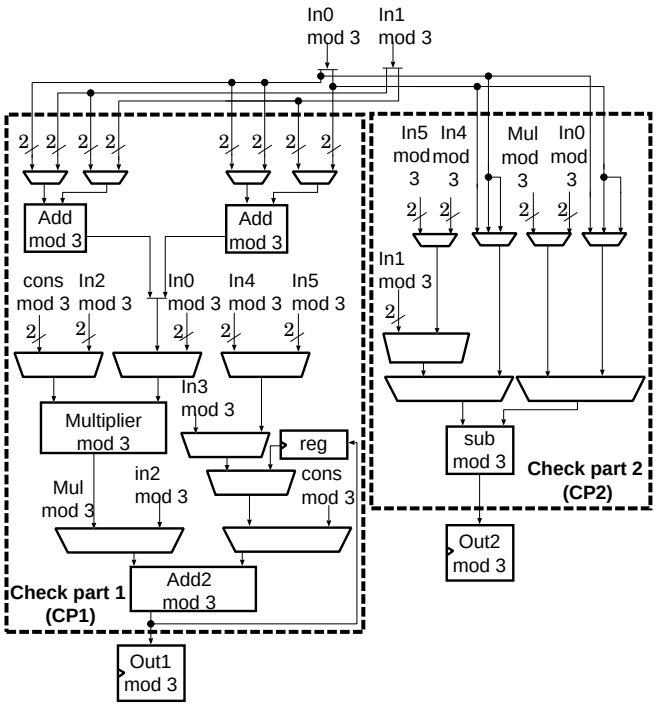

Figure 4.5: Self-checking hardware to check Out1 and Out2.

Fig. 4.6 illustrates how the replicas are combined to realize a self-checking DPU. We will explain the scheme with the data flowing from the top to the bottom. Initially, four 18-bit inputs (only one input is shown to enhance visibility), each containing 16 data bits and 2 check bits arrive at the input of the DPU. At this stage, the data bits and the check bits are separated. 16 data bits are sent to the original DPU, Cp3-4, and *MSB mod 3* blocks. The MSB mod 3 block generates $|msb|_3$—the residue mod 3 for 8 most significant bits of the 16-bit operand which is sent to the Cp1 and Cp2. At the same time, the Cp1 and Cp2 also receive the check bits (from the input). Considering that the residue mod 3 of an entire word equals the sum of the residues mod 3 of all its parts, the Cp1 and Cp2 generate the residue mod 3 for least significant bits (LSBs), $|lsb|_3$ according to

$$| \, |msb|_3 + |lsb|_3 \, |_3 = |operand|_3 \tag{4.1}$$

and

$$| \, |lsb|_3 = |operand|_3 - |msb|_3 \, |_3 \tag{4.2}$$

performed using the adder/subtractor mod 3 of Fig. 4.4. Having available the $|msb|_3$ and $|lsb|_3$, the replicas perform the calculations simultaneously with the original DPU. Finally, the results calculated from the replicas are compared to the outputs of the original DPU to detect errors.

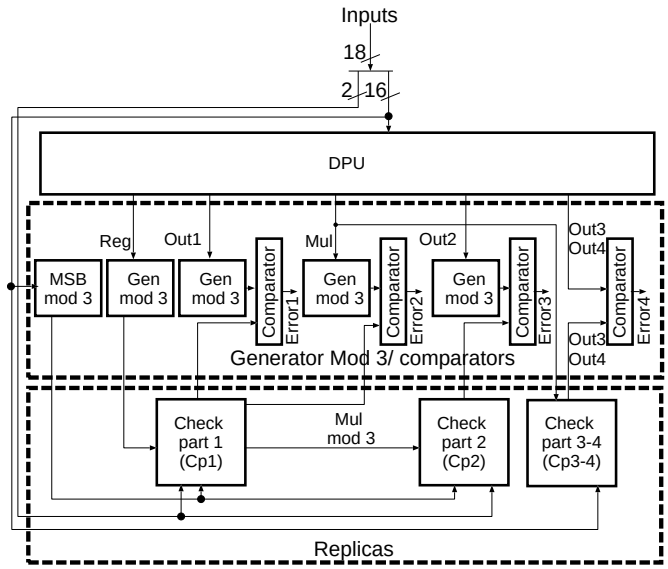

Figure 4.6: Self-checking DPU using residue code mod 3 and duplication.

4.4.3 Fault-Tolerant DPU

To realize a fault-tolerant DPU, we considered two alternatives: (i) recomputation and (ii) duplication (by combining two self-checking DPUs). For

recomputation, the error signal from the self-checking DPU is fed to the DRRA register files (see Section 2.1.1). If the error signal is activated, the data transmission is stalled and recomputation is executed. A detailed discussion of recomputation is beyond the scope of this thesis, and some details can be found e.g. in [9]. The fault-tolerant DPU can also be realized by combining two self-checking DPUs, as shown in Fig. 4.7: the output selector allows to move forward only the error free output. The architectural modifications needed to realize this architecture dynamically will be discussed later in Section 4.5.

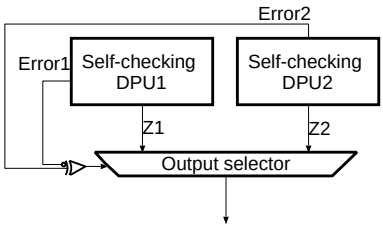

Figure 4.7: Fault-tolerant DPU built using two self-checking DPUs.

4.4.4 Permanent Fault Detection

To detect a faulty DPU, we have used the state machine containing three states, shown in Fig. 4.8:

1. As long as no fault is detected in the system, the state machine remains in state *No error*. If an error signal is activated, the state machine changes its state to *Tempft*, denoting detection of a temporary fault.

2. Once in the *Tempft* state, a counter is initialized. Upon consecutive occurrence of errors in the same DPU, the counter is incremented. Should the value of the counter exceed a pre-defined threshold, a permanent fault is declared and the state machine shifts to *Update RTM* state, where RTM denotes the run-time resource manager.

3. In Update RTM state, the RTM updates information about detected permanent faults.

The reason for choosing this widely used methodology for detecting on-line permanent faults was relative ease of its implementation [55][39].

4.5 Private Reliability Environments

To efficiently meet the reliability requirements of multiple applications, the proposed architecture dynamically creates a separate fault-tolerant partition for each application, called private reliability environment (PRE). To

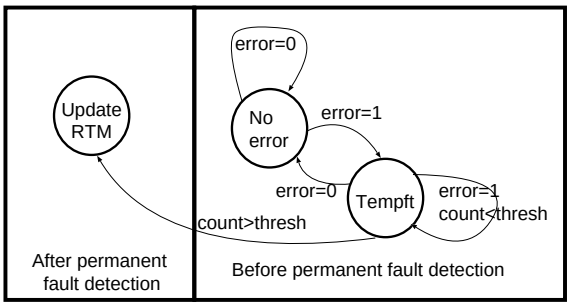

Figure 4.8: Permanent fault detection state machine.

clarify the concept of the PRE, let us consider the case of a DRRA instance that hosts simultaneously two applications, shown in Fig. 4.9. It is assumed that each of these two applications requires a different reliability level. Therefore, assuming the worst-case for both of them, and hence applying the same fault-tolerance techniques to both the applications, would clearly waste energy and area. The proposed technique reduces the overhead involved by morphing into private reliability environment 1 and private reliability environment 2 for application 1 and application 2, respectively. To create PREs, the fault-tolerance needs of each application are stored along with its configware in the global configuration memory.Depending on the reliability needs, a separate thread in the RTM configures a set of *Fault-Tolerance agents (FTagents)* which activate/deactivate different parts of the fault-tolerance infrastructure to meet the exact fault-tolerance needs of the mapped application. The FTagent will be detailed later in Sections 4.5.2 and 4.5.3.

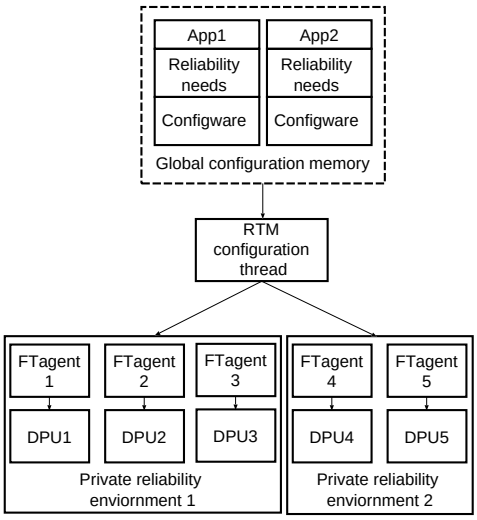

Figure 4.9: Private reliability environments.

Table 4.2: Fault-tolerance levels.

Reliability level	Technique used	Faults covered
RL1	None	No fault-tolerance
RL2	Res mod 3	SEU detection
RL3	Res mod 3 + state machine	SEU and permanent fault detection/diagnosis
RL4	Res mod 3 + DMR	SEU detection/correction
RL5	Res mod 3 + DMR + state machine	SEU and permanent fault detection/correction

4.5.1 Reliability Levels

We have defined five different fault-tolerance levels (RL1-RL5) with growing fault-tolerance strength, and hence requiring higher and higher overhead, as shown in Table 4.2. The lowest level RL1 offers no fault-tolerance, so the replicas and residue mod 3 generators (see Fig. 4.6) are switched off thus consuming no dynamic energy. In RL2, the replicas, residue mod 3 generators, and logic duplication units are all activated to allow for SEU detection. In RL3, besides SEU detection, permanent faults causing single bit data errors are also detected, by activating the state machine of Fig. 4.8. In RL4, two self-checking DPUs are combined to realize a fault-tolerant DPU that can detect and correct SEUs as well as tolerate permanent faults causing single bit data errors. Finally, the highest level RL5, besides the protection level offered by RL4, it also allows to diagnose permanent faults and signal them to the RTM which then can trigger an appropriate action.

4.5.2 Fault-Tolerance Agent (FTagent)

To ensure that each application is provided with the required reliability level autonomously, we have embedded a *fault-tolerance agent* (FTagent) into each self-checking DPU (see Fig. 4.10). The FTagent is a passive entity that is controlled by the RTM. Each FTagent, denoted by $FTa_{i,j}$, is connected to the FTagents $FTa_{i+1,j}$ and $FTa_{i,j+1}$, by a 1-bit feedback wire, where i and j represent the row and column number of an FTagent. The feedback wire is used to send an error signal when two self-checking DPUs are dynamically combined to implement a fault-tolerant DPU (capable of detecting and correcting SEUs). The details of how the fault-tolerant DPU is realized at run-time, will be discussed below in Section 4.5.3. To map an application, the RTM sends its reliability requirements to the FTagents which activate only those parts of the self-checking DPU circuitry that are essential to provide the required reliability level. Thereby, the dynamic energy consumption is significantly reduced compared to static fault-tolerance.

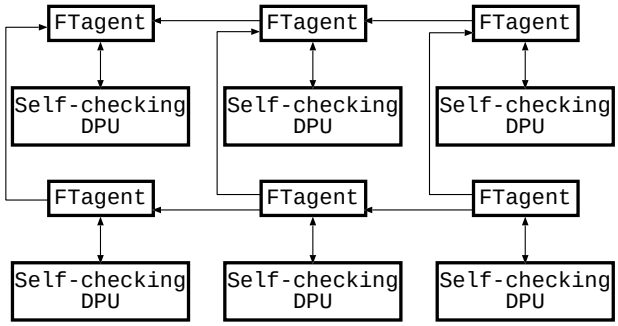

Figure 4.10: Fault-tolerance agent integration.

4.5.3 Run-Time Private Reliability Environments Generation

Fig. 4.11 illustrates the FTagent circuitry that allows to dynamically shift between different reliability levels. The RTM controls the FTagent using three control bits: (i) Enable Self-Checking (ESC), (ii) Enable Permanent fault Detection (EPD), and Enable Fault-Tolerance (EFT). Table 4.3 shows the bit values and the corresponding reliability levels. The ESC bit controls Multiplexer 1 and Demultiplexer 2 to decide whether the self-checking replicas should be enabled. The EPD bit decides whether the state machine should be activated to allow for permanent fault detection. To implement the fault-tolerant reliability levels RL4 and RL5, two self-checking DPUs are dynamically combined, as discussed previously in Section 4.4.3. The proposed architecture allows to combine the self-checking DPU, $DPU_{i,j}$, with one of the neighboring DPUs, $DPU_{i+1,j}$ or $DPU_{i,j+1}$, where i and j represent the DPU row and column numbers, respectively. To interchange between self-checking and fault-tolerant reliability levels, the Enable Fault-Tolerance (EFT) bit is used. In the fault-tolerant mode, the outputs of both self-checking DPUs are connected to the same line. The Output Select (OtS) bit determines which of the two outputs should be forwarded to the common line. The value of the OtS bit itself can be determined with the help of Table 4.4, where OrD, ErN, and Error denote respectively the DPU order, an error in the neighboring DPU (shown in Fig. 4.10), and an error in the same DPU. For example, consider the self-checking $DPUi, j$ combined with one of the self-checking DPUs $DPU_{i+1,j}$ or $DPU_{i,j+1}$, to realize a fault-tolerant DPU. The self-checking $DPU_{i,j}$ will have $OrD = 0$, while the self-checking $DPU_{i+1,j}$ or $DPU_{i,j+1}$ will have $OrD = 1$. In absence of any error ($ErN = 0$ and $Error = 0$), the first DPU (i.e. the DPU with $OrD = 0$) outputs data while the DPU with $OrD = 1$ outputs high

Table 4.3: Control bits and the corresponding reliability level.

Control bits			Reliability
ESC	EPD	EFT	level
0	0	0	RL1
1	0	0	RL2
1	1	0	RL3
1	0	1	RL4
1	1	1	RL5

impedance signal. If erroneous data is detected in one of the DPUs (indicated by the positive ErN or the Error bit), the error free data is forwarded. The minimized logic circuitry generating the OtS bit is given by

$$OtS = \overline{Error} \cdot OrD + \overline{Error} \cdot ErN \tag{4.3}$$

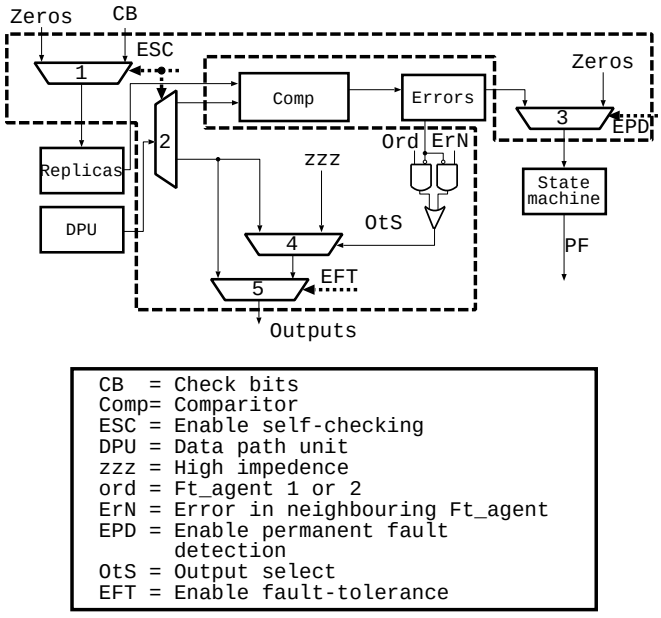

Figure 4.11: Interface of a fault-tolerance agent with a self-checking DPU.

4.5.4 Formal Evaluation of Energy Savings

To visualize the potential savings of the proposed method, first we will present here a simplistic energy model, whereas more accurate synthesis results, obtained using Synopsys Design Compiler, will be given in Section 4.8. The energy, $E_{ft}(i)$, needed to execute an application, $A(i)$, on a static

Table 4.4: Truth-table of the output select signal OtS.

Inputs			Output
OrD	ErN	Error	OtS
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

fault-tolerant architecture is given by

$$E_{ft}(i) = E_a(i) + E_{ft}(max), \tag{4.4}$$

where $E_a(i)$ is the energy required to execute $A(i)$ and $E_{ft}(max)$ is the energy required to provide fault-tolerance to the application needing the maximum reliability level.

The total energy, E_{ftt}, needed to execute a set of applications, A, by a CGRA with static reliability level is given by

$$E_{ftt} = \sum_{i=1}^{A}(E_a(i) + E_{ft}(max)). \tag{4.5}$$

The energy, E_{fr}, required to execute an application on a platform with flexible reliability level is given by

$$E_{fr}(i) = E_a(i) + E_{RL}(i) + E_{conf}, \tag{4.6}$$

where $E_{RL}(i)$ is the energy consumed by the i^{th} application requiring a given reliability level and E_{conf} is the additional energy required to configure the reliability levels.

The total energy, E_{frt}, needed to ensure suitable fault-tolerance methods for a set of applications, A, hosted by a CGRA (with support for flexible reliability levels) is given by

$$E_{frt} = \sum_{i=1}^{A}(E_a(i) + E_{RL}(i) + E_{conf}). \tag{4.7}$$

Eqns 4.5 and 4.7 indicate that for a set of applications, A, an architecture with flexible reliability promises a decrease in energy consumption provided that $E_{ft}(max) - (E_{ft}(RL)) + E_{conf}) > 0$. Since in many application domains,

the reliability requirements of different applications vary significantly (e.g., a car braking system and a video decoder have significantly different reliability needs), on-demand fault-tolerance promises massive energy savings, provided that the actual implementation guarantees that E_{conf} is indeed relatively low. As shown in Figs 4.11, the on demand circuitry is composed of simple multiplexers that consume negligible energy (see Section 4.8).

4.6 Configuration memory protection

To support advanced features like configuration caching [109, 88], runtime parallelism and indirect reconfiguration [58] modern CGRAs host big configuration memories (50% in ADRES [124], 40% in MuCCRA [7], 30% in DRRA [112]), making configuration scrubbing an interesting technique for CGRAs. Various scrubbing techniques (discussed in Section 4.2) have different overheads and benefits (as will be shown later in sections 4.7 and 4.8). Therefore, instead of implementing a dedicated scrubber, we present an architecture to support multiple scrubbing techniques (shown in Section 4.2). In Section 4.8) we will show the the architecture to support multiple scrubbing schemes requires negligible overheads compared to a dedicated scrubber.

4.6.1 Morphable Configuration Infrastructure

Fig. 4.12 depicts a high-level overview of the DRRA configuration infrastructure. The configware is fed to the DRRA sequencers either directly by LEON3 or indirectly by DiMArch. To feed the sequencers directly, LEON3 performs three steps: (i) it loads the configware from the configware bank to the Horizontal Bus (HBUS); (ii) it asserts the sequencer addresses directly to DRRA using RowMultiC (see [123]); and (iii) it directs the memory sequencers which are present in each column of the DRRA to copy the data from the HBUS to vertical bus VBUS, thus effectively broadcasting data to all sequencers. To feed the DRRA via DiMArch, the configware is first loaded to DiMArch by the memory sequencers using the DiMArch network. Then, the configware from the DiMArch memory banks is transferred to the DRRA sequencers, depending on the instructions of the LEON3. It will be shown later in Sections 4.6.2 and 4.8 that this architecture supports scrubbing techniques discussed in Section 4.2 incurring minimal overhead.

4.6.2 Scrubbing Realization in DRRA

Table 4.5 summarizes how scrubbing methods based on location and intelligence are implemented. Fig. 4.13 depicts the datapaths to realize various

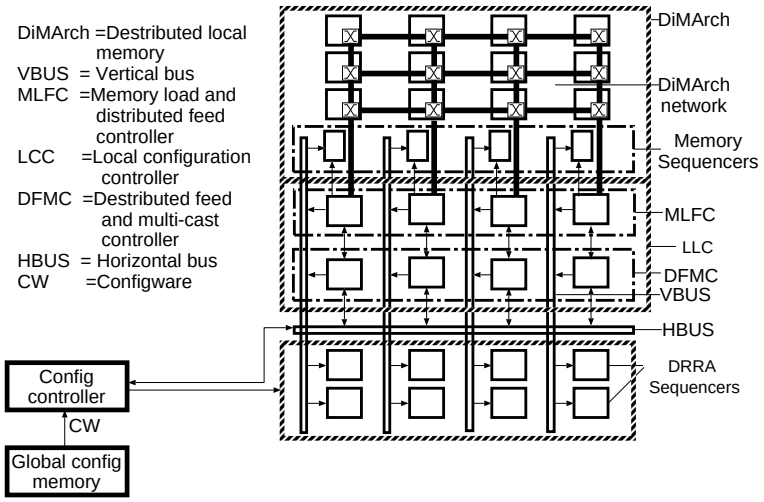

Figure 4.12: Private Configuration Environment (PCE) infrastructure

scrubbing techniques. To implement the external scrubber, the LEON3 processor picks the configuration words from the global configuration memory and transfers them via AHB, application controller, and local controllers to the DRRA sequencers. To realize the internal scrubber, the DiMArch sequencers are initially programmed by the LEON3 processor. Once programmed, the memory sequencer can configure DRRA sequencers during each cycle (without any support from the processor). Both the configuration memories (global configuration memory and DiMArch) are volatile memories that are filled when the system is powered on. In this thesis, we assume that the first configuration (i.e. copied from non-volatile memory to global configuration memory) is correct. Techniques to ensure the validity of the first configuration will be considered in the future.

To realize various levels of intelligence, the scrubbing algorithms presented Section 4.2 were implemented on LEON3 processor and the DiMArch sequencers respectively for the internal and external scrubber.

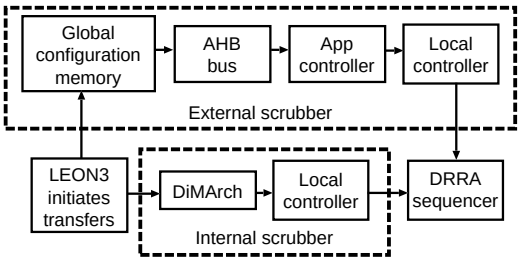

Figure 4.13: Architecture for internal and external scrubbers

To support a heterogeneous/homogeneous scrubber (Section 4.2), we

Table 4.5: Summary of how various scrubbing techniques are realized

Classification criteria	Scrubber	Implementation
Location	External	Sequencers scrubbed from Global Configuration Memory (GCM)
	Internal	Sequencers scrubbed from DiMArch
Intelligence	Blind	Sequencers scrubbed blindly after statically calculated period from DiMArch or GCM
	Readback	Configware from sequencers read and checked after statically calculated period
	Error invoked	Configware from sequencers read and checked after error detection

exploit the memory sequencers present in DiMArch. Each memory sequencer can be programmed to scrub the sequencers at a different frequency (by LEON3). Therefore, each application on the DRRA fabric can have a scrubbing scheme tailored to its reliability needs. The required scrubbing frequency can be calculated on the basis of the methods proposed in [75][115][114].

4.7 Formal Modeling of Configuration Scrubbing Techniques

In this section, we will formally analyze the memory timing and energy overhead of each scrubbing technique. The basic assumption is that a given sequencer can be scrubbed without affecting all the remaining sequencers. The main motivation for providing these formalizations is to provide a generic framework to estimate the impact of scrubbing strategy on memory requirements, reconfiguration cycles, and reconfiguration energy. The actual results on a representative platform will be given in the next section.

4.7.1 Memory Requirements

The configuration memory requirements depend mainly on whether a reliable memory SEU hardened or protected by ECCs is used. The total number

of bits needed for an unprotected configuration memory is

$$CM_{nft} = \sum_{i=0}^{seq-1} W_i * l_{CW}, \tag{4.8}$$

where seq, W_i, and l_{CW} respectively denote the total number of sequencers in DRRA, the number of configuration words of the i^{th} sequencer, and the length of a configuration word. The size of the configuration memory required to implement configuration scrubbing, using a reliable duplicate is

$$CM_{dup} = \sum_{i=0}^{seq-1} (W_i * l_{CW}) + \sum_{i=0}^{seq-1} (W_i * l_{FCW}), \tag{4.9}$$

where l_{FCW} denotes the length of a configuration word in memory. Here, we assume that the reliable memory is realized using an error detecting and correcting code protection, which implies $FCW > CW$. Since the blind scrubbing requires a reliable duplicate, Eqns 4.8 and 4.9 indicate that the blind scrubbing requires twice the area of an unprotected memory. The other schemes (readback and error invoked scrubber) can also be realized with a reliable duplicate memory. However, modern memory schemes commonly employ EDCs and/or ECCs to reduce the overhead, and the size of thus protected configuration memory is

$$CM_{ECC} = \sum_{i=0}^{seq-1} (W_i * l_{CW} + l_{ECC}), \tag{4.10}$$

where l_{ECC} is the length of the check part of the EDC/ECC employed. From Eqns 4.8, 4.9, and 4.10 it is obvious that for memory requirements the following inequalities hold: $CM_{nft} < CM_{DUP} < CM_{ECC}$. Hence, a blind scrubber is expected to require larger memory than the readback or error invoked scrubber.

4.7.2 Scrubbing Cycles

The number of scrubbing cycles is mostly dependent on the datapath employed for configuration (i.e. the location of the scrubber). The time needed to configure an application using an external scrubber is given by

$$t_{ext} = \sum_{i=0}^{seq-1} (Wi * Cyc_{GCM} + Cyc_{AHB} + Cyc_{app} + Cyc_{lcc}), \tag{4.11}$$

where Cyc_{GCM}, Cyc_{AHB}, Cyc_{app}, and Cyc_{lcc} denote the number of cycles respectively needed by the global configuration memory, the AHB bus, the

application controller, and the local controller, to transfer a sequencer instruction. The time needed to configure an application using an internal scrubber is

$$t_{int} = \sum_{i=0}^{seq-1} (Wi * Cyc_{DiMArch} + Cyc_{lcc}), \tag{4.12}$$

where $Cyc_{DiMArch}$ and Cyc_{lcc} denote the number of cycles needed respectively by DiMArch and the local controller to transfer a sequencer instruction. Therefore, from Eqns 4.11 and 4.12 we can infer that $t_{ext} \gg t_{int}$ provided that $Cyc_{DiMArch} \ll Cyc_{GCM} + Cyc_{AHB} + Cyc_{app}$. It will be shown in Section 4.8 that it is indeed the case.

4.7.3 Energy Consumption

In this section, we will present a simplistic energy consumption model to visualize the impact of using scrubbing on energy consumption. The actual energy estimates, obtained using Synopsys Design Compiler, will be reported in Section 4.8.

The configuration energy needed to configure DRRA is

$$E_{cfg} = \sum_{i=0}^{seq-1} W_i * E_{CW}, \tag{4.13}$$

where E_{CW} is the energy required to transport a configuration word from the memory to the sequencer.

The energy needed to scrub the configuration memory depends on both the location of the scrubber and the technique employed to implement scrubbing.

The configuration energy needed for blind scrubbing is

$$E_{bl} = SCR_{cyc} * EWB_{cfg}, \tag{4.14}$$

where SCR_{cyc} is the number of times the configuration is written during the application execution (it depends on the reliability needs and external noise level).

The configuration energy needed for readback scrubbing is

$$E_{rb} = SCR_{cyc} * (ERD_{cfg} + EWR_{cfg}) * E_{cmp}, \tag{4.15}$$

where ERD_{CFG}, EWR_{cfg}, and E_{cmp} respectively denote the energies needed to read the configuration word, to write to the configuration memory, and to perform comparison. EWR_{cfg} and E_{cmp} depend on whether DWC or ECC is employed. Eqns 4.14 and 4.15 indicate that the blind scrubbing consumes less energy than the readback scrubbing.

The configuration energy needed for error invoked scrubber is

$$E_{sys} = SCR_{cyce} * (ERD_{cfg} + EWR_{cfg}) * E_{cmp}, \tag{4.16}$$

where SCR_{cyce} is the number of the scrubbing cycles which for error invoked scrubber depends on the number of detected errors. It should be noted that although the architectural components (sequencers, instructions, AHB bus etc.) used to realize these formalizations were applied specifically for DRRA, the formalizations presented can be adapted for other architectures by replacing the appropriate components, e.g. the distributed sequencers can be replaced by the centralized configuration memory for ADRES.

4.8 Results

In this section we will separately discuss the benefits and overheads of using adaptive sub-modular redundancy and configuration scrubbing adaptively. The architecture and formalizations used in this thesis are generic and, basically, should be applicable to most grid based CGRAs as well. Since the existing CGRAs vary greatly in architecture, it is not possible to provide concrete generic results as well. Therefore, we have chosen DRRA as a representative platform because of the following reasons: (i) it is well documented [112], (ii) it has been used in both industry [92] and academia [112], and (iii) we had available all its full architectural details from RTL codes design to physical layout.

4.8.1 Sub-modular redundancy

To analyze the benefits of using residue mod 3 codes rather than modular redundancy techniques, we have synthesized the self-checking and fault-tolerant versions of DPUs. The area and power overhead of unprotected, self-checking, and fault-tolerant versions of DPU are shown in Table 4.6 and depicted in Fig. 4.14. The second column of Table 4.6 shows the parameters of the self-checking DPU using DMR and residue mod 3 circuits. The self-checking DPU using DMR was realized by duplicating the entire DPU followed by a comparator of the results. The self-checking DPU using residue mod 3 was realized by using circuitry shown in Figs 4.5 and 4.6. The table clearly shows that the residue mod 3 circuitry requires significantly smaller overhead in terms of area (59%) and power (57%) compared to DMR. The fault-tolerant DPU using TMR was realized by triplicating the entire DPU followed by a 2-out-of-3 voter selecting the correct result. The fault-tolerant DPU using residue mod 3 was realized using circuitry shown in Figs 4.7 and 4.10. It is seen that two self-checking residue mod 3 DPUs can be combined to realize a fault-tolerant DPU with lesser overhead than TMR. It should

Table 4.6: Area and power overhead of self-checking and fault-tolerant circuits using residue code mod 3, DMR, and TMR.

	DPU	Self-checking		Fault-tolerant	
		DMR	Mod 3	TMR	Mod 3
Area $[\mu m^2]$	13563	27868	19890	40984	39978
Power $[mW]$	4.5	9.16	6.61	13.78	13.2

be noted that the architecture is capable of dynamically shifting between self-checking and fault-tolerant versions.

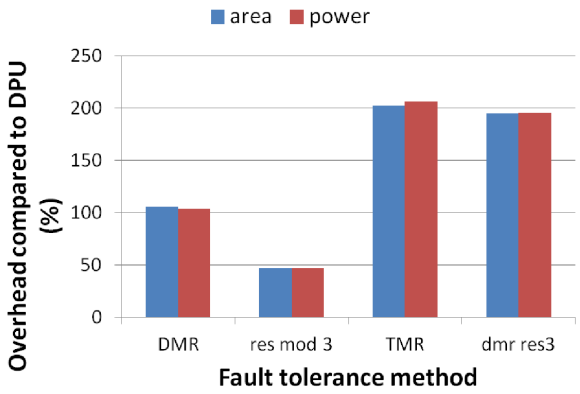

Figure 4.14: Overhead evaluation of self-checking and fault-tolerant DPUs using residue mod 3 code, DMR, and TMR.

Fig. 4.15 shows the area breakdown of self-checking DPU. It can be seen that most of the extra area (74%) is consumed by the duplicated DPU logic. As for the residue mod 3 circuits of Fig. 4.6, most of the extra area (18%) consume the residue generators mod 3. The adders, subtractors, and multipliers mod 3 consume only negligible area compared to the remaining part of the self-checking circuitry.

To evaluate the energy/power consumption benefits of the fault-tolerance on-demand, we performed gate level simulations by mapping three representative applications (FFT, matrix multiplication, and FIR filtering) on DRRA. Figs 4.16 and 4.17 show the energy consumed by the self-checking and fault-tolerant circuitry for executing the above three benchmarks. It can be seen that using on-demand fault-tolerance incurs the power overhead varying from 3.125% to 107%, depending on the reliability level. In summary, the proposed energy aware fault-tolerance approach allows to save up to 107% energy overhead (compared to the worst case approach) and 58% area (compared to known state-of-the-art techniques offering flexible reliability levels).

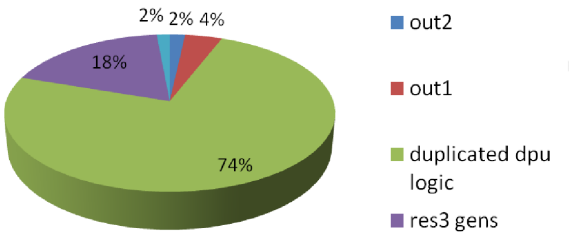

Figure 4.15: Area breakdown for overall fault-tolerant circuitry.

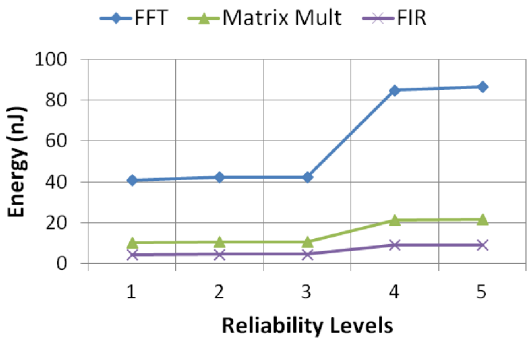

Figure 4.16: Energy consumption for various applications.

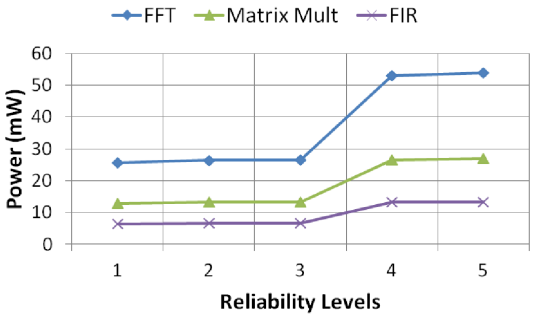

Figure 4.17: Energies of different algorithms tested.

4.8.2 Scrubbing

Configuration Time

To analyze the configuration time of various configuration scrubbing techniques in real applications, we have mapped four sample algorithms/applications on the DRRA: (i) Fast Fourier Transform (FFT), (ii) Matrix Multiplication (MM), (iii) Finite Impulse Response (FIR) filter, and (iv) Wireless LAN (WLAN) transmitter. For the FFT and MM, multiple versions with different levels of parallelism (serial, partially parallel, and fully parallel) were simulated. Since the number of scrubbing cycles directly depends on the location of scrubber, we simulated the operation of both the internal and external scrubber. Table 4.7 shows the number of cycles required to scrub each application. It is clearly seen that the external scrubber requires significantly larger configuration time (up to 38 times more) compared to the internal scrubber, which justifies the assumption made in Section 4.7.2 ($t_{ext} \gg t_{int}$). The reason is that the internal scrubber bypasses the processor intervention. Fig. 4.18 depicts these trends graphically.

Table 4.7: Number of cycles required by the external and internal scrubber

Algorithm/ application	External scrubber [# of cycles]	Internal scrubber [# of cycles]
FFT64 serial	3120	80
FFT64 partially parallel	5655	145
FFT2048	13500	500
MM serial	819	21
MM partially parallel	1326	34
MM parallel	1716	44
FIR	546	14
WLAN	6435	165

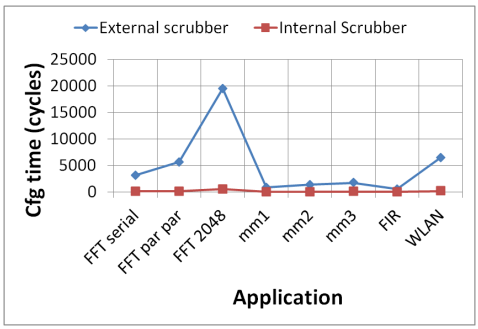

Figure 4.18: Scrubbing cycles external vs internal scrubber

Memory Requirements

To evaluate memory requirements of each scrubbing class, we simulated each technique using DWC and Hamming ECCs. The reason behind using the Hamming codes is their widespread use to protect configuration data in FPGAs [64]. Table 4.8 and Fig. 4.19 show memory requirements of scrubbing, when memory is unprotected, using duplication, and using ECCs. It can be seen that duplication incurs significantly larger memory overhead compared to ECCs (100% for duplication and only 19.44% for ECCs). It should be noted that while ReadBack Scrubber (RBS) and Error Invoked Scrubber (EIS) can employ either duplication or ECCs, the BLind Scrubber (BLS) can only employ duplication. Therefore, the BLS generally requires higher overhead than the RBS and EIS. Justifying the assumption made in Section 4.7.1.

Table 4.8: Memory requirements of different scrubbers

Algorithm/ application	Unprotected (bits)	Duplication BLS, RBS, EIS (bits)	ECCs RBS, EIS (bits)
FFT64 serial	2880	5760	3440
FFT64 partially parallel	5220	10440	6235
FFT2048	18000	36000	21500
MM serial	756	1512	903
MM partially parallel	1224	2448	1462
MM parallel	1584	3168	1892
FIR	504	1008	602
WLAN	5940	11880	7095

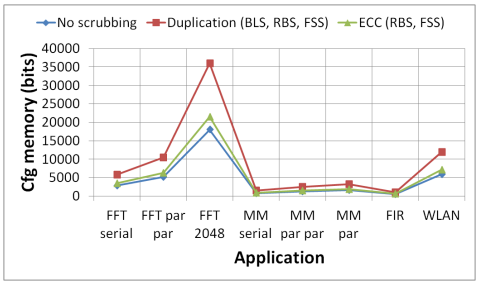

Figure 4.19: Configuration memory requirements for various scrubbers

Area and Power Overhead

To estimate the area and power overhead related to using scrubbers, we have synthesized the DRRA fabric with configuration scrubbing support for

65 nm technology at 400 MHz frequency using Synopsys Design Compiler. Area and power requirements of each component (discussed briefly in Section 4.3 and detailed in [119]) are shown in Table 4.9. Overall, the results reveal that scrubbing incurs negligible overhead (3% area and 4% power).

Table 4.9: Area and power consumption for memory based scrubbing

	ACC	LCC	DRRA cell
Power $[\mu W]$	38.68	163.48	5029
Area $[\mu m^2]$	1735	1470	85679

To estimate additional overhead incurred by the ECCs, we also synthesized the Hamming coder/decoder. Area and power requirements for each of the components are shown in Table 4.10 and in Figs. 4.20 and 4.21 which show that the Hamming coder/decoder consumes a significant portion of the overall additional scrubbing circuitry (58% area and 89% power).

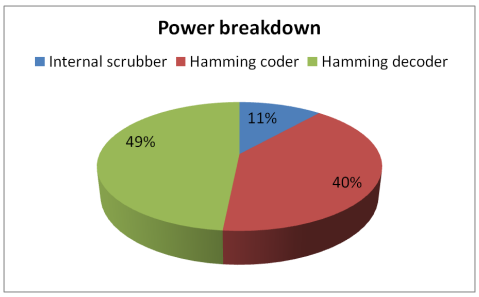

Figure 4.20: Power breakdown for a scrubber

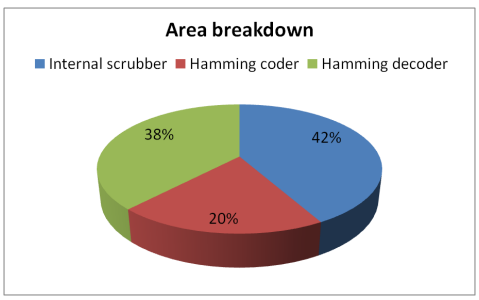

Figure 4.21: Area breakdown for a scrubber

4.9 Summary

In this chapter, we have presented an adaptive fault-tolerance mechanism that provides the on-demand reliability to multiple applications hosted by

Table 4.10: Area and power consumption for Error Correcting Codes (ECCs)

	Hamming encoder	Hamming decoder
Power $[\mu W]$	719	866
Area $[\mu m^2]$	1527	2904.12

a Coarse Grained Reconfigurable Architecture (CGRA). To provide on-demand fault-tolerance, the reliability requirements of an application were assessed upon its entry. Depending on the assessed requirements, one of the five fault-tolerance levels can be provided: (i) no fault-tolerance, (ii) temporary fault detection, (iii) temporary/permanent fault detection, (iv) temporary fault detection and correction, or (v) temporary/permanent fault detection and correction. In addition to modular redundancy (employed in the state-of-the-art CGRAs offering flexible reliability levels), we have also presented the architectural enhancements needed to realize sub-modular residue mod 3 redundancy. Indeed, residue mod 3 coding allowed to reduce the overhead of the self-checking and fault-tolerant versions by 57% and 7%, respectively. To shift autonomously between different fault-tolerance levels at run-time, a fault-tolerance agent was introduced for each DPU. This agent is responsible for reconfiguring the fault-tolerance infrastructure upon arrival of a new application or changing external conditions. Finally, the polymorphic fault-tolerant architecture was complemented by a morphable scrubbing technique to protect the configuration memory. The obtained results suggest that the on-demand fault-tolerance can reduce energy consumption up to 107%, compared to the highest degree of available fault-tolerance (for an application actually needing no fault-tolerance). Future research on a fault-tolerant DRRA will move in two directions: (i) the architecture and algorithms to support a comprehensive remapping algorithm will be implemented and (ii) the algorithms for self-adaption (scrubber that can adapt itself to provide the needed reliability level using the minimal energy) will be studied and implemented.

Chapter 5

Private reliability environment for NoCs

5.1 Introduction

Until now we only discussed how the PREX framework was applied to CGRAs. However, although CGRAs are emerging as high performance energy efficient alternatives to FPGAs, they are difficult to program compared to both FPGAs and processors (since the design flows and compilers for CGRAs are not mature). Therefore, we also decided to test the effectiveness of the proposed framework on relatively more mature network on chip platforms. Table 5.1 lists the major differences between a NoC and a CGRA platform. The table simply shows that while CGRA platforms are essential to allow fast computations demanded by modern 3-G, 4-G standards, the NoC based platforms provide an easily programmable.

Table 5.1: Major differences between CGRA and NoC platforms

Platform	Switching strategy	Computational units	Speed	Programming efficiency
NoC	Packet-switching	LEON3 processor	Low due to load store architecture	High due to processor based platform and static network
CGRA	Circuit-switching	DPUs	High speed due to arbitrary long datapaths	Low due to arbitrary datapaths with in-deterministic synchronization

This chapter presents an architecture to provide customized reliability to different applications hosted by a packet switched NoC. In NoCs different entities communicate by exchanging packets. Information contained in

a packet can be classified into two classes: (i) control information (source address, destination address etc.) and (ii) data information (actual data payload to be transmitted). To guarantee the smooth delivery, control information in general require high reliability while the protection level of the data information should ideally be application dependent [133]. Proposed adaptive fault tolerance methods attempt to reduce the fault tolerance overhead by providing different level of protection to control and data fields [105, 133]. Henceforth, we will refer to these methods as *Intra Packet Fault Tolerance approaches* (IPF). Many recently proposed NoCs support multiple traffic classes [65, 23]. On the basis of functionality, the traffic classes can be divided into *data traffic* and *control traffic*, containing data and control packets, respectively. The data packets hold computation information such as intermediate results, while the control packets deliver control information such as lowering of voltage. In addition, a new traffic class can emerge when when a new application, with different protection needs (than those already running), enters the platform. The reliability requirement of a packet depends on its functionality and parent application. Consider for example that in streaming applications control traffic needs higher reliability because loss or corruption of a control packet can lead to system failure. However, infrequent loss or corruption of data packets has little effect on the quality or can even be reproduced in software. On the other hand, in a critical application like car breaking system, both data and control traffic need high reliability.

Inspired from the IPF methods, we proposed *Inter Packet Fault tolerance* (IAPF). IAPF provides different fault tolerance strengths to multiple traffic classes, considering packet as a single entity, thereby reducing the energy overhead, significantly. To identify different traffic types, a two layer low cost identification circuitry is used. The two layers identify the parent application and the control/data type of each packet, respectively. Upon identification, packets are directed towards the path offering needed reliability. We have chosen frequently used methods to tolerate temporary and permanent faults. To combat temporary faults, we use *Error correcting codes* (ECC). ECC utilizes information redundancy to detect and/or correct errors [14, 133]. Specifically, we use hamming codes to detect and correct temporary faults in each switch. To address permanent faults in interconnects, we use a spare wire between each pair of switches similar to [77, 131]. If a permanent fault is detected, the interconnect is reconfigured to the spare wire. Using fault tolerance infrastructure and a hierarchical control layer, our architecture offers four dynamically changeable reliability levels: (i) no fault tolerance, (ii) end to end fault tolerance providing Double Error Detection Single Error Correction (DEDSEC) for temporary faults, (iii) per-hop fault tolerance providing DEDSEC for temporary faults, and (iv) per-hop fault tolerance providing DEDSEC for temporary faults and spare wire replacement for permanent faults. It should be noted that more fault

levels can easily be integrated to existing architecture. We achieve considerable reductions in energy overhead (from 95 % to 52 %) for implemented applications (wave front, FFT, HiperLAN, and matrix multiplication) with an acceptable area overhead (up to 5.3 %), for providing on-demand fault tolerance (in comparison to overall fault tolerance circuitry).

Motivational Example:

As a concrete motivating example, we present here a case study of 64-point FFT, mapped on a 3-processor NoC, as shown in Figure 5.1. The FFT is parallelized by pipelining the butterfly stages [57]. The communications between the processors are realized using a Distributed Shared Memory (DSM). One of the processors, called system processor, acts as the system manager that controls the other two processors, performing computations. The system processor also hosts a smart power management algorithm to choose the optimal voltage/frequency operating point. In the figure, the control and data packet exchanges, between the processors, are represented by dotted and solid arrows, respectively. The data packets hold intermediate results, and control packets deliver synchronization and voltage/frequency scaling information. It can be seen that the data packets are significantly greater in number compared to the control packets. For many streaming applications (e.g. WLAN, HiperLAN), that use 64-point FFT, infrequent loss or corruption of data packets has little effect on the quality or can even be reproduced in software. However, an erroneous control packet can cause system failure (e.g. by turning off the processor). As opposed to existing fault-tolerance techniques (providing same reliability to all the packets), we exploit this difference in reliability requirements, to reduce the fault tolerance energy overheads, by providing on-demand fault tolerance (to each traffic class). Like the FFT example taken here, the data packets are likely to be the dominant the traffic class in most applications, as asserted by the famous 80-20 % rule [85]. The 80-20 rule states that 80 % of the execution time is consumed by 20 % of the program code. The time consuming portions of the code are typically data computations in nested loops. Parallelism is typically exploited by mapping the loop iterations on multiple processing elements (which need to exchange data). Thereby making the data packets dominant traffic class.

5.2 Related work

Since the last decade, fault tolerant NoCs have been a subject of extensive research [78]. In this section, we will review only the most prominent work on Energy aware fault tolerant NoCs. Depending on the attribute to be modified, techniques to reduce fault tolerance energy overheads can be either Voltage Adaptive (VA) or the Fault tolerance scheme Adaptive (FA). *VA*

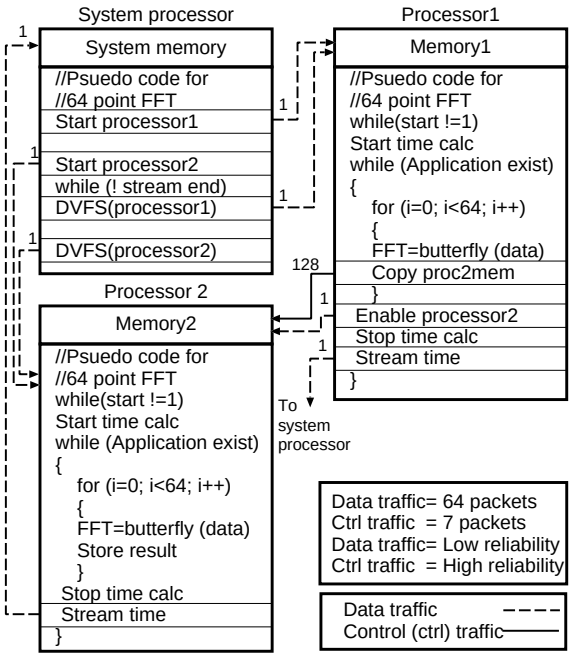

Figure 5.1: Motivational example for control/data traffic

approaches, considering that transmission voltage has a conflicting influence on energy efficiency and circuit dependability, adjust the voltage level (on the basis of e.g. error rate or noise) to minimize the energy consumption. *FA* approaches adjust the fault tolerance scheme (and hence the energy overhead) to match reliability needs.

Voltage adaptive: Worm et al [125] proposed a technique to scale supply voltage based on observed error pattern. Considering that the voltage level directly affects reliability, they suggested that a smaller voltage would be sufficient for transmission in a less noisy execution condition. Therefore, they increased/decreased the voltage based on existing noise. This work was later used in [126] to propose a self-calibrating on-chip link. The proposed architecture achieved high-performance and low-power consumption by dynamically adjusting the operating frequency and voltage swing. Error detection was combined with retransmission to ensure reliability.

Fault tolerance scheme adaptive: Li et al [79] showed that retaining the voltage and changing the fault-tolerance scheme provides a larger improvement in reliability per unit increase in energy consumption. Based on their findings, they presented a system capable of dynamically monitoring noise and shifting between among three fault tolerance levels of different intensity (Triple Error Detection (TER), Double Error Detection (DED), and parity). The idea behind their strategy was to monitor the dynamic

variations in noise behavior and use the least powerful (and hence the most energy efficient) error protection scheme required to maintain the error rates below a pre-set threshold. Zimmer and Jantsch [133] proposed a method for dynamically adapting between four different quality of service levels. They provided different protection levels to different packet fields. They suggested that, since packet control part needs higher reliability, the encoding scheme for the header should be chosen first so that the minimum reliability constraint is met. The number of wires required for header encoding limits those remaining for payload transmission. Rossi et al [105] included end to end fault tolerance on specific parts of NoC packet to minimize energy and timing overhead. This thesis uses the end to end and per hop strategy inspired from their work. Lehtonen et al [76] employed configurable circuits for adapting to different fault types. They used reconfigurable links to tolerate transient, intermittent, and permanent errors. A method for dynamically shifting between codes of different strengths was presented in [130] that tolerated temporary faults. This method adapts error detection and correction at runtime. Later the work was improved to handle both permanent and temporary faults [100]. The proposed scheme combines ECC, interleaving and infrequently used spare wires to tolerate faults.

From the related work it can be seen that the existing fault tolerance FA schemes reduce energy overheads by changing fault tolerance level on the basis of information class within a packet. Our approach (which can be considered as a subset of FA schemes) make decisions to adapt reliability on the basis of traffic class of each packet. This apparently small change in granularity of decision making significantly enhances awareness (and hence the intelligence) of the system, as will be shown in Section 5.6.5. Section 5.7 shows that our scheme promises significant reduction in energy overheads at cost of minimal area/timing overheads.

Compared to the related work, we made following major contributions:

1. We presented on-demand fault tolerance that scans each packet for its reliability needs and directs it to the path offering the required protection. Thus, the energy overhead to provide fault tolerance is significantly reduced compared to state-of-the-art adaptive techniques FA techniques [133, 105, 76] (having no information about the traffic class).

2. We present an enabling management and control backbone that provides a foundation for the above concept by configuring the fault tolerance circuit to meet the reliability requirements.

5.3 Hierarchical control layer

As already mentioned in Chapter 1, we have chosen McNoC to test the effectiveness of our method. To refresh our memory, we will briefly discuss its architecture again. The overall architecture of McNoC is shown in Figure 5.2. Broadly, McNoC can be divided into two different parts: (i) network on chip, and (ii) power management infrastructure. McNoC uses the Nostrum network-on-chip as communication backbone [93, 97, 83]. It uses regular mesh topology and provides *hot potato* X-Y routing [37]. A power management system has been built on top of Nostrum by introducing The power management system allows to manipulate voltage and frequencies using APIs. A detailed description of GRLS can be found in [23].

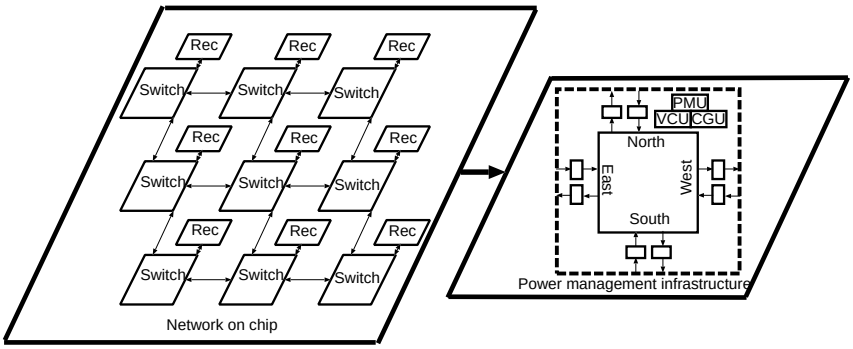

Figure 5.2: McNoC architecture

To enable adaptivity we added an intelligence layer on the McNoC system architecture as shown in Figure 5.3. This layer is composed of one *cell agent* per node, one *cluster agent* controlling a number of cell agents and a *system agent* managing entire platform. The main purpose of this layer was to provide various services like fault tolerance and power management orthogonally to the traditional NoC functions (like packet switching etc.). In this section we will describe this layer briefly. A detailed description about how this layer controls adaptive fault-tolerance and power-management will be given later in Section 5.6 and Chapter 7, respectively.

Cell agents are simple, passive entities implemented primarily in hardware to provide on-demand monitoring services such as reporting average load of a switch, to the cluster agent (explained later in Chapter 7). Each cluster agent manages a number of cell agents to bring about e.g. DVFS functionality. The system agent is the general manager of all monitoring. Operations like application mapping are performed by the system agent. The cluster agent is responsible for managing each application in case multiple applications are running in a single platform. The joint efforts of the system, cluster and local agents realize the adaptivity of the system e.g. autonomous trade-off between power, energy and timing requirements of the

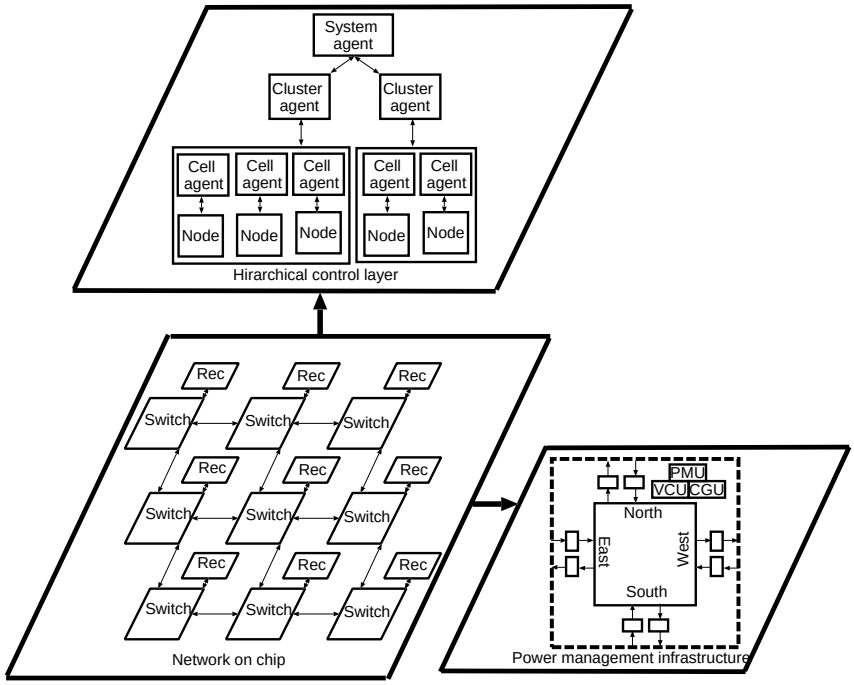

Figure 5.3: McNoC architecture

application. In terms of functionality, the agent layer is orthogonal to the data computation. The underlying NoC backbone, regardless of the exact implementation (topology, routing, flow control or memory architecture), performs the conventional data communication, while the agent subsystem monitors the computation and communication. The separation of agent services provides portability of the system architecture to different NoC platforms, thus leading to improved design efficiency.

5.4 Fault Model and infrastructure

In this thesis, we detect and correct three type of faults in NoCs: (i) single bit temporary faults in storage buffers, (ii) temporary faults in a link, and (iii) permanent faults in a link. A single bit temporary faults in storage buffers or a link, can cause a bit flip in the packet to be transmitted and is modeled as single event upsets (SEUs). Permanent fault in a link can induce error in the packet bits traversing the wire. These faults are modeled as Stuck-At-faulT (SAT). The motivation for protecting buffers and wires is that they consume most silicon in high performance [106] and low power [89, 46, 67, 73] NoCs, respectively. Thereby, they are most susceptible to faults. Moreover, the other components of a NoC (i.e. routing logic and network interface) can be protected locally, using commonly used Built-In-

Self-Test (BIST) methods [31], independent of the proposed scheme.

5.4.1 Protection against Temporary Fault in Buffers/links

For protection against single bit temporary errors, we use Error Correcting Codes (ECC). Specifically we employ hamming codes which are DEDSEC codes. Motivation behind the choice of the ECC was that hamming codes or its variants are frequently used in NoCs, to combat temporary faults [14, 76]. It should be noted that any other ECC scheme and/or interleaving can be used. Same ECC can be employed to detect/correct temporary faults in wires and the buffers. Here, the purpose is not to propose the most efficient ECC, but to present a generic methodology to reduce fault tolerance energy overheads. Figure 5.4 shows the architecture of a fault tolerant switch. *HC* and *HD* stand for hamming coder and decoder, respectively. Whenever, a packet leaves a switch, it is encoded with hamming code using a hamming encoder. Whenever a packet enters a switch, it is decoded to extract the original packet.

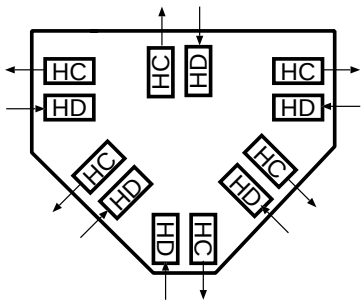

Figure 5.4: Fault tolerant NoC switch

5.4.2 Protection against permanent faults in links

To overcome a permanent fault in one of the wires, we employ a spare wire between each pair of switches, similar to [77]. If a permanent fault is detected, the data around the faulty wire is directed towards the spare wire as shown in Figure 5.5. In the figure T and R indicate respectively the transmitting and receiving end. To reduce router complexity and balance the delay within routed wires the reconfiguration ripples through the bus instead of directly mapping the faulty wire to the spare. The faulty wire (wire with permanent fault) is detected at the receiving end by continuous occurrence of temporary fault at the same wire. Upon detection of a faulty wire, the receiver switches to spare wire and informs the transmitter end about the faulty wire (by sending a *faulty wire* packet, shown in Table 5.3, directed towards the transmitter). After receiving the faulty wire

packet, the transmitter also switches to the spare wire and sends *spare wire switched* packet to the receiving end. To ensure safe communication, from the transmission of *faulty wire* packet to the reception of *spare wire switched* packet, the receiver rejects all packets during this duration. This process is facilitated by the cell agent and will be explained in Section 5.6.1.

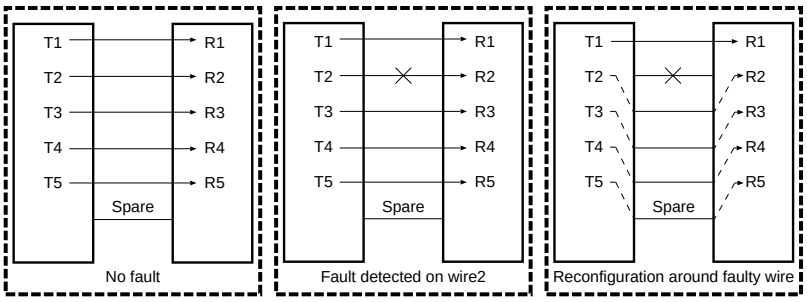

Figure 5.5: Reconfiguration to spare wire

If another permanent fault is discovered, the application is mapped to another processor. Again, the reason for choosing this methodology was the ease of implementation. Methods like split transactions [76] can be employed to increase the efficiency, however they are beyond the scope of this thesis. The state machine, shown in Figure 5.6, is used to support the switching to spare wire and the remapping functionality. This state machine specifically represents the reconfiguration functionality at the receiving end of faulty wire. The reconfiguration at the transmitting end and remapping is accomplished by the agent based monitoring and management system, explained in Section 5.6. The state machine is divided into three stages:

1. As long as no fault is detected in the system, the state machine remains in state *No error*. If a temporary fault is detected, by a non zero syndrome, the state machine changes its state to *Tempft 1*. In this state, a counter is initialized and the syndrome stored. Upon consecutive occurrence of errors at the same bit location, the state machine moves to *Tempft 2*. If the value of counter exceeds a pre-defined threshold, a permanent fault is inferred. Upon detection of permanent fault the state machine signals switching to spare wire (state *Switch wire*) and changes to state *wchd* in stage 2,

2. The state machine remains in state *wchd* as long as no other single wire fault is detected. If another permanent fault is detected, the state machine moves to state *Remap req* in stage 3 passing through states *wchd ft1* and *wchdft 2*, similar to stage 1.

3. This state signals remapping to the control layer.

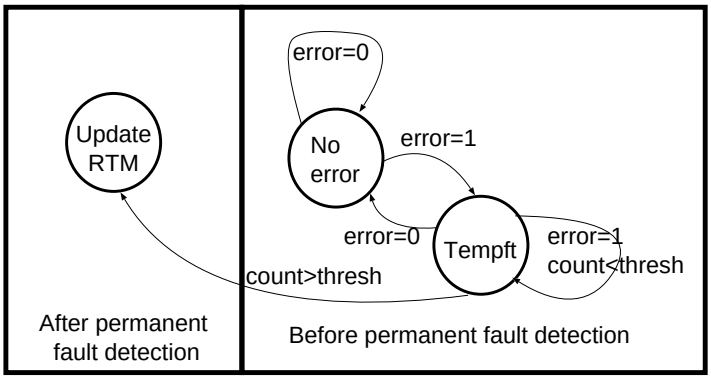

Figure 5.6: Permanent fault detection state machine

Table 5.2: Fault tolerance levels

Code	Fault tolerance level	Energy Overhead
00	None	None
01	End to end (Temporary)	Low
10	Per-hop (Temporary)	Medium
11	Per-hop (Temporary and permanent)	High

5.5 On-demand fault tolerance

Depending on the reliability needs, each packet is provided only the required fault tolerance strength, thereby reducing the energy overhead, considerably. We call this method on-demand fault tolerance. On the basis of fault tolerance strength, and hence the overhead, we have provided four different fault tolerance levels as shown in Table 5.2. In per hop strategy, the packet passes through the fault tolerance circuitry at each hop. This strategy can detect both single bit temporary errors and single wire permanent faults per hop. End to end fault tolerance scheme is employed to ensure energy efficiency in packets requiring low reliability. This scheme can tolerate single bit error in the entire source to destination path. On-demand fault tolerance is accomplished in two stages: (i) packet identification and (ii) providing needed protection.

5.5.1 Packet identification

The packet identification involves traffic type and parent application identification. The traffic type identification circuitry determines whether the packet is control or data packet. The parent application circuitry determines the parent application a packet belongs to.

Control/data traffic identification

This strategy is specifically targeted for applications needing different protection levels for the control and data packets (e.g. streaming applications). Packet type identification uses a mux, a demux and a Hop Count Comparator (HCC in the figure) shown at the top of Figure 5.7. To distinguish data packets from control packets, high hopcount values are reserved for the control packets. The motivation behind choosing hopcount to identify control packet is that hopcount is also used to prioritize packets in case of contentions. Therefore, by reserving high values in hopcount field for control packets, they always have higher priority than the data packets. To support this mechanism, the router only increments the hopcount value, HV, of the data packets if $HV < Res_{min} + 1$. Where Res_{min} is the smallest of the reserved values.

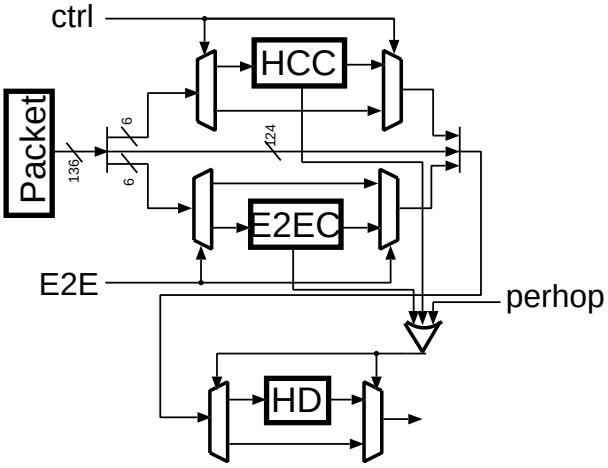

Figure 5.7: Multi path reconfigurable fault tolerance circuitry

Parent application identification

This strategy is specifically designed for situations when multiple applications with different reliability needs are running simultaneously on a platform. To distinguish packets from different applications, either of the two circuits shown in Figure 5.8 can be used. Circuit in Figure 5.8-a (referred to ABFA from hereon in), is very efficient and flexible but, as will be shown later in this section, not scalable. Circuit in Figure 5.8-b (referred to ABFB from hereon in) is scalable but not as efficient for smaller projects involving less than 70 processors.

ABFA uses a special, $proc * f$ bit register, called index reg, which can support 2^f different fault tolerance levels. Where $proc$ is the number of processors present in the platform. Each row in the index reg indicates the

fault tolerance need of application hosted by that processor e.g. $proc3 = 01$ indicates that processor 3 needs end to end temporary fault tolerance (from Table 5.2). When all the applications in the platform need the same reliability, the application identification circuitry is deactivated and packets pass uninterrupted through a bypass path (not shown in figure). As soon as an application needing a different protection level enters the platform, ABFA is switched on and the corresponding rows of index reg are updated. When activated, ABFA compares the reliability needs of the source and the destination processor and provides the greater of the two values. The main problem with this method is that the size of the index reg is dependent on the number of processors, making it unscalable.

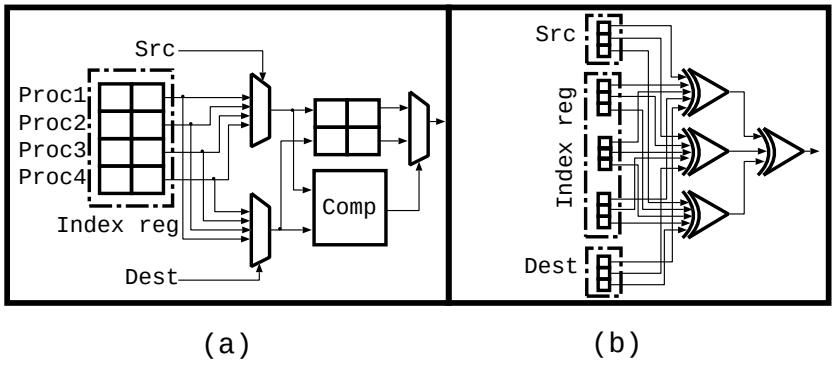

Figure 5.8: Application fault tolerance level identifier

ABFB, shown in Figure 5.8-b offers scalability at the cost of flexibility. In the this method, a pre determined maximum number of processors PRE_{max} per application is decided at design time. A $PRE_{max} * a$ bit register is embedded in each cell agent. Where a represents the bits needed to represent the source or destination address. The controlling cluster agent (explained in Section 5.6.2) fills the register with the addresses of the processors controlled by the cluster agent, collectively called *cluster group*. When ABFB is activated, the source and destination addresses of each packet are compared to other addresses in the cluster group. If the incoming packet belongs to a different group, it is assigned the maximum fault tolerance level supported by the platform. Though less efficient, this approach still promises substantial reduction in energy overhead provided bulk of packets are exchanged between processors belonging to the same application. To estimate the area and power overheads of the proposed circuits (ABFA and ABFB), we synthesized their multiple versions (with different NoC nodes). For power estimates, the default 20 % switching activity was used. The obtained area and power are shown in figures 5.9 and 5.10. The figures reveal that for small projects (up to 70 nodes), ABFA promises lesser area

and power overheads. For projects exceeding 70, ABFB is more efficient in terms of both area and power.

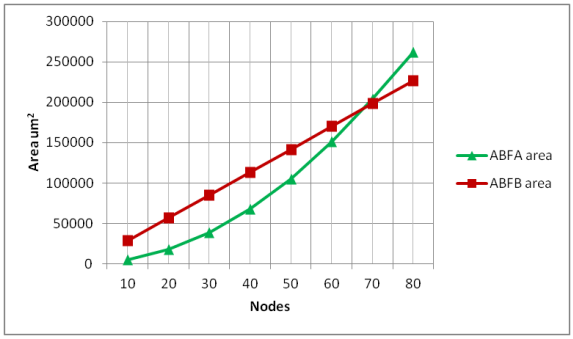

Figure 5.9: Area comparison between ABFA and ABFB

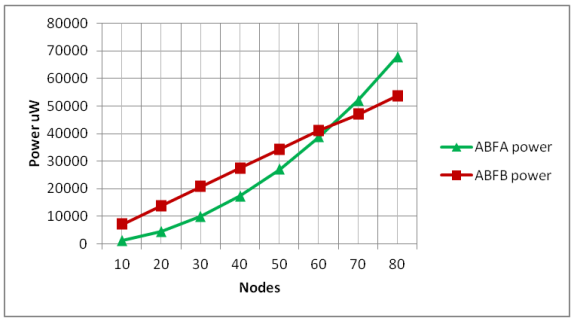

Figure 5.10: Power comparison between ABFA and ABFB

5.5.2 Providing needed protection

Once the packet is identified, the packet is given needed protection by configuring the muxes/demuxes shown in Figure 5.7. Depending on the traffic type, the control, end to end, and per-hop signals (ctrl, E2E, and perhop in the figure) are adjusted. Any of the four fault tolerance levels can be provided to the packets. The details of how this circuit is configured, will be presented in Section 5.6.1.

5.5.3 Formal evaluation of energy savings

To visualize the potential savings of the proposed method (for a generic NoC/application), we will present here a simplistic energy model. The actual energy estimates using, Synopsys design compiler, by executing real application (HiperLAN, matrix multiplication, wavefront, and FFT) on McNoC will be shown in Section 5.7. The formalizations can serve as a guide to determine when to bypass the packet identification circuits, presented

in sections 5.5.1 and 5.5.1, is useful. Let $E_c(i)$ and $E_d(j)$, be the energy required by control and data packet, respectively, to traverse the NoC. Energy, E_t, needed for providing fault tolerance to all the packets traversing the NoC using traditional methods is given by equation:

$$E_t = \sum_{k=1}^{C+D} E_{ft} + (\sum_{i=1}^{C} E_c(i) + \sum_{i=1}^{D} E_d(j)). \tag{5.1}$$

Where, C and D is the number of control and data packets, respectively. E_{ft} is the energy required for providing fault tolerance. The equation can be reduced to:

$$E_t = (C + D) * E_{ft} + (\sum_{i=1}^{C} E_c(i) + \sum_{i=1}^{D} E_d(j)). \tag{5.2}$$

Energy, E_{CID}, needed to identify and provide fault tolerance to each packet type, separately, is given by equation

$$E_{CID} = \sum_{i=1}^{C}(E_c(i) + E_{ftc} + E_{id}) + \sum_{i=1}^{D}(E_d(j) + E_{ftd} + E_{id}), \tag{5.3}$$

where, E_{id}, is the energy needed to identify the packet. E_{ftc}, and E_{ftd}, represent the energy consumed for providing fault tolerance to control and data packets, respectively. The equation can be reduced to:

$$E_{CID} = (C + D) * (E_{ftc} + E_{ftd} + E_{id}) + \sum_{i=1}^{C}(E_c(i)) + \sum_{i=1}^{D}(E_d(j)) \tag{5.4}$$

Since in many applications (e.g. streaming applications), the number of control packets is significantly lower than the data packets (shown in Section 5.7) and data packets can traverse unprotected, this equation promises massive energy savings provided E_{id} is lesser than E_{ft}. We will show in Section 5.7 that E_{id} is composed of a simple comparator needing very low energy. To cover the corner cases, where the control packets are frequent and/or each packet class needs the same reliability, the identification circuitry can be bypassed.

To formalize the potential savings of using ABFA and ABFB, let $E_a(i)$, be the energy overhead of i^{th} application and E_{max} be the overhead to provide fault tolerance to the application needing maximum reliability. Energy, E_{app}, for providing fault tolerance to all application using traditional method is given by equation

$$E_{app} = \sum_{i=1}^{app}(E_a(i) + E_{max}), \tag{5.5}$$

where, *app* are the total applications running simultaneously. Energy, E_{PID}, needed to provide fault tolerance to all applications individually is given by equation

$$E_{PID} = \sum_{i=1}^{app}(E_a(i) + E_{id} + E_{ft}(i)). \tag{5.6}$$

This method can reduce energy overhead provided $E_{id} + E_{ft}(i) < E_{max}$. If all applications running on the system have same reliability requirements, bypass paths should be activated. Overall, equations 5.3 and 5.6, promise significant overhead reduction provided multiple traffic classes with different reliability needs traverse the NoC.

5.6 Monitoring and management services

The monitoring and management services are provided by a three-tier hierarchical control layer shown in Figure 5.2. In this section we will explain, in detail, the architecture and functionality of the control layer. Remember from Section 5.3, the control layer is composed of three types of agents. Figure 5.11 shows the functionality of each of the agent type.

5.6.1 Cell agent

Cell agent is a passive entity, implemented in hardware and connected with each switch of the NoC. In terms of fault tolerance, the functionality of cell agent can be divided into six parts: (i) if a permanent fault is detected by the state machine, shown in Figure 5.6, it configures the receiving switch to shift to the spare wire and sends a packet containing the syndrome to the neighboring switch at the transmitting end, (ii) if a packet containing the syndrome has been sent to the transmitting end, it rejects all packets till the reception of *spare wire switched* packet from the transmitting end, (iii) if an incoming packet is identified as *faulty wire* packet, it configures the switch to shift to spare wire, specified by the packet, and sends a *spare wire switched* packet directed towards the source of received packet, (iv) if an incoming packet is identified as *spare wire switched* packet, it restarts accepting packets, (v) if a permanent fault is detected in another wire, it sends remap packet to the system agent, and (vi) upon request from the application/system agent, it configures the fault tolerance circuitry shown in Figure 5.7, to set the fault tolerance strength. Here, we only focus on shifting to spare wire, a corresponding fault tolerance protocol is beyond the scope of this thesis and for that an intrested reader can refer to [16].

The interface of the cell agent with the switch is shown in Figure 5.12. The cell agent is further divided into 2 sub-agents (i) Power Management agent (PM agent) and (ii) the Fault Tolerance agent (FT agent). The PM

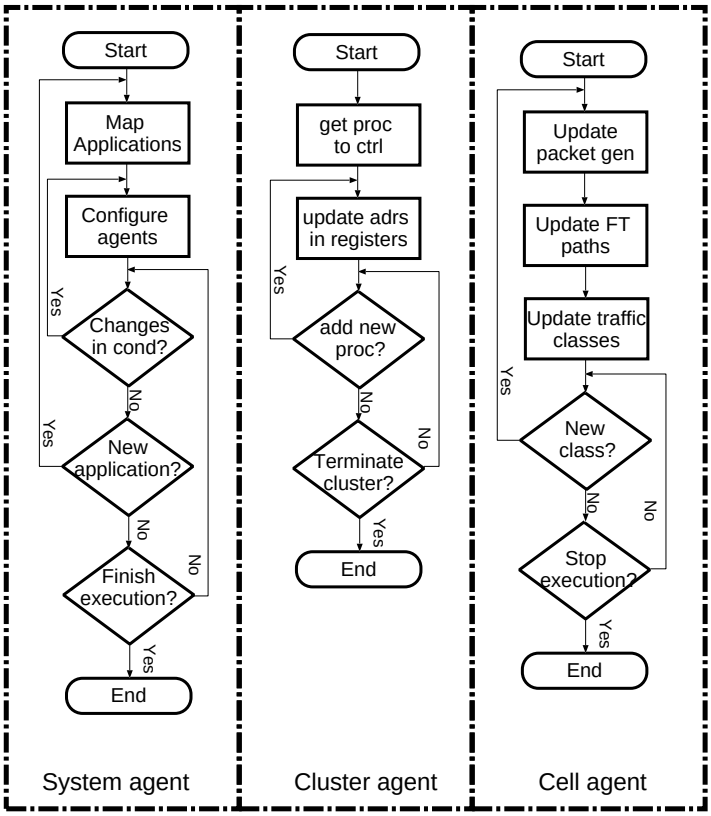

Figure 5.11: Functionality of the system, cluster, and cell agent

Table 5.3: Traffic interchange between cell agent and switch

Packet	source	destination
peak load	PM agent	application agent
faulty wire (syndrome)	FT agent	neighboring node
remap	FT agent	system agent
load request	cluster agent	PM agent
set DVFS	cluster agent	PM agent
set region	system agent	PM agent
spare wire switched	FT agent	neighboring node

agent and the FT agent work independently to provide power management and fault tolerance services, respectively. The only interference between the agents occur at the switch boundary, when the packet is to be inserted into the network. At this point the contentions need to be resolved using appropriate priorities. Here, we will focus on the functionality of the FT agent and the PM agent will be discussed only when it affects the FT agent. A detailed functionality of PM agent will be discussed in Chapter 7. All types of packet transfer between the agents and the network are shown in Table 5.3.

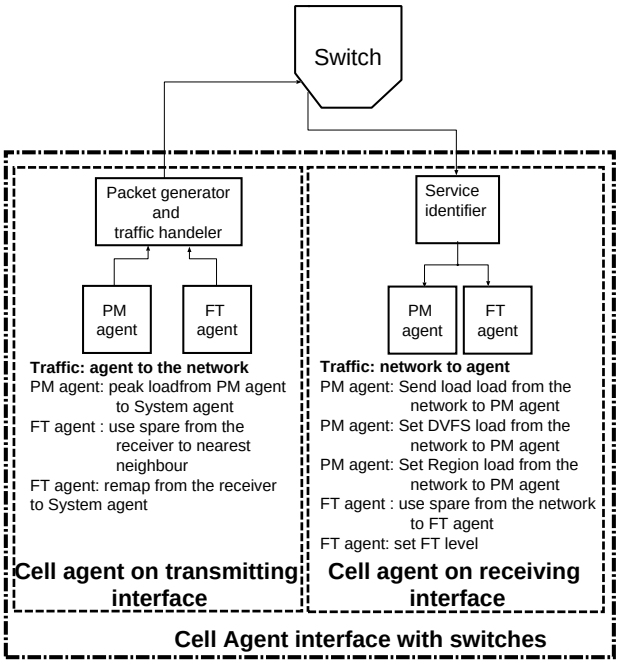

Figure 5.12: Cell agent interface to the switch

Packets from the switch to the agent pass through a service identifier.

The service identifier directs the packet to the appropriate sub-agent (PM agent or the FT agent). The packets from the agent to the switch pass through a packet generator and traffic handler unit. The packet generator packs the packets with appropriate destination and priority, before sending them to the switch. For our experiments, the highest priority is given to remap packets, followed by the spare wire (faulty wire, switch to spare wire, wire switched) and power management (peak load, load request, set DVFS, set region) packets. In present implementation of McNoC, the hopcount field has 6 bits. Therefore, on the basis of priority, values 63, 62, and 61 are reserved for the remap, spare wire, and power management packets, respectively. In a single cycle, up to four remap or change wire packets and one sendload can contend for the switch. Three FIFOs with different priorities, shown in Figure 5.13, are used to resolve contentions. The generate packet unit generates a NoC packet depending on the information received.

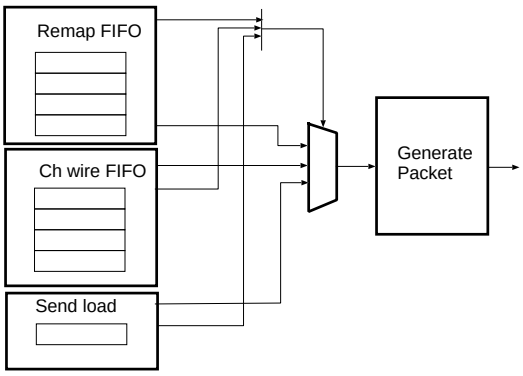

Figure 5.13: block diagram of packet generator

5.6.2 Cluster agent

A cluster agent is present as a separate thread for each application running on a NoC platform. Cluster agent runs parallel to the computations on one of the processors executing the application. The number of cluster agents is equal to the number of applications executing simultaneously on NoC. In our experiments we use 4 cluster agents to control wavefront, FFT, HiperLAN and matrix multiplication respectively. In terms of fault tolerance, the cluster agent has two main functionalities: (i) to dynamically update the index reg in ABFB with the addresses of the processors controlled by the cluster agent and (ii) to provide turn off signals to the cell agents if the application finishes. The cluster agent also performs power management, but that will be discussed later in Chapter 7.

5.6.3 System agent

The system agent is the general manager of the system and is implemented as a separate thread on one of the processors of the NoC in software. For each application, it updates the cell and cluster agent about the fault tolerance requirements of traffic types present in the NoC. As soon as the fault tolerance needs change (e.g. due to entry of new application), it updates the packet generator of cell agents to ensure that the control packets with appropriate hop count values (and hence priorities) are generated. Upon receiving remap message, it remaps the application to a different part of NoC and sends the turn off message to the cluster agent.

5.6.4 Inter-agent communication protocol

To realize on-demand fault-tolerance, an agent may need to apprise other agent(s) about an observed event. As shown in Figure 5.14, this information exchange takes place via inter-agent communication protocol. To simply the illustration, all the communications are shown by arrows (they actually take place using the NoC). In the context of fault tolerance, the agents communicate with each other in three scenarios: (i) an application with different reliability needs (from already hosted applications) enters the platform, (ii) first faulty wire (in a link) is detected by a cell agents, and (iii) second faulty wire (in a link) is detected by a cell agent. Upon arrival of an application with new fault tolerance needs, the system agent informs all cluster agents (by sending Set Fault Tolerance (SFT) packet) to update the Fault Tolerance Level (FTL) of cell agents, controlled by it. When a wire with permanent fault is detected by the a cell agent at receiving switch (see Section 5.6.1), it requests the cell agent at transmitting switch to use the spare wire. After shifting to spare wire, the the cell agent at transmitting end updates the cell agent at receiving end about it. When a link with two faulty wires is discovered, the the cell agent at receiving end updates the system agent about it.

5.6.5 Effects of granularity on intelligence

Remember from Section 5.2 that principal difference between IAPF and IPF approaches is the granularity of decision making. As shown in Table 5.4, increasing the granularity to packet level offers significant flexibility, otherwise unachievable by IPF approaches. While both IPF and IAPF schemes can modify their fault tolerance strengths depending on the noise, IPF approaches inherently lack the ability differentiate packets belonging to different traffic classes. They can neither differentiate between packets belonging to different applications (with different reliability needs) nor between packets containing different information (control/data). By using the proposed

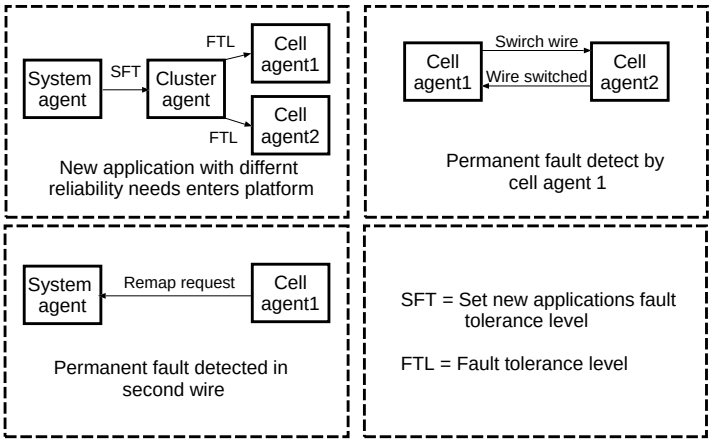

Figure 5.14: Communication protocol between agents

Table 5.4: Comparison between voltage scaled, IPF, IAPF, and IPF+IAPF schemes

Awareness	IPF	IAPF	IPF+IAPF
Varying noise levels	yes	yes	yes
Different packet classes	no	yes	yes
Different application classes	no	yes	yes
Different fields in packets	yes	no	yes

IAPF approach the system is able to transparently distinguish packets belonging to different traffic classes and adjust the fault tolerance intensity accordingly. However, due to higher granularity, IAPF alone fails to recognize different fields within a packet. Integrating IPF and IAPF approaches enables the system to aptly analyze the packet and hence reduce the energy overhead of fault tolerance. It will be shown in the next section that the addition of this additional analysis step reduces energy overhead, considerably, at the cost of minimal area and timing penalty.

5.7 Results

In this section, we will evaluate the benefits and costs of on-demand fault tolerance.

5.7.1 Experimental setup

We performed experiments by mapping four representative applications (wavefront, FFT, HiperLAN, and matrix multiplication) on McNoC platform. The applications were coded in C and mapped to multiple number proces-

Table 5.5: Ratio of control to data packets

App	CPUs	CPs	DPs	CPs (%)
WF	3	7	2040	0.34
WF	6	13	4080	0.31
WF	9	19	6120	0.31
FFT	3	7	192	3.64
FFT	6	13	384	3.38
HLAN	4	9	320	2.81
HLAN	7	15	504	2.97
MM	3	7	192	3.64
MM	6	13	240	5.40
MM	12	25	264	9.46

sors, as shown in column $CPUs$ of Table 5.5. Along with the actual application C code, the per-core energy/power management algorithm, used in [59] (will be explained in Chapter 7), was also implemented on each processor. The algorithm chooses the optimal operating point, by selecting the minimum voltage/frequency required to meet the application deadlines. Therefore, along with Data Packets (DP) three type of Control Packets (CP) were generated: (i) synchronization packets, (ii) fault tolerance packets, and (iii) power management packets. The hopcount field of each control packet was modified to the reserved value, before leaving the source node, by the cell agent attached to it. Thereby, the control and data packets were provided different levels of protection.

5.7.2 Ratio of control to data packets

Table 5.5 shows total Data Packets (DPs) and Control Packets (CPs) generated while executing each of the benchmarks. The DPs were used in exchange of intermediate computation results and the CPs were used to provide fault tolerance and power management. It is clearly seen that only a negligible portion of packets were control packets (from 0.34% to 9.46%). Hence, justifying the assumption made in Section 5.5.3.

5.7.3 Cost benefit analysis

To evaluate the reduction in overhead, achieved by the proposed technique, the fault tolerance hardware was synthesized. Worst case and on-demand adaptive fault tolerance levels, explained in Section 5.5, were tested. Table 5.6 and Table 5.7 show the energy consumed by the fault tolerance circuitry for executing the four benchmarks. In the tables, WCP, WCE, OPA, OEA,

Table 5.6: Energy consumption for worst case and on-demand fault tolerance

App	WCP (μJ)	WCE (μJ)	OPA (μJ)	OEA (μJ)	OPB (μJ)	OEB (μJ)
WF(3)	114.9	168.5	5.6	11.1	30.5	60.7
WF(6)	229.8	337.1	11.2	22.5	60.8	121.2
WF(9)	344.6	505.6	16.8	33.0	91.2	181.86
FFT(3)	11.2	15.9	0.9	1.6	3.3	6.4
FFT(6)	22.3	31.7	1.7	3.1	6.6	12.7
HLAN(4)	18.5	26.4	1.3	2.4	5.3	10.4
HLAN(7)	29.1	41.6	2.1	3.9	8.5	16.5
MM(3)	11.1	15.8	0.9	1.6	3.3	6.4
MM(6)	14.2	19.8	1.37	2.4	4.4	8.5
MM(12)	16.2	21.8	2.14	3.5	5.6	10.6

Table 5.7: Reduction in energy overhead by using on-demand fault tolerance

App	OPA (%)	OEA (%)	OPB (%)	OEB (%)
WF(3)	95.1	93.4	73.5	63.9
WF(6)	95.1	93.5	73.5	64.0
WF(9)	95.1	93.5	73.5	64.0
FFT(3)	91.9	89.9	70.3	59.5
FFT(6)	92.1	90.2	70.5	59.8
HLAN(4)	92.6	90.8	71.1	60.6
HLAN(7)	92.5	90.8	70.9	60.4
MM(3)	91.9	89.9	70.3	59.5
MM(6)	90.3	88.0	68.7	57.1
MM(12)	86.8	83.7	65.1	51.6

OPB and OEB represent worst case per hop, worst case end to end, on-demand per hop using circuit ABFA, on-demand end to end using circuit ABFA, on-demand per hop using circuit ABFB (OPB), and on-demand end to end using circuit ABFB, respectively. The number inside the parenthesis, after the benchmark, represents the number of processors used in parallel to execute the algorithms

For per hop protection, parallel parts of application were mapped contiguously while for end to end fault tolerance, the parallel parts were one hop apart. For on-demand fault-tolerance, only the control packets were provided DEDSEC while the data packets traversed the network unprotected. From the tables it is obvious that both the ABFA and ABFB promise considerable reduction in energy overheads compared to the worst case fault

Table 5.8: Area and power consumption of different components of fault tolerant circuit

	HC	HD	AIF	ABFA	ABFB
Power μW	1333	1924	29	109	705
Area μm^2	3729	5369	60	450	2828

tolerance. ABFA circuitry outperforms ABFB circuit, since the synthesis was done for 16 nodes. Following Figure 5.10, for NoCs with more than 70 processors, the ABFB circuit should be a better solution.

To estimate additional overhead incurred by on-demand fault tolerance, we synthesized two versions of fault tolerance hardware, using ABFA and ABFB, respectively. Area and power requirements for each of the components is shown in Table 5.8 and Figure 5.15. For both versions, Hamming Coder (HC) and the Hamming Decoder (HD) was found to be most costly in terms of area (84.70 % and 79.9 %) and power (95.86 % and 75.9 %). The AIF circuit for detecting control/data packet had negligible overhead. The overhead for parent application detection was largely dependent on whether ABFA or ABFB was used. In Figure 5.15, ABFA promises lesser overhead (4.68 % area, 3.21 % power) compared to ABFB (23.59 % area and 17.69 % power) because a NoC with 16 nodes was synthesized. Remember from Section 5.5.1, the overhead of ABFA is dependent on the number of processors present. For a NoC exceeding 70 processors, ABFA would be more costly than ABFB while the overhead of ABFB would remain constant. Overall, results confirm that on-demand fault tolerance can achieve significant energy savings with negligible additional costs (5.3 % area for ABFA and 24.9 % area for ABFB). It should be noted that although these experiments were conducted on Nostrum (that reduces energy consumption by eliminating output buffers), same overheads are expected for a high performance NoC (with buffered outputs). Since regardless of the buffer size, the fault tolerance architecture and ECC remains the same.

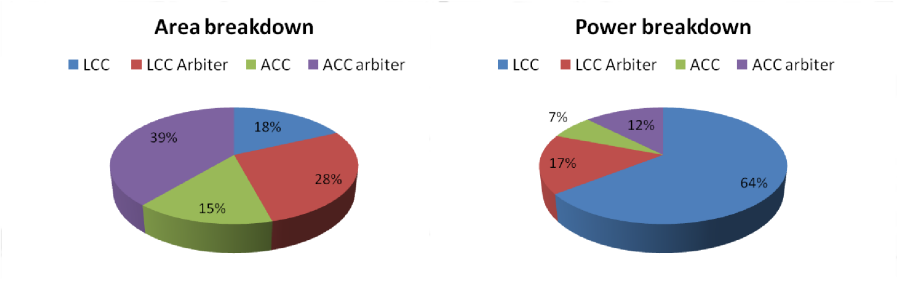

Figure 5.15: Area and power overhead of fault tolerance circuitry

5.8 Summary

In this chapter, we presented an adaptive fault tolerance mechanism, capable of providing the on-demand protection to multiple traffic classes present in a NoC platform. On-demand fault tolerance was attained by passing each packet through a two layer , low cost, class identification circuitry. Upon identification, the packet was provided one of the four fault tolerance levels: (i) no fault tolerance, (ii) end to end DEDSEC, (iii) per hop DEDSEC, or (iv) per hop DEDSEC with permanent fault detection and recovery. To manage the process autonomously, a three-tier control backbone, was introduced. It was responsible for reconfiguring the fault tolerance infrastructure upon arrival of each new traffic class. The obtained results suggest that the on-demand fault tolerance incurs a negligible penalty in terms of area (up to 5.3%) compared to the fault tolerance circuitry, while being able to provide a significant reduction in energy (up to 95%) by providing protection only to the control traffic.

Chapter 6

Private operating environments for CGRAs

6.1 Introduction and Motivation

As the dark silicon era fast approaches, where the thermal considerations will allow only a part of chip to be powered on, aggressive power management decisions will become critical. This chapter presents the architecture and algorithms that allow aggressive power management in CGRAs. The proposed solution, called Private Operating Environments (POEs), allows various applications (hosted by a CGRA) to enjoy the voltage/frequency operating points tailored to its needs. Specifically, we deal with the case when a CGRA hosts multiple applications, running concurrently (in space and/or time), and each application has different performance requirements. Some applications enjoy relaxed timing budget, and can afford to run at a low voltage/frequency operating point. While other applications, have stringent timing deadlines that require the CGRA to operate at maximum frequency/voltage. To efficiently host these applications, requires a platform that is geared to dynamically and with agility create arbitrary voltage/frequency partitions. Various requirements of such a platform are depicted in Table 6.1. In this section we will briefly describe the each function and its requirements. The detailed implementations will be given later in this chapter.

For efficient operating point selection, we have employed *Dynamic voltage and frequency scaling (DVFS)* [3]. DVFS enhances energy efficiency by scaling voltage and/or frequency to match the runtime performance requirements. To realize DVFS, requires voltage regulators and frequency dividers. Depending on the granularity of power management, DVFS can be either coarse-grained or fine-grained [41]. In *coarse-grained DVFS*, the operating point of entire platform is scaled to match frequency/voltage re-

Table 6.1: Private operating environment requirements

Function	Requirement
Frequency/voltage scaling	(i) Voltage controller (ii) Frequency divider
Data-flow management	(i) Intermediate buffers (ii) Frequency regulation
Metastability management	Synchronizes
Runtime parallelism	(i) Application model (ii) Multiple profiles
DVFS/parallelism intelligence	Algorithms and feedback loop

quirements of the application needing maximum performance. *Fine-grained DVFS* allows to modify the frequency/voltage of different parts of chip separately. Therefore, for contemporary platforms, fine-grained DVFS offers better energy efficiency by exploiting fine-grained workload locality [70]. However, realization of fine-grained DVFS is strongly challenged by factors (e.g. data-flow management and metastability) when data crosses clock boundaries [33]. In this chapter, we will show how these synchronization overheads can be reduced by exploiting the reconfiguration features offered by modern CGRAs. Synchronization overheads arise from extra buffers and handshakes needed to provide reliable communication between different *islands*, (i.e. different parts of a platform with different operating points). Our solution relies on runtime generation of a bitstream, which configures one of the existing cells (hardware resource) as an isolation cell, called *dynamically reconfigurable isolation cell* (DRIC). The DRIC serves to synchronize data exchanges between different islands. Reduction in overheads is achieved by eliminating the need for most of additional dedicated hardware. To reduce energy consumption even further we utilize Autonomous Parallelism, Voltage, and Frequency Selection (APVFS). that in addition to selecting optimal frequency/voltage, also parallelizes the applications at runtime [57]. To implement runtime parallelism, various versions (i.e. implementations of an application with different degree of parallelism) are stored. High energy efficiency is achieved by dynamically choosing the version that requires the least voltage/frequency to meet the deadlines on available resources.

Motivation: To illustrate motivation for using fine-grained DVFS, DRIC, and APVFS consider Figure 6.1, showing a CGRA with six processing elements (PEs). The figure depicts a typical scenario, in which WLAN transmits data to an MPEG decoder. Each of these applications is mapped to a different part of device and requires different throughput (performance). Fine-grained DVFS promises a considerable increase in energy efficiency by providing a separate voltage/frequency to each of these parts. Fine-

grained DVFS, however, requires additional synchronizers embedded in all PEs, causing unnecessary overheads. The proposed technique reduces these overheads, significantly, by configuring a DRIC to synchronize communication between the PEs, which actually communicate (PE2 and PE4 in the figure). APVFS can enhance the energy/power efficiency even further by shifting to a parallel version with lower frequency/voltage (F3, V3 in the figure).

The proposed scheme is generic and in principle applicable to all grid based CGRAs. To report concrete results, we have chosen dynamically reconfigurable resource array (DRRA) [112], as a representative CGRA. Simulating many practical applications revealed a significant reduction in power and energy consumption, compared to traditional DVFS techniques. Matrix multiplication (with three versions) showed the most promising results giving up to 23% and 51% reductions in consumption (for the tested applications), respectively. Synthesis results confirm that our solution offers considerable reductions in area overheads compared to state of the art DVFS techniques.

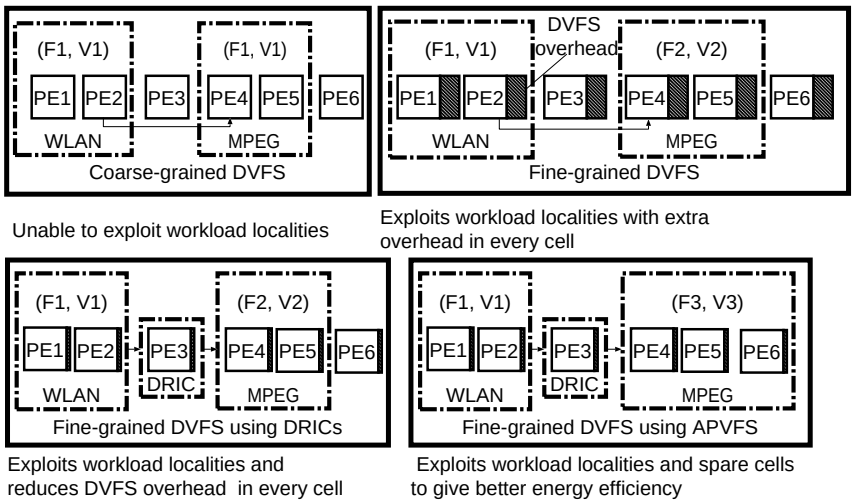

Figure 6.1: CGRA hosting multiple applications

6.2 Related Work

Since our work relates both to reduction in DVFS overheads and optimal version selection (dynamic parallelism), we review the most prominent work from both areas that is relevant to our approach.

Reduction in DVFS overheads: DVFS has been an area of extensive research in recent years for system on chip [3]. Unfortunately, only few works deal with implementing DVFS on CGRAs [68], [104]. Liang et al.

[41] and Amir et al. [102] proposed using reconfigurable links to reduce the overheads imposed by the synchronizers in network on chips (NoC). The reconfigurable links bypass the synchronization circuitry if two cells operate at same frequency. Both of these methods require dedicated reconfigurable buffers and the reconfiguration is mainly used to minimize the latency. Yang et al. [101] exploit DVFS to reduce reconfiguration energy in runtime reconfigurable devices. This method is in principal opposite to what we try to achieve (i.e. use reconfiguration to reduce DVFS overheads). Warp processor [84] monitors the program at runtime and generates instructions to configure its hardware for computation intensive parts. The fast execution of certain parts creates idle slacks. The voltage is later scaled to take advantage of these idle slacks. We use runtime creation of DRICs inspired from this method. To the best of our knowledge, a technique that exploits the reconfiguration to reduce dedicated hardware, needed for fine-grained DVFS, is missing (in CGRA domain).

Runtime parallelism: Traditionally, the platforms were provided with only one configuration per task considering the worst case [38]. Nagarajan et al. [107], explored the possibility of dynamic parallelism, by employing an array of small processors. The proposed architecture allowed different levels of parallelisms ranging from single thread on multiple processors to running many threads on a single core. P. Palatin et al. [95], presented component-based programming paradigm to support run-time task parallelism. In the MORPHEUS project [122], Thoma et al. developed dynamically reconfigurable SoC architectures. Teich et al. [121] presented a paradigm (called invasive computing) to parallelize/serialize tasks on a CGRAs.

DVFS + Runtime parallelism: Although both DVFS and runtime parallelism have been researched thoroughly, only few works combine parallelism with DVFS, to allow aggressive voltage/frequency scaling. Couvreur [128] presented a two phase method to enhance energy efficiency, by combining dynamic version selection and DVFS on a CGRA, called ADRES. The work was later improved in [129] by providing criteria for selection of optimal versions. However, this method incurred prohibitive memory and reconfiguration costs, limiting the versions that can be stored. To reduce storage and reconfiguration requirements, we [57] suggested to store only a single version and represent the remaining versions by their differences from the original. In [60], we presented the architecture and algorithm to dynamically realize Autonomous Parallelism Voltage and Frequency Selection (APVFS). The proposed architecture/algorithm promised high energy efficiency by parallelizing a task, whenever adequate resources are available. However, the task parallelism relied on solely on greedy algorithm, that can (in some cases) be more costly than simple DVFS (see Section 6.9 and [62]). In [62], we presented energy aware task parallelism to address the problem faced by the greedy algorithm. In this chapter, we combine the architecture

proposed in [60] and with the parallelism intelligence presented in [62].

Compared to related work, this thesis has four major contributions:

- We present Architecture and implementation of fine-grained dynamic voltage and frequency scaling (DVFS), using low latency rationally related frequencies, on a CGRA;
- We propose energy aware task parallelism, that parallelizes a task only when its parallel version offers reduction in energy;
- We integrate energy-aware task parallelism with operating point intelligence, to select optimal parallelism, voltage, and frequency at runtime; and
- We present a complete HW/SW solution that serves to realize all the above concepts.

6.3 DVFS infrastructure in DRRA

We have chosen Globally Ratio-synchronous Locally Synchronous (GRLS) [20] design style to implement DVFS in DRRA. The main motivation for choosing GRLS is that it promises higher performance compared to GALS systems (requiring handshake) and higher efficiency compared to me-synchronous systems (requiring all modules to be at same frequencies).The only restriction is that it requires that all clocks on the chip run at frequencies which are sub-multiple of a so called global virtual frequency, F_H. For a detailed discussion of GRLS and other clocking strategies, we refer to [20]. It should be noted that although we specifically use GRLS for this thesis, the proposed methods are, in principal, applicable to GALS as well.

A power management system has been built on top of DRRA by introducing a wrapper around every cell as shown in Figure 6.2. The wrapper is used to ensure safe communication between nodes operating at different frequencies and to realize Dynamic Voltage and Frequency Scaling (DVFS). The access point to provide the power services is given by the power management unit. The power management unit, depending on voltage select and frequency select signals, uses voltage control unit and clock generation unit to control the voltage and the clock frequency, respectively.

6.3.1 Voltage Control unit

To implement voltage scaling, we have used *quantized supply voltage* levels. In this technique, multiple global supply voltages are generated on-chip or off-chip and distributed throughout the chip using parallel supply voltage

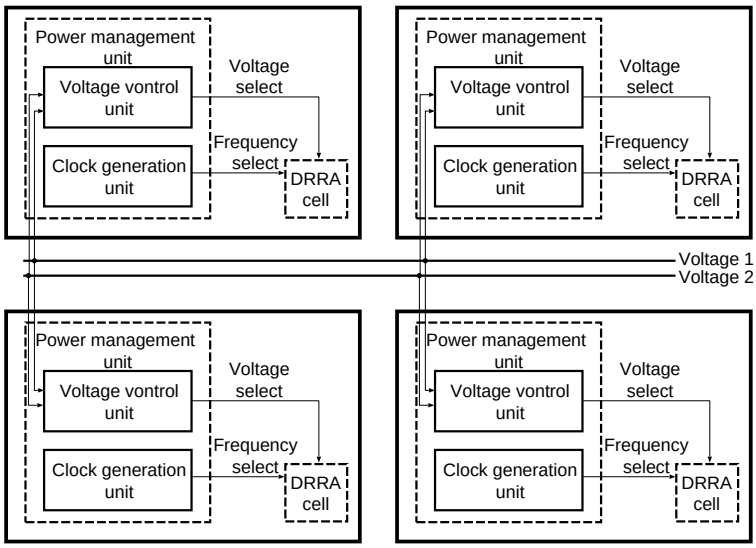

Figure 6.2: DVFS infrastructure in DRRA

distribution grids. To realize voltage switching, a local Voltage Control Unit (local VCU) is embedded in every DRRA cell, as shown on Figure 6.3. Each local voltage control unit contains a PMOS power switch and the necessary logic to drive it. The power switches select one of the global supply voltages as the local supply voltage for the module. This allows quantized voltage regulation. The central Voltage Control Unit (central VCU) powers the distribution grid with different voltage levels. Depending on the system requirements, any number of rails can be formed and distributed. For this work, we have chosen two operating voltages.

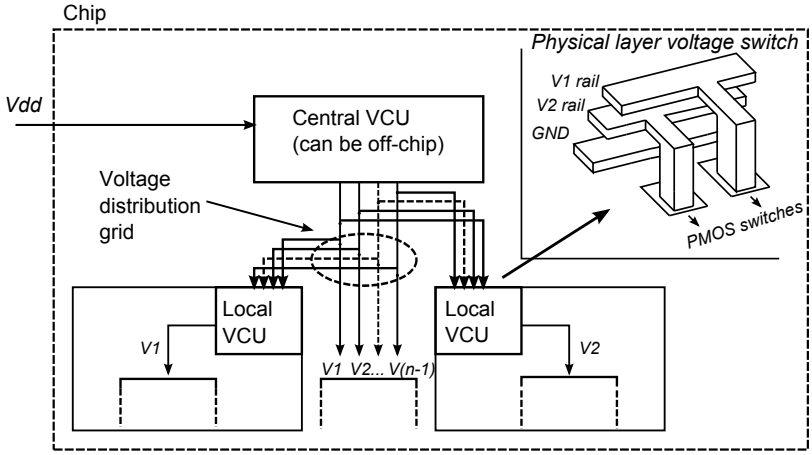

Figure 6.3: Voltage control unit

6.3.2 Clock generation unit

For frequency scaling, we have used a Clock Generation Unit (CGU) inspired from [20]. The hardware for the clock generation unit is shown in Figure 6.4. The CGU receives the selected clock from the local voltage control unit. It uses the Frequency Select (Fs) signal, from the runtime resource manager, to set a division threshold. To generate the output clock, $Clko$, of desired frequency a counter is incremented every cycle, and compared to the division threshold value. If $count = Fs$, the counter is reset and a toggle Flip Flop (FF) enabled. The toggle Flip Flop (FF) derives the $Clko$ signal.

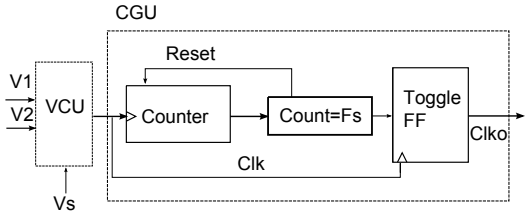

Figure 6.4: Clock generation unit

6.4 Data flow management

Whenever two islands with different frequencies/voltages communicate, synchronization is required. The synchronization requires data-flow and metastability management. We have employed *Dynamically Reconfigurable Isolation Cells (DRICs)* to meet these requirements at runtime. Where a DRIC can be any spare DRRA cell, that is dynamically configured by the runtime resource manager (see Chapter 2) to synchronize communications between two islands. In this section, we discuss how a DRIC manages data-flow later in Section 6.5 we will discuss how it caters metastability. For data flow management, the DRIC has two responsibilities: (i) to regulate the data transmission and (ii) to provides storage buffers (in form of reg-files) for intermediate data. To formulate the need for data-flow management, consider that two islands with different frequencies need to exchange data. Let F_t and F_r be the transmitter and the receiver frequencies, respectively. As long as $F_t \leq F_r$, the transmitter can transmit safely without any regulation. If $F_t > F_r$, a regulation mechanism is needed to prevent the loss of data.

6.4.1 Dynamically Reconfigurable Isolation Cell (DRIC)

Transmission rate management has two major requirements (i) a transmission algorithm to decide when the data should be sent/stalled and (ii) a buffer to store intermediate data. Additionally, to dynamically create the arbitrary partitions (frequency/voltage islands) the architecture should be

able to meet these requirements between any two cells of DRRA. We have used Dynamically Reconfigurable Isolation Cells (DRICs) to meet the above requirements. To create a DRIC, the runtime resource manager (see Chapter 1) configures a spare CGRA cell as isolation cell, whenever a new island is created. An isolation cell contains three parts: (i) an FSM dictating when to transmit data (based on the regulation algorithm used), (ii) a buffer to store data, and (iii) a link connecting the transmitter, DRIC and the receiver. In DRRA, the configware to manage data-flow (for DRIC) can be generated using only the reg-file and SB instructions (see Chapter 2 to recall the functionality of reg-file and SB instructions). Final composition of reg-file instructions is dependent on the transmission algorithm used for synchronization. Contents of the SB instructions are determined by the physical location DRIC. The process of DRIC generation is shown in Figure 6.5 and explained in Sections 6.4.1 and 6.4.1.

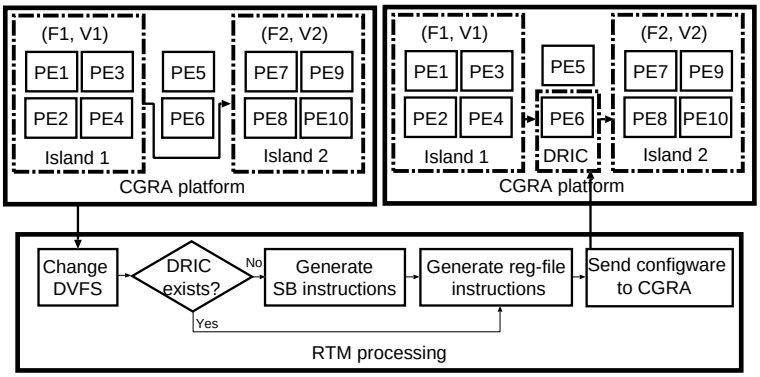

Figure 6.5: DRIC generation and placement

Register-file (reg-file) instruction generation

We have employed the reg-files to realize the transmission algorithm. The GRLS clocking strategy, used in our architecture, uses the frequency regulation algorithm presented in [24]. Algorithm 1 shows a modified version of this algorithm. $N_t = F_h/F_t$ and $N_r = F_h/F_r$ are called transmitter and receiver division ratios. Where F_h, F_t, and F_r are the global, transmitter, and the receiver frequencies, respectively. Comments m and o indicate whether the line is our modification or original code from [24]. The algorithm allows a transmitter to send data only when $send = 1$. To implement the regulation algorithm (using DRIC), we considered three alternatives: (i) configure DPUs, SBs and reg-files to directly mimic the regulation algorithm and control data flow accordingly, (ii) map the regulation algorithm as a separate thread on RTM (i.e. in software) and control data transmission by sending the calculated value of *send* to DRRA, and (iii) use a hybrid of above

approaches. The first alternative requires at least two cells to implement a DRIC, making it costly (in terms of area, time and power). The second alternative is too costly because it involves software based calculation of *send* whenever two islands communicate. We therefore, resort to third alternative. In our technique, the RTM does initial calculation to generate custom configware and uploads it to DRRA. After initialization, DRRA regulates data transmission internally (without any assistance from RTM). The DRIC configware is generated by RTM in three intermediate steps: (i) RTM exploits the fact that the sequence of '0s' and '1s' (in variable *send*), calculated by regulation algorithm is periodic with period P; The period, P, for given N_t and N_r, is calculated using equation:

$$P = N_r/HCF, \tag{6.1}$$

where HCF is the Highest Common Factor (HCF) of the transmitter and the receiver division ratios, (ii) the loop in Algorithm 1 is executed P times and the corresponding values of *send* (either '0' or '1') are stored in an array named *sendseq*, and (iii) the delay and reg-file instructions are generated for every '0' and '1' stored in sendseq, respectively.

Although the above steps are performed in software, they have negligible overheads (compared to overall application execution time), since they are executed in background only once (when DRIC is created). The process of reg-file instruction generation for $N_r = 20$ and $N_t = 8$ is shown in Figure 6.6. Using Equation 6.1 we get $P = 5$. The regulation algorithm generates the sequence 1010010100100100..... For $P = 5$, the sequence reduces to 10100. As a result, DRIC configware will contain 2 reg-file and 3 delay instructions.

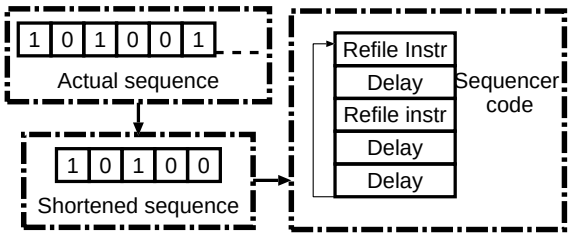

Figure 6.6: Generation of DRIC configware from regulation algorithm

Switch-box (SB) instruction generation

SB instructions need to be generated whenever an island is created. These instructions serve to connect the transmitter and the receiver with the DRIC. Based on the location of the transmitter and the receiver, the RTM calculates the optimal position (in terms of minimum hop-counts) for placing the

```
Data: N_r, N_t
Result: Array containing minimum sequence that needs to be stored
c=Nr;
P = N_r/HCF ;                                          /* m */
if N_r <= N_t then                                     /* o */
    send=1; ;                                          /* o */
else                                                   /* o */
    for i ← 0 to P do                                  /* m */
        if c>Nr-Nt then                                /* o */
            send=1 ;                                   /* o */
            c=c-(Nr-Nt);                               /* o */
        else                                           /* o */
            send=0 ;                                   /* o */
            c=c+Nt ;                                   /* o */
        end
        sendseq[i]=send ;                              /* m */
    end
end
```

Algorithm 1: Modified regulation Algorithm

DRIC. The calculated optimal position is used to generate the configware for connecting the DRIC with the islands exchanging data.

6.4.2 Intermediate Storage

The storage requirement of a DRIC depends on the frequency difference between the communicating islands and the traffic burstiness. This storage is essential for all multi-frequency interfaces. In case the applications hosted by the frequency islands require a bigger buffer, two DRICs can be combined to meet the storage requirements. The algorithm to dimension the buffer lies outside the scope of this thesis.

6.5 Metastability management

The transmission algorithm, presented in the previous section, prevents the data overflow by constraining the transmitter. In this section, we will discuss how our architecture handles metastability, i.e. setup and hold violations. To cater the setup and hold violations, that occur due to voltage/frequency scaling, we employ ratio-synchronous metastability manager, inspired from [20]. The system level view of the metastability manager is shown in Figure 6.7 (a). If the communicating cells have different operating points (indicated by different island control line), the data is passes through the metastability

manager. Otherwise it is sent directly to the receiver.

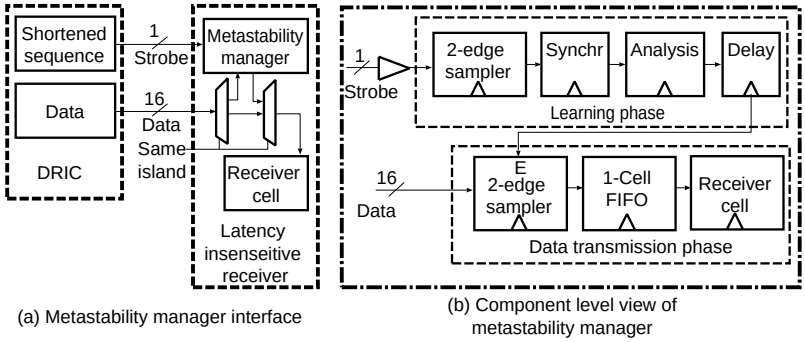

Figure 6.7: Metastability manager integration

6.5.1 Operating principle

As shown in Figure 6.7 (b), the functionality of the metastability manager can be divided into two phases (i) learning phase and (ii) data transmission phase. During the learning phase, it analyzes a 1-bit *strobe* signal to determine the time instants at which data can be safely sampled. The method relies on the periodicity of ratio-synchronous clocks; i.e. the relationship between the transmitter and the receiver frequencies repeats periodically. Based on the analysis results of the learning phase, the valid data is transmitted to the receiver in second phase. Here, we will first explain each component of the learning phase followed by its implementation. The data transmission phase will be explained with implementation.

Strobe sampling and synchronization

The strobe signal originates from Dynamically Reconfigurable Isolation Cells (DRIC). The DRIC uses the same shortened sequence (see Figure 6.6), generated for data-flow management, to drive the strobe signal. The shortened sequence is stored in a 16-bit circular buffer (considering the maximum possible size of shortened sequence), that shifts its contents after every cycle. The value of the least significant bit, in the circular register is assigned to a toggle flip-flop that drives the strobe signal. Simply put, the strobe signal toggles with each data transmission. The strobe signal itself is sampled, by the metastability manager, after a delay of TW, at every (positive and negative) clock edge. The motivation and method to determine TW will be given later in this section. Since the source and destination of the strobe signal have a different operating point, it is passed through high-latency multi-stage synchronizers, to ascertain its validity.

Strobe analyzes

The strobe analyzes phase relies on the fact that the divider algorithm (see Section 6.4) guarantees that the data sample obtained at either positive or negative clock edge is valid [20]. This phase determines whether to sample data at positive or negative clock edge. Let $S = (s_0, ..., s_i)$ denote a set of samples, of delayed strobe signal (obtained at every clock edge). It was shown in [20] that if $s_i \neq s_{i-1}$, the delayed strobe signal transitioned between the time instants $t_{i-1} - t_{su}$ and $t_i + t_{ho}$. Where t_{su} and t_{ho} denote setup and hold times, respectively. In other words if $s_i \neq s_{i-1}$, the data can be safely sampled at s_i.

6.5.2 Hardware Implementation

The RTL level implementation of metastability manager is shown in Figure 6.8 (b). We will explain the figure from left to right, considering how the data flows. A strobe line is connected via delay line to two flip-flops, one positive and other negative edge-triggered. To synchronize the strobe samples, each flip-flop is connected to a cascades of additional flip-flops. The output of the flip-flop cascade is compared with the sample arrived half a cycle earlier. The results of the comparator are fed to another a chain of flip-flops. The outputs of this flip-flops chain are connected to multiplexers controlled by $sel = KN_T - N_S - 1$ signal. Where N_S and N_T denote number of synchronization stages (typically 2 or 3 flip-flops) and transmission ratio, respectively. $K = \lceil N_S/N_T \rceil$ is the smallest integer that guarantees $KN_T - N_S - 1 \geq 0$. The circuit of Figure 6.8 (b) outputs Sp and Sn signals to ensure that the hardware of Figure 6.8 (a) transmits valid data. If $Sp = 0 (Sn = 0)$, $vp(vnz)$ is cleared, otherwise the value of the shortened sequence is stored in $vp(vnz)$ and the value of the data signal is stored in $dp(dnz)$. The dnz and vnz signals are synchronized to the receiver clock domain. $vp(vn)$ indicates that a valid data item has just been sampled on the positive (negative) edge of the clock. The ds register acts as a one cell buffer to absorb the bursts of data sampled on two consecutive edges. When dp and dn contain a word, the oldest (dn) is output and the newest is saved in the ds register. $vs = 1$ when the ds register contains a valid item.

6.6 Dynamic Parallelism

In this section, we will explain how an application with each task containing multiple versions is modeled. Additionally, we will also explain the potential problems with existing state of the art parallelism techniques (when combined with DVFS).

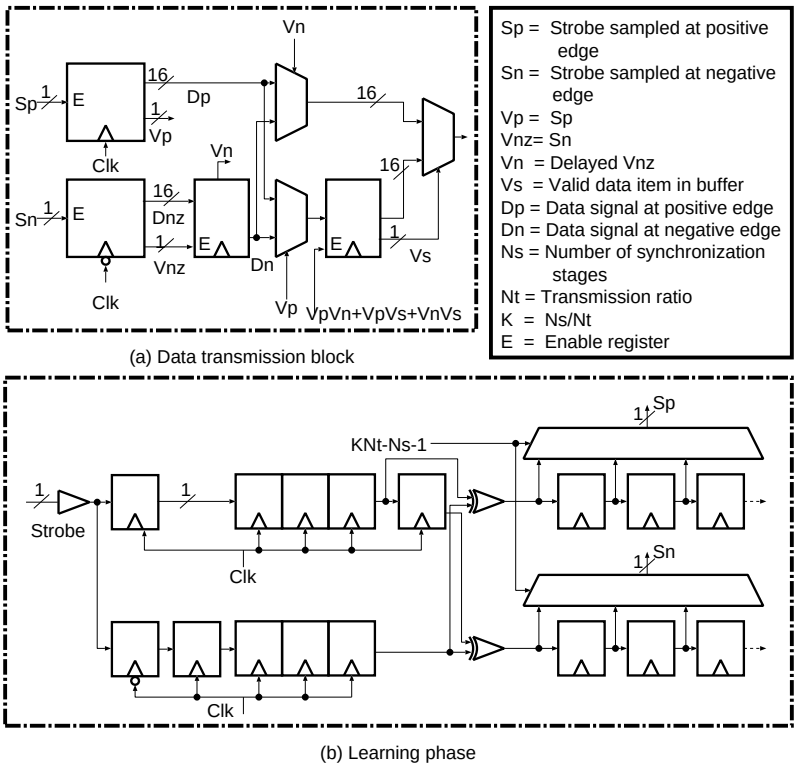

Figure 6.8: Metastability manager

6.6.1 Model

Before we introduce the intelligence to control the power management infrastructure, we will present an intuitive way to model an application containing multiple tasks, where each task can be parallelized/serialized. In addition, we will also show a simplistic delay and energy model to visualize how the dynamic parallelism affects energy.

Application and delay model

An application A can be described by a *directed acyclic graph*, as shown in Figure 6.9. It is an enhanced version of the directed acyclic graph proposed in [74], that modeled application containing only the tasks with single version. A directed acyclic graph is a quadruple $< T; V; W; C >$, where T is a set of tasks, corresponding to its nodes. V_i represents the set of versions (implementations with different degree of parallelism) for each task $t_i \in T$. The weights, $w_{i,j}$, of a node t_i (shown below the nodes), represent the execution time of each version, $v_{(i,j)}$. A task with multiple versions $t_i(v_{(i,j)})$ is expressed as:

$$t_i(v_{(i,j)}) = \{t_i(v_{(i,j)}) \in T \mid w(t_i(v_{(i,j)})) > w(t_i(v_{(i,j+1)}))\}, \tag{6.2}$$

where $i = 1, ..., T$ and $j = 1, ..., V_i$. Each edge, $C(t_i, t_j)$, represents the precedence constraints between tasks t_i and t_j. Consider an application A, pipelined into T tasks, such that that each task executes on a different part of the device. After the pipeline is filled, net execution time of the application w_A can be approximated as:

$$w_A \approx max(w(t_i(v_{(i,vm)})), \tag{6.3}$$

where $max(w(t_{(i,vm)}))$ is the mapped task version requiring maximum execution time. Simply put, an application is represented by a set of tasks. Each task contains multiple versions with different degree of parallelism. During execution, any task version can be mapped to the device. Overall application execution time is approximately equal to the mapped task version, with maximum execution time.

Energy model

To visualize the effect of parallelism on energy efficiency, we present here a simplistic energy model. The actual energy estimates, calculated by Synopsys Design Compiler on synthesized DRRA fabric, will be given in Section 6.9. Overall dynamic energy consumption is composed of two components (i) computation energy and (ii) communication energy. Consider a CGRA with multiple processing elements. The supply voltage and frequency, for a

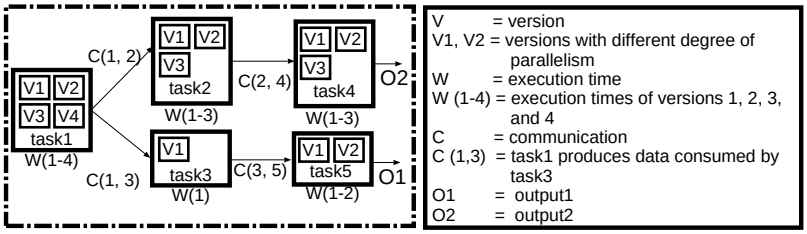

Figure 6.9: Directed acyclic graph representing tasks with multiple versions

processing element, are represented as VDD_i and F_i, respectively. Where, i denotes the processing element number. Using this notation, the dynamic energy consumptions for computations can be written written as:

$$E_i(VDD_i, F_i) = SW_i * F_i * VDD_i^2 * Ac_i, \tag{6.4}$$

where Ac_i is the time for which i^{th} processing element remains active and SW_i stands for the total switched capacitance per cycle. Equation 6.4 clearly shows that the energy consumption can be reduced by lowering the frequency and/or voltage. The for an application, the lowest allowed voltage/frequency is determined by its performance requirements. Parallelism induces speedup allowing to scale the voltage/frequency even further thereby reducing the net energy consumption.

To model the communication energy, we use the bit energy matrix proposed by Benini and Mecheli [127]. The bit energy matrix estimates the communication energy, for a packet switched NoC to be:

$$E_{bit} = E_{Lbit} + E_{Bbit} + E_{Sbit}, \tag{6.5}$$

where E_{Lbit}, E_{Bbit}, and E_{Sbit}, represent the energy consumed by the link, buffer and switch fabric, respectively. For a circuit switched NoC (employed in many CGRAs [112]), the bit energy matrix can be simplified to:

$$E_{bit} \approx E_{Lbit}. \tag{6.6}$$

Since E_{Bbit}, and E_{Sbit}, are negligible for circuit switched networks (because after a route is configured, all packets follow the same route). It will be shown that the parallel versions require additional communication energy, since the data has to travel longer.

6.6.2 Optimal DVFS granularity

Remember from Section 6.1, depending on the granularity of power management, DVFS can range from coarse-grained to fine-grained. Considering the costs and benefits of fine/coarse grained DVFS, it was shown in [60] that for CGRAs the DVFS is most effective when is done at *application level*.

These results have been further quantified by our experiments in Section 6.9). In application level DVFS, the operation point of an entire application (e.g. WLAN, MPEG4) is scaled. Implementing DVFS at a finer granularity (e.g. interleaver, scrambler) would require additional power hungry buffers (DRICs in our case), that diminish the benefits of DVFS.

6.6.3 Problems with unconstrained parallelism

Existing techniques that aim to enhance energy efficiency by employing parallelism, use greedy algorithm [128, 129, 57, 60]. However, the greedy algorithm blindly parallelizes tasks causing two potential problems: (i) unproductive resource allocation and (ii) inter-task resource arbitration.

Unproductive resource allocation problem

The existing techniques, that combine parallelism with DVFS, take decisions to parallelize and/or serial a task, based on greedy algorithm. The greedy algorithm blindly shifts a task $t_i(v_{(i,j)})$ to its parallel version $t_i(v_{(i,(j+1)})$, provided the required resources are available. Unfortunately, the parallel version $t_i(v_{(i,j+1)})$ guarantees a reduction in overall application execution time w_A only if it requires maximum time; i.e. $w(t_i(v_{(i,j)})) = max(w(t_i(v_{(i,vm)})))$ (from Equation 6.3). Since, DVFS is done at application level (for motivation see Sections 6.6.2 and 6.9 or [60, 62]), without a reduction in overall application execution time, voltage/frequency cannot be lowered. At the same voltage and frequency $t_i(v_{(i,j+1)})$ is likely to result in excessive energy consumption due to additional data communication cost of the parallel version (see Equation 6.6). Moreover, if a resource is allocated to a task, it cannot be can be turned off to save static energy. Therefore, it is essential to judiciously decide whether to parallelize a task. The problem can be formulated as follows: Given an application A with set of tasks T, subject to availability of resources available and parallel versions, parallelize a task $t_i \in T$, only if parallelizing it increases overall application throughput.

Inter-task resource arbitration problem

To visualize this problem, consider an instance of a CGRA platform with limited free resources. In a mapped application, A, multiple tasks can be parallelized. However, the free resources are only sufficient to parallelize some of the tasks. In this scenario, the resource manager should allocate the free resources to the task(s) promising the highest energy efficiency. The problem can be formulated as follows:
Given an application A with set of tasks $t_p \in T$ requiring some resources to shift to a parallel version, subject to availability of resources, parallelize the task $t_i \in t_p$ promising maximum energy efficiency.

To illustrate these drawbacks of greedy approach, consider Figure 6.10. Figure 6.10 (a) depicts an instance of CGRA that hosts two applications, simultaneously. Application 1 contains three tasks, where tasks 1 and 3 can be parallelized. Figure 6.10 (b) shows another instance of CGRA in which application 2 finishes execution, leaving behind free resources sufficient to parallelize both Task1 and Task3. The greedy algorithm will blindly parallelize both the tasks, even though they have no impact on the application throughput. Without a speedup, the operating point, of the application, cannot be lowered. At same voltage and frequency, a parallel versions is likely to be more expensive in terms of both dynamic (resulting from additional data communication costs) and static (since an allocated resource cannot be turned off) energy. Figure 6.10 (c), shows an instance when limited free resources are available. The greedy algorithm will be unable to decide, which task to parallelize.

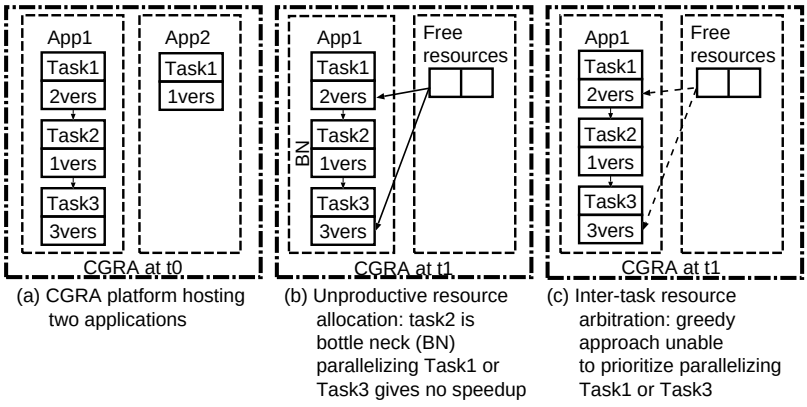

Figure 6.10: Shortcomings of greedy algorithm

6.7 Parallelism Intelligence

As a solution to the above mentioned problems, we present energy aware task parallelism. The proposed solution relies on resource allocation graphs and autonomous parallelism, voltage, and frequency algorithm (APVFS), to make parallelism decisions. In this section, we will show how to parallelize tasks intelligently, later in Section 6.8, we will show the criteria to choose voltage/frequency.

6.7.1 Architectural enhancements

The greedy algorithm requires no information about the application behavior as it parallelizes tasks blindly. The proposed approach, aims to guide the resource manager in dynamically allocating resources to tasks, such that

each resource allocation reduces energy consumption of overall application. As shown in Figure 6.11, our approach relies on a compile-time generated resource allocation graph (RAG). The runtime resource manager uses this RAG as a guide to orchestrate parallelism at runtime. The RAG contains information about the execution time and data dependencies of tasks. Based on this information, and application deadlines, the runtime resource manager alters frequency/voltage and manipulates parallelism, as will be discussed later in this section.

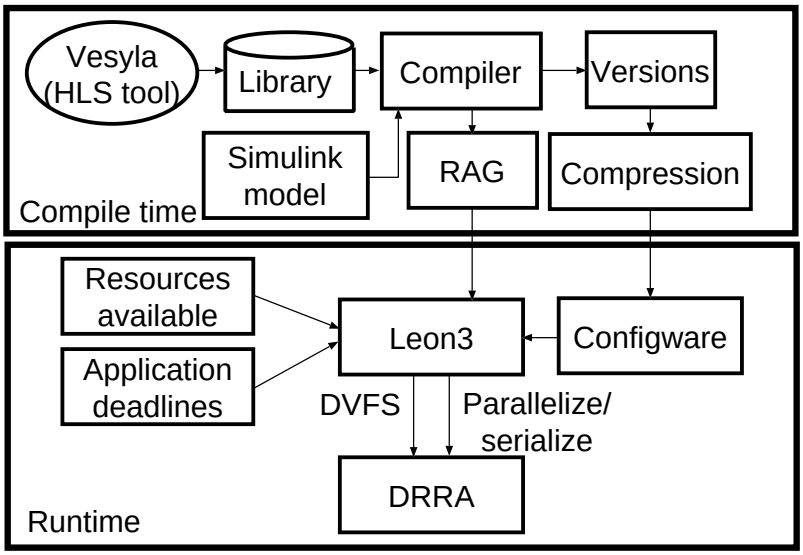

Figure 6.11: Modified programming flow for energy aware task parallelism

6.7.2 Resource allocation graph (RAG) model

The resource allocation graph (RAG) ensures that a resource is only allocated to a task, if it decreases overall application execution time. To accomplish this, RAG imposes complete ordering on task parallelism, by using a one dimensional linked list shown in Figure 6.12. The RAG is composed of five main components: (i) main nodes, (ii) entry edge, (iii) mapping header, (iv) sub node, and (v) sub edge. The functionality of these components is summarized in Table 6.2. RAG contains a set of main nodes connected by directed edges, called entry edges. A main node, along with the nodes to its left, represent the *application parallelism state*. Application parallelism state identifies the version of each task at a main node. The main-node also contains information about the application execution time at that node. The left most main node represents all tasks with their serial version, and therefore has the maximum execution time. The execution time of a main node decreases from left to right. The entry edges show the additional re-

sources needed for moving to the main node towards right. A pointer called, mapping header, is used to identify the task version currently mapped to the platform. The mapping header points to one of the main nodes. During execution, if the resources specified by the entry edge become free (because another application finishes execution), the mapping header moves towards right main node. The tasks, indicated by the sub nodes, are parallelized. To each parallelized task, the resources allocated by the sub edge, are allocated.

To clarify the functionality of resource allocation graph (RAG), consider for example the RAG shown in Figure 6.13 (bottom right block). The mapping header points to main node 3. It means that currently tasks 2, 5, 3, and 6 have versions 3, 2, 2, and 2, respectively, mapped to the platform. Rest of the tasks have their serial version mapped to the platform. The application parallelism state will remain in the same till at least three additional resources are available.

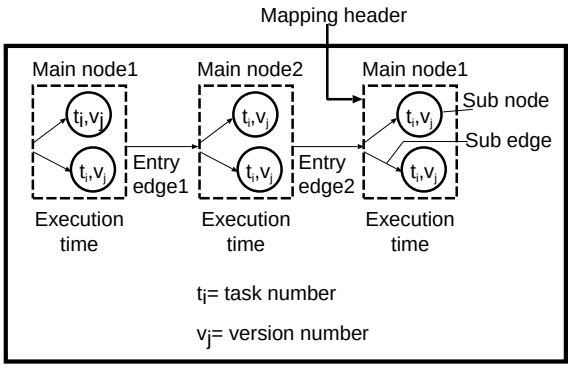

Figure 6.12: Resource allocation graph model

Table 6.2: Functionality of various RAG components

Component	Functionality
Main node	(i) Identifies the tasks to allocate the free resources (ii) A main mode with all the main nodes to its right identify the version of each task
Left most main node	Represents all tasks with in serial version
Right most main node	Represents all tasks with maximum parallelism
Mapping header	Shows the mapped version of each task
Main node execution time	Shows the execution time of an application when all the tasks using the version specified by the main node
Entry edge	Specifies the number of resources needed for a shift to main node towards right
Sub node	Indicates the task to parallelize
Sub edge	Indicates the resources to each task in sub node

6.7.3 RAG generation

RAG is generated, at compile time, from the directed acyclic graph of an application. The proposed solution accommodates the directed acyclic graph with multiple outputs, with different deadlines. As shown in Algorithm 2 and illustrated in Figure 6.13, the RAG is created in three main steps.

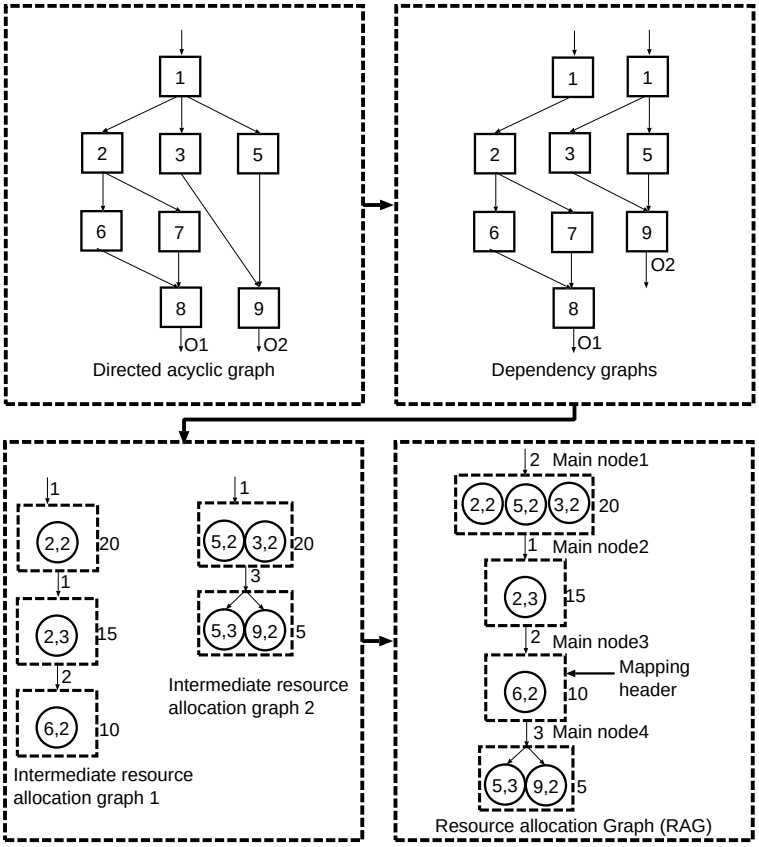

Figure 6.13: Resource allocation graph (RAG)

In the first step, a separate *dependency graph*, is created, for each application output. Each dependency graphs, thus created, contain the tasks on which an output depends. A task t_i or an output o_j is considered to be dependent on another task t_k, if t_i or o_j consumes the data produced by t_k. In the second step, each dependency graph is converted to an *intermediate resource allocation graph*. An intermediate resource allocation graph is modeled the same way as resource allocation graph, discussed earlier in this section. The first main node of an intermediate resource allocation graph, represents all tasks (from the corresponding dependency graph), in their serial versions. For generating the rest of the main nodes the execution times of each task, is profiled and stored with the dependency graphs. To create

the second node, the task(s), t_{max}, with maximum execution time in the dependency graph, is isolated. The overall application execution time approximately equals t_{max}. The parallel version of t_{max} forms the second main node of intermediate resource allocation graph. In the dependency graph, the execution time of t_{max} is updated to the execution time of its parallel version. If the dependency graph contains multiple tasks with maximum execution time, a sub node for each task is created and placed inside the main node. The rest of the nodes of each intermediate resource allocation graph are created the same way. The process continues until a task found which cannot be parallelized. In the third step, the intermediate resource allocation graphs are merged into a single resource allocation graph. Like step 2, this step is also carried out iteratively. In each iteration, all the intermediate resource allocation graphs are searched to find the main node with maximum execution time. This main node, along with its entry edge, is moved to the resource allocation graph (RAG). Therefore, a new node is added to the RAG in each iteration. The process continues till all nodes from intermediate resource allocation graphs are finished. To preserve the dependency constraints, with exception of the first main nodes (in intermediate resource allocation graphs), a main node can only be moved to the RAG, if its predecessor already exists in RAG.

6.8 Operating point intelligence integration

The dynamic voltage and frequency scaling presented in [60] and [62] rely on monitoring the deadlines at runtime. This method has two drawbacks: (i) it forces the application to miss a deadline, and is therefore applicable for only soft deadline applications and (ii) it requires multiple counters that consume additional dynamic energy. The main motivation for using the counters was that storing the complete application profile with all the versions is very costly. We will show here that by using the RAG, the storage requirements are reduced, or the profile can be generated at the runtime.

6.8.1 Integrating voltage and frequency in RAG

The proposed algorithm caters the overheads of the runtime monitoring using counters by adding the operating point information in the compile time generated resource allocation graph. Remember that each main node in RAG represents an entire application (to find the version of each task a node with the nodes to its right). The main node also contains application execution time (in cycles) at a particular state. Therefore, given the application deadline and available frequencies, the lowest frequency that meets applications deadlines can be easily calculated.

Input: DAG representing an application ;
Output: Resource allocation graph (RAG) ;
`/* Generate dependency graphs */`
Construct, D, dependency graphs for each application output and calculate execution times of all task versions;
`/* Generate intermediate resource allocation graph */`
for $j \leftarrow 0$ **to** D **do**
  for *continuous* **do**
    Isolate the bottleneck task(s), t_{max} ;
    entry edge= 0 **for** $m \leftarrow 0$ **to** L **do**
      `/* L is a set of tasks,` tl_i`, with execution time =` t_{max}`, and each` tl_i `is a sub node */`
      Find the resources R_m needed to parallelize t_{max} ;
      Create a sub node with the weight of sub edge $=R_m$;
      $entryedge = entryedge + R_m$;
    end
    Create a new main node with entry edge, sub nodes, and sub edges;
  end
  Break loop ;
end
`/* Generate Resource Allocation Graph (RAG) */`
In the intermediate resource allocation graphs, find main nodes, MN_{max}, with maximum execution time;
if $MN_{max} > 1$ **then**
  Combine all Main Nodes (MN) in to a single node ;
end
while $\exists$ MN *in intermediate resource allocation graphs* **do**
  Find the MN_{max} ;
  if *the predecessor of the main node already in RAG* **then**
    Add mn_l as a new node in RAG ;
  end
end

Algorithm 2: Resource allocation graph (RAG) generation

Data: Available frequencies $Freq$, application deadlines A_{dl}, RAG main nodes MN with execution time in cycles MN_{et}
Result: RAG main nodes with corresponding frequency and voltage
Selected frequency ;
$F_s = Freq(0)$;
for $i \leftarrow 0$ **to** MN **do** `/* loop through all main nodes */`
  for $j \leftarrow 0$ **to** $Freq$ **do** `/* loop through all frequencies */`
    exe_{time}= $MN_{et}(i)/Freq(j)$;
    if exe_{time} <= A_{dl} **then**
      F_s=Freq(j) ;
    else
      Add F_s with the MN(i);
      break frequencies loop;
    end
  end
end

Algorithm 3: Generating RAG with frequency and voltages

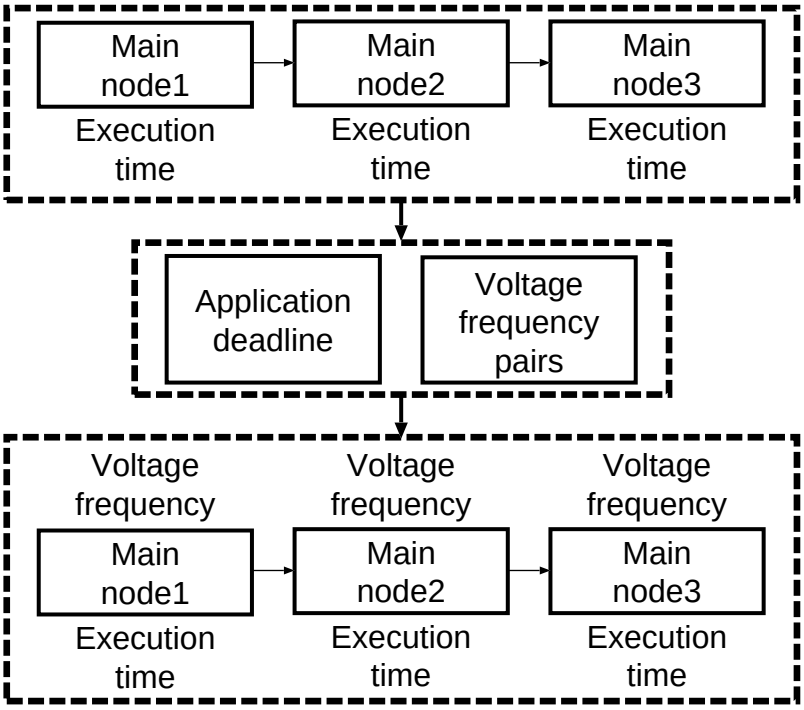

Figure 6.14: Resource allocation graph (RAG) with voltage and frequencies

6.8.2 Quantifying Feasibility of profiling

The main motivation for monitoring the runtime performance in our previous works [60, 62] was to avoid the excessive memory overhead of profiling all the versions with all the frequencies. The total values that needed to be stored is given by $versions * frequencies * modes$. Where mode represents the operating mode of an applications; e.g. WLAN be mapped with either BPSK, QPSK, or 16-QAM mode and the processing speed depends on the chosen mode. In the proposed approach, the memory requirement is simply equal to the number RAG nodes. The memory requirements for the presented algorithm is shown in Figure 6.15.

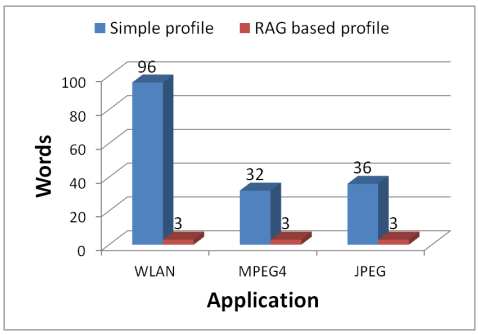

Figure 6.15: Memory requirements to generate profile for RAG based parallelism

6.8.3 Autonomous Parallelism, Voltage, and Frequency Selection (APVFS)

To demonstrate the effectiveness and overheads of using our scheme, we have used APVFS algorithm, shown in Figure 6.16. In the figure, Rf, Rep, Aa, Ar, V, and F refer to free resources, resources needed to enter the next RAG node, actual execution time, deadline, voltage, and frequency, respectively. Depending on the runtime deadlines and available resources, the algorithm iteratively finds a mapping offering high energy efficiency. The algorithm operates in three steps: (i) RAG forward traversal, (ii) RAG backward traversal, and (iii) parallelism lookup table traversal. In RAG forward traversal, the RAG is traversed from the first main node to the last main node till an entry edge with weight greater than free resources is found. The mapping pointer (MP) is placed at the source main node, of this edge. In RAG backward traversal, the RAG is traversed from this main node to the first node, to generate a parallelism lookup table. The parallelism lookup table is a look-up table with single column. The index of the table indicates the task number and its value denotes the mapped version. During the back traversal, the task versions, indicated by the RAG sub nodes, is placed in

the the task cell, if its empty. If a task cell is already full (indicating that a version with higher energy efficiency is already present) no action is taken. In the parallelism lookup traversal step, the task versions present in the filled cells are mapped to the device. For empty cells, the most serial versions of the tasks are mapped.

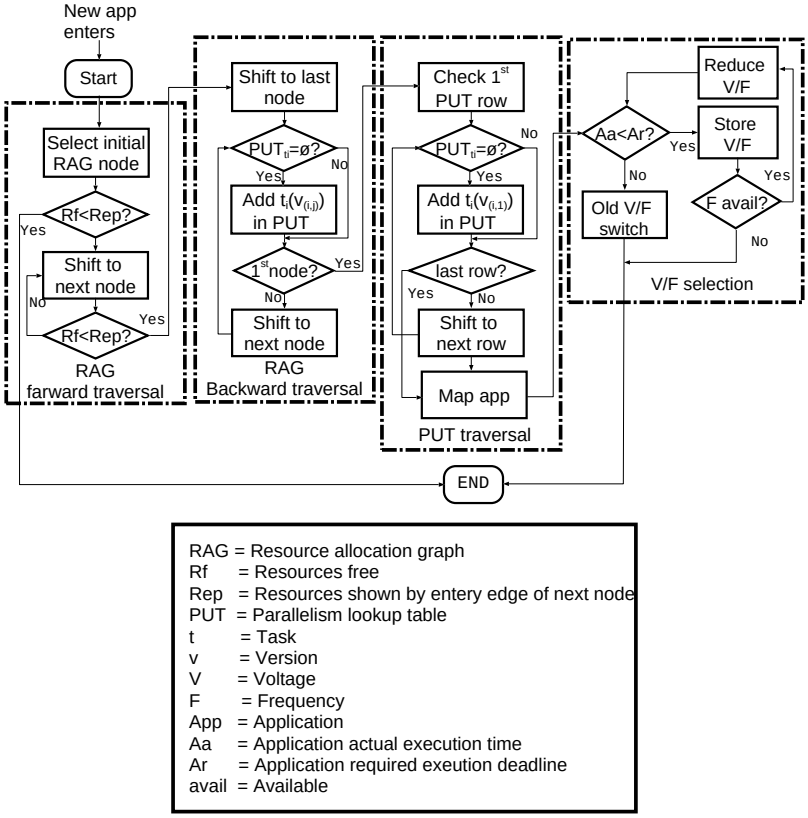

Figure 6.16: Autonomous parallelism, voltage, and frequency selection (APVFS)

6.9 Results

To identify the voltages and their corresponding supported frequencies, DRRA fabric was synthesized. The technology supports voltages from 1.1 V to 1.32 V. The synthesis results revealed that DRRA can run up to a frequency of 1.2 GHz and 1 GHz at 1.32 V and 1.1 V, respectively.

6.9.1 Energy and power reduction

To determine the power and energy consumption, gate level Switching Activity Files (SAIF) were recorded. The power analysis was performed on the synthesized DRRA fabric with the generated SAIF files.

Independent islands

Here, we will show the benefits of combining parallelism with DVFS, for the algorithms that do not communicate with each other (i.e. they do not require DRICs/buffers for synchronizing the clock domains). Later in section 6.9.1, we will show how the application level DVFS using DRICs reduce the overheads, compared to static buffers. We used matrix multiplication, FIR filter, and FFT as representative algorithms, motivated by their extensive use in many DSP applications (like WLAN, image enhancement etc.). To exploit parallelism, matrix multiplication with three versions, serial (ser), partially parallel (parpar), and parallel (par) was used. The benchmarks were experimented with no DVFS, traditional DVFS (TDVFS; i.e. DVFS without parallelism), DVFS with runtime parallelism (PVFS shown in [60, 62]), and the pre-profiled DVFS (PPVFS; presented in Section 6.8.3). Synthetic deadlines were used to analyze whether a shift to different version/voltage/frequency should be made. Initially, maximum frequency (1.2 GHz) and voltage (1.32 V) was assigned to all cells of the fabric. Figures 6.17 and 6.17 show energy and power consumption, after applying no DVFS, TDVFS, PVFS and PPVFS. Since the matrix multiplication had three versions, we have shown it separately, as well, to amplify the effect of dynamic parallelism. It can be seen that by applying PVFS and PPVFS the power and energy consumption of matrix multiplication reduces by 23% and 51%, respectively. The proposed PPVFS iterates quickly to the quickly to the optimal energy and power. Figures 6.17 depicts a scenario when FIR and FFT also enter the platform platform at time instants 9 and 13, respectively. Again the proposed algorithm iterates quickly compared to the TDVFS and PVFS without missing a single deadline.

Communicating islands

To evaluate the energy reductions for algorithms/applications, at different operating points, that communicate with each other, we mapped the WLAN transmitter to DRRA (see [62]). In our experiments, the interleaver and the IFFT had respectively 2 and 5 versions. The actual deadline of WLAN i.e. $4\mu secs$ was used. To quantify the energy/power reductions, promised by our approach, we compared it to three DVFS algorithms: (i) traditional DVFS (TDVFS), (ii) dynamic parallelism voltage and frequency scaling using greedy algorithm with application level DVFS (GPVFS), and (iii) dy-

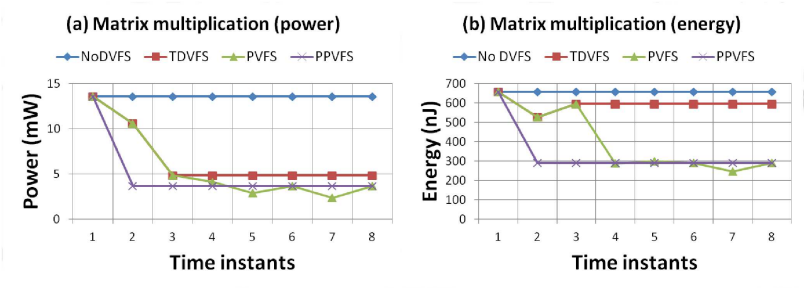

Figure 6.17: Energy and power savings by applying APVFS on matrix multiplication with multiple versions

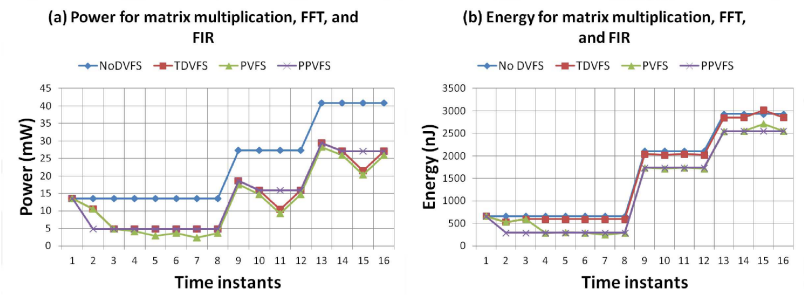

Figure 6.18: Energy and power savings by applying APVFS multiple algorithms

namic parallelism voltage and frequency scaling using greedy algorithm with task level DVFS (TPVFS). The results are shown in Figure 6.19. The figure shows power and energy consumed with different number of free resources. For 13 resources, all the algorithms show similar behavior, since none of the application tasks can be parallelized. When 17 resources are available, the interleaver can be parallelized. Both GPVFS and TPVFS parallelize the interleaver. Unfortunately, TPVFS, increases both the power and energy consumption as a result of additional buffers needed for synchronizing different frequency islands. GPVFS is unable to perform any voltage or frequency scaling, since it would violate $4\mu sec$ deadline of WLAN. APVFS leaves the extra resources free. These free resources can can be powered off to reduce static power/energy. For 19 resources, APVFS, parallelizes the FFT (which was actually the bottleneck in application performance), providing reduction in power and energy since both the voltage and frequency can be scaled at this point. GPVFS is unable to utilize these resources, since it would parallelize the Interleaver first. At this point, it can be seen that APVFS saves 28% power and 36% energy compared to GPVFS. It should be noted that if more resources are available, GPVFS will continue to assign resources till all the 5 versions are exhausted.

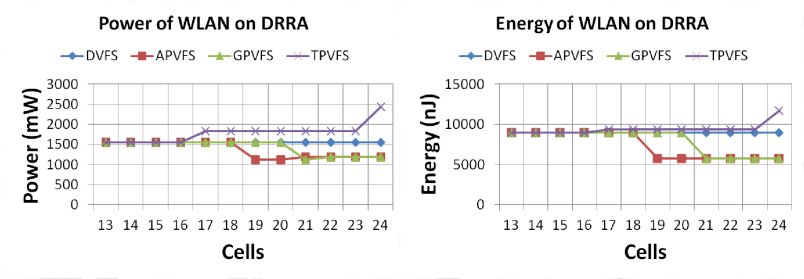

Figure 6.19: Energy and power consumption of WLAN on DRRA

Resource utilization

To evaluate the resource utilization, promised by our technique compared to the greedy approach, both the techniques were simulated. For simulations, MPEG4 [66] and WLAN [62] were used. Figure 6.20 shows the resources allocated to the applications and their corresponding throughputs. The figure clearly illustrates that while energy aware task parallelism allocates a resource(s), only if it promises a speedup, the greedy approach suffers from unproductive resource allocations for both the applications. For MPEG4, the greedy approach makes unproductive allocations when 16, 18, 20 and 22 resources are free. For WLAN, an unproductive allocations is made for 15 free resources, and the effect ripples till 21 free resources are available. It is is due to these unproductive resource allocations that the greedy approach

consumes excessive energy/power (as seen in Section 6.9.1).

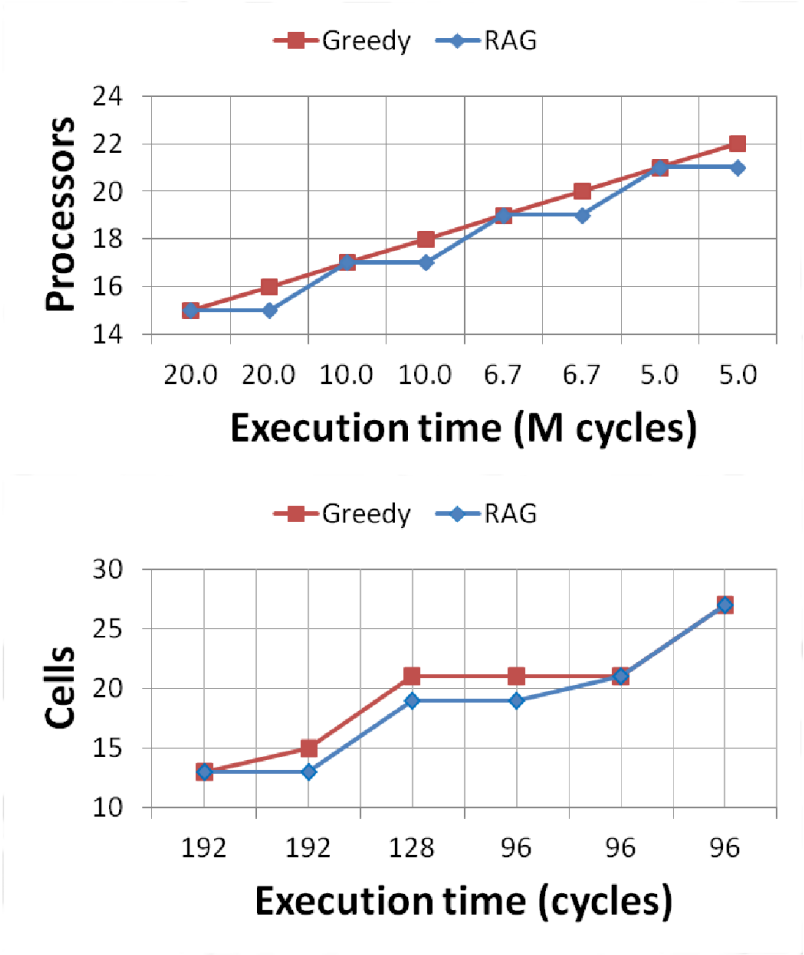

Figure 6.20: Resources required for speedup RAG vs greedy approach

Reduction in configuration memory requirements

Finally, our method promises significant savings in configuration memory requirements compared to state of the art compression method proposed in [57]. This method, called Compact Generic Intermediate Representation (CGIR), compresses data by storing configware for only a single version. The rest of the versions are stored as differences from the original version. Remember, from Section 6.7, the RAG isolates the versions which actually reduce power/energy. Therefore, all the redundant versions are discarded. As a result, APVFS promises considerable configuration memory savings. The proposed method promises significant (up to 36 %) memory savings compared to state of the art for implementing IFFT in WLAN. Figure 6.21

clearly illustrates the trend that as the number of stored versions increase, our method promises a higher compression compared to CGIR.

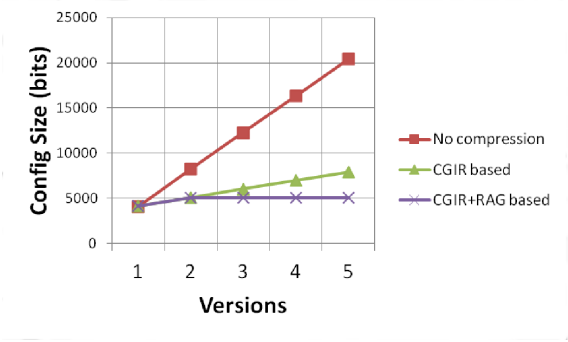

Figure 6.21: Compression achieved by using RAG based DVFS

6.9.2 Overhead analysis

DRIC overheads

To analyze the benefits (in terms of area) of the proposed method (ISDVFS), employing DRICs, compared to the traditional DVFS (TDVFS), we synthesized different versions of DRRA. For each version different number of cells and islands were chosen. The synthesis results are illustrated in Figure 6.22. In the figure, ISDVFS_1, ISDVFS_2, ISDVFS_3, and ISDVFS_4 refer to a fabric with 1, 2, 3, and 4 DRICs, respectively. It is seen that while overheads of TDVFS increase linearly with size of fabric, the cell addition has negligible effect on ISDVFS. The overheads for ISDVFS are more dependent on the number of communicating islands. The figure reveals that ISDVFS incurs lesser overhead provided only a single DRIC is employed for 10 or more cells. Since even simple applications like WLAN transmitter (with serial IFFT) require 16 cells, for most real world applications our approach promises significant reductions in area overheads. The proposed approach incurs additional timing overhead, when a DRIC is initially mapped to the device. This overhead is dependent on the size of DRIC configware, generated by RTM. For 15 frequency levels, used in this thesis, the maximum size of DRIC configware can be 15 words and would require 15 cycles to be mapped. This overhead is a negligible compared to overall application execution time. DRIC generation itself does not require any overhead since it occurs transparently in background.

APVFS overheads

The proposed approach incurs additional timing overhead during forward, backward, and parallelism lookup table traversals. However, the traversals

are done in the background, while the application is running. The application has to stall for only $A_s = SW * LC$ secs. Where SW and LC denote the number of words in configware and the time required for loading a word, respectively. At $400MHz$, the reconfiguration of the serial and partially parallel versions of WLAN require $A_s = 9\mu sec$, and $A_s = 10\mu sec$, respectively (i.e. only 3 frames will be lost during reconfiguration). The memory overhead of storing RAG is $M_{RAG} = N_{bit} * (\sum mainnodes + \sum subnodes + \sum enteryedges)$ bits, where N_{bit}, represents the bits required to store a node. For WLAN, considering $N_{bit} = 32bits$, $M_{RAG} = 576bits$. Since this overhead is only 14% of the reductions in configuration memory (shown in section Section 6.9.1), overall, APVFS requires less memory than previously presented approaches. For the GRLS clocking strategy, used in DRRA, the voltage switching time is approximated to be $20ns$, while a frequency can be changed in a single cycle.

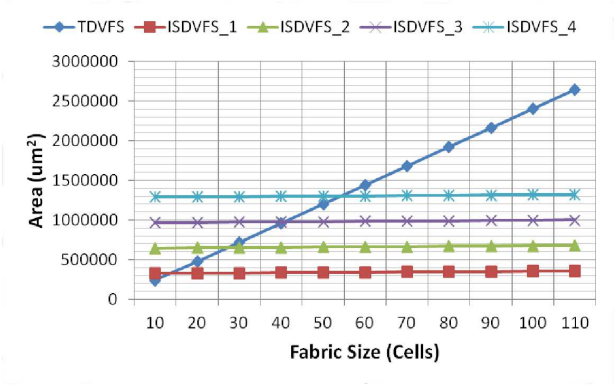

Figure 6.22: Area comparison ISDVFS vs TDVFS

6.10 Summary

In this chapter, we have presented architecture and implementation of energy aware CGRAs. The proposed architecture promises better area and power efficiency, by employing DRICs and APVFS. The DRICs utilize reconfiguration to eliminate the need for most of the dedicated hardware, required for synchronization, in traditional DVFS techniques. APVFS ensures high energy efficiency by dynamically selecting the application version which requires the minimum frequency/voltage to meet the deadline on available resources. Simulation results using representative applications (Matrix multiplication, FIR, and FFT) showed up to 23% and 51% reduction in power and energy, respectively, compared to traditional designs. Synthesis results have confirmed significant reduction in DVFS overheads compared to state of the art DVFS methods. Future research on energy-aware architectures

will involve investigating the effect APVFS has on energy-aware mapping.

Chapter 7

Private operating environment for NoCs

7.1 INTRODUCTION

In this chapter we will explain how the power management intelligence was integrated to the McNoC platform. As already explained, McNoC already hosted an architecture to support multi VDD/multi frequency partitions of NoC. The architecture uses GRLS principles to ensure that validity when crossing clock domains. In additions the architecture allowed to use simple commands like *change_DVFS* to scale the voltage and/or frequency. Unlike the private operating environments for the CGRAs, where the intra application communication patterns are predictable, in packet switched NoCs both inter and intra application communication patterns are unpredictable. Therefore, instead of profiling, to achieve autonomous adaptivity we decided to integrate a feedback loop in existing McNoC architecture. The proposed feedback loop monitors the traffic loads at runtime and based on the loads autonomously find the optimal voltage and frequency that meets the application deadlines (for each switch).

This chapter presents the essential architectural support to enable the automation of power management services. The need for scalability has dictated the use of a hierarchical agent monitored NoC. The proposed architecture contains several levels of controllers, called agents (see Chapter 5), with hierarchical scope and priorities, to provide both coarse and fine-granular observability and reconfigurability. Conceptually, agents are monitoring and reconfiguration functions, which can be realized as software, hardware or a hybrid of both. The conventional NoC platform (consisting of data, memory and communication) is considered as a resource supervised by the agents. As explained in previously in Chapter 5, the hierarchical monitoring services are performed by three types of agents: (i) system agent, (ii) cluster

agent, and (iii) cell agent. The system agent, which determines the adaptive policy for the whole NoC. It is implemented in software agent, with specific instructions designed monitor and reconfigure the NoCs. The cluster agents are only used for the fault tolerance services already mentioned in Chapter 5. The cell agents monitor (e.g. traffic loads) and reconfigure the local resources (e.g. voltage and frequency) based on the command from the system agent. The communication between agents are implemented on existing NoC channels. The agents are fully integrated in a RTL-level cycle-accurate NoC simulator with LEON3 processing elements and distributed shared memory.

To test the efficacy of our solution, we have used best-effort per-core DVFS (dynamic voltage and frequency scaling), as a representative algorithm. The architecture and the algorithm were tested on a few applications (matrix multiplication, FFT, wavefront, and hiperLAN transmitter). The software and hardware overheads were evaluated to show the scalability of the system architecture.

7.2 RELATED WORK

The coming dark silicon era has made DVFS a subject of intensive research. Existing works focus previous on a specific algorithms or monitoring to determine the optimal power. Unfortunately, works that deal with systematic approach for generic monitoring and reconfiguration architecture that allows integration of various services (e.g. fault tolerance and DVFS) are fairly limited.

Ciordas [28] proposed a monitoring-aware system architecture and design flow for NoC. This work focused on hardware-based probes for transaction debugging and QoS (Quality-of-Service) provision. Our work, however, presents a SW/HW (software/hardware) co-design approach to the monitoring and reconfiguration, with services for non-functional design goals, such as power and energy consumption. Sylvester [116] presented an adaptive system architecture, ElastIC, for self-healing many-core SoC. Where each core was designed with observable and tunable parameters, for instance power monitors. A centralized DAP (diagnostic and adaptivity processing) unit dynamically was employed to test and reconfigure the cores with degraded performance. However, [116] does not explore the architectural support. A two-level controlling architecture was presented by Dafali [32]. They used a centralized configuration manager to determines the management policies (of the whole network), while each local manager performed reconfiguration based on the management policies. However, this work only focused on the design of self-adaptive network interface, without the system level discussion of power efficiency or dependability. Liang et al. [42] proposed a

functional overview of hierarchical agent monitoring design paradigm. This work presented an instruction-level architectural design and implementation specifically for NoC platforms. However, they only focused on general principles to realize functional partition. Hoffmann [50] presented a so-called "heartbeat framework". This approach presented a way for applications to monitor their performance and make that information available to external observers. The progression of the application is symbolized as a heartbeat. By monitoring the intervals between heartbeats, the platform observer and the application can be aware of the system performance. We integrate this application labeling approach into our system architecture, where the system agent monitors the application execution time by checking the labeled timestamps.

Compared to these existing works, we made following major contributions:

- We presented an scalable hardware architecture to provide monitoring and reconfiguration services using hierarchical agents.

- We presented an instruction-level architectural design that enables the system architecture to be integrated into NoC design flow.

7.3 ARCHITECTURAL DESIGN

The functions of system and cell agents were implemented as software instructions and hardware components, respectively. For the software based system agents we designed various instructions to monitor and reconfigure. For hardware based cell agents, we designed an interface with the software and the the necessary primitives to control the resources under the directives of system agent.

7.3.1 Application Timestamps

To allow applications monitoring, meta-data was added in the instructions (e.g. to denote the progression of the application). Fig. 7.1 depicts an example of adding timestamps in the applications. In particular, the starting and finishing time of the application and the critical sections are labeled with special instructions, so that the occurrence of these events can be monitored by the system agent.

These timestamps labeling instructions are implemented as memory write instructions. Specific data is written to a memory location of the system agent, to notify the occurrence of the event. The allocation of the memory address is performed during compilation.

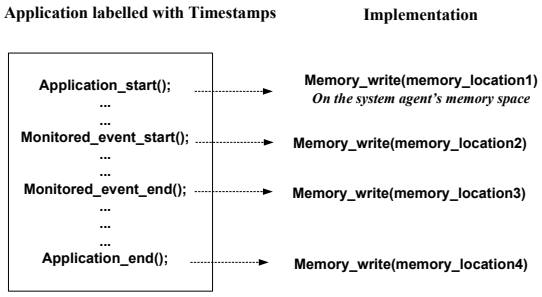

Figure 7.1: Labeling Timestamps in the Application

7.3.2 System Agent

The system agent works as the "general manager" for monitoring and reconfiguration services. Depending on the design requirement, the system agent performs operations like task mapping, process scheduling, run-time power management and fault tolerance. The need to perform these diverse operations has motivated us to implement the system agent is implemented as a dedicated processor in NoC, so that the agent functions can be reloaded dynamically. For smaller projects, the system agent can be implemented as a separate thread. The system agent monitors the application progress and the system parameters, and reconfigures the system according to an adaptive algorithms. To accomplish this, the system agent first checks the start of the application (or a frame in streaming applications), which is implemented as a blocking memory read. The application will label the timestamps when it starts (Section 7.3.1). To monitor a certain parameter after the application starts, the system agent first issues a command to check the run-time value of the parameter. The command is written to the memory location of the intended network node, so that the corresponding cell agent will receive the command. Similarly, the system agent issues a number of parameter-checking commands, implemented as non-blocking memory writes. To make the reconfiguration decisions, the system agent waits on the report of the monitored parameters by the corresponding cell agents (as memory writes; Section 7.3.3). These waiting operations are implemented as blocking reads. When a read completes, the system agent performs reconfiguration based on the run-time parameter values. The waiting of multiple parameters are parallel processes, since the parameters may be returned in random orders. When all required monitoring and reconfiguration operations are finished, the system agent waits for the completion of the application. However, in case one monitored parameter is the execution time of an application frame, the monitoring operation may be finished after the frame ends.

Table 7.1 lists the detailed C instructions (on a Leon 3 processor) on the system agent to implement monitoring and power management.

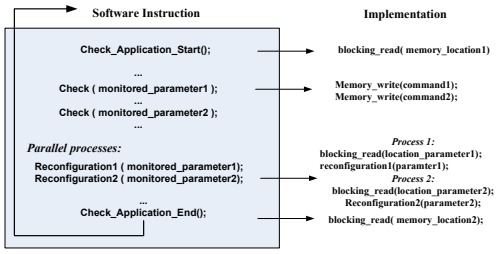

Figure 7.2: Monitoring and Reconfiguration Software on System Agent

Table 7.1: Experimented Instructions for Monitoring and Power Management on System Agent (a LEON3 processor)

Instruction	Function
wait(memory_location)	Wait for the occurrence of an event (the application writes the corresponding memory location)
get_load(row, column, switch)	Check the run-time workload of a particular switch
reset_load(row, column, switch)	Refresh the workload record of a particular switch
set_window(row, column, switch, windowsize)	Set the monitoring window
set_priority(row, column, switch, priority)	Set the priority of agent command in the network arbitration
DVFS_change(memory_location, clk_sel, vol_sel)	Change the voltage and frequency of a particular switch (denoted by the memory location)

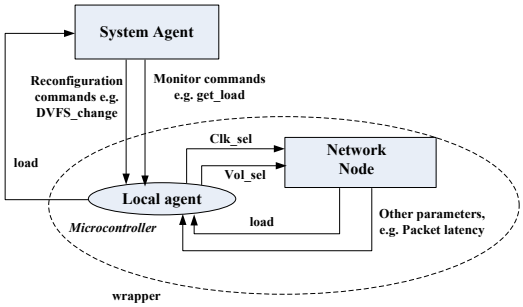

Figure 7.3: Schematics of cell Agent and its Interfaces to System Agent and Network Node

7.3.3 Cell Agents

Cell agents are distributed hardware entities embedded in each network node. They receive commands from the system agent to activate the monitoring and reconfiguration operations. Each cell agent, after receiving the monitoring commands from the system agent, reads the required parameters from the local resource (Fig. 7.3). Similarly, when receiving a reconfiguration command, it actuates the reconfiguration, for instance by setting the power switch and frequency generator. The interfaces to various parameters for monitoring and reconfiguration are hardwired, so that the network node can be used as a modularized component integrable into any NoC systems.

7.3.4 Architectural Integration

The agent intelligence layer is the architectural integration of the system agent and the distributed cell agents, with time-stamp-labeled application (Fig. 7.4) .

The application programmers specify the timestamps of monitored events in the application, for instance the starting/end times of each frame. The system designers write software instructions for monitoring and reconfiguration operations with high-level abstraction. These operations are sent to and implemented by cell agents, which are hardware entities present in each network node. The wrapping of the cell agent and the resource is design specific. For instance, if parameters from both the processing element and the router are needed for the monitoring and reconfiguration, the cell agent is attached to the whole node. Since the monitoring and reconfiguration are infrequently issued compared to data communication [29], we can reuse the existing NoC interconnect for inter-agent communication.

Due to the SW/HW co-design and modularized architectural integration, the agent intelligence layer is highly scalable. The cell agent wrapper can be applied to any NoC node (or a particular NoC component, e.g. router),

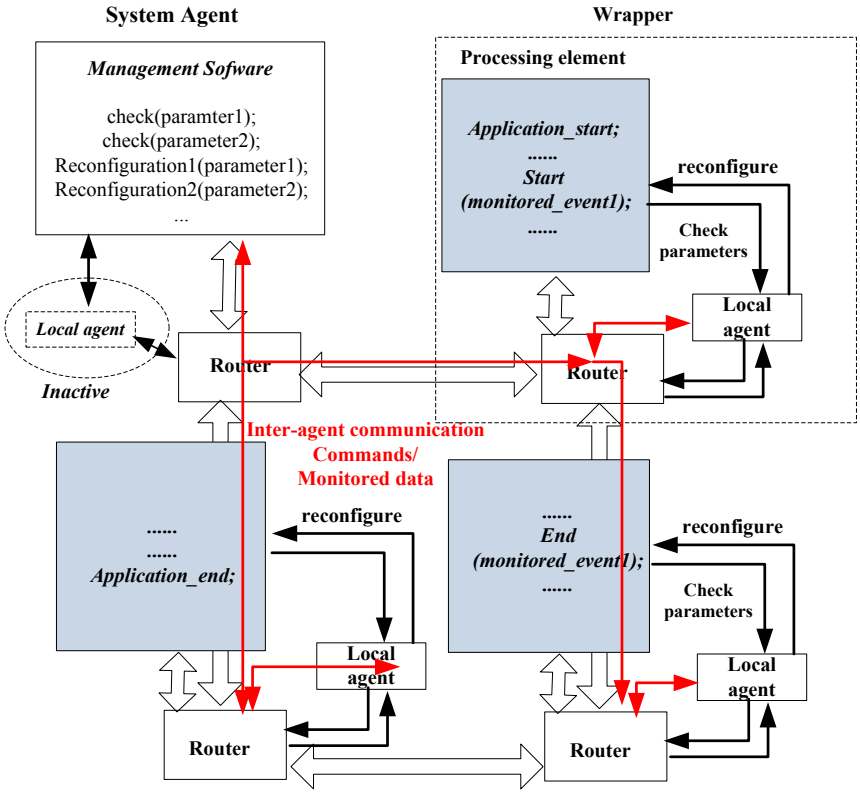

Figure 7.4: Integrating Hierarchical Agents as an Intelligence Layer

and be used as a building block to construct a NoC of arbitrary size without incurring additional overhead. The software-based system agent, on the other hand, can be written with various monitoring and reconfiguration instructions as needed for the application.

7.4 SELF-ADAPTIVE POWER MANAGEMENT

To demonstrate the effectiveness and overheads of using dual-level agents, we have used best-effort per-core DVFS on the existing NoC platform. Based on the specified parameters (e.g. peak load and average load), the cell agents trace run-time system information. Upon the request of the system agent, they return the recorded values. Depending on the provided information and the application performance constraints, the system agent adjusts the voltage and/or frequency to optimize the power and energy consumption.

7.4.1 Best-effort Per-Core DVFS (BEPCD)

The adaptive power management using distributed DVFS with run-time application performance monitoring, abbreviated as BEPCD, is illustrated in Fig. 7.5. *P*, *S*, *LT*, *F* and *Ts* represent processor, switch, *low traffic switches* (the switch with the lowest workload), switch frequency and *threshold time* (the application latency), respectively. The terms inside parenthesis represent the function to be performed on the entity to the left (e.g. *P(any) starts?* means if any of the processors starts). Simply put, the process is performed in three steps: (i) the initialization of voltage and frequency of each switch and the setting of application latency requirement, (ii) run-time tracing of the workload of each switch and the application latency (Section 7.3.2), (iii) if the latency is lower than the constraint, DVFS is applied to the switch with the lowest workload.

7.4.2 Experiment Setup

To identify the voltages and their corresponding supported frequencies, the switches were synthesized (Table 7.2). The technology supports voltages from 1.1 V to 1.32 V. The synthesis results reveal that the routers are capable of supporting up to 300 MHz frequency at 1.32 V and up to 200 MHz frequency at 1.1 V. Based on GRLS clocking in the NoC platform, the allowable frequencies are 300, 200, 100, 50, 40, and 20 MHz (exact divisors of $F_H = 600MHz$, least common multiplier of 300MHz and 200MHz).

Four applications (matrix multiplication, FFT, wavefront, and hiperLAN transmitter) are mapped on a 3x3 mesh-based NoC. The absence of DSPs in existing NoC platform prevents us from meeting the deadline (4 μs/frame) of hiperLAN transmitter. Thus we set the deadline as the minimal latency of

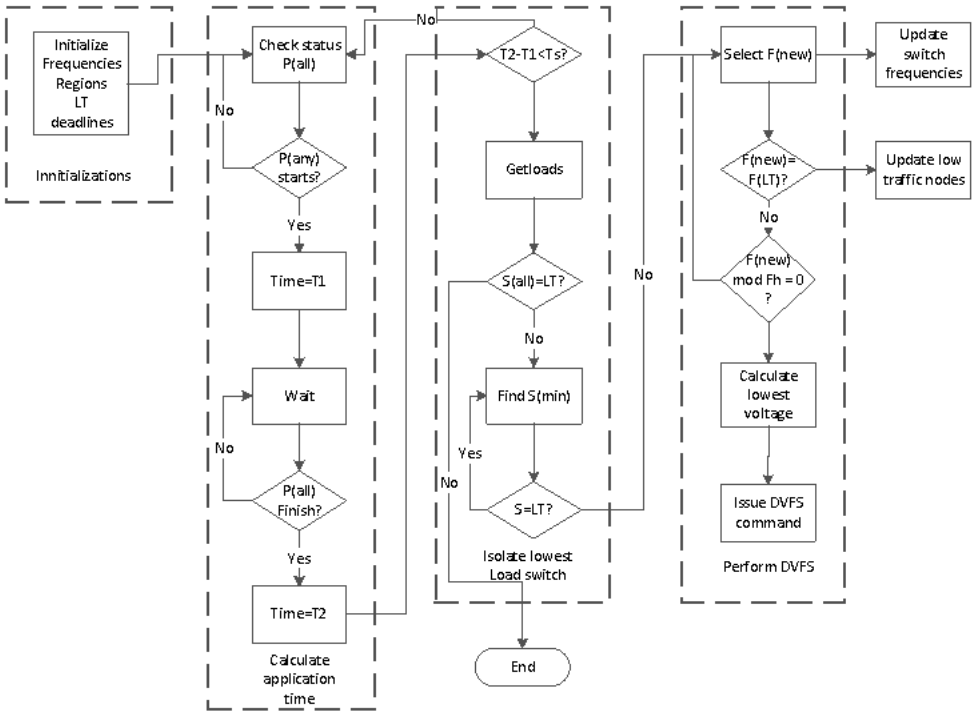

Figure 7.5: Per-Core DVFS for Best-effort Power Management with Runtime Performance Monitoring

Table 7.2: Voltage frequency pairs

Voltage (V)	Frequency (MHz)	Timing constraints
1.32	400	violated
1.32	**300**	**met**
1.32	200	**met**
1.1	400	violated
1.1	300	violated
1.1	**200**	**met**
1.1	**100**	**met**
1.1	**50**	**met**
1.1	**40**	**met**
1.1	**20**	**met**

the application on the NoC platform (39 μs) , when all routers are configured with the highest frequency.

To analyze the power and energy consumption, the switching activity files are generated for each application from Cadence NCSim. The power analysis is performed by Synopsys design compiler on the synthesized NoC routers with the generated switching activity files.

7.4.3 Experiment Result

Four benchmarks (matrix multiplication, FFT, wavefront, and hiperLAN) were experimented with BEPCD algorithm. Initially, the system agent assigned max frequency (300 MHz) and voltage (1.32 V) to all switches. At each iteration, the application execution time was monitored and if it did not violate the timing deadline, the next lower voltage/frequency pair from Table 7.2 was assigned to the lowest traffic switch (in terms of peak load in a time window of 40 cycles).

Tables 7.3, 7.4, 7.6, and 7.5 show the energy and power savings of each of the four benchmarks. In the tables, the second column shows the switch number which changes its voltage/frequency followed by *"f"* or *"vf"*. "f" indicates a frequency change, while "vf" shows that both the voltage and frequency change.

The power and energy trends for each of the four applications are clearly depicted in Figure 7.6. It is seen that as a consequence of BEPCD, the NoC quickly iterates towards the minimum power for each of the application. If the targeted switch is present in the critical path, as expected, the application execution time (AET) increases with a decrease in voltage/frequency (iteration 3 to 6 and 7 to 9 in Table 7.3, iteration 2 and 4 in Table 7.6). The AET remains unaffected if the switch does not come in the critical path (Table 7.5, iteration 6 to 13 Table 7.6). In some situations, the memory contention is reduced with voltage/frequency decrease, then AET may also decrease (iteration 7 and 10 Table 7.3, iteration 7 and 10 in table 7.4, and iteration 3 in Table 7.6).

The BEPCD performs iterations only till the application meets deadline. To cater for the sudden changes in time (iteration 6 in Table 7.4) resulting from massive memory contention (iteration 6 Table 7.4), the algorithm performs an additional iteration to check if a further reduction in frequency would reduce time. If no reduction is encountered, switch is reverted to original frequency and no further DVFS commands are given. The plots confirm clearly significant advantages of our proposed strategy (from 21% to 33% decrease in energy and from 21% to 36% decrease in power consumption).

Table 7.3: Energy and power savings for matrix multiplication

Iteration	Switch	Time (ns)	Energy (mJ)	Power mW	Energy saving %	Power savings %
1	-	105834	1.73	16.35	0	0
2	1vf	105834	1.67	15.84	3.11	3.11
3	3vf	106808	1.26	11.84	26.90	27.56
4	3f	107415	1.21	11.31	29.78	30.82
5	3f	112134	1.25	11.20	27.39	31.46
6	3f	116373	1.27	10.99	26.07	32.76
7	1f	101815	1.11	10.97	35.46	32.91
8	2vf	108774	1.92	16.96	31.11	32.97
9	2f	113100	1.17	10.41	31.97	36.34
10	2f	111134	1.15	10.38	33.32	36.50
11	2f	111467	1.57	10.38	33.12	36.53

Table 7.4: Energy and power savings for FFT

Iteration	Switch	Time (ns)	Energy (mJ)	Power mW	Energy saving %	Power savings %
1	-	381615	17.40	45.61	0	0
2	3vf	381615	15.87	41.60	8.78	8.78
3	3f	381615	15.67	41.07	9.95	9.95
4	3f	381616	14.10	36.96	18.95	18.95
5	1vf	377320	13.66	36.21	21.49	20.59
6	1f	430525	15.54	36.11	10.68	20.83
7	1f	381616	16.69	35.89	21.29	21.29
8	2vf	381616	13.69	35.89	21.29	21.29
9	2f	381154	13.68	35.89	21.39	21.29
10	2f	376549	12.01	31.89	30.99	30.06

Table 7.5: Energy and power savings for HiperLAN

Iteration	Switch	Time (ns)	Energy (mJ)	Power mW	Energy saving %	Power savings %
1	-	39000	1.77	45.61	0	0
2	1vf	39000	1.62	41.60	8.78	8.78
3	3vf	39000	1.60	41.07	9.95	9.95
4	3f	39000	1.44	37.06	18.73	18.73
5	3f	39000	1.42	36.54	19.88	19.88
6	3f	39000	1.42	36.42	20.13	20.13
7	1f	39000	1.41	36.21	20.59	20.59
8	-	39000	1.40	35.90	21.29	21.29
9	-	39000	1.40	35.90	21.29	21.29
10	-	39000	1.40	35.90	21.29	21.29
11	-	39000	1.40	35.90	21.29	21.29
12	-	39000	1.40	35.90	21.29	21.29

Table 7.6: Energy and power savings for wavefront

Iteration	Switch	Time (ns)	Energy (mJ)	Power mW	Energy saving %	Power savings %
1	-	91970	1.51	16.50	0	0
2	3vf	110234	1.37	12.50	9.15	31.94
3	3f	106529	1.28	12.03	15.51	37.09
4	3f	110294	1.32	11.97	12.96	37.79
5	-	110294	1.32	11.97	12.96	37.79
6	-	110294	1.32	11.97	12.96	37.79
7	-	110294	1.32	11.97	12.96	37.79
8	-	110294	1.32	11.97	12.96	37.79
9	-	110294	1.32	11.97	12.96	37.79
10	-	110294	1.32	11.97	12.96	37.79

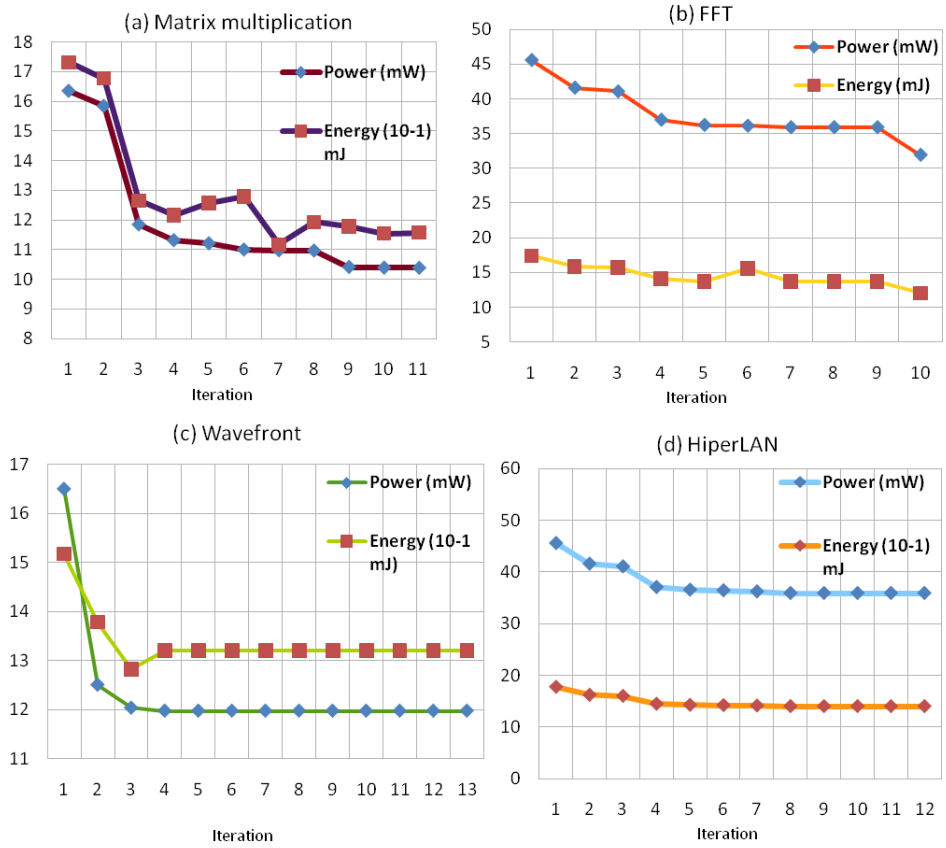

Figure 7.6: Energy and power comparison for (a) matrix multiplication, (b) FFT, (c) wavefront, and (d) hiperLAN

7.4.4 Overhead Analysis

To evaluate the overhead of the dual-level agent intelligence layer, we need to analyze the area overhead of microcontroller-based cell agent (Fig. 7.3) and the instruction overhead of software-based system agent (Fig. 7.2).

At 300 MHz frequency with 1.32 V operating voltage, Synopsys design compiler shows an area of 1459 μm^2 for each cell agent, which is negligible (4 %) as compared to the router area (33806 μm^2). The cell agent does not contribute to any timing overhead as it is not present in the critical path of the switch. Concerning the software overhead of the system agent, it only amounts to 279 lines of C code on Leon 3 processor for the BEPCD algorithm.

We can see from the overhead analysis that, dual-level agent monitoring incurs minimal hardware area overhead and software instruction overhead. Thus the system architecture is scalable to large-sized NoCs with a diversity of monitoring and reconfiguration functions.

7.5 Summary

In this chapter, we have presented the design and implementation of a generic and scalable self-adaptive NoC architecture. The system is monitored and reconfigured by dual-level agents with SW/HW co-design and synthesis. The system agent is implemented in software, with high-level instructions tailored for issuing adaptive operations. The cell agent is attached to each network node and implemented as a microcontroller. The cell agent provides tracing and reconfiguration of the local circuit parameters, based on the run-time adaptation commands from the system agent. The dual-level agents make a joint effort to achieve the performance goals of the application, where the monitored events are labeled with timestamps. The separation of the intelligence layer from NoC infrastructure makes the approach generic and improves the design efficiency. The SW/HW co-design and synthesis effectively reduces the hardware overhead while offering flexibility for adaptive operations.

We demonstrated the effectiveness and the scalability of the system architecture with best-effort dynamic power management using distributed DVFS. In this case study, the application execution time and the run-time workloads of all routers are directly monitored by the agents. The router with the lowest workload will be switched to a lower voltage and/or frequency when there is a positive slack of application latency (per frame/stream). The experiments were performed with four benchmarks (matrix multiplication, FFT, wavefront, and hiperLAN transmitter), on a cycle-accurate RTL-level NoC simulator. We showed that the adaptive power management saves up to 33% energy and up to 36% power. The hardware overhead of each

cell agent is only 4% of a router area.

In the future work, we will present a complete design chain for the system architecture, including application mapping, scheduling followed by run-time monitoring and reconfiguration. The inter-agent communication shall also be provided with guaranteed services.

Chapter 8

Conclusion

In this chapter, the concluding remarks of all the preceding chapters will be presented to consistently depict the overall picture of the achievements. In addition, a few remaining open problems and directions for future research will also be presented.

8.1 Contributions

The main contribution of this thesis was to present a framework for creating dynamic heterogeneity in CGRAs and NoCs. The dynamic heterogeneity allowed to adapt the platform resources, depending on the application, needs at runtime. Thereby, the proposed framework addressed the emerging design issues like dark silicon and fault tolerance. In particular, we dynamically manipulated the voltage, frequency, reliability, and configuration architecture to optimize the area and power consumption. To systematically tackle this problem we divided the VRAP framework into three parts: (i) Private Configuration Environments (PCE), (ii) Private Reliability Environment (PRE), and (iii) Private Operating Environments (POE). To provide concrete results, PRE and POE were evaluated on both NoCs and CGRAs, while PCEs were analyzed analyzed only on CGRAs.

PCE provided on demand configuration infrastructure by employing a morphable data/configuration memory controlled by a hierarchical controllers. By configuring the memory, four configuration modes, with different memory requirements and reconfiguration time, were realized: (i) direct feed, (ii) direct feed multi-cast, (iii) direct feed distributed, and (iv) multi context. The obtained results suggest that significant reduction in configuration memory requirements (up to 58 %) can be achieved by selecting the most appropriate mode. Synthesis results revealed that the PCE incurred negligible penalty (3 % area and 4 % power) compared to a DRRA cell.

PRE was designed to provide on demand reliability to each applica-

tion, at runtime, for CGRAs and NoCs. To implement on-demand fault-tolerance for CGRAs, the reliability requirements of an application were assessed upon its entry. Depending on the assessed requirements, one of the five fault-tolerance levels was provided: (i) no fault-tolerance, (ii) temporary fault detection, (iii) temporary/permanent fault detection, (iv) temporary fault detection and correction, or (v) temporary/permanent fault detection and correction. In addition to modular redundancy (employed in the state-of-the-art CGRAs offering flexible reliability levels), this thesis presented the architectural enhancements needed to realize sub-modular (residue mod 3) redundancy. The residue mod 3 codes allowed to reduce the overhead of the self-checking and fault-tolerant versions by 57% and 7%, respectively. To shift autonomously between different fault-tolerance levels, at run-time, a fault-tolerance agent was introduced for each element. This agent was responsible for reconfiguring the fault-tolerance infrastructure upon arrival of a new application or changing external conditions. The polymorphic fault-tolerant architecture was complemented by a morphable scrubbing technique to prevent fault accumulation. The obtained results suggest that the on-demand fault-tolerance can reduce energy consumption up to 107%, compared to the highest degree of available fault-tolerance (for an application actually needing no fault-tolerance). For NoCs, this thesis presented an adaptive fault tolerance mechanism, capable of providing the on-demand protection to multiple traffic classes. On-demand fault tolerance was attained by passing each packet through a two layer, low cost, class identification circuitry. Upon identification, the packet was provided one of the four fault tolerance levels: (i) no fault tolerance, (ii) end to end DEDSEC, (iii) per hop DEDSEC, or (iv) per hop DEDSEC with permanent fault detection and recovery. The results suggest that the on-demand fault tolerance incurs a negligible penalty in terms of area (up to 5.3%) compared to the fault tolerance circuitry, and premisses a significant reduction in energy (up to 95%) by providing protection only to the control traffic.

Private operating environments was provided for both CGRA and NoC. In CGRA domain, this thesis presented architecture and implementation of energy aware CGRAs. The proposed architecture promised better area and power efficiency, by employing Dynamically Reconfigurable Isolation Cells (DRIC)s and Autonomous Parallelism Voltage and Frequency Selection algorithm (APVFS). Simulation results using representative applications (Matrix multiplication, FIR, and FFT) showed up to 23% and 51% reduction in power and energy, respectively, compared to traditional designs. Synthesis results have confirmed significant reduction in DVFS overheads compared to state of the art DVFS methods. In NoC domain, this thesis presented the design and implementation of a generic agent-based scalable self-adaptive NoC architecture to reduce power. The system employed dual-level agents with SW/HW co-design and synthesis. The system agent

was implemented in software, with high-level instructions tailored to issue adaptive operations. The effectiveness and the scalability of the system architecture was demonstrated using best-effort dynamic power management using distributed DVFS. The experiments revealed that the adaptive power management saved up to 33 % energy and up to 36 % power. The hardware overhead of each local agent is only 4 % of a router area.

8.2 Future work

The future work can take two main directions: (i) additional private environments can be realized and (ii) design time environment generation to complement the runtime VRAP framework. In this thesis, we have focused on PCE, PRE, and POE. The VRAP framework can be extended to integrate new environments, for upcoming technology trends. In particular, we envision Private Thermal Environments (PTEs) and Private Compression Environments (PComEs) would be useful. The new environments would adapt the system resources to optimize respectively the device temperature and the compression hierarchy. VRAP is useful for mixed criticality applications. Where criticality can be in terms of reliability, performance, or reconfiguration overheads. If it is known that a particular platform will host applications varying only in specific type of criticality (reliability, performance, or reconfiguration requirements), incorporation of all the private environments will be redundant. For such conditions, a design time environment generator can be developed to avoid the needless redundancies.

Bibliography

[1] X. Chen, Z. Lu, A. Jantsch, S. Chen. Run-time partitioning of hybrid distributed shared memory on multi-core network-on-chips. In *3rd International Symposium on Parallel Architectures, Algorithms and Programming*, PAAP '10, pages 39–46, Washington, DC, USA, 2010. IEEE Computer Society.

[2] Waqar Ahmed. *Core Switching Noise for On-Chip 3D Power Distribution Networks Doctoral Thesis in Sweden, 2012.* PhD thesis, Royal Institute of Technology (KTH), 2012.

[3] R. Airoldi, F. Garzia, and J. Nurmi. Improving reconfigurable hardware energy efficiency and robustness via DVFS-scaled homogeneous MP-SoC. In *Proc. IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 286 –289, May 2011.

[4] M.A. Al Faruque, R. Krist, and J. Henkel. Adam: Run-time agent-based distributed application mapping for on-chip communication. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 760–765, June 2008.

[5] D. Alnajiar, Younghun Ko, T. Imagawa, H. Konoura, M. Hiromoto, Y. Mitsuyama, M. Hashimoto, H. Ochi, and T. Onoye. Coarse-grained dynamically reconfigurable architecture with flexible reliability. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 186–192, 2009.

[6] D. Alnajjar, H. Konoura, Y. Ko, Y. Mitsuyama, M. Hashimoto, and T. Onoye. Implementing flexible reliability in a coarse-grained reconfigurable architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,*, PP(99):1–1, 2012.

[7] H. Amano, Y. Hasegawa, S. Tsutsumi, T. Nakamura, T. Nishimura, V. Tanbunheng, A. Parimala, T. Sano, and M. Kato. MuCCRA chips: Configurable dynamically-reconfigurable processors. In *Proc. IEEE Asian Solid-State Circuits Conference (ASSCC)*, pages 384–387, 2007.

[8] Hideharu Amano, Masayuki Kimura, and Nobuaki Ozaki. Removing context memory from a multi-context dynamically reconfigurable processor. In *Proc. IEEE International Symposium on Embedded Multicore Socs (MCSoC)*, pages 92 –99, Sept. 2012.

[9] Muhammad Moazam Azeem, Stanislaw J. Piestrak, Olivier Sentieys, and Sébastien Pillement. Error recovery technique for coarse-grained reconfigurable architectures. In *Proc. IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 441–446, 2011.

[10] N. Banerjee, C. Augustine, and K. Roy. Fault-tolerance with graceful degradation in quality: A design methodology and its application to digital signal processing systems. In *IEEE Int. Symp. Defect and Fault Tolerance of VLSI Systems (DFTVS)*, pages 323–331, 2008.

[11] Jürgen Becker and Reiner Hartenstein. Configware and morphware going mainstream. *Journal of Systems Architecture*, 49(4?6):127 – 142, 2003.

[12] M. Berg. The NASA Goddard space flight center radiation effects and analysis group Virtex 4 scrubber. Annu. Xilinx Radiation Test Consortium (XRTC) Meeting, 2007.

[13] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K.A. LaBel, M. Friendlich, H. Kim, and A. Phan. Effectiveness of internal versus external SEU scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis. *IEEE Trans. Nucl. Sci.*, 55(4):2259–2266, August 2008.

[14] D. Bertozzi, L. Benini, and G. De Micheli. Error control schemes for on-chip communication links: the energy-reliability tradeoff. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 24(6):818–831, 2005.

[15] Muhammad Bhatti, Cécile Belleudy, and Michel Auguin. Hybrid power management in real time embedded systems: an interplay of dvfs and dpm techniques. *Real-Time Systems*, 47:143–162, 2011. 10.1007/s11241-011-9116-y.

[16] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen. DyNoC: A dynamic infrastructure for communication in dynamically reconfugurable devices. In *International Conference on Field Programmable Logic and Applications*, pages 153 – 158, aug. 2005.

[17] Shekhar Borkar. Microarchitecture and design challenges for gigascale integration. In *Proc. 37th Annu. IEEE/ACM Int. Symp. Microarchitecture*, page 3, 2004.

[18] C. Pilotto, J.R. Azambuja, and F. L. Kastensmidt. Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications. In *Proc. 21st Annual Symposium on Integrated Circuits and System Design*, pages 199–204, 2008.

[19] Jean-Michel Chabloz. *Globally-Ratiochronous, Locally-Synchronous Systems*. PhD thesis, Royal Institute of Technology (KTH), 2012.

[20] Jean-Michel Chabloz and Ahmed Hemani. Distributed DVFS using rationally-related frequencies and discrete voltage levels. In *Proc. International symposium on Low power electronics and design (ISLPED)*, pages 247–252, 2010.

[21] Jean-Michel Chabloz and Ahmed Hemani. Lowering the latency of interfaces for rationally-related frequencies. In *ICCD*, pages 23–30, 2010.

[22] Jean-Michel Chabloz and Ahmed Hemani. A gals network-on-chip based on rationally-related frequencies. In *ICCD*, pages 12–18, 2011.

[23] J.M. Chabloz and A. Hemani. *Scalable Multi-core Architectures*, chapter Power Management Architecture in McNOC, pages 55–80. Springer Science Business media LLC, 2012.

[24] A. Chakraborty and M.R. Greenstreet. Efficient self-timed interfaces for crossing clock domains. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 78 – 88, May 2003.

[25] Xiaowen Chen, Zhonghai Lu, A. Jantsch, and Shuming Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 39–44, 2010.

[26] Chen-Ling Chou and R. Marculescu. Incremental run-time application mapping for homogeneous nocs with multiple voltage levels. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, pages 161–166, Sept 2007.

[27] I-Hsin Chung, Che-Rung Lee, Jiazheng Zhou, and Yeh-Ching Chung. Hierarchical mapping for hpc applications. In *Parallel and Distributed*

Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pages 1815–1823, May 2011.

[28] Calin Ciordas. *Monitoring-Aware Network-on-Chip Design*. PhD thesis, Eindhoven University of Technology, 2008.

[29] Calin Ciordas, Andreas Hansson, Kees Goossens, and Twan Basten. A monitoring-aware network-on-chip design flow. *J. Syst. Archit.*, 54:397–410, March 2008.

[30] Katherine Compton. *Reconfigurable Computing the Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers, 2008.

[31] E. Cota, F.L. Kastensmidt, M. Cassel, M. Herve, P. Almeida, P. Meirelles, A. Amory, and M. Lubaszewski. A high-fault-coverage approach for the test of data, control and handshake interconnects in mesh networks-on-chip. *Computers, IEEE Transactions on*, 57(9):1202 –1215, sept. 2008.

[32] R. Dafali and J.-P. Diguet. Self-adaptive network interface (sani): Local component of a noc configuration manager. In *Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig '09*, pages 296–301, 2009.

[33] W J. Dally and J. W Poulton, editors. *Digital System Engineering*. Cambridge University Press, 1998.

[34] Andreas Dandalis and Viktor K. Prasanna. Configuration compression for FPGA-based embedded systems. In *Proc. Ninth international symposium on Field programmable gate arrays*, pages 173–182, New York, NY, USA, 2001. ACM.

[35] A. DeHon. Dynamically programmable gate arrays: A step toward increased computational density. In *Proc. Fourth Canadian Workshop on Field-Programmable Devices (FPD)*, pages 47–54, 1996.

[36] Nasim Farahini. An improved hierarchical design flow for coarse grain regular fabrics. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2011.

[37] U. Feige and P. Raghavan. Exact analysis of hot-potato routing. In *33rd Annual Symposium on Foundations of Computer Science*, SFCS '92, pages 553–562, Washington, DC, USA, 1992. IEEE Computer Society.

[38] M.D. Galanis, G. Dimitroulakos, and C.E. Goutis. Mapping DSP applications on processor/coarse-grain reconfigurable array architectures. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, page 4 pp., May 2006.

[39] A.-A. Ghofrani, R. Parikh, S. Shamshiri, A. DeOrio, Kwang-Ting Cheng, and V. Bertacco. Comprehensive online defect diagnosis in on-chip networks. In *Proc. IEEE VLSI Test Symposium (VTS)*, pages 44–49, 2012.

[40] L. Guang. *Hierarchical agent-based adaptation for self-aware embedded computing systems*. PhD thesis, . Ph.D. thesis, University of Turku, Finland, 2012.

[41] Liang Guang, E. Nigussie, and H. Tenhunen. Run-time communication bypassing for energy-efficient, low-latency per-core DVFS on network-on-chip. In *Proc. IEEE International SOC Conference (SOCC)*, pages 481 –486, Sept 2010.

[42] Liang Guang, Ethiopia Nigussie, Jouni Isoaho, Pekka Rantala, and Hannu Tenhunen. Interconnection alternatives for hierarchical monitoring communication in parallel socs. *Microprocessors and Microsystems*, 34(5):118–128, Aug 2010.

[43] H. Singh, M.H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M.C. Filho. Morphosys: An integrated reconfigurable system for data-parallel computation-intensive applications. *IEEE Trans. Comput.*, 49(5):465–481, May 2000.

[44] H.Amano, T.Inuo, H.Kami, T.Fujii, and M.Suzuki. Techniques for virtual hardware on a dynamically reconfigurable processor - An approach to tough cases. In *Field Programmable Logic and Application Lecture Notes in Computer Science*, pages 464–473, Berlin, 2004.

[45] S. Hauck, Zhiyuan Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 138–146, 1998.

[46] Mitchell Hayenga, Natalie Enright Jerger, and Mikko Lipasti. SCARAB: a single cycle adaptive routing and bufferless network. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO 42, pages 244–254, New York, NY, USA, 2009. ACM.

[47] J. Heiner, N. Collins, and M. Wirthlin. Fault tolerant ICAP controller for high-reliable internal scrubbing. In *Proc. IEEE Aerospace Conf.*, pages 1–10, 2008.

[48] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb. FPGA partial reconfiguration via configuration scrubbing. In *Proc. Int. Conf. Field Programmable Logic and Applications (FPL 2009)*, pages 99–104, Prague, Czech Rep., 31 Aug. – 2 Sept. 2009.

[49] I. Herrera-Alzu and M. López-Vallejo. Design techniques for Xilinx Virtex FPGA configuration memory scrubbers. *IEEE Trans. Nucl. Sci.*, 60(1):376–385, February 2013.

[50] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 347–348, New York, NY, USA, 2010. ACM.

[51] M. Huebner, M. Ullmann, F. Weissel, and J. Becker. Real-time configuration code decompression for dynamic FPGA self-reconfiguration. In *Proc. International Parallel and Distributed Processing Symposium*, 2004.

[52] Intel. Microprocessor quick reference guide. http://www.intel.com/pressroom/kits/quickreffam.htm.

[53] ITRS. International technology roadmap for semiconductors 2011 edition: Executive summary. http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf, 2011.

[54] Eric Jackowski. FFT survey, March 2010.

[55] S. M. A. H. Jafri, Liang Guang, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen. Energy-aware fault-tolerant network-on-chips for addressing multiple traffic classes. In *Proc. Euromicro Conf. Digital System Design (DSD)*, pages 242–249, 2012.

[56] S. M. A. H. Jafri, S.J. Piestrak, O. Sentieys, and Sebastien Pillement. Design of a fault-tolerant coarse-grained reconfigurable architecture: A case study. In *Proc. Int. Symp. Quality Electronic Design (ISQED)*, pages 845–852, 2010.

[57] S.M.A.H. Jafri, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen. Compact generic intermediate representation (CGIR) to enable late binding in coarse grained reconfigurable architectures. In *Proc. International Conference on Field-Programmable Technology (FPT),*, pages 1 –6, Dec. 2011.

[58] S.M.A.H. Jafri, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen. Compression based efficient and agile configuration mechanism for coarse grained reconfigurable architectures. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 290 –293, may 2011.

[59] Syed M. A. H. Jafri, Liang Guang, Axel Jantsch, Kolin Paul, Ahmed Hemani, and Hannu Tenhunen. Self-adaptive noc power management with dual-level agents - architecture and implementation. In *PECCS*, pages 450–458, 2012.

[60] Syed M. A. H. Jafri, Ozan Ozbak, Ahmed Hemani, Nasim Farahini, Kolin Paul, Juha Plosila, and Hannu Tenhunen. Energy-aware CGRAs using dynamically reconfigurable isolation cells. In *Proc. International symposium for quality and design (ISQED)*, pages 104–111, 2013.

[61] Syed M. A. H. Jafri, Stanislaw J. Piestra, Ahmed Hemani, Kolin paul, Juha Plosila, and Hannu Tenhunen. Energy-aware fault-tolerant cgras addressing application with different reliability needs. In *Proc. Euromicro conference on digital system design (DSD)*, 2013.

[62] Syed.M.A.H. Jafri, Muhammad Adeel Tajammul, Ahmed Hemani, Kolin Paul, Juha Plosila, and Hannu Tenhunen. Energy-aware-task-parallelism for efficient dynamic voltage, and frequency scaling, in cgras. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 104–112, 2013.

[63] Ricardo Jasinski. Fault-tolerance techniques for SRAM-based FPGAs. *Comput. J.*, 50(2):248–248, March 2007.

[64] L. Jones. *Single Event Upset (SEU) detection and correction using Virtex-4 devices*. Xilinx Ltd., San Jose, CA, January 2007. Application Note XAPP714.

[65] M. R. Kakoee, V. Bertacco, and L. Benini. ReliNoC: A reliable network for priority-based on-chip communication. In *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 1–6, 2011.

[66] G. Khan and U. Ahmed. Cad tool for hardware software co-synthesis of heterogeneous multiple processor embedded architectures,. *Design Automation for Embedded Systems*, 12:313–343, 2008.

[67] J. Kim. Low-cost router microarchitecture for on-chip networks. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 255 –266, dec. 2009.

[68] Jungsoo Kim, Sungjoo Yoo, and Chong-Min Kyung. Program phase and runtime distribution-aware online DVFS for combined Vdd/Vbb scaling. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 417 –422, April 2009.

[69] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 123–134, 2008.

[70] Yoonjin Kim and R. N. Mahapatra. Reusable context pipelining for low power coarse-grained reconfigurable architecture. In *Proc. IEEE Int. Symp. Parallel and Distributed Processing IPDPS 2008*, pages 1–8, 2008.

[71] Yoonjin Kim, Ilhyun Park, Kiyoung Choi, and Yunheung Paek. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proc. Int. Symp. ISLPED'06 Low Power Electronics and Design*, pages 310–315, 2006.

[72] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jurgen Teich. A highly parameterizable parallel processor array architecture. In *Proc. IEEE International Conference on Field Programmable Technology (FPT)*, pages 105–112, 2006.

[73] A.K. Kodi, A. Sarathy, and A. Louri. ideal: Inter-router dual-function energy and area-efficient links for network-on-chip (noc) architectures. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 241 –250, June 2008.

[74] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

[75] Ju-Yueh Lee, Cheng-Ru Chang, Naifeng Jing, Juexiao Su, Shijie Wen, R. Wong, and Lei He. Heterogeneous configuration memory scrubbing for soft error mitigation in FPGAs. In *Proc. International Conference on Field-Programmable Technology (FPT)*, pages 23–28, 2012.

[76] T. Lehtonen, P. Liljeberg, and J. Plosila. Online reconfigurable self-timed links for fault tolerant NoC. *VLSI Design*, 2007, 2007.

[77] T. Lehtonen, D. Wolpert, P. Liljeberg, J. Plosila, and P. Ampadu. Self-adaptive system for addressing permanent errors in on-chip interconnects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(4):527–540, 2010.

[78] Teijo Lehtonen. *On Fault Tolerance Methods for Networks-on-Chip*. PhD thesis, University of Turku Department of Information Technology, 2009.

[79] Lin Li, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Adaptive error protection for energy efficiency. In *International Conference on Computer Aided Design ICCAD*, pages 2–7, 2003.

[80] C. Liang and X. Huang. SmartCell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications. *EURASIP Journal on Embedded Systems*, 2009.

[81] D. Lim and M. Peattie. *Two flows for partial reconfiguration: Module based or difference based*. Xilinx Ltd., May 2004. Application Note XAPP290.

[82] D. Lipetz and E. Schwarz. Self checking in current floating-point units. In *Proc. IEEE Symposium on Computer Arithmetic (ARITH)*, pages 73–76, 2011.

[83] Z. Lu, R. Thid, M. Millberg, E. Nilsson, and A. Jantsch. NNSE: Nostrum network-on-chip simulation environment. In *Swedish System-on-Chip Conference (SSoCC),*, pages 1–4, March 2005.

[84] R. Lysecky. Low-power warp processor for power efficient high-performance embedded systems. In *Proc. Design, Automation and Test in Europe Conference Exhibition (DATE)*, pages 1 –6, April 2007.

[85] R. Lysecky and F. Vahid. A configurable logic architecture for dynamic hardware/software partitioning. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, volume 1, pages 480 – 485 Vol.1, Feb. 2004.

[86] M. A. Tajammul, M. A. Shami, A. Hemani, S. Moorthi. A NoC based distributed memory architecture with programmable and partitionable capabilities. In *Proc. 28th NORCHIP Conf.*, pages 1–6, Tampere, Finland, 15–16 Nov. 2010.

[87] A. Martin-Ortega, M. Alvarez, S. Esteve, S. Rodriguez, and S. Lopez-Buedo. Radiation hardening of FPGA-based SoCs through self-reconfiguration and XTMR techniques. In *Proc. 4th Southern Conference on Programmable Logic*, pages 261–264, 2008.

[88] M.Motomura. A dynamically reconfigurable processor architecture. In *Microprocessor Forum,*, October 2002.

[89] Thomas Moscibroda and Onur Mutlu. A case for bufferless routing in on-chip networks. In *Proc. International symposium on Computer architecture (ISCA)*, ISCA '09, pages 196–207, New York, NY, USA, 2009. ACM.

[90] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, and T. Mitra. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proc. of the 50th Annual Design Automation Conference (DAC)*, 2013.

[91] N. Farahini, S. Li, M. A.l Tajammul, M. A. Shami, G. Chen, A. Hemani, W. Ye. 39.9 GOPs/Watt multi-mode CGRA accelerator for a multi-standard base station. In *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, 2013.

[92] N. Farahini, S. Li, M. A.l Tajammul, M. A. Shami, G. Chen, A. Hemani, W. Ye. 39.9 GOPs/Watt multi-mode CGRA accelerator for a multi-standard base station. In *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, 2013.

[93] E Nielsson. Design and implementation of hot/potato switch in a network on chip. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2002.

[94] V. Nollet and D. Verkestt. A quick safari through the MPSoC Runtime management jungle. In *Proc. IEEE/ACM/IFIP Workshop Embedded Systems for Real-Time Multimedia ESTIMedia 2007*, pages 41–46, 2007.

[95] Pierre Palatin, Yves Lhuillier, and Olivier Temam. CAPSULE: Hardware-assisted parallel execution of component-based programs. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 247 –258, Dec. 2006.

[96] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):36:1–36:24, December 2011.

[97] S. Penolazzi and A. Jantsch. A high level power model for the nostrum NoC. In *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, pages 673 –676, 0-0 2006.

[98] S.J. Piestrak. Design of residue generators and multioperand modular adders using carry-save adders. *IEEE Transactions on Computers,*, 43(1):68–77, 1994.

[99] M. Pirretti, G.M. Link, R.R. Brooks, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. Fault tolerant algorithms for network-on-chip interconnect. In *IEEE Computer society Annual Symposium on VLSI*, pages 46 – 51, Feb 2004.

[100] Y. Qiaoyan and P. Ampadu. Transient and permanent error co-management method for reliable networks-on-chip. In *Fourth ACM/IEEE Int Networks-on-Chip (NOCS) Symp*, pages 145–154, 2010.

[101] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. Using dynamic voltage scaling to reduce the configuration energy of run time reconfigurable devices. In *Proc. Design, Automation and Test in Europe Conference Exhibition (DATE)*, pages 1 –6, April 2007.

[102] A.-M. Rahmani, P. Liljeberg, J. Plosila, and H. Tenhunen. Developing reconfigurable FIFOs to optimize power/performance of voltage/frequency island-based networks-on-chip. In *Proc. IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 105 –110, April 2010.

[103] G.K. Rauwerda, P.M. Heysters, and G.J.M. Smit. Towards software defined radios using coarse-grained reconfigurable hardware. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):3 –13, jan. 2008.

[104] G.K. Rauwerda and G.J.M. Smit. Implementation of a flexible RAKE receiver in heterogeneous reconfigurable hardware. In *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, pages 437 – 440, Dec. 2004.

[105] D. Rossi, P. Angelini, and C. Metra. Configurable error control scheme for NoC signal integrity. In *Proc. 13th IEEE Int. On-Line Testing Symp. (IOLTS)*, pages 43–48, 2007.

[106] Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. An analysis of on-chip interconnection networks for large-scale chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 7(1):4:1–4:28, May 2010.

[107] K. Sankaralingam, R. Nagarajan, R. Mcdonald, R. Desikan, S. Drolia, M.S. Govindan, P. Gratz, D. Gulati, H. Hanson, Changkyu Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S.W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proc. Annual IEEE/ACM*

International Symposium on Microarchitecture (MICRO], pages 480 –491, Dec. 2006.

[108] T. Sano, Y. Saito, and H. Amano. Configuration with self-configured datapath: A high speed configuration method for dynamically reconfigurable processors. In *Proc. Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 112–118, 2009.

[109] T. Sato, H. Watanabe, and K. Shiba. Implementation of dynamically reconfigurable processor DAPDNA-2. In *Proc. IEEE international symposium on Design, Automation and Test 2005 (VLSI-TSA-DAT)*, pages 323–324, 2005.

[110] M. A. Shami and A. Hemani. Partially reconfigurable interconnection network for dynamically reprogrammable resource array. In *Proc. IEEE 8th Int. Conf. ASIC ASICON '09*, pages 122–125, 2009.

[111] M. A. Shami and A. Hemani. Classification of massively parallel computer architectures. In *Proc. IEEE Int. Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 344–351, May 2012.

[112] Muhammad Ali Shami. *Dynamically Reconfigurable Resource Array.* PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2012.

[113] L.T. Smit, G. J M Smit, J.L. Hurink, H. Broersma, D. Paulusma, and P.T. Wolkotte. Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 421–424, Dec 2004.

[114] Jinho Suh, Murali Annavaram, and Michel Dubois. MACAU: A Markov model for reliability evaluations of caches under single-bit and multi-bit upsets. In *Proc. IEEE 18th Int. Symp. High-Performance Computer Architecture (HPCA '12)*, pages 1–12, Washington, DC, USA, 2012.

[115] Jinho Suh, Mehrtash Manoochehri, Murali Annavaram, and Michel Dubois. Soft error benchmarking of L2 caches with PARMA. *SIGMETRICS Perform. Eval. Rev.*, 39(1):85–96, June 2011.

[116] D. Sylvester, D. Blaauw, and E. Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Design & Test of Computers*, 23(6):484–490, 2006.

[117] M. A. Tajammul, M. A. Shami, and A. Hemani. Segmented bus based path setup scheme for a distributed memory architecture. In *Proc. IEEE 6th Int. Symp. Embedded Multicore SoCs (MCSoC)*, pages 67–74, Sept. 2012.

[118] M.A. Tajammul, M.A. Shami, A. Hemani, and S. Moorthi. NoC based distributed partitionable memory system for a coarse grain reconfigurable architecture. In *International Conference on VLSI Design (VLSI Design),*, pages 232 –237, Jan. 2011.

[119] Muhammad Adeel Tajammul, Syed M. A. H. Jafri, Ahmed Hemani, Juha Plosila, and Hannu Tenhunen. Private configuration environments for efficient configuration in CGRAs. In *Proc. Application Specific Systems Architectures and Processors (ASAP)*, Washington, D.C., USA, 5–7 June 2013.

[120] Michael B. Taylor. Is dark silicon useful? harnessing the four horesemen of the coming dark silicon apocalypse. In *Design Automation Conference*, 2012.

[121] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, pages 241–268. 2011.

[122] F. Thoma, M. Kuhnle, P. Bonnot, E.M. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K.D. Muller-Glaser, and J. Becker. MORPHEUS: Heterogeneous reconfigurable computing. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 409 –414, aug. 2007.

[123] V. Tunbunheng, M. Suzuki, and H. Amano. RoMultiC: Fast and simple configuration data multicasting scheme for coarse grain reconfigurable devices. In *Proc. IEEE International conference on Field-Programmable Technology (FPT)*, pages 129–136, 2005.

[124] F.-J. Veredas, M. Scheppler, W. Moffat, and Bingfeng Mei. Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes. In *Proc. Int. Conf. Field Programmable Logic and Applications (FPL 2005)*, pages 106–111, Tampere, Finland, 24–26 Aug. 2005.

[125] F. Worm, P. Ienne, P. Thiran, and G. De Micheli. An adaptive low-power transmission scheme for on-chip networks. In *Proc. 15th Int. Symp. System Synthesis*, pages 92–100, 2002.

[126] F. Worm, P. Ienne, P. Thiran, and G. De Micheli. A robust self-calibrating transmission scheme for on-chip networks. *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, 13(1):126–139, 2005.

[127] T.T. Ye, L. Benini, and G. De Micheli. Analysis of power consumption on switch fabrics in network routers. In *Proc. 39th Design Automation Conference (DAC)*, pages 524 – 529, 2002.

[128] Ch. Ykman-Couvreur, E. Brockmeyer, V. Nollet, T. Marescaux, F. Catthoor, and H. Corporaal. Design-time application exploration for MP-SoC customized run-time management. In *Proc. International Symposium on System-on-Chip*, pages 66 –69, Nov. 2005.

[129] Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor, and H. Corporaal. Pareto-based application specification for MP-SoC customized run-time management. In *Proc. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, pages 78 –84, July 2006.

[130] Q. Yu and P. Ampadu. Adaptive error control for nanometer scale network-on-chip links. *IET Computers & Digital Techniques*, 3(6):643–659, November 2009.

[131] S. Y. Yu. *Fault tolerance in adaptive real-time computing systems.* PhD thesis, Stanford University, December 2001.

[132] Zain-ul-Abdin and B. Svensson. Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. *Microprocessors & Microsystems*, 33:161–178, March 2009.

[133] H. Zimmer and A. Jantsch. A fault model notation and error-control scheme for switch-to-switch buses in a network-on-chip. In *Proc. First IEEE/ACM/IFIP Int Hardware/Software Codesign and System Synthesis Conf*, pages 188–193, 2003.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-3164-3
ISSN 1239-1883

Syed M. A. H. Jafri

Vritual Runtime Application Partitions for Resource Management in Massively Parallel Architectures