Field Programmable Gate Arrays with Hardwired Networks on Chip

---

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op

dinsdag 6 november 2012 om 15:00 uur

door

MUHAMMAD AQEEL WAHLAH

Master of Science in Information Technology
Pakistan Institute of Engineering and Applied Sciences (PIEAS)
geboren te Lahore, Pakistan.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. K.G.W. Goossens

Copromotor:
Dr. ir. J.S.S.M. Wong

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. K.G.W. Goossens	Technische Universiteit Eindhoven, promotor
Dr. ir. J.S.S.M. Wong	Technische Universiteit Delft, copromotor
Prof. dr. S. Pillement	Technical University of Nantes, France
Prof. dr.-Ing. M. Hubner	Ruhr-Universitat-Bochum, Germany
Prof. dr. D. Stroobandt	University of Gent, Belgium
Prof. dr. K.L.M. Bertels	Technische Universiteit Delft
Prof. dr.ir. A.J. van der Veen	Technische Universiteit Delft, reservelid

ISBN: 978-94-6186-066-8
Keywords: Field Programmable Gate Arrays, Hardwired, Networks on Chip

Printed in The Netherlands

Acknowledgments

Today when I look back, I find it a very interesting journey filled with different emotions, i.e., joy and frustration, hope and despair, and laughter and sadness. At the same time, I feel that I am lucky enough to have some great people around, without whom the journey could not have been possible. I would like to express my gratitude to all of them as following.

First of all I would like to convey my gratitude to Kees Goossens, my promoter and supervisor, for his erudite and invaluable supervision with sustained inspirations and incessant motivation. He guided me to explore the challenging research problems while giving me the complete flexibility, which provided the rationale to unleash my ingenuity and creativity along with an in-depth exploration of various research issues. Despite being a busy person, he still managed to extract time to provide me with his sufficient feedback. His encouragement and meticulous feedback wrapped in constructive criticism helped me to keep the impetus and to remain streamlined on the road of research that resulted in the triumphant completion of this work.

I would also like to thank the PhD committee, i.e., Kees Goossens, Sebastien Pillment, Dirk Stroobandt, Michael Hubner, Koen Bertels, and Stephan Wong for investing their precious time to read the thesis and providing me with their valuable feedback.

I am grateful to Higher Education Commission (HEC) Pakistan for financially supporting my research work during the initial four years of my PhD that enabled me to work and do research in the Computer engineering department of Technical University of Delft, one of the leading universities in the world.

I would like to pay my thanks to all of the colleagues from the Computer engineering department for their discussions and feedback. In particular, I want to thank Dr. Jae Young Hur for the many discussions, motivational talks and valuable guidance during my first two years of PhD. I also want to extend my thanks to Dr. Chunyang Guo for being such a nice friend and office mate in all those PhD years. Furthermore, I want to acknowledge the support of our chair secretary Lidwina Tromp, and administrators Erik de Vries and Eef Hartman to provide a good working environment.

I would like to pay my deepest gratitude to my parents (Muhammad Siddique Wahlah and Razia Sultana) and my siblings (Anwar-us-Saeed, Riffat Shahid, Tasneem Khalid, Muhammad Shafique, Naseem Atif), and my in-laws (Razia

Naveed, Afzal Naveed, and Saba Naveed) for their never-ending support, sincere prayers, and encouragement throughout my Ph.D studies. In particular, I am thankful and pay salute to my parents (Muhammad Siddique Wahlah and Razia Sultana) for their unconditional love and exceptional sacrifice. I always found them standing beside me whenever I needed them. I must say that I can not thank enough to Almighty Allah, Who gave me such great parents.

Finally, I get to the persons who I owe the most for the completion of this journey. My wife Tahira Aqeel, who always stood beside me through this long journey. I must say that she endures all the efforts that were put in to produce the thesis. I would not have reached this point without her loving and caring support, and I want to take this opportunity to thank her from the core of my heart. I also want to present bundle of thanks and love to my little three years old princess Ayesha Aqeel, whose smile and little acts always freshens up my mind and brightens up my days. More so often she makes me feel how beautiful life could have been, and how much blessed a person I am.

I dedicate this thesis to all of my family members, and my advisor Prof. Kees Goossens.

Field Programmable Gate Arrays with Hardwired Networks on Chip

Muhammad Aqeel Wahlah

Abstract

Technology down-scaling and platform-based designs have enforced a number of application and architecture trends for system-on-chip (SoC) designs. A modern SoC is now a multi-functional machine that can execute a large number of complex applications by using tens or even hundreds of intellectual properties (IPs). Meanwhile, due to a number of constraints, e.g., short time to market, fickle market demands, and high non-recurring engineering (NRE) costs to name a few, Field Programmable Gate Arrays (FPGAs) have gained popularity to implement SoC designs. The applications in an SoC can be dynamically started and stopped thus forming multiple use-cases. The applications can also have diverse Quality-of-Service (QoS) constraints ranging from non real-time to soft, firm, and hard real-time constraints. At the same time the IP cores in an SoC are heterogenous in nature and run at diverse clock frequencies. The IPs can be microprocessors, DSP slices, memories, and ALU units, etc. The increasing number and diversity of applications and IPs require a powerful onchip communication architecture for quick integration and appropriate QoS. In contemporary FPGAs the onchip interconnect would be *soft*, i.e., programmed in the configurable fabric.

The above-mentioned application and architecture trends have triggered a series of problems. (1) An increasing number of applications on an FPGA often requires dynamic reconfiguration of an application, which in turn can produce interference with other running applications. (2) The increasing complexity of an application may mean that it can not be mapped entirely on the FPGA, which in turn can encounter loss of state of data during intra-application dynamic partial reconfiguration. (3) The diverse natures of applications make it difficult to fulfill the Quality-of-Service constraints of an application. (4) Similarly, it is hard to achieve (physical) timing closure in an SoC, because of the

increasing number and diversity of the IP cores. (5) The technology downscaling leads to FPGA architectures that are more prone to faults, e.g., configuration memories and logic elements in an FPGA can be stuck at a particular value. (6) Because communication architecture and IPs are both mapped as soft IPs in the same logic plane of the FPGA, their placement has many restrictions to allow for dynamic partial reconfiguration.

In this thesis, we aim to address the above-mentioned problems by proposing the architecture and design flow of a new FPGA. As the main contribution of the thesis, we propose the FPGA architecture with a hardwired network on chip (HWNoC), and multiple test, configuration, and functional regions (TCFRs). We call it hardwired, because the NoC in an FPGA is built in silicon and not by using the reconfigurable elements. By having a HWNoC we can have a globally asynchronous locally synchronous (GALS) environment, which in turn ensures that data is not lost during inter-IP communication. The HWNoC separates the communication and computation in two disjoint planes, which alleviates restrictions on the placement of IPs. As the second contribution of the thesis, we show how we can use the HWNoC to transport unified test, configuration, and functional data to TCFRs, for testing, faster configuration, and interference-free communication during execution of applications. As the third contribution of the thesis, we demonstrate that how the proposed design flow ensures predictable application behavior by fulfilling the QoS constraints. We also present a 3-tier reconfiguration model that uses the HWNoC, which ensures contention-free communication at architecture level, to overcome the problems of interference and state-loss during inter-application and intra-application reconfiguration respectively. Another contribution of the thesis is that it proposes a non-intrusive test methodology that uses the HWNoC as a test access mechanism to test the presence of faults reliability of FPGA architecture. In other words, the proposed methodology makes sure that applications are always reconfigured and executed on a reliable region of an FPGA, and without effecting the other running applications.

Table of contents

Acknowledgments **3**

Abstract . . . . . **i**

List of Tables . . . . . **ix**

List of Figures . . . . . **xi**

List of Algorithms . . . . . **xvii**

1 Introduction **1**
- 1.1 Trends 2
  - 1.1.1 Application Point of View 2
  - 1.1.2 Architecture Point of View 6
  - 1.1.3 Summary 11
- 1.2 Problems 12
  - 1.2.1 Application Point of View 12
  - 1.2.2 Architecture Point of View 13
  - 1.2.3 Summary 15
- 1.3 Requirements 16
- 1.4 Techniques 17
  - 1.4.1 Hardwired Network on Chip 19
  - 1.4.2 Design Flow to Bind Applications on FPGA 21
  - 1.4.3 Composable and Persistent-State Dynamic Reconfiguration using 3-Tier Model 21
  - 1.4.4 Online FPGA Testing 22
  - 1.4.5 Summary 23
- 1.5 Problem Statement 24
- 1.6 Thesis Organisation 24
- 1.7 Thesis Contributions 25

2 Background on FPGA & Networks on Chip 27
2.1 Background: Field Programmable Gate Array 27
2.1.1 FPGA Architecture 27
2.1.2 FPGA Design Flow 32
2.2 Background: Networks on Chip 35
2.2.1 NoC Architecture 35
2.2.2 NoC Design Flow 44
2.3 Conclusions 46

3 Proposed Solution and Related Work 47
3.1 Proposed Solution: FPGA with Hardwired NoC 47
3.1.1 Proposed Architecture 47
3.1.2 Proposed Design Flow 50
3.2 Technique: Hardwired Network on Chip 53
3.2.1 Overview 54
3.2.2 Motivation 54
3.2.3 Related Work on Conventional FPGA with Soft & Hard Interconnect 55
3.2.4 Positioning with the State of the Art 57
3.2.5 Related Work on Custom Reconfigurable Architectures 59
3.2.6 Positioning with the State of the Art 61
3.3 Technique: Binding of Applications to FPGA 62
3.3.1 Overview 62
3.3.2 Motivation 64
3.3.3 Related Work 65
3.3.4 Positioning with the State of the Art 67
3.4 Technique: Composable and Persistent-State Dynamic Reconfiguration 69
3.4.1 Overview 69
3.4.2 Motivation 70
3.4.3 Related Work 72
3.4.4 Positioning with the State of the Art 74
3.5 Technique: Online Testing 76
3.5.1 Overview 77
3.5.2 Motivation 78
3.5.3 Related Work 81
3.5.4 Positioning with the State of the Art 82
3.6 Conclusions 84

4 FPGA Architecture with a Hardwired Network on Chip 85
4.1 Overview 85
4.2 Hardwired NoC Architecture 88
4.3 Test Configuration Functional Region Architecture 89
4.3.1 Minimum Test Configuration Regions 89
4.3.2 Bus Macros 90
4.3.3 Clock Domain Crossing FIFOs 91
4.3.4 Bitstream Manager 91
4.3.5 Clock / Reset Manager 92
4.4 Control Processor Architecture 94
4.5 Hard Soft Partitioning 97
4.5.1 Hardwired NoC Partitioning 97
4.5.2 TCFR Partitioning 99
4.5.3 Control Processor Partitioning 101
4.6 Implementation versus Modeling 101
4.6.1 Hardwired NoC Implementation versus Modeling . . . 101
4.6.2 TCFR Implementation versus Modeling 102
4.6.3 Control Processor Implementation versus Modeling . . 104
4.7 Hardwired NoC Extensions 104
4.7.1 Soft & Multi FPGA NoC 104
4.7.2 Applicability Extensions 105
4.8 Architectural Limitations 106
4.9 Results and Analysis 107
4.9.1 Network Interface Variations 108
4.9.2 Router Variations 110
4.9.3 Test Configuration Functional Region Variations . . . 110
4.9.4 Design Space Exploration with Constant TCFR Size . 111
4.9.5 Design Space Exploration with Variable TCFR Size . 114
4.9.6 Area & Functional Performance Comparison of Soft & Hard NoC 116
4.10 Conclusions 119

5 Preparing the FPGA System at Compile Time 121
5.1 Architecture and Application Specifications 121
5.1.1 Architecture Specifications 121
5.1.2 Application Specifications 122
5.1.3 Required Objectives 123
5.2 PUMA: (Road to) Unified Placement, Mapping, and Allocation 124

5.2.1 Preprocessing: Database Creation 126
5.2.2 Traversing the Application and Creating Clusters . . . 127
5.2.3 Solution Space Extraction 130
5.2.4 Candidate Solution Finding 133
5.2.5 Solution Construction 139
5.2.6 Cluster Resource Reservation 143
5.3 Limitations 143
5.4 Results And Analysis 144
5.4.1 Performance: Success Rate 145
5.4.2 PUMA Scalability 147
5.5 Conclusions 148

6 Run-Time FPGA System Adaptation 149
6.1 System Configuration & Programming: Overview 149
6.1.1 FPGA With Soft Interconnect 151
6.1.2 FPGA With Hard Interconnect 151
6.1.3 Summary 153
6.2 3-Tier Model for Composable & Persistent-State Run-Time Reconfiguration 153
6.2.1 Responsibilities Across the 3 Tiers 153
6.2.2 Enforcing the Inter-Application Composability 155
6.2.3 Run Time Application Reconfiguration 156
6.2.4 Assuring the Intra-Application Persistent-State Transition 159
6.2.5 Summary 167
6.3 Limitations 168
6.4 Evaluation and Results 168
6.4.1 Configuration, Programming, & Functional: Comparison 169
6.4.2 Conventional and Proposed Architecture Comparison for Larger Systems 172
6.5 Conclusions 174

7 Online Testing of FPGA Architecture 175
7.1 The Test Methodology 176
7.1.1 TCFR Testing 177
7.1.2 Perform HWNoC Test 181
7.2 Limitations 181

7.3 Results And Analysis . . . 181
7.3.1 A Non-Intrusive Test Methodology . . . 182
7.3.2 Performance: Fault Detection Latency . . . 184
7.3.3 Spatiotemporal Cost . . . 185
7.3.4 TCFR Area Impact on Performance & Cost . . . 186
7.3.5 Comparison with the State of the Art . . . 187
7.4 Conclusion . . . 190

8 H.264 Encoder Case Study . . . 193
8.1 Design Time Specifications . . . 193
8.1.1 H.264 Specifications . . . 193
8.1.2 FPGA Specifications . . . 194
8.2 Compile Time Binding of H.264 to FPGA . . . 195
8.2.1 Cluster Creation . . . 195
8.2.2 QoS Ensured Cluster Binding . . . 196
8.2.3 Cost of QoS Guarantees . . . 197
8.3 Run Time H.264 Dynamic Reconfiguration . . . 198
8.3.1 Temporal Analysis of Application Binding . . . 199
8.3.2 Persistent State Intra-Application . . . 200
8.3.3 Composable Inter-Application . . . 203
8.4 Conclusions . . . 203

9 Conclusions . . . 205
9.1 Thesis Summary . . . 205
9.2 Thesis Contributions . . . 207
9.3 Open Issues and Future Directions . . . 208

Bibliography . . . 209

A Glossary . . . 225
A.1 List of Abbreviations . . . 225
A.2 List of Terminology . . . 227
A.3 List of Legends . . . 229

B System XML specification . . . 230
B.1 Architecture specification . . . 230
B.2 Application Specification . . . 232

List of Publications . . . 234

Samenvatting **237**

Curriculum Vitae **239**

List of Tables

1.1 Overview Of Trends, Problems, Requirements, and Techniques 18

3.1 Our Work Positioning with respect to the State of the Art on Traditional FPGAs. 57
3.2 Our Work Positioning with respect to the State of the Art on Traditional FPGAs. 63
3.3 Our Work Positioning with respect to the State of the Art on Traditional FPGAs. 68
3.4 Our Work Positioning with respect to Composable Dynamic Reconfiguration Approaches. 75
3.5 Our Work Positioning with respect to Persistent-State Dynamic Reconfiguration Approaches. 76
3.6 Our Work Positioning with respect to the State of the Art on Traditional FPGAs. 83

4.1 Hard Soft Partitioning of FPGA with Hardwired NoC. 98
4.2 Modeling Vs Implementation of the Proposed Architecture. . . 102
4.3 Specifications of the Target FPGA Architecture. 111
4.4 Soft and Hard Values of Different Components in FPGA. . . . 112
4.5 Results of Design Space Exploration with Variable TCFR Size. 115
4.6 Area of Network on Chip Components. 117

5.1 Success Rate over Multiple Applications and FPGA Dimensions. 148

6.1 Configuration, Programming, and Functional Comparison of Conventional and Proposed Architectures. 172

7.1 IP Synthesized Area, Frequency, and Bitstream Frames. 182

7.2 Cost Evaluated for **the Complete** FPGA after Varying TCFR Area 186

8.1 Application IP Synthesized Area, Frequency and Reconfiguration Time 194

8.2 Application IP Frequency and Reconfiguration Time 199

List of Figures

1.1 A Simple Application. 3
1.2 Video Application Standards Become More Complex. 4
1.3 SoC Architecture Example. 5
1.4 Design Productivity Gap [68]. 6
1.5 System on Chip Predicted Future Performance [69]. 7
1.6 FPGA Virtex Family Logic Densities over the Years [155,156, 158,161,162]. 8
1.7 FPGA Virtex Family Architectural Evolution over the Years [155,156,158,161,162]. 9
1.8 FPGA Architecture and Application on FPGA. 10
1.9 Interconnect Delay over Different Process Technologies [67]. . 11
1.10 3-Tier Behavior and Interaction in Multiple Use-cases 22

2.1 Architecture of Conventional FPGA. 28
2.2 Configurable Logic Block Architecture. 29
2.3 Different Types of Wires to Connect Logic Blocks. 30
2.4 High Level View of FPGA Architecture with Application on it. 31
2.5 Design Flow of Binding Application on a Conventional FPGA. 33
2.6 Network on Chip Architecture 36
2.7 Architecture of Master and Slave Buses in NoC. 38
2.8 Architecture of Master and Slave Network Interface Shells. . . 40
2.9 Network Interface Kernel Architecture. 41
2.10 Router Architecture. 43

2.11 Aethereal NoC Design Flow. 45

3.1 Abstract View of the Proposed Solution (FPGA with Hardwired NoC). 48
3.2 Architecture of the Proposed FPGA. 49
3.3 Architecture of the Proposed FPGA Architecture and Application on it. 50
3.4 Our Design Flow for the Proposed FPGA Architecture. 51
3.5 Restricted IP Placement due to the Presence of Soft Functional Interconnect. 55
3.6 Motivational Case Study for Unified Placement, Mapping, and Allocation. 64
3.7 Motivation for 3-Tier Reconfiguration Model with HWNoC. . 71
3.8 Motivation for our Online Test Scheme. 79

4.1 Overview Diagram of the Proposed FPGA Architecture. . . . 86
4.2 IP with one Master and one Slave Port, without Reprogramming and Reconfiguration Privileges, and its NI Shell and Kernel. 87
4.3 Detailed Functional Architecture of a Minimum Test Configuration Region. 90
4.4 Bitstream Manager to Write Bitstreams in a TCFR. 92
4.5 Clock Tree in a Test Configuration Functional Region. 93
4.6 Control Processor Communication with TCFRs. 94
4.7 Details Architecture of the Control Processor. 95
4.8 Data Forwarders in the SystemC Model of a Test Configuration Functional Region. 103
4.9 Architectural Extensions of the Hardwired NoC. 105
4.10 NI Kernel with Variable FIFO Depths. 108
4.11 NI Kernel with Variable Time-Slots. 108
4.12 NI Kernel with Variable Ports. 109
4.13 Router Area Overhead with Variable Number of Ports. 110

4.14 Bitstream Manager Area Overhead with Variable Sizes of a TCFR. 111
4.15 Soft NoC Cost for a Virtex-4 FPGA with Variable TCFR Sizes. 114
4.16 Soft NoC Benefit for a Virtex-4 FPGA with Variable TCFR Sizes. 116
4.17 HWNoC Cost for a Virtex-4 FPGA with Variable TCFR Sizes. 117
4.18 HWNoC Benefit for a Virtex-4 FPGA with Variable TCFR Sizes. 118

5.1 FPGA with Hardwired NoC: (A) High level Architecture, (B) Architecture Resource Details. 122
5.2 An Example Instance of Two IPs on FPGA Nodes and Connection Path in Between Them. 123
5.3 High Level Flow of our PUMA Scheme. 125
5.4 An Example Application Task Graph and its Clusters. 128
5.5 Example that Shows the Binding of Clusters on our FPGA. . . 134
5.6 Finding the Candidates Solutions. 135
5.7 PUMA Success Rate with Variable Communication and Area Demands. 144
5.8 Binding Results of Applications with Variable Standard Deviations w.r.t. the Communication Throughput Demands. 145
5.9 Impact on the Binding Success of Applications by Increasing Inter-IP Dependencies. 146
5.10 PUMA Success Rate with High Area (i.e., 50% and 70% Area of FPGA) and Variable Communication Requirements. 147
5.11 PUMA Success Rate with Low Area (i.e., 15% and 30% Area of FPGA) and Variable Communication Requirements. 148

6.1 Conventional Configuration and Programming with (A) Non-Programmable Soft Functional Interconnect, and (B) Programmable Soft Functional Interconnect. 150
6.2 New Configuration and Programming with Programmable Hardwired Network on Chip. 152

6.3 3-Tier Reconfiguration Model with an Overview of Responsibilities of each Tier. 154
6.4 Application Configuration by Using the System Manager. . . . 157
6.5 Starting a Soft IP. 158
6.6 Interaction between an Application Manager and its Application. 159
6.7 Application with Sub Applications and its Interaction with an Application Manager. 160
6.8 Procedural Description to Assure Persistent State by Using Application Manager. 162
6.9 Programming Protocol Structure. 164
6.10 An Example Case Study of Application Manager Operating on Input Data. 166
6.11 Procedure to Program NoC Connection. 170
6.12 Configuration Time Comparison Between the Soft and Hard Architectures. 173
6.13 Programming Time Comparison Between the Soft and Hard Architectures. 173

7.1 Run Time Flow for the Test Process. 176
7.2 Test IP placed in our FPGA with Different Abstract Level Details. 179
7.3 Applications in Different TCFRs. 183
7.4 Details of Interleaved Test, Load, and Execute for Multiple Applications. 184
7.5 Different FPGA Architectures with Variable TCFR Area and Count. Also Showing Fault Detection Latency Per TCFR. . . 187
7.6 Per TCFR: Fault Detection Latency (*mili sec*). 188
7.7 Per TCFR: Spatiotemporal Overheads. 189

8.1 H.264 Task Graph with Communication Demands. 194
8.2 Specification of the Target FPGA Architecture. 195
8.3 Showing H264 Clusters Created by using PUMA. 196

8.4 Compile Time Binding of H.264 to the Target FPGA Architecture. 197
8.5 Showing Communication Cost that is Paid for the H.264 Binding. 198
8.6 Showing Hop Count between the IP that Communicate with Each Other. 199
8.7 Showing: Temporal Analysis for SA1 and SA2 201
8.8 Bitstream Loading with Fixed Latency with Departure Time at Control Processor (X-axis) and Arrival Time at TCFR (Y-axis). 202
8.9 Persistent State Intra-Application Analysis. 202
8.10 Showing: (A) Composable Inter-Application Reconfiguration, (B) Allocated Time Slots 203

A.1 Showing Different Figures that are Used in the Thesis. 229

List of Algorithms

5.1 Calculation of Effective Throughput between two FPGA Nodes. 126
5.2 Cluster Creation Process. 129
5.3 Finding the Solution Space for a Cluster. 131
5.4 Determining the Placement of Source IP of a Cluster. 132
5.5 Determining the Solution Space for an IP of a Cluster. 133
5.6 The Process to Find Candidate Solutions. 136
5.7 Allocation Pruning Process. 138
5.8 Construction of the Best Solution. 140
5.9 Calculating Area Cost Matrix to Determine the Best Solution. . 141
5.10 Resource Reservation Process for the Best Solution. 142

1 Introduction

Over the years, the down-scaling of silicon process technologies has followed Moore's law [96, 97], due to which millions of transistors can be placed on a single chip of few millimeter dimensions [34, 65]. System designers have exploited the increased transistor densities by building systems on a single chip (SoC), with enhanced features and increased complexities [10, 33]. The SoCs have proliferated into almost every walk of our life in the form of embedded systems [54, 55], such as cell phones, PDA, GPS, MP3 players, video / still cameras, and many more. A modern day SoC can comprise numerous heterogenous intellectual properties (IPs) to execute multiple applications [69, 138]. The on-chip interconnect that enables different IPs to communicate with each other plays a pivotal role in achieving the desired performance for an SoC [48, 55].

From a target platform viewpoint, Field Programmable Gate Arrays (FPGAs) are increasingly popular to implement SoC designs [85]. The FPGA-based SoCs promise a solution to short time to market, fickle market demands, tight fiscal constraints, and high non-recurring engineering (NRE) costs [73]. A modern day FPGA architecture can offer application-specific integrated circuit (ASIC) like features [8,162], by embedding *hardwired*[1] IP blocks [8,162], e.g. DSP units, MAC units, memory blocks, etc. These computational blocks achieve performance gains for FPGA systems compared to their *soft*[2] implementation. However, as we shall discuss, the current FPGA architectures still face critical challenges in meeting the requirements of scalability, composability, predictability, and reliability required for SoC designs.

To fulfill the requirements, we propose a new FPGA architecture with a hardwired network on chip (HWNoC), and multiple test, configuration, and func-

[1] We define an IP as *hardwired* or *hard* when it is directly implemented in silicon.
[2] A *soft* IP is mapped on the reconfigurable resources (e.g. CLBs) of FPGA.

tional regions (TCFRs). The HWNOC serves as the system-level communication architecture and transports test, configuration, and functional data to TCFRs, so as to test, configure, and execute the applications on TCFRs. The proposed architecture has been simulated in SystemC (and is not implemented in the real FPGA hardware). This differs from current FPGA chips that, as we shall explain in Section 2.1, have only a single test and configuration architecture. Additionally, the conventional FPGAs do not have hardwired communication architecture. Instead, the FPGA-based SOCs make use of *soft* communication architecture (e.g., bus, cross bar, and NoC) to transport inter-IP data.

The rest of this chapter is organised as follows. We start with describing the trends of SOCs, Section 1.1. We then point out the problems that have emerged due to these trends, Section1.2. We continue by discussing the key requirements to overcome the problems to implement the FPGA-based SOC in Section 1.3. We then present the techniques to fulfill the requirements in Section 1.4. Afterwards, we state the problem that is the focus of the thesis in Section 1.5. At the end, we list the organisation and contributions of our thesis in Section 1.6 and Section 1.7, respectively.

1.1 Trends

In this section we explain the SOC trends from the application point of view, and from the architecture point of view.

1.1.1 Application Point of View

SOC functionality is defined by the set of its *applications*[3]. An application is comprised of multiple (hardwired) IPs, which can have data and control ports to process the *functional data*[4] and *control data*[5]. The IPs use logical connections to communicate with each other, as shown in Figure 1.1.

The recent trends indicate a large number of applications in SOC designs [10, 138]. For instance, OMAP SOC from Texas Instruments can be used for video and speech processing, location-based services, security, gaming, and

[3] An *application* can be defined as a program that is designed to perform a specific function.

[4] *Functional data* (or simply data) stands for the data that is computed or stored by the IPs.

[5] *Control data* is used to program the IPs by writing to their memory-mapped input output (MMIO) registers.

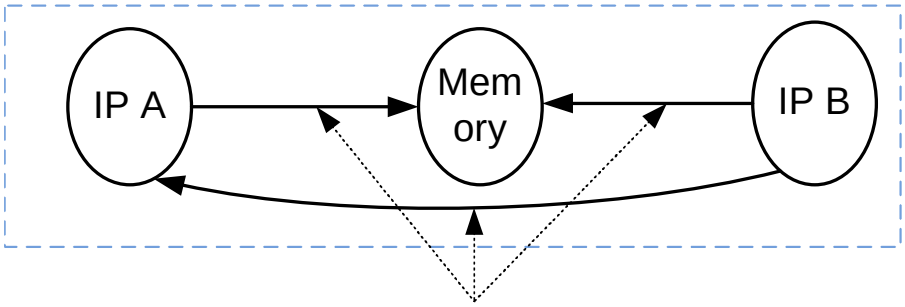


Figure 1.1: A Simple Application.

multimedia [138]. Similarly, the present day cell phones, which were traditionally used to receive and place phones calls, are now capable of conducting video conferencing, messaging, web browsing, storing pictures, and many more functions [55]. Today's FPGAs [158, 161] are used to implement SoCs that can run complex use-cases[6] [45, 100]. The applications can be started and stopped independently (e.g., on user command). As applications are often developed by different companies, it is desirable that they can be designed and tested independently. Therefore, the absence of interference is required for this, so that applications can be safely loaded at run time[7] and without affecting the already running applications [5, 49, 81]. In short, SoCs have *many applications* that are *dynamically started / stopped* as per user demand.

Typically, applications that execute on FPGA or ASIC architecture, can have diverse performance constraints on the basis of which SoC applications can be classified as *control-oriented* or *streaming* [105, 152] applications. Control-oriented applications often have non real-time constraints[8], whereas the streaming applications often have real-time Quality-of-Service (QoS) requirements and are widely used in embedded systems in the form of video, audio, and gaming. The real-time QoS constraints of streaming applications should be met in a timely manner to ensure a *predictable application behavior* [12]. The QoS constraints of an application are related to its *throughput*[9]

[6] A *use-case* is defined as the set of applications that execute in parallel at a given time.

[7] *Run time* is defined as the time during which an application executes.

[8] Control-oriented applications at times can have hard real-time guarantees, e.g., in automotive industry [108] and aerospace

[9] *Throughput* is the average data transfer rate that is required over a communication connection.

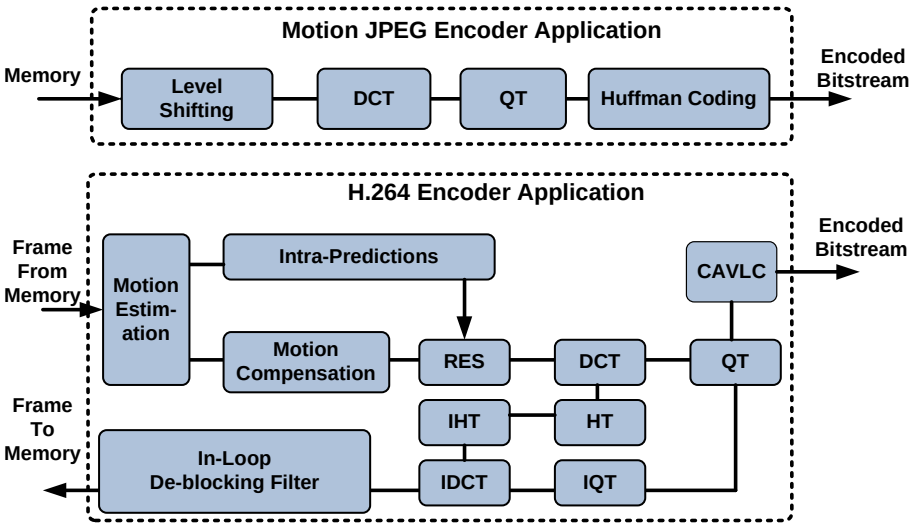


Figure 1.2: Video Application Standards Become More Complex.

and *latency*[10] requirements [12], and can fall into *soft*, *firm*, and *hard* categories. In applications with *soft* real-time quality constraints, the temporal behavior is not critical to preserve the functional correctness of SoC. For instance, during the video conferencing involving H.264 application, an occasional frame's processing deadline miss can be tolerated. On the other hand, the applications with *firm* and *hard* real-time quality constraints can not afford such a deadline miss. In these applications, the temporal behavior is critical to preserve the functional correctness of the SoC. Notably, the applications with *firm* and *hard* QoS requirements differ with each other in a safety or security aspect. The applications with *firm* requirements are not safety critical and can be found in consumer electronics, e.g., a Software-Defined Radio (SDR) [98]. On the contrary, in applications with *hard* real-time requirements, along with satisfaction over the quality constraints, an additional aspect of customer safety is also introduced. The applications with hard constraints occur in the automotive industry [108] and aerospace. Hence we can say that SoCs can have *applications with diverse natures*, i.e., with different performance constraints. For the thesis, we consider streaming applications that execute on FPGA architectures, and have soft and firm QoS constraints [95].

The end-user influence has also become a driving force in implementing SoC designs. The end users are pushing the vendors for better service quality [12],

[10] *Latency* stands for the amount of time data takes to traverse the communication connection.

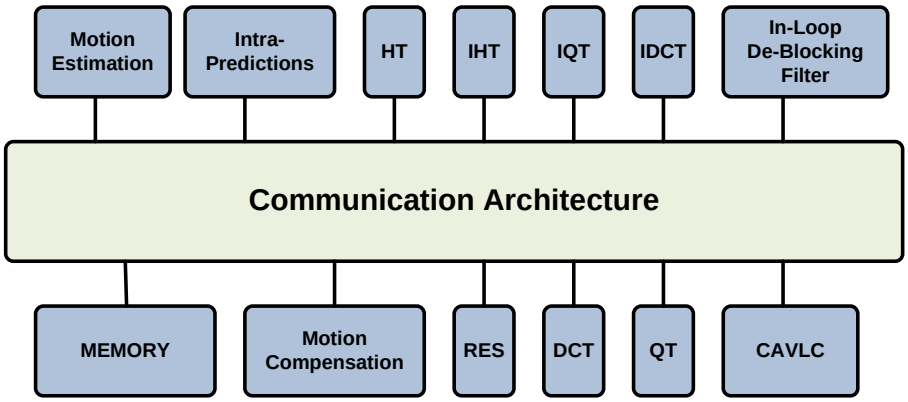


Figure 1.3: SoC Architecture Example.

such as higher video resolutions. For instance, the authors in [91] show that H.264 [75] Intra Prediction modes outperform the previously used motion JPEG 2000 [74], in terms of both subjective (visual appearance) and objective (Peak Signal to Noise Ratio) video quality. However, the high quality of H.264 Intra Prediction modes induce high computation as well as communication requirements [59]. Figure 1.2 shows an abstract comparison (in terms of IP blocks) between H.264 and motion JPEG. It shows that the H.264 encoder task graph contains more computational IPs than that of an earlier video coding standard of motion JPEG. Hence we can conclude that SoC *applications are becoming more complex*.

In recent years the SoC *product life cycle* has shortened due to the rapid technology changes [69]. The product life cycle is defined as the period in which the product is: (i) introduced through marketing, (ii) grows in sales, (iii) attains the maturity during which sales revenue stabilizes, and at some point reaches (iv) a saturation or decline stage [23]. *Time to market has become a critical factor*, because shorter time to market enables a company to launch its products ahead of its competitors.

These application trends have enforced a number of architecture trends for the SoC implementation of these applications. We explain these in the next Section 1.1.2.

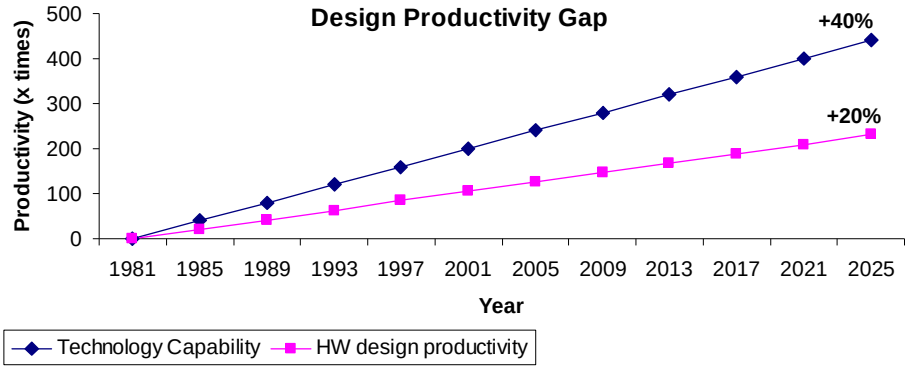


Figure 1.4: Design Productivity Gap [68].

1.1.2 Architecture Point of View

From the architecture point of view, a SoC comprises multiple IPs and a communication architecture [10, 33, 152]. The IPs implement computation or storage to execute the set of applications[11]. The IPs in SoC can be programmable processors, on-chip memories, digital signal processing units, dedicated hardware, peripherals, and internal / external interfaces such as SelectMap, ICAP, etc. [10, 162]. The IPs are made by multiple vendors and, therefore, can have different clocks and interfaces, e.g., AXI [9], DTL [110] for IPs from Philips and NXP, and PLB [157] for the μBlaze family from Xilinx. The IPs communicate (send control and functional data) with each other by making use of standard communication protocols (e.g. Advanced eXtensible Interface (AXI) and Device Transaction Level (DTL)) implemented by a communication architecture such as bus [10, 33], cross-bar switch [62, 151], or a Network-on-Chip (NoC) [13, 43, 48]. Figure 1.3 is an example of a SoC architecture, where the IPs of H.264 video encoder connect to the communication architecture. In short, a single SoC chip can have *many IPs* with *diverse natures* in terms of clock frequencies and interfaces, and a *communication architecture is used to implement inter-IP communication*, i.e., transporting functional and control data.

As a result of the technology down-scaling, the architecture of SoC chip has taken a giant leap during the past twenty years or so. Modern SoCs can comprise tens of IP cores [55], and the designers have enforced platform-based design to implement such complex systems [69]. The platform-based design relies on high reuse of IPs and performs scalable IP integration in a plug-and-

[11] In our discussions Memory is considered as an IP.

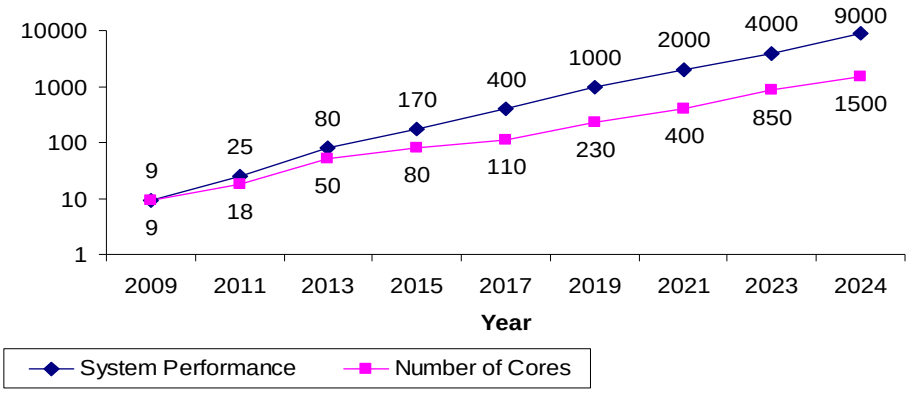


Figure 1.5: System on Chip Predicted Future Performance [69].

play fashion [123]. Through platform-based design, system designers try to reduce the design productivity gap, which indicates the difference in between the available and used number of transistors on a chip [55]. Figure 1.4 shows that the number of transistors on a chip double every 24 months (annual increase of 40%) [68], but the hardware design productivity (of VLSI designers) increases annually with 20%. Importantly, the predicted trends [18, 69] show that future systems will be far complex than the existing ones. Figure 1.5 illustrates one such trend that is mentioned in [69]. It shows that in comparison with an existing SoC in year 2009, a future SoC in year 2024 would posses approximately 150 times more processing elements to obtain a performance of 9000 times better. Hence we can say that the current and predicted trends indicate SoCs with *many IP cores*.

In recent years, FPGAs have emerged as target architectures to implement SoC designs [76, 85]. For instance, the modern FPGAs can now be found in the fields of communications [114], medicine [3, 27], radio astronomy [24], particle physics [35], and high performance computing [7, 27, 28], etc. The architecture of FPGA can be divided into two physical planes: the logic and configuration planes[12]. The *logic plane* executes the desired application(s), whereas the *configuration plane* (re)configures[13] the desired application on the logic plane.

The increasing FPGA popularity is due to the FPGA architectures, which are at the forefront of technology down-scaling. This trend can be observed from

[12]The detailed discussion on FPGA architecture can be found in Section 2.1.1.

[13]We define *(re)configuration* as the installation of new functionality in the FPGA by sending a bitstream to a reconfiguration region.

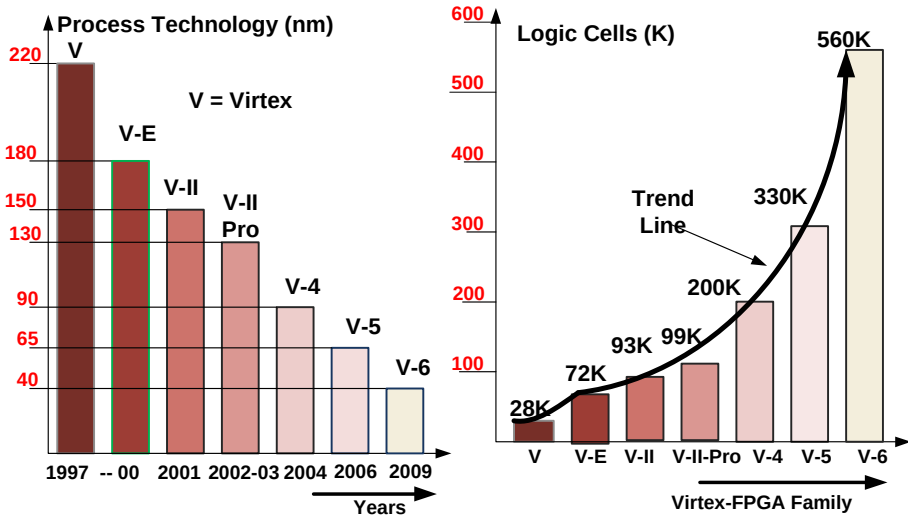


Figure 1.6: FPGA Virtex Family Logic Densities over the Years [155, 156, 158, 161, 162].

Figure 1.6A, which shows that modern FPGA features have scaled down from 220 nm to 40 nm during the last 15 years[14]. Consequently, the logic density of FPGA chip has increased by approximately 2000% over the same period, Figure 1.6B. The decreasing number of successful ASIC design starts also motivate the use of FPGAs for SoC implementations. As stated in [113], the number of successful ASIC design starts have significantly reduced from 4000 in 1997 to approximately 1000 in year 2008. The decline of ASIC designs is mainly because of: (i) longer time to market that has become one of the key element in deciding the success of SoC designs [69], (ii) high non-recurring-engineering (NRE) cost [22], (iii) and increased mask plus wafer costs [22]. In short, the modern FPGA*s use the most advanced semiconductor processes* and have become popular for SoC designs.

It is also important to note that the architectures of modern FPGAs are no longer a mere combination of configurable interconnection network and reconfigurable logic blocks (CLBs). The modern FPGA architectures also contain a number of hardwired blocks [162]. Figure 1.7 illustrates the architectural evolution for one of the Xilinx Virtex families. Over the years, in addition to increasing the reconfigurable logic density, the Virtex families [158, 161, 162] have been embellished with an increasing number and size of ASIC-like hard

[14]The new Virtex-7, which is not part of Figure 1.6, is even of smaller dimension of 28 nm [163].

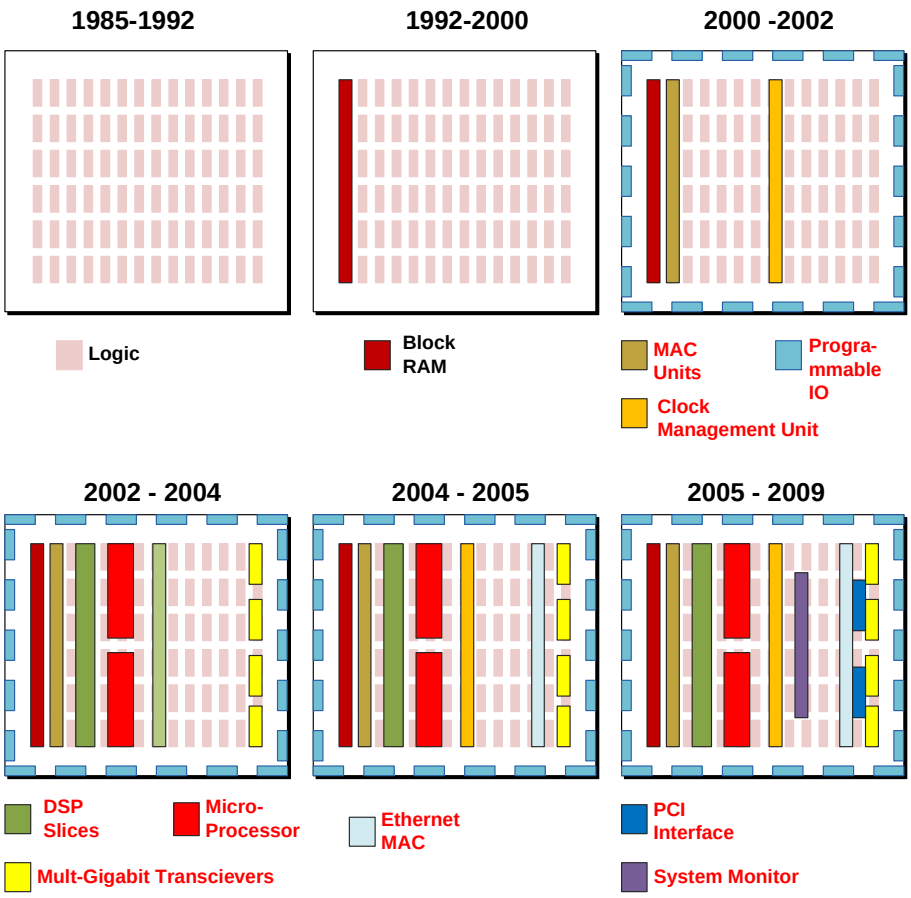


Figure 1.7: FPGA Virtex Family Architectural Evolution over the Years [155, 156, 158, 161, 162].

IP blocks. These include on-chip block RAMs, DSP slices, digital clock managers, programmable IO, programmable processors, Ethernet MAC, system monitor, transceivers, and PCI Interfaces. In existing FPGA architectures, application IPs can be *hard* or *soft*. However, the inter-IP communication architecture (e.g., NoC), which transports control and functional data among the IPs, is *soft only*. This means, in existing FPGA architectures, the bus, switch, or NoC is configurable and mapped on the reconfigurable resources of FPGA, i.e., CLBs, switch-matrices, and interconnection wires.

For the convenience of the reader, an abstract view of FPGA architecture is shown in Figure 1.8A. An FPGA is comprised of reconfigurable interconnect and logic blocks (CLBs), and programmable hard IP blocks. The Figure 1.8B

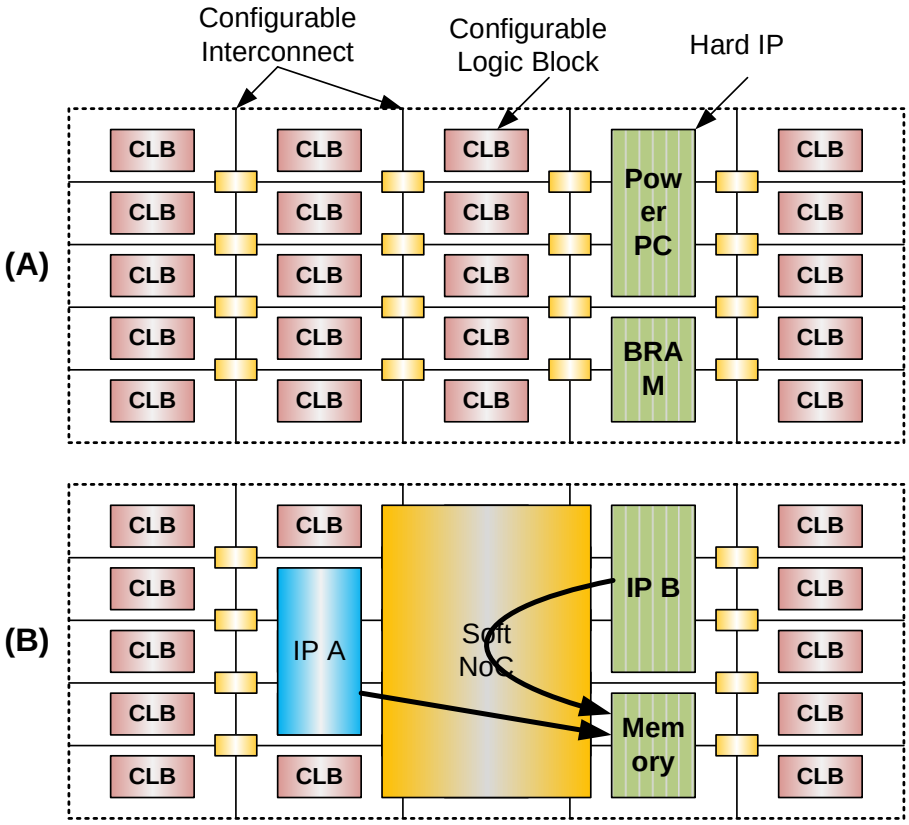


Figure 1.8: (A) Abstract View of FPGA Architecture, and (B) Application on FPGA.

shows an application (of Figure 1.1) that is mapped on FPGA architecture. In Figure 1.8B, application IPs are mapped on soft and hard blocks of the FPGA, and a soft network on chip (consisting of network interfaces and routers)[15] is used for inter-IP communication.

Existing ASIC and FPGA architectures belong to the deep sub-micron (DSM) regime, where the delays due to long wires have become prominent. With each developing process technology, the gap between the interconnection delay and the gate delay is increasing [67]. Figure 1.9 shows this trend, according to which the gap between the interconnection delay and the gate delay is expected to increase from 2:1 for 180 nm to 9:1 for 65 nm technology. This indicates that the *communication is becoming the key performance bottleneck* in the deep-sub-micron regime.

[15]The detailed discussion on NoC architecture can be found in Section 2.2.1.

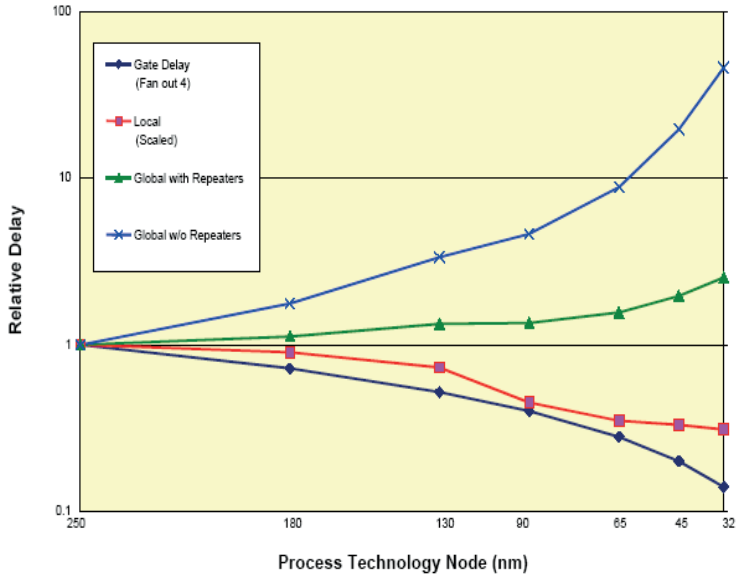


Figure 1.9: Interconnect Delay over Different Process Technologies [67].

1.1.3 Summary

Recapitulating the preceding discussions, we see that the technology down-scaling has enabled system designers to converge multiple applications in a single chip. Therefore, a number of application and architecture trends have emerged to implement SoCs.

From the application point of view, SoC can have an *increasing number of complex applications*. A user can *dynamically start / stop applications*, thus forming multiple use-cases. A single SoC can be used for multiple purposes, which means *applications can impose diverse Quality-of-Service constraints*. Meanwhile, time to market has become important due to shorter product life cycles.

From the architecture point of view, SoC applications require computation and storage resources in the form of IPs. The IPs in turn require communication resources for inter-IP communication to transport control and functional data. Importantly, the technology down-scaling and platform based designs have enabled the integration of *many IPs*. These IPs, which reach tens or even hundreds in number, can have *diverse clock frequencies*. Meanwhile, FPGA*s are now popular* (as target architecture) to implement SoC designs. The delays

in the interconnection wires of ASIC and FPGA architectures have become prominent, due to the deep sub-micron regime. This increases the *importance of communication in comparison with the computation*.

1.2 Problems

The above-mentioned trends give rise to a number of problems that need to be addressed. The problems are explained below, and can be classified from application and (FPGA) architecture points of view.

1.2.1 Application Point of View

The first problem is due to many applications in SoC. Due to the increasing number of applications, all the applications might not fit in a single FPGA simultaneously. This means at a particular time instance, not all applications are executing on FPGA architecture. To execute the applications that do not reside on FPGA, dynamic partial reconfiguration[16] is performed. However, the dynamically configured applications might interfere with the execution of applications that already execute on the FPGA [60, 125]. The interference can arise due to the rerouting of signal paths (i.e., the wires that connect different CLBs at inter-IP level) of existing applications, when a new application is dynamically reconfigured [125]. Moreover, the interference can be in the form of resource conflicts between the newly reconfigured application and already executing applications. For instance, the new application can induce resource conflicts in the communication architecture, or by sharing the same set of CLBs that are in use by the executing applications. In short, we can say that the presence of many applications can lead to dynamic reconfiguration of applications, which can introduce the problem of *inter-application interference*.

Due to the increasing complexity of applications, a single application might not even fit on a FPGA, which means the resources required by the application exceed the available FPGA resources. In this situation, a complex application is divided into multiple sub-applications [144]. For a single execution of the application, one by one, all of its sub-applications are swapped in FPGA. A sub-application is swapped out when it completes a (partial) computation. Afterwards, the next sub-application is swapped in. This implements

[16] *Dynamic partial reconfiguration* allows the reconfiguration of selected area of FPGA without shutting down the applications that run on rest of FPGA.

dynamic partial reconfiguration at the intra-application level. However, without an adequate partial dynamic reconfiguration process the state information of a sub-application, which is swapped out of FPGA, might be lost [107]. Hence the presence of a complex application that does not fit on FPGA can trigger dynamic reconfiguration at sub-application level, which can introduce the problem of *intra-application state loss*. Dynamic inter / intra application reconfiguration lead to interference, which is a problem because QoS may not be met, verification is harder since you need all applications and all interleaving / usecases to test. Hence we wish no interference between applications.

Applications can have different area and Quality-of-Service constraints. Some applications might occupy large area, whereas some applications might occupy small area in an FPGA. Moreover, due to the diverse Quality-of-Service constraints, the *functional interconnect* resources (e.g., communication links, buffers, etc.) that are shared between multiple applications are utilized differently by the SoC applications. In other words, the applications with high throughput have high resource requirements as compared to applications with low throughput. The diverse natures of applications, therefore, can produce high fragmentation of resources, which makes it difficult to meet area and Quality-of-Service constraints. The problem is aggravated with each additional application. A new application introduces additional requirements in terms of area and throughput, which in turn can increase the probability of resource conflict at an application level. This again makes it *hard to ensure area and Quality-of-Service* requirements for SoCs that have diverse as well as many applications.

As the number and complexity of applications increase in SoC, more time and efforts are required to solve the issues that can come while reconfiguring and / or executing applications on FPGA. This means the complexity and number of applications translate into *long design time*, which directly impacts the time to market trend of SoC implementation.

1.2.2 Architecture Point of View

Each IP that is integrated in a SoC operates synchronously, i.e., an IP has its own independent clock. However, an IP can have a different clock frequency than the functional interconnect. This introduces multiple clock domains, due to which the timing closure problem can arise during inter-IP communication. Importantly, the growing diversity of IPs introduces an increasing number of clock domains, making it *improbable to achieve a single clock domain* in a SoC.

The problem of IP integration can get aggravated due to the growing number of IPs, because the number of inter-IP communication connections increases. The communication architecture is therefore required to serve more number of communication connections. This in turn can raise a number of issues, e.g., (a) redesign of the communication architecture to accommodate an increasing number of connections, (b) contention of data that belong to different connections, and (c) congestion of data that flows through the communication architecture. In short, we can conclude that more number of IPs can introduce the problem of *more communication connections*.

Generally, the timing closure of an IP in isolation is not problematic, because its size is limited by the size of a single clock domain, and its lay-out is confined in space. However, the communication architecture (e.g., NoC) can introduce unpredictable delays during inter-IP communication [41]. This is due to the *soft* nature of the communication architecture, which means the reconfigurable elements (i.e., CLBs, switch-matrices, and interconnection wires) are used to construct the communication architecture. Since the global communication wires are long and span the chip, and should operate at high speed, it is difficult to achieve timing closure, i.e., to synthesise, and place and route them without timing violations.

The functional interconnect serves as the backbone for the SoC, because it provides transportation of data for all applications. The dynamic reconfiguration of an application involves the reconfiguration of its IPs and updating i.e., reconfiguration[17] and / or reprogramming[18] the functional interconnect [125, 144]. Ideally, updating the functional interconnect should not affect the executing applications. However, if it becomes inevitable to stop / pause the execution of already running application during the reconfiguration of new application, then the functional interconnect must be updated as fast as possible [144]. It is important to note that reconfiguration is a slow process as compared to reprogramming. When the reconfigured part of functional interconnect occupies significant resources of FPGA. It will induce high reconfiguration overhead in terms of bitstream size and latency. A high reconfiguration overhead might, therefore, lead to highly delayed response to execute the user application. In short, the soft nature of functional interconnect leads to the problem of *high reconfiguration overhead*.

Additionally, a soft communication architecture, which is placed in different areas of FPGA, can also pose restrictions on the placement of application IPs.

[17] *Configuring* an IP means loading its bitstream in the configuration plane.
[18] *Programming* an IP means changing the state of its registers.

The reason is that due to the presence of a soft communication architecture (in the same logic plane), an IP of a particular dimension might not be placeable. In this situation the IP has to be partitioned into multiple smaller IPs, so that they can be placed in the FPGA, as illustrated in Section 3.2. To solve the problem of restricted IP placement, which is worse with the soft functional interconnect, communication and computation should be separated from each other. In short, the soft nature of functional interconnect causes the problem of *restricted IP placement*.

The existing FPGA architectures have small feature size, and are therefore, prone to faults [1,2,37,142]. The chances of FPGA to become faulty increases, if it is used for mission-critical systems or exposed to harsh external conditions (e.g., cosmic radiations). The radiations can flip the bits in the memory cells of the configuration plane, thus resulting in wrong values for the memory cells. A value in the memory cells of configuration plane can be propagated to the logic plane [86], i.e., in CLBs, IPs, and interconnection network, resulting in an unreliable logic plane. In other words, due to small feature sizes and harsh external conditions, *faults can arise* in FPGAs.

1.2.3 Summary

Recapitulating the above discussion, we see that the application and architecture trends have raised a number of problems.

From the application point of view, *an increasing number of applications* might trigger dynamic reconfiguration at inter-application level. The dynamic reconfiguration process might interfere with already executing applications. Meanwhile, due to the *increasing complexity of applications*, an application can be dynamically reconfigured at sub-application level. In this case, the problem of state loss at intra-application level can arise during dynamic reconfiguration. At the same time, due to the *diversity of applications*, it becomes hard to fulfill area and Quality-of-Service constraints. Finally, an increasing number and diversity of applications is translated into long design times.

From the architecture point of view, an *increasing number and diversity of IPs* introduce difficulties in the integration of IPs in a SoC. Particularly, due to the diversity of IPs multiple clock domains exist in a SoC, which create the problem in achieving the global synchronisation. The *technology down-scaling* has resulted in small feature sizes for FPGA architectures, which are more prone to faults. Moreover, *the soft nature of the functional interconnect* can give rise to multiple problems that include: a) hard to achieve timing closure for inter-

IP communication, b) high reconfiguration overhead, and c) restrictions on the placement of IPs in FPGA.

1.3 Requirements

Base on problems in the previous Section 1.2, we impose nine requirements that should be fulfilled for a successful SoC design.

1. *Globally Asynchronous Locally Synchronous (GALS) techniques* are required to solve the problem of *single clock domain*. In a GALS environment, the synchronous IPs communicate with each other in an asynchronous fashion. This can be exercised by using asynchronous wrappers or bisynchronous FIFOs to connect two distinct clock domains [82].

2. *Scalable IP integration* is required to solve the problem *hard to achieve IP integration*. This can be achieved by using a communication architecture, supported by a design flow, with an inherent modular and scalable[19] nature.

3. *Communication and computation should be separate* from each other to alleviate restrictions on IP placement. This can be achieved at the physical level, i.e., communication architecture and computational IPs both do not coexist in the same logic plane.

4. *Fast updates for the communication architecture* should be performed to reduce the reconfiguration time of a dynamically started new application.

5. *Composable dynamic reconfiguration* is required to overcome the problem of interference during inter-application dynamic reconfiguration. *Composable dynamic reconfiguration* ensures that no interference is experienced during the steady state or dynamic run time reconfiguration. This can be achieved when the principles [80] of error containment, non-interfering interactions, and stability of prior services are fulfilled, while applications are executed or dynamically started and stopped in FPGA. Here, *avoiding error containment* means that errors in one application are not propagated to other application(s). *Stability of prior services*

[19] *Scalability* is the ability of something (hardware of software) to adapt to increased demands [116].

means that a dynamically inserted application has no impact and conflict with the logic and communication plane resources of the existing application(s). *Non-interfering interactions* mean that application during its execution time does not affect the other applications as long as their allocation remains unchanged.

6. *Persistent-state dynamic reconfiguration* is required to overcome the problem of state loss during intra-application dynamic reconfiguration. *Persistent-state dynamic reconfiguration* makes sure that data is not lost during intra-application dynamic reconfiguration [94, 107]. This means the state information (spread at multiple places in the system) of the sub-application must be saved, when it is swapped out. It is essential to avoid unpredictable behavior of the system.

7. *Predictability* is required to offer good application QoS, and avoid unpredictable application behavior and architecture. At the application level, predictability is required to fulfil QoS, i.e., throughput and latency constraints [48]. At the architectural level, predictability is required to resolve the timing closure issues of the soft communication architecture.

8. *Reliable architecture* for the target FPGA to ensure that the applications always execute on a fault-free FPGA.

9. *Automation*, which refers to having parts of the design process done by tools, is required to overcome the problem of high design times. Automating the process of binding[20] of application to FPGA can directly impact the time to market by reducing the design and verification efforts.

1.4 Techniques

To fulfill the above requirements, we propose a number of techniques that are shown in Table 1.1. We position the trends, problems, requirements, and techniques, where the order in which techniques are explained defines the order of Table 1.1 rows.

[20] An application is said to be: (i) *placed* when its IPs are placed on FPGA logic plane, (ii) *mapped* when its IP ports are connected to the functional interconnect, and (iii) *allocated* when its IPs can communicate (after programming the NoC) with each other as per QoS constraints. We term the whole process of placing, mapping, and allocation as *binding*.

Table 1.1: Overview Of Trends, Problems, Requirements, and Techniques

Trends	Problems	Requirements	Techniques
Diverse IPs	Many Clock Domains	GALS Environment (1)	Hardwired Network on Chip
Many IPs	Many Connections	Scalable IP Integration (2)	Hardwired Network on Chip
Soft Inter-IP Interconnect (Layout Trend)	Restricted IP Placement	Separate Communication & Computation (3)	Hardwired Network on Chip
Soft Inter-IP Interconnect (Update Trend)	High (Re)Configuration Overhead	Fast Updates for Communication Architecture (4)	Hardwired Network on Chip
Soft Inter-IP Interconnect (DSM Trend)	Hard to Meet Timing Closure	Predictability (7)	Hardwired Network on Chip
Short Time to Market	High Design Times	Automation (9)	Design Flow to Bind Applications on FPGA
Diverse Applications	Hard to Meet Area & QoS Constraints	Predictability (7)	Design Flow to Bind Applications on FPGA
Multiple Applications	Inter-Application Interference	Composable Dynamic Reconfiguration (5)	3-Tier Model for Composable Dynamic Reconfiguration
(Too) Large Applications	Intra-Application State Loss	Persistent-state Dynamic Reconfiguration (6)	3-Tier Model for Persistent-State Dynamic Reconfiguration
Small FPGA Feature size	Increasing Faults	Reliable Architecture (8)	Online Testing

1.4.1 Hardwired Network on Chip

In a FPGA architecture, the presence of an embedded hardwired system level interconnect can fulfill a number of requirements that have discussed before. The embedded system level interconnect provides inter-IP communication, and can be a hardwired network on chip (HWNOC). In the following discussion, we see that how the presence of a HWNOC in a FPGA is helpful to fulfil the requirements of a *GALS environment, scalable IP integration, decoupled communication and computation, fast updates for the inter-IP communication architecture, predictable architecture, and automation*.

To cross a clock domain from IP to NoC, the HWNOC decouples communication from computation by using bi-synchronous FIFOs. By doing this, the hardwired NoC provides a globally asynchronous locally synchronous platform, where all IPs can run at their (variable) clock speeds.

In our newly proposed FPGA chip, the hardwired NoC can serve multiple connections to transport inter-IP communication data, simultaneously [145]. Moreover, the HWNOC exhibits a scalable and modular architecture, and is made up of reusable blocks (i.e., network interfaces, routers, and connecting links) [41]. An increasing number of IPs can demand a bigger interconnect than the HWNOC topology, the dimensions of which have been decided by the FPGA *manufacturers* at the fabrication time of FPGA chip. However, the HWNOC due to its modular nature can be extended into the reconfigurable logic plane of FPGA and without changing the existing parts, as shall also discuss in Section 4.7 and show in Figure 4.9. This means that redesign of the HWNOC architecture is not required as the number of IPs increases.

In our FPGA chip, the hardwired interconnect does not occupy space in the FPGA reconfigurable plane. This means the restrictions on IP placement, which are caused by the layout of the conventional *soft* function interconnect, no longer exist. Instead, more IPs can be placed, because the FPGA reconfigurable plane will only be reserved for the SOC IPs and not for inter-IP communication architecture. In short, the communication and computation are physically disjoint because of the hardwired nature of the HWNoC. The presence of HWNOC as such does not impose constraint on the design of an IP except that in our architecture, an IP consists of a data-path to execute a specific functionality and a protocol shell to exchange data with the HWNOC. The connection between a protocol shell and the HWNOC is *soft* that is made up of FPGA reconfigurable elements such as interconnection wires and switch

matrices[21].

The dynamic addition / removal of applications can trigger an update in the inter-IP communication architecture. An FPGA with a hardwired interconnect (e.g., HWNOC) requires less time to update, as compared to an FPGA with a soft interconnection (e.g., a soft bus, crossbar or NoC). This is because the hardwired NoC can be updated by simply programming the registers. On the contrary, a *soft* interconnect, only if not a soft NoC, can only be updated after loading the bitstream. Importantly, the programming of registers can be done at a faster speed as compared to the bitstream loading [41]. Thus requiring shorter time cycle to update the inter-IP communication architecture during dynamic reconfiguration process.

The architecture of HWNoC is predictable, because the timing closure issues are solved at design time[22] of FPGA. For instance, the global wires that can be required (to connect adjacent routers) in case of a soft functional interconnect are no longer required for the hardwired NoC. Instead, these are replaced with optimised segmented wires with well-defined timing characteristics. Therefore, the HWNoC can transport the data from one IP to another IP in a predictable amount of time, provided application (QoS compliant) resources are allocated at compile time.

The design and verification times of complex systems are continue to grow. The HWNoC can reduce these by providing a pre-verified, and stable communication architecture with tested electrical parameters. The presence of HWNoC, therefore, helps the automation process by introducing short design and verification time cycles for FPGA-based SOC.

However, the cost of the above-mentioned benefits of the HWNoC should not be prohibitively high. Moreover, embedding a HWNoC should not affect the architecture of the primitive reconfigurable blocks of the FPGA, i.e., CLBs and switch-matrices. This will then encourage the vendors to introduce FPGAs with hardwired NoC by embedding as a separate IP, and without changing the FPGA reconfigurable logic plane.

[21] Please refer to Section 2.1.1 for the detailed discussion on the architecture of FPGA reconfigurable plane.

[22] As explained in Section 2.1.2, we split *design time* and *compile time* in two distinct phases, here we intend the former only.

1.4.2 Design Flow to Bind Applications on FPGA

The design flow can meet a number of requirements that include *automation* and *predictability*. In the following discussion, we discuss these one after the other.

The design flow overcomes the problem of long design times by automating the process of binding SoC applications to FPGA. First, the design flow dimensions the FPGA architecture from the specifications, i.e., FPGA topology and dimensions. Then, from the given application specifications, i.e., task graph and Quality of Service requirements, the design flow determines the binding of application to FPGA architecture, at compile time[23]. For each application, the binding algorithm has three responsibilities: (a) on which logic elements of FPGA an application IPs are to be placed, (b) to which ports of the hardwired NoC the IPs are to be mapped, (c) which paths are to be allocated, in the hardwired NoC, to transport data between the IPs.

The binding solution, therefore, takes into account the required (application) resources and available (FPGA) resources across both the logic and communication planes, simultaneously. The binding algorithm, which is triggered during the compile time phase of the design flow, ensures the predictability. In other words, the binding of application is performed only and only if its QoS constraints are fulfilled at compile time. For this purpose, our binding solution divides FPGA into two virtual planes (i.e., logic and communication). The logic resources (i.e., area) are required to place the IPs, whereas the communication resources (i.e., ports and throughput connections) are required for inter-IP communication.

It is important that the binding solution should have: (a) high performance i.e., high success-rate while binding applications with diverse QoS constraints, (b) low cost in terms of logic fragmentation and communication allocation.

1.4.3 Composable and Persistent-State Dynamic Reconfiguration using 3-Tier Model

To implement composable and persistent-state dynamic reconfiguration, we use a 3-tier reconfiguration model [143], as shown in Figure 1.10. Figure 1.10 illustrates the time-space relation, as well as abstracted interaction among the three tiers of our reconfiguration model, which are explained below.

[23] *Compile time* is defined as the time during which the user specifications are being translated into the executable code (for hardware and software), see Section 2.1.2.

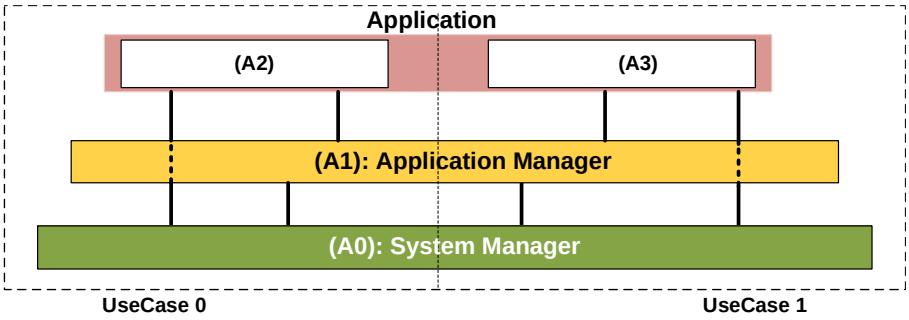


Figure 1.10: 3-Tier Behavior and Interaction in Multiple Use-cases

The 3-Tier reconfiguration model makes use of the System Manager (SM) as the foundation layer, and an Application Manager (AM) per application to ensure composability across the applications. The SM provides the services for application-specific actions, which include: configuration of application and programming the HWNoC to transport data for application IPs. Prior to configuring an application, the system manager ensures that one application can never affect another. An AM provides intra-application services, which include I/O and storage to client sub-application. An AM also enforces data-integrity between the sub-applications that are dynamically swapped in/out.

However, it is challenging to implement a composable and persistent-state system even with a 3-tier reconfiguration model, because the 3-tier reconfiguration model faces the challenges to maintain scalability and predictability with an increasing number of applications. As more and more applications are integrated, the number of dynamic reconfiguration instances of applications also increase. This in turn put more work load on the system manager that is responsible for the reconfiguration of applications. In addition, with the increasing number of applications, the level of resources sharing increases [49, 144] that in turn increases the probability of unintended behavior of applications, due to the inter-application interference.

1.4.4 Online FPGA Testing

To ensure a reliable architecture for a mission-critical FPGA system, we enforce online testing[24] of FPGA. However, due to the mission-critical nature of FPGA system, the online testing can not be performed on the whole FPGA

[24] *Online testing* verifies the FPGA chip while the system is operational. It can be further classified into structural and functional tests.

chip, simultaneously [1, 142]. Conventionally, it is performed after dividing FPGA into multiple (logical) regions. A test region can be: (i) a single configurable logic block (CLB) [38], (ii) or single / multiple columns of CLBs [2, 129]. This means at a certain point in time, some FPGA regions are under test (region under test (RUT)), and some regions are not under test (RNUT).

A non-intrusive online test scheme is challenging [146] to implement since the online test process should be transparent to RNUTs. This means (a) programming and execution of SoC applications that reside on RNUTs is not disrupted, and (b) (if required) bitstreams of new SoC applications are configured onto RNUTs. In other words, during online testing, the normal operation of RNUTs is not affected in terms of programming, execution, and configuration. Additionally, an ideal online test scheme should possess certain characteristics. For instance, it should detect high percentage of faults (ideally 100% faults). It should have small fault detection latency (ideally 1 cycle). Moreover, the *spatial* and *temporal* overheads that are induced by an online test scheme should be insignificant. Here, spatial overhead accounts for an additional test hardware [1], e.g., test pattern generators (TPGs) and output response analysers (ORAs), which are needed to perform the online verification of FPGA region. The TPGs and ORAs can be made up of FPGA reconfigurable resources, i.e., CLBs and interconnection wires, etc. [2]. On the other hand, the temporal overhead is determined by the time required to configure and use the spatial overhead. The online test scheme should be non-intrusive for RNUTs, have a low fault-detection latency, and have low spatio-temporal overheads.

Our online test scheme, like conventional schemes, tests FPGA architecture in region-wise fashion, but by using the hardwired network on chip as the test access mechanism. Meanwhile, as we shall discuss in Section 3.5, our online test scheme is non-intrusive, has a high percentage of fault detection, low fault-detection latency, and minimum spatio-temporal overheads.

1.4.5 Summary

Recapitulating the above discussion, we see that four techniques are used to fulfill the requirements of Section 1.3.

By having a hardwired network on chip we can: (a) have a GALS environment, (b) integrate IPs in a scalable way, (c) separate communication and computation in two disjoint physical planes, (d) ensure fast updates for the functional interconnect, (e) have a predictable communication architecture with

well-defined timing closure, and (f) help the automation process. The detailed overview, motivation, related work, and positioning with the state of the art of (hardwired) NoC can be found in Section 3.2.

By having a design flow, we aim to: (a) automate the process of binding of SoC applications to FPGA from design time specifications, (b) ensure application level predictability by binding an application, only and only if, its QoS constraints are fulfilled. The detailed discussion on the proposed design flow can be found in Section 3.1.2. The detailed overview, motivation, related work, and positioning with the state of the art automated binding approaches can be found in Section 3.3.

By using a 3-tier model, we aim to fulfill the requirements of: (a) composability, i.e., dynamically inserted (sub)applications do not interfere with other executing (sub)applications, (b) persistent-state, i.e., the state information (spread at multiple places in the system) of the sub-application must be saved, when it is swapped out and restored when swapped in. The detailed overview, motivation, related work, and positioning with the state of the art of 3-tier reconfiguration model can be found in Section 3.4.

By using an online test scheme, we aim to: (a) ensure that applications always start execution on a reliable FPGA architecture, (b) keep a non-intrusive behavior for the already running applications. The detailed overview, motivation, related work, and positioning with the state of the art of online test schemes can be found in Section 3.5.

1.5 Problem Statement

In this thesis we address the problems listed in Section 1.2, resulting in requirements of Section 1.3. We propose to solve them by the techniques of Section 1.4 (and Chapter 3), as shown in Table 1.1.

1.6 Thesis Organisation

This thesis is organised as follows.

Chapter 2 provides background information about FPGAs and NoCs.

Chapter 3 describes the details of our proposed solution. Moreover, we provide the overview, motivation, and positioning of the techniques that use the proposed solution (FPGA with hardwired NoC) to fulfil the requirements of

Section 1.3.

Chapter 4 provides the architecture of our proposed FPGA. It begins with explaining the hardwired NoC architecture, followed by expounding the FPGA logic plane architecture, and at the end illustrating the hard and soft partitioning of components in both the logic and the communication planes of FPGA. The results and analysis of the proposed architecture are also provided.

Chapter 5 explains our unified placement, mapping, and allocation scheme that at compile time binds applications to FPGA after ensuring the predictable performance for them. The results and analysis of the proposed application binding scheme are also provided.

Chapter 6 illustrates the booting procedure for FPGA, and a 3-tier model for composable and persistent-state run time application reconfiguration. The results and analysis about the configuration of the proposed architecture are also provided.

Chapter 7 explains the procedure for online testing of FPGA. It is performed to ensure a reliable architecture for the executing application. The results and analysis are also provided for the proposed online test scheme.

Chapter 8 shows the complete picture, i.e., from design time specifications to compile time application to FPGA binding, and run time reconfiguration, execution, and testing procedures of a real world state-of-the-art H.264 encoder.

Chapter 9 concludes our work and presents directions for the future work.

1.7 Thesis Contributions

Apart from identifying the trends, problems, and requirements imposed by the future FPGA-based SoCs (Chapter 1), the main contributions of the thesis are as follows. The citations indicate where the contribution has been published before.

1. An FPGA architecture with hardwired NoC and multiple test configuration functional regions (TCFRs), where each TCFR has its own configuration circuit [41, 145], see Chapter 4.

2. A HWNoC as the unified communication architecture to transport test, configuration, control, and functional data [143, 144, 146], see Chapter 6 and Chapter 7.

3. A binding scheme that unifies the process of placement, mapping, and allocation while binding application to FPGA [147], see Chapter 5.

4. A 3-tier model to perform composable and persistent-state dynamic run time reconfiguration for applications [143, 144], see Chapter 6.

5. An online test scheme that finds out the reliability of the proposed FPGA architecture, and without producing intrusiveness with executing applications [146], see Chapter 7.

2

Background on FPGA & Networks on Chip

In this chapter we provide the preliminary background information about the architecture and design flow of FPGAs in Section 2.1. Afterwards, we explain the architecture and design flow of NoCs in Section 2.2. Lastly, we provide the concluding remarks of the chapter in Section 2.3.

2.1 Background: Field Programmable Gate Array

In this section, we describe the architecture and the design flow of FPGAs. We have used a Xilinx FPGA for our analysis and experiments [159]. Therefore, the discussion in this section will use Xilinx specific terminology, but other FPGAs work similarly.

2.1.1 FPGA Architecture

The architecture of an FPGA can be divided into a logic plane to execute the desired functionality, and a test configuration plane to configure and test the desired functionality on the logic plane. In the following discussion, we will explain the architecture of both the planes that are shown in Figure 2.1.

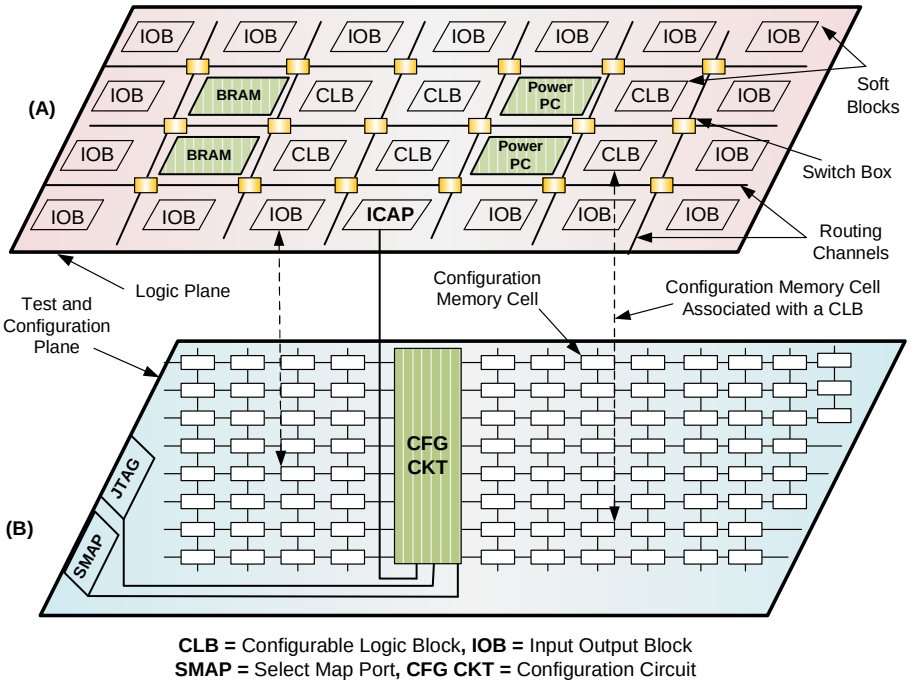


Figure 2.1: Architecture of Conventional FPGA with (a) Logic, and (b) Test and Configuration Planes [124].

Logic Plane Architecture

Modern FPGAs [158] comprise an array of *input output blocks* (IOBs)[1], *soft and hard logic blocks*, and *interconnection network*, Figure 2.1A. The *soft* logic blocks are configurable and called configurable logic blocks (CLBs), whereas the *hard* blocks are programmable and can include Block RAMs and processor units (e.g. *PowerPC*), etc. The interconnection is configurable and is used to connect the (hard and soft) logic blocks.

The Configurable Logic Blocks (CLBs) are the main logic resource to implement sequential as well as combinatorial logic. In FPGA (e.g. Virtex-4), a CLB consists of four interconnected slices that are grouped in pairs, see Figure 2.2A. Each slice[2] consists of two logic-function generators (or look-up tables), registers, and multiplexers, as shown in Figure 2.2B. The logic function

[1]The *Input Output Blocks* (IOBs) are not relevant for the thesis. Therefore, we do not discuss them in the following.

[2]The remaining details of the slice, e.g., carry logic are omitted to keep things simple.

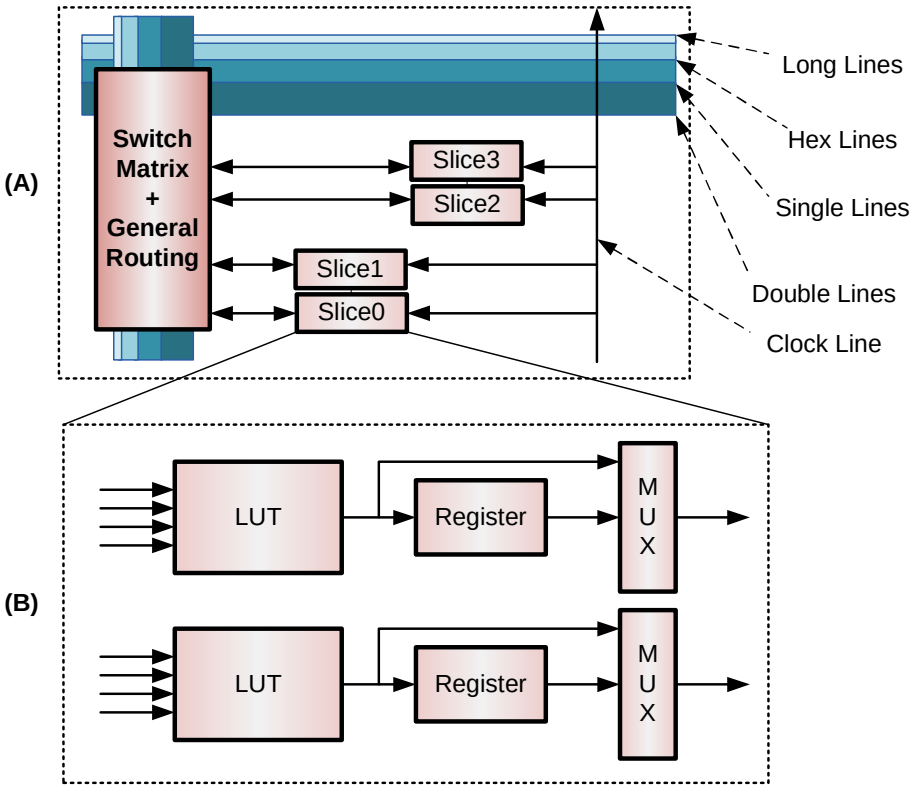


Figure 2.2: Architecture of (A) Configurable Logic Block (CLB), and (B) a CLB Slice [159].

generators or look-up tables (LUTs) have 4 inputs and one output. This means that a 4-input LUT can implement any arbitrarily defined four-input Boolean function.

The *interconnection network*, which connects the logic blocks, is comprised of a network of *routing channels* and *switch matrices*, as shown in Figure 2.1A. The Figure 2.1A shows that each logic block is surrounded by routing channels that are connected via switch boxes. The routing channels contain wires of different lengths to provide local and global routing resources. Local routing resources are used to connect adjacent logic blocks. Global routing resources, on the other hand, are used to connect the non-adjacent logic blocks, i.e., the blocks that are separated by multiple logic blocks. The wire segments in a routing channel can be categorised as long, hex, double, and direct, see Figure 2.3. The switch boxes are configured to connect the wire segments to form

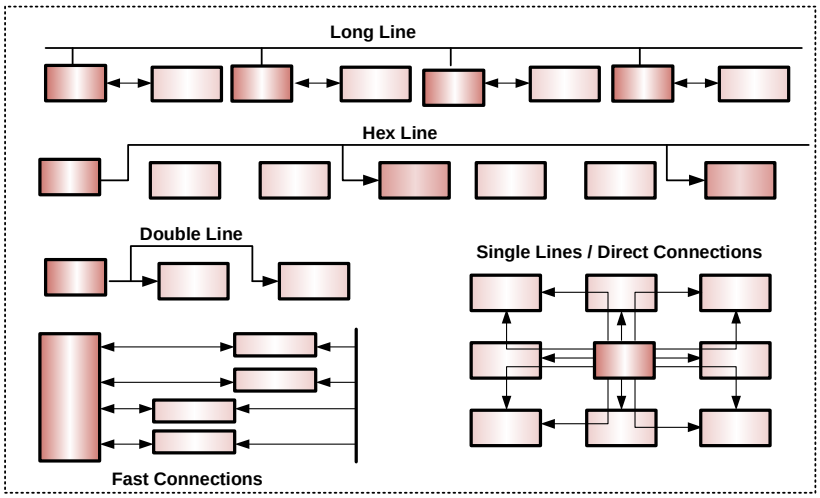


Figure 2.3: Different Types of Wires to Connect Logic Blocks.

signal paths between the logic blocks.

It is important that the elements (i.e. CLBs, IOBs, and interconnection network) of the logic plane are associated with memory cells in the configuration plane, as shown in Figure 2.1B. The architecture of the configuration plane is explained in the following discussion.

Test Configuration Plane Architecture

The test configuration plane architecture [124] comprises *memory cells* to store the configuration bits, and the *configuration circuit* to load the bitstream in the memory cells, see Figure 2.1B.

The process of loading the bitstream to FPGA memory cells is called *configuration*. FPGA configuration, in turn, implements the desired functionality on the FPGA logic plane. However, it is important to note that the configuration memory is arranged in *frames*, which are the smallest addressable unit of the configuration memory [154]. This means the configuration process must be performed in a frame-wise manner. An FPGA device has a specified *frame count*, *frame length*, and *bitstream size*. For instance, the configuration memory of Virtex-4-XC4VLX200 device is divided into 39,120 frames, each of which consists of 41 32-bit words [154]. The bitstream size[3] is, therefore,

[3]The *bitstream size* equals the number of configuration frames times the number of words

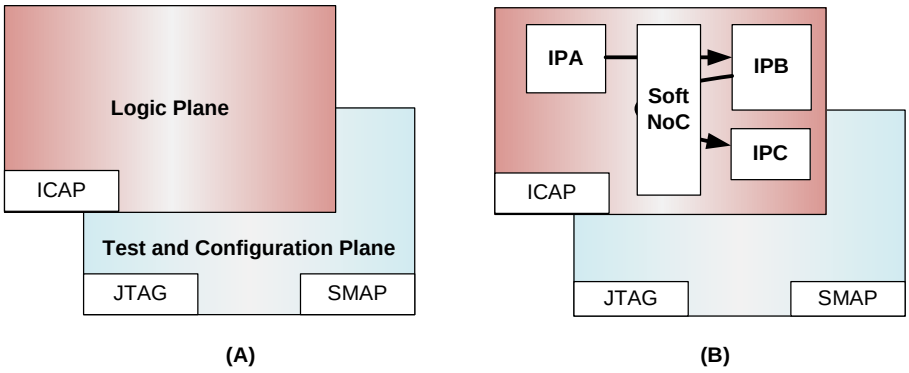


Figure 2.4: (A) High Level View of Conventional FPGA Architecture, and (B) Application on FPGA.

1,494,204 words.

The FPGA configuration circuit is used to load the bitstream to the desired configuration memory location. The configuration circuit is comprised of a number of units that include state machine, packet processor, address decoder, and multiple configuration registers etc. (not shown here to keep the diagram simple) [124]. A number of interfaces can be used to configure the elements of the logic plane. In case the bitstream is loaded by using an external host PC, then a JTAG or SelectMap port is used. An internal configuration port (ICAP) is also available, by using which the FPGA is configured by an internal entity in the logic plane, e.g., the *PowerPC* processor.

It is important that the same configuration circuit and *JTAG* port can be used to transport the *test data*. The test data consists of test bitstreams and stimuli to verify the correctness of the logic plane regions, which can be used by multiple applications.

Application on FPGA Architecture

In this section, we briefly explain the roles of different FPGA components to execute applications.

For the convenience of the reader, an abstract view of FPGA architecture is shown in Figure 2.4A. An FPGA consists of a logic plane, and a test and configuration plane. (1) The configuration data (or bitstream) for an application is transported by using the *configuration circuit in test and configuration plane*

per frame [154].

and any of the *JTAG, Select Map, or ICAP* ports. (2) It is important that the same configuration circuit and any of the *JTAG, Select Map, or ICAP* ports can be used to transport *test data* to test an application regions. (3) After configuration, application IPs and the associated functional interconnect are placed in the FPGA logic plane, see Figure 2.4B. The soft functional interconnect can be a network on chip (consisting of network interfaces and routers)[4], which is used for inter-IP communication. It is important that in the conventional FPGA architecture both the IPs and the functional interconnect exist in the logic plane of an FPGA, as shown in Figure 2.4B.

2.1.2 FPGA Design Flow

The design flow to configure applications on an FPGA can be divided in three phases [153], i.e., design time, compile time, and run time, as shown in Figure 2.5. In the following discussion, we explain the phases separately.

Design Time

Design time includes the definition of the hardware and software architecture, i.e., the common use of design time. We however split off the compilation and synthesis phases from design time. During the design time, the input specifications are provided which include: (a) target FPGA architecture, (b) application IPs, and (c) user constraints. The IPs can be specified by using schematic or Hardware Description Language (HDL), such as VHDL or Verilog. For complex designs, a HDL is used because it isolates the designer from the details of the hardware, providing a faster way for the design specifications. The user constraints are provided in a User Constraint File (UCF), and include placement and timing constraints. The placement constraints can constraint the placement of IPs on the target FPGA, whereas the timing constraints are used to constraint the path timings at intra-IP and inter-IP levels.

Compile Time

The compile time phase can be subdivided into synthesis, binding, and bitstream generation processes.

Synthesis: After a design is specified it is synthesised. In the *Synthesis* phase, the VHDL / Verilog code is *translated into a device netlist* format. This means

[4]The detailed discussion on NoC architecture can be found in Section 2.2.1.

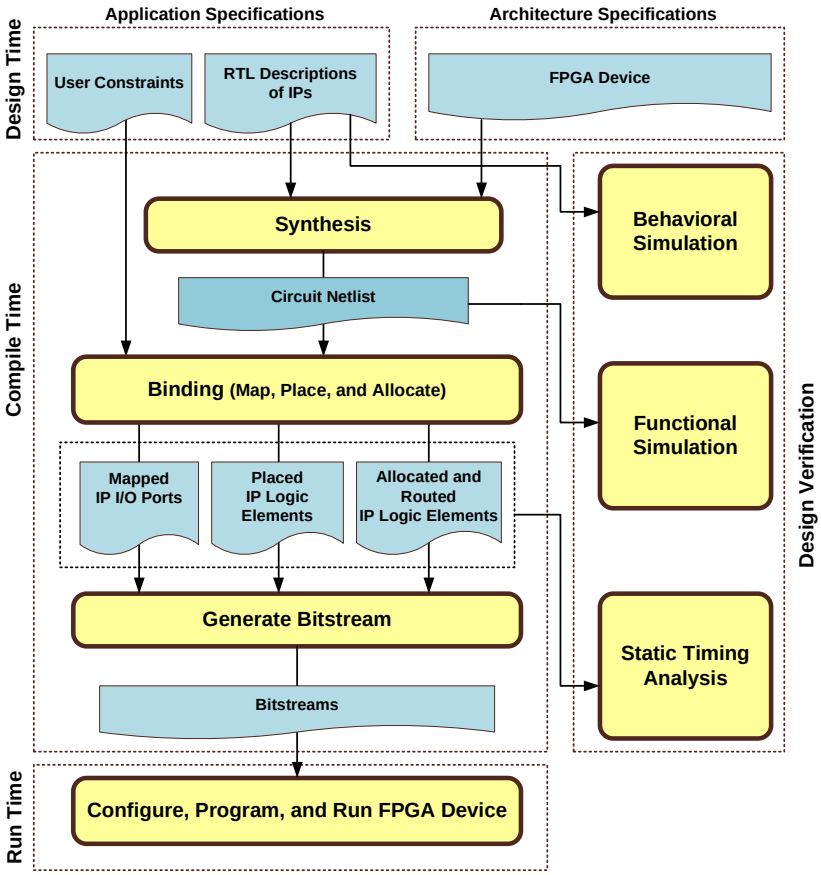


Figure 2.5: Design Flow of Binding Application on a Conventional FPGA.

that a complete circuit is generated that is comprised of gates, flip flops, etc. The resulting netlist is saved to an NGC (Native Generic Circuit) file for Xilinx Synthesis Technology (XST). The synthesised design can be verified by using *Behavioral Simulation* (RTL Simulation) that verifies the RTL (behavioral) code.

Binding: Once the synthesised design is available, it is implemented on the target FPGA device. For this purpose, the *Binding* process performs the mapping of synthesised circuit onto the target device. This process consists of three steps: (a) translate, (b) map, and (c) place and route.

The *Translate* process *assigns ports* in a synthesised design to the physical elements of the target device and specify timing requirements of a design. The physical elements could be pins, switches, buttons, etc. For this purpose, the

Translate process is provided with the input netlist files that are generated during the synthesis process. A *user constraint file* (UCF) is also provided that is specified during the design time. The output of the *Translate* process is saved as an NDG (Native Generic Database) file.

The *Map* process *maps the logic blocks* of the NGD file onto the logic blocks of the target FPGA device. The logic blocks of the target FPGA can be configurable logic blocks (CLBs), and input output blocks (IOB), etc. For this purpose, the *Map* process divides the whole circuit (of the NGD file) into sub-blocks such that they can fit into the FPGA logic blocks. The output of the *Map* process is an NCD (Native Circuit Description) file.

The *Place and Route* process *connects the logic blocks* that are obtained from the *Map* process. In other words, it allocates the FPGA interconnection resources for inter-IP and intra-IP data transportation. The output of the *Place and Route* process is a completely routed NCD file.

During the *Binding phase*, the design can be verified at various stages, as shown in Figure 2.5. This could be *Functional simulation*, which is performed after the *Translate* process. Functional simulation gives information about the logic operation of the circuit. Moreover, *Static Timing Analysis* can also be performed after the *Map* or *Place and Route* process. This is performed to obtain the timing reports (signal path delays) for the input design.

Generate Bitstream: The *Generate Bitstream* process converts the output of the *Place and Route* process into a *bitstream*. Conventionally, the configuration bitstream consists of multiple packets, where each packet contains commands and configuration data. From a routed NCD file, the *Generate Bitstream* process produces a .BIT file.

Run Time

At run time, an external host PC or an internal processing unit (e.g., a *PowerPc*) can be used to configure the device. The FPGA device is configured using the bitstream that was generated at compile time. However, it is not necessary to configure the whole FPGA, instead by using dynamic partial reconfiguration an FPGA can be partially configured [125, 141]. In partial reconfiguration, the bitstream loading of an IP or an application can be performed. Moreover, a non-relocatable bitstream, if required, can be loaded to the same FPGA region for more than once. In Chapter 6, we explain and compare the configuration process of the conventional FPGA and our proposed FPGA.

2.2 Background: Networks on Chip

Different system level interconnects can be used for inter-IP communications. These include point-to-point [63], busses [16, 125], cross-bars [21, 62], and networks on chip [56,78,88–90,92,106,149]. We propose a *hardwired network on chip* in an FPGA. Therefore, we provide the background for an NoC that includes NoC architecture and its design flow. The subsequent discussions are ÆTHEREAL [43, 44, 48] NoC specific, because we use ÆTHEREAL NoC as the hardwired NoC proposed in Section 3.1. However, any NoC with the following requirements can be used.

2.2.1 NoC Architecture

The ÆTHEREAL NoC can comprise multiple IPs, buses, network interfaces (shell and kernel), and routers. Moreover, the NoC can have a regular / irregular topology. Figure 2.6A shows a simple NoC architecture with a regular mesh topology. In the following discussion, we provide an overview of communication inside the NoC architecture by using an example. In Figure 2.6, we will discuss the role of different blocks of NoC when an IP (IPA) wants to communicate with another IP (IPC).

In ÆTHEREAL NoC a *logical connection* is between a single master and single slave, e.g., in Figure 2.6B a logical connection is established to transport data between IPA and IPC. IPA can communicate with IPC by issuing a *read or write transaction* and it is the job of a logical connection to transport the transaction. Each transaction consists of multiple request and (optional) response *message*. A write *request message* writes the data on a specific memory address, whereas read request message indicates the amount of data to be read from a specific address. In return, the *response message* can contain acknowledgment for the write operation, or data for the read operation. Therefore, a connection consists of two channels: one request channel and one response channel. The *request channel* is used to transport request messages, whereas the *response channel* is used to transport response messages.

Referring back to Figure 2.6B, let us say IPA initiates a write transaction, which is received by the *local bus*. The local bus, based on address, *forwards the request messages* (i.e., command, address, and data) to the required NI shell (in this case Shell 2a). Depending upon the address map, the shell then serialises the message into individual words using the Point-to-Point Streaming Data (PPSD) [47]. The serial data is then transported to the respective

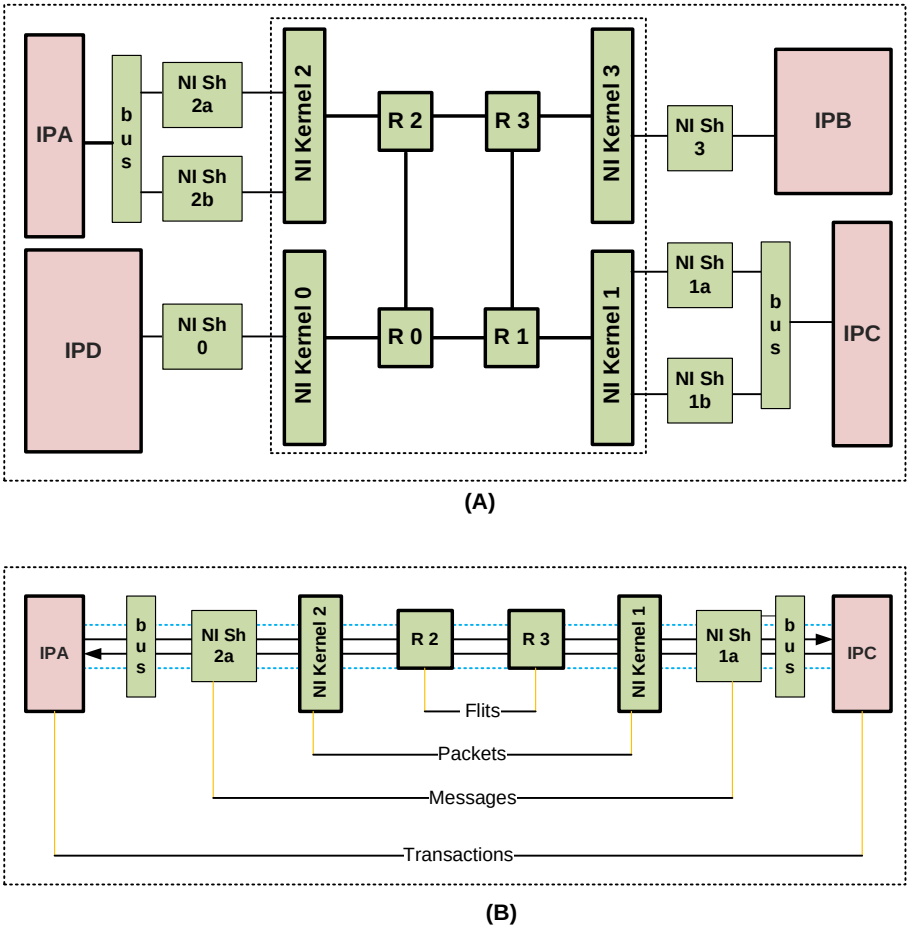


Figure 2.6: (A) Architecture of Network on Chip (NoC), and (B) Different Abstractions of Communication Data in NoC during Inter-IP Communication [47].

input queue of the NI kernel (i.e., between protocol shell and Kernel2). It is important that the *input queue is associated with the connection* that is established between IPA and IPC. The data items stay in the input queue until the connection is scheduled by the NI kernel scheduler, which arbitrates the input queue data on the basis of time-division multiplexed (TDM) slots. The NI kernel then converts the PPSD data into *packets*, and injects the packets in the router network in the form of *flow control digits* (flits), where each flits is a data unit of 3 words. The flits are routed through the router network, as determined by the packet header, until they reach the NI kernel of IPC (i.e. Kernel 1). The destination NI kernel then depacketises the received flits and puts the payload, i.e., the streaming data, in the output queue. The respective NI shell (i.e. Shell 1a) at IPC, then deserialises the streaming data into a request. This in turn indicates the local bus of IPC about the write command, flags, address, and the incoming write data. The bus then transfers the data to the attached IPC.

After writing data, IPC *sends an acknowledgment* to IPA by generating a response message. The NI shell (i.e. Shell 1a) of IPC adds a message header and serialises the response message into streaming data that is sent back through the NIs and routers (also referred as network). On the other side of the network, the response message is reassembled by the master shell and forwarded to the bus. The bus then forwards the received request to IPA.

In the following discussion, we explain the role and architecture of NoC components.

Local Buses: Role

The local buses are attached to IPs that initiate transactions and receive transactions, and are used to implement distributed memory communication [47]. An IP can use a single port to send sends data to multiple ports of multiple IPs by using a master bus, whose architecture is shown in Figure 2.7A. Conversely, a single port of an IP can be used to receive data from multiple IPs ports by using a slave bus, whose architecture is shown in Figure 2.7B.

Local Buses: Architecture

A master bus connects one memory-mapped initiator port to multiple target ports. A master bus comprises an address decoder, multiple (de)multiplexers, FIFO, and a memory-mapped programmable port, as shown in Figure 2.7A. A

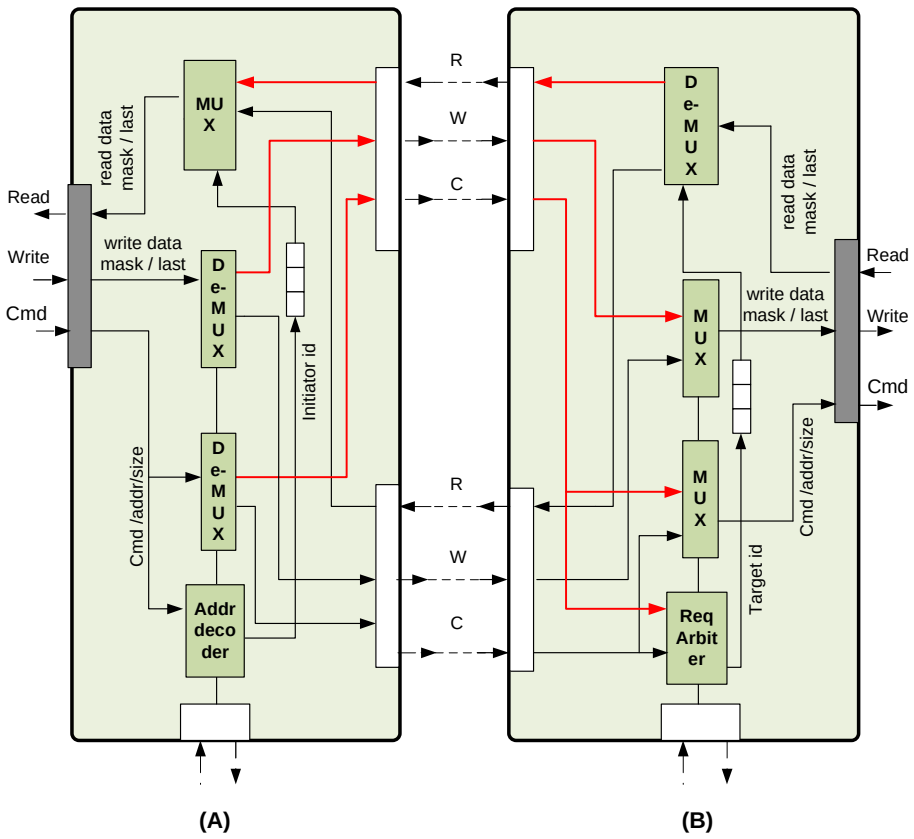


Figure 2.7: Architecture of (A) Master Bus Attached to an Initiator Port, (B) Slave Bus Attached to a Target Port.

local bus, based on the address of the request, multiplexes the request to the appropriate target port. The FIFO inside the bus is used for response ordering, which means responses are returned to the (initiator) IP in the order in which the requests were issued. The response ordering is implemented by storing the identifier of the target port when request was issued. Multiple transactions may be outstanding from one master to multiple slaves.

The architecture of a master bus (i.e. address decoder, and FIFO depth) can be dimensioned by determining the number of concurrent target ports accessed by the attached initiator port, see as we discuss in Section 2.2.2 and Figure 2.11. After the dimensioning phase, the number of target ports that are reachable from each initiator port is fixed and can not be altered at run time. However, the buses are programmable at run time which means the initiator port can be

programmed to access any slave port in the network. At run time, the selection of a target port can be made by programming the address decoder through a memory-mapped programmable port.

A slave bus connects one memory-mapped target port to multiple initiator ports. The slave bus, like the master bus, contains a number of (de)multiplexers, FIFO, and a memory-mapped programmable port, as shown in Figure 2.7B. However, in a slave bus the request arbiter is used in place of the address decoder. The request arbiter is responsible for multiplexing of requests according to a slave specific policy.

Network Interface Shell: Role

The network interface shells bridge between the *memory mapped* IP ports and the *streaming ports* of the network interface kernel. An NI shell converts specific IP port protocols, such as AXI [9] and DTL [110] to a stream of data with the PPSD protocol for request, and vice versa for responses. In our architecture, each shell contains two finite-state-machines (FSMs) named *encoder* and *decoder*, as shown in Figure 2.8. These are used to implement the valid / ready handshakes per command, read / write data groups, and their (de)serialization to / from the NI kernel ports, etc.

Network Interface Shell: Architecture

In the NoC architecture, communication is performed using a transaction-based protocol. The master IP issues request messages (e.g., read and write commands, and data) that are executed by the respective slave IPs. The slave IP (optionally) responds with a response message that can contain status of the command execution and a possible data. Therefore, NI shells are attached to either a master or slave IP and implement the inter-IP communication protocol, see Figure 2.8. A master shell comprises a request encoder and a response decoder, whereas a slave shell comprises a request decoder and a response encoder.

For a request channel, a request (that comprises read / write commands and their flags, read / write addresses, and a possible write data set) is received as input by the request encoder of the master shell. A request encoder then serialises the transaction into multiple one-word data that is presented to an input port of the attached NI kernel. An NI kernel then uses the router network to send the data to the destination NI kernel. The destination NI kernel after

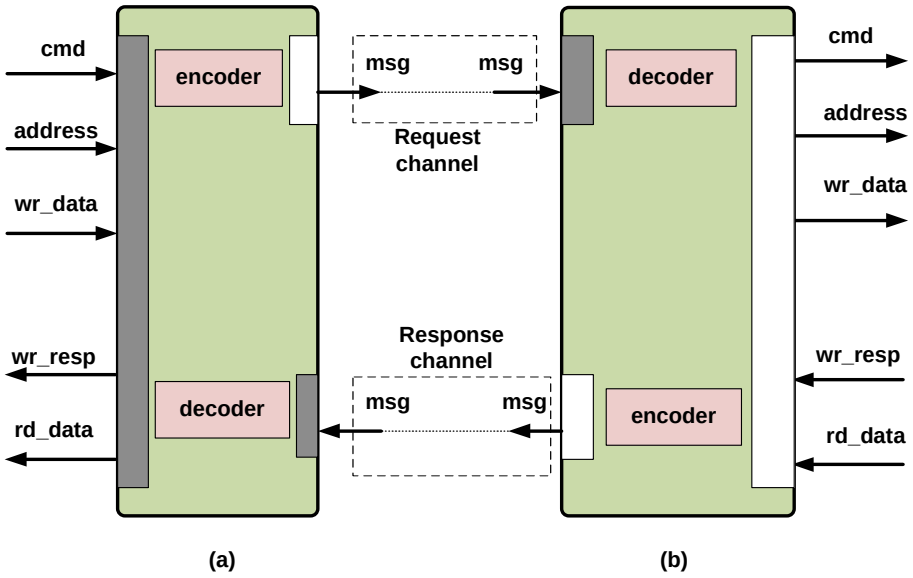


Figure 2.8: Architecture of (A) Master, and (B) Slave Network Interface Shell Architectures.

receiving the words, present them to the slave NI shell. The request decoder at the slave NI shell, does the opposite of the request encoder, i.e. it de-serialises the message words to indicate the attached IP about the appropriate read / write commands, flags, address and a possible incoming write data. For the response channel, a response that comprises write response, flags, and a possible read data is serialised by the response encoder of the slave shell to a response message. The response message after reaching the NI kernel of the master NI shell, is deserialised into appropriate flag and data, Figure 2.8.

Network Interface Kernel: Role

The Network interface kernel [121] is responsible to send / receive packets to / from a router and sending / receiving messages to / from shells. In the ÆTHEREAL architecture, a network interface kernel makes use of virtual point-to-point connections between the IPs to transport data over the router network. A connection constitutes two channels: one request channel and one response channel. We provide guaranteed services for the communication traffic. Therefore, each channel has its own quality of service (QoS), i.e., bandwidth and latency guarantees. To ensure the fulfillment of required QoS, appropriate re-

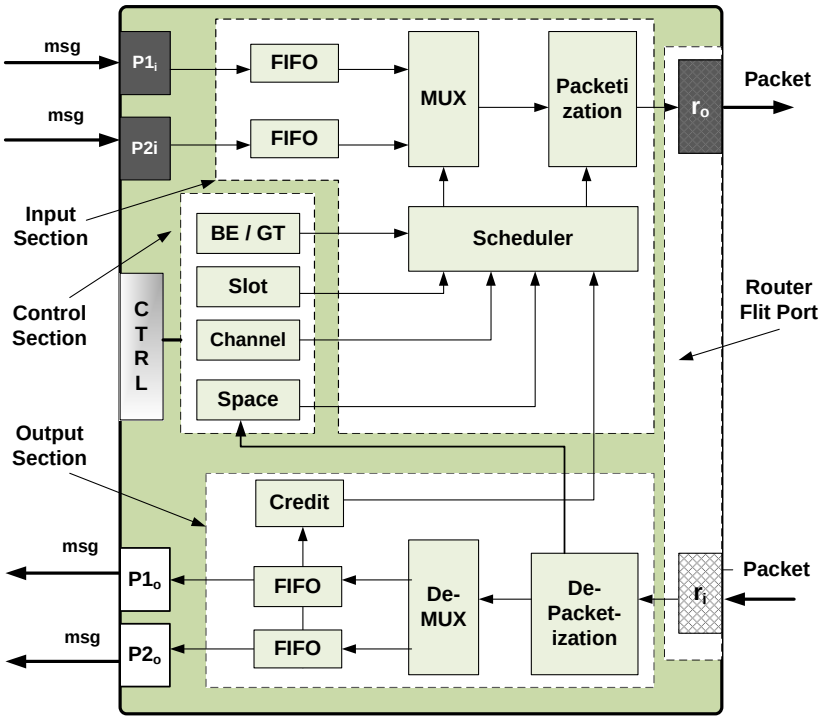


Figure 2.9: Network Interface Kernel Architecture.

source reservations are made in the source and the destination NI kernel, and over the router network. Each request / response channel uses two queues to store data, one in each of the two NI kernels. Hence for each connection, four queues in total are reserved in the source and the destination network interface kernels. End-to-end flow control is provided between the source and the destination NI kernel to avoid data loss and deadlock [51]. The flow control mechanism ensures that the source NI kernel never sends data over a request/response channel, unless there is enough space available in the corresponding queue of the destination NI kernel [121].

Network Interface Kernel: Architecture

The architecture of the network interface kernel can be classified into three sections, i.e., a control section, an input section, and an output section, see Figure 2.9. In the following discussion, we present the architectural details and inter-dependencies of these three sections.

The *control section* comprises a *memory-mapped port*, and three tables that include a channel table, a space table, and a slot table. The channel and space tables have one entry per network interface kernel port, whereas the slot table is of arbitrary but fixed length. The slot table is cyclically traversed by the scheduler to determine from which NI kernel port data should be injected into the router network. The channel table contains the path of each channel that its data traverse through the router network. The space table implements the credits for end-to-end flow control (credits to send back to other NI) between the NI kernels for connections.

The *input section* comprises multiple input ports and queues, a scheduler unit, and a packetisation unit. Each input port is associated with an input queue, and each input queue is associated with a channel. The input queue is used to store and forward data from a shell to the router network. The size of each queue is a design time parameter and is implemented, e.g., by using custom-made ASIC first-in first-out (FIFOs) [121]. In each time-slot, the scheduler checks if the data is available for the channel and there exist enough credits to send the data. After finding the data and credits, the scheduler forwards the data to the packetisation unit. In each time-slot the schedular can send 3 words (one flit) of data to the packetisation unit. The packetisation unit afterwards constructs a packet, which at maximum consists of 8 flits [48]. Importantly, the first flit of the packet contains a one-word packet header and a payload of two words. A packet header contains the path through the router network, the queue id at the destination NI, and number of credits for NI-to-NI flow control purpose.

The *output section* comprises a de-packetisation unit, multiple output queues and ports, and a credit table. The credit table, like space and channel tables, contains one value per port. A credit table value indicates the free buffer space available in a local output queue. The de-packetisation unit after receiving a packet flit, inspects the packet header to find out the remote buffer space. This information is used to update the space table. The scheduler at the source kernel, then takes into account this value before sending data words to the destination kernel. Apart from inspecting the packet header, the de-packetisation unit forwards the payload data to the appropriate output queue. From an output queue, data is consumed by the corresponding output port. As soon as data words are consumed by the output port, the credit table is updated with the number of credits that need to be delivered to the scheduler. These credits, after being piggy-backed in the packet header, are sent back to the sending NI through a reverse channel.

The input/output ports always appear in pairs and, hereafter in our discussions

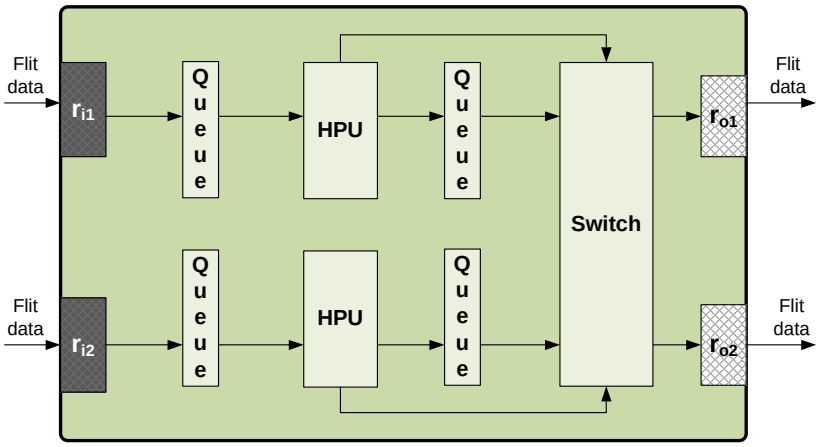


Figure 2.10: Router Architecture.

the term port will be used for the input / output port pair, e.g., P1 stands for both $P1_i$ and $P1_o$ in Figure 2.9.

Router: Role

A router in NoC architecture receives a packet, and based on the specified output port in the header forwards it to an attached NI kernel or (if any) to another router. The ÆTHEREAL NoC router [48] has fixed latency for the input data, and uses contention free routing. This in turn ensures an uncorrupted, lossless, and ordered data transfer within the router network.

Router: Architecture

The router [48] architecture has no routing table and no notion of time-division multiplexed (TDM) slots, Figure 2.10. Each input port of a router has one-word queue to store the input data. For each word of input data, the router induces a delay of three cycles. The synchronisation of the input data is performed at the first stage. The second stage makes use of a header parsing unit (HPU) to determine the output port based on the path encoded in the packet header. In other words an HPU associates an output port with an input port, and signals this to the cross bar switch. In the third pipeline stage the cross bar switch sends the data from the input queue to the selected output port. The selected output port forwards data from the input port until an End-of-Packet

(EoP) is encountered. Since routing is contention-free, no arbitration is performed, and packets never wait in router queues.

2.2.2 NoC Design Flow

The NoC design flow is shown in Figure 2.11, and it is comprised of three phases [48], which are explained below.

Design Time

At design time, the inputs are provided in the form of application and architecture specifications. The architecture specifications include interfaces for IPs, and NoC topology. The application specifications are provided in the form of connections. Each connection represents communication between two IP ports of an application, and has specific Quality-of-Service (QoS) requirements on the NoC interconnect. Additional application specifications include uses-cases that represent different combinations of applications that can run in parallel, and constraints on mapping an IP ports to a specific network interface.

A NoC architecture is generated, which is comprised of buses that bridge between network and IP ports, network interface shells, network interface kernels, routers, and links to connect NoC elements. Additionally, NoC resources are dimensioned in accordance with the input application(s). These include number of ports and queue sizes in network interface kernels, and decoder for buses. A control infrastructure, which is part of NoC, is also generated to program the generated NoC architecture (including buses).

Compile Time

In this phase, the physical resources of NoC are assigned to logical connections. As a first step the ports of IPs, which belong to a connection, are mapped to NI ports. It is followed by allocation of each connection, i.e., TDM slots in network interface kernels, and paths through the router network. The output of the compile time phase is an application to architecture binding. The binding here stands to map and allocate the applications all the use-cases.

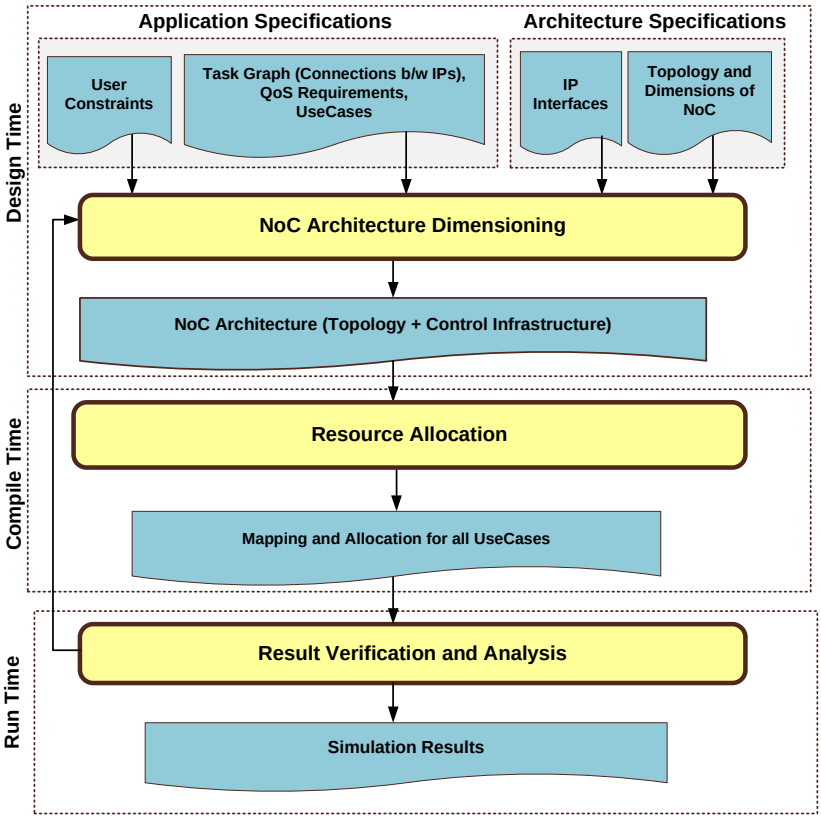


Figure 2.11: Aethereal NoC Design Flow.

Run Time

At run time, the performance of each application is verified. This is performed after computing the worst-case minimum throughput, maximum latency, and minimum buffer sizes per application connection. In this step, it is checked whether the resultant application to NoC binding can fulfil the required worst-case application(s) performance constraints. In case the application(s) constraints are not met, the design flow re-dimensions the NoC resources (buffers in particular). A cycle-accurate SystemC simulation is conducted to assess the average performance for a particular execution trace.

2.3 Conclusions

In this chapter, we provided the preliminary background about an FPGA architecture, which can be divided into; a logic plane to execute the required functionality, and a configuration plane to configure/program the logic plane (Section 2.1.1). Next, we explained the design flow for generating application bitstreams to be placed on a conventional FPGA architecture (Section 2.1.2). Afterwards, we explained the background of a conventional NoC architecture which is comprised of multiple network interfaces and routers (Section 2.2.1). We then provided the design flow for a conventional NoC to generate an application specific network on chip from design time specifications (Section 2.2.2).

3

Proposed Solution and Related Work

In this chapter we explain the architecture and the design flow of the proposed solution in Section 3.1. We then explain the techniques (Section 3.2, Section 3.3, Section 3.4, and Section 3.5) that make use of the proposed solution to fulfill the requirements, as mentioned earlier in Section 1.3 together with related work for each. In Section 3.6, we provide the concluding remarks of the chapter.

3.1 Proposed Solution: FPGA with Hardwired NoC

Our proposed solution is an FPGA with a hardwired network on chip (HWNoC). As shown in Figure 3.1, an FPGA with a hardwired NoC requires new architecture and design flow, because such a solution deals with both the FPGA and NoC architecture and design flow issues. Through this thesis, we provide the architecture and the design flow for such a solution. Importantly, the proposed solution fulfills SoC requirements that are mentioned in Section 1.3.

3.1.1 Proposed Architecture

The existing FPGA architectures [158, 161, 162] have a single test and configuration plane, and a single logic plane. Moreover, in existing FPGAs the IPs and the communication architecture (e.g., bus or NoC) coexist in the same logic plane. We propose innovations in the current FPGA architecture by (a) dividing it into multiple test configuration functional regions (TCFRs), and (b) embedding a hardwired network on chip (HWNoC) [41, 145]. Figure 3.2 shows an abstract view of our proposed FPGA architecture.

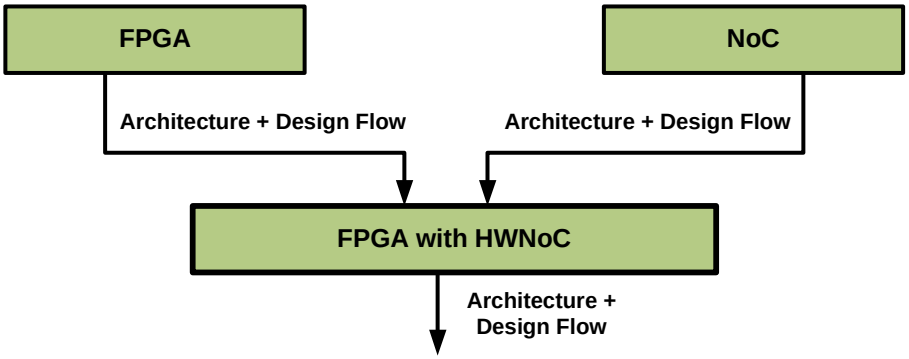


Figure 3.1: Abstract View of the Proposed Solution (FPGA with Hardwired NoC).

In our proposed architecture, each TCFR represents a unified test, configuration, and functional region. The hardwired NoC, which consists of routers and network interfaces, transports the unified test (Chapter 7), and configuration, programming, and functional data (Chapter 6) to TCFRs. The dimensioning of TCFRs in our proposed FPGA is application independent. Though the dimensions of TCFRs in our proposed FPGA are fixed at design time, but these can range from 1 k LUTs to 32 k LUTs and even more if required. It is important that the TCFRs are isolated at test and configuration levels, but are not isolated at functional level. This means, an IP can span fully or partially multiple TCFRs, as illustrated in Figure 3.2. Moreover, the logic plane of a TCFR is identical to the logic plane of a conventional FPGA, i.e., consists of look-up tables (LUTs), programmable switches, and wires.

There must be at least one IP that can program the system. This can be a CPU that bootstraps the system by programming the HWNoC, or a hard (secure) boot module [31, 145]. In our FPGA system, we make use of a *Control processor* to bootstrap the system, Figure 3.2. Additionally, the control processor can test, configure, and program the TCFRs by using the hardwired NoC.

More specifically we:

1. propose the architecture of the FPGA logic plane, which is a combination of multiple TCFRs.
2. propose the architecture of the FPGA communication plane, which is an on-chip hardwired NoC.
3. define the interaction between the two planes in terms of test, configuration, programming, and functional data.

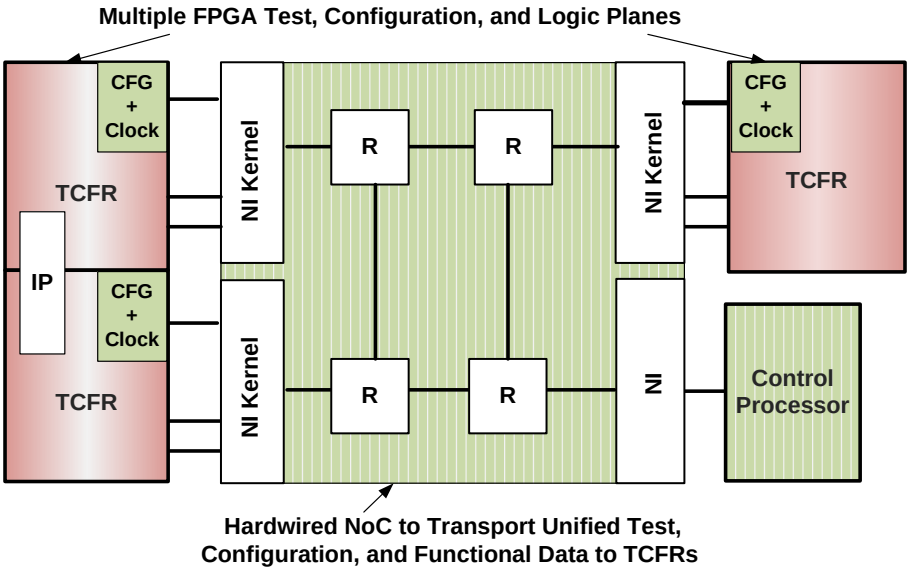


Figure 3.2: Architecture of the Proposed FPGA.

4. propose a hard and soft partitioning in both the FPGA planes.
5. provide hard / soft tradeoff in both the FPGA planes.

We describe the details of the above-mentioned FPGA architecture in Chapter 4.

Applications on the Proposed Architecture

In this section, we explain how IPs of an application are placed and communicate with each other in the proposed FPGA architecture. For the convenience of the reader, an abstract view of the proposed FPGA architecture is shown in Figure 3.3A. The proposed FPGA comprises multiple test configuration functional regions (TCFRs), where each TCFR consists of a logic plane, and a test and configuration plane. The configuration data (or bitstream) for an application is transported to any TCFR by using the unified hardwired network on chip. Each TCFR has its local configuration circuit that, after receiving the bitstream, (re)configures the required logic plane resources. It is important that the same hardwired network on chip is used to transport test data to test an application TCFRs. After configuration, application IPs are placed in the logic

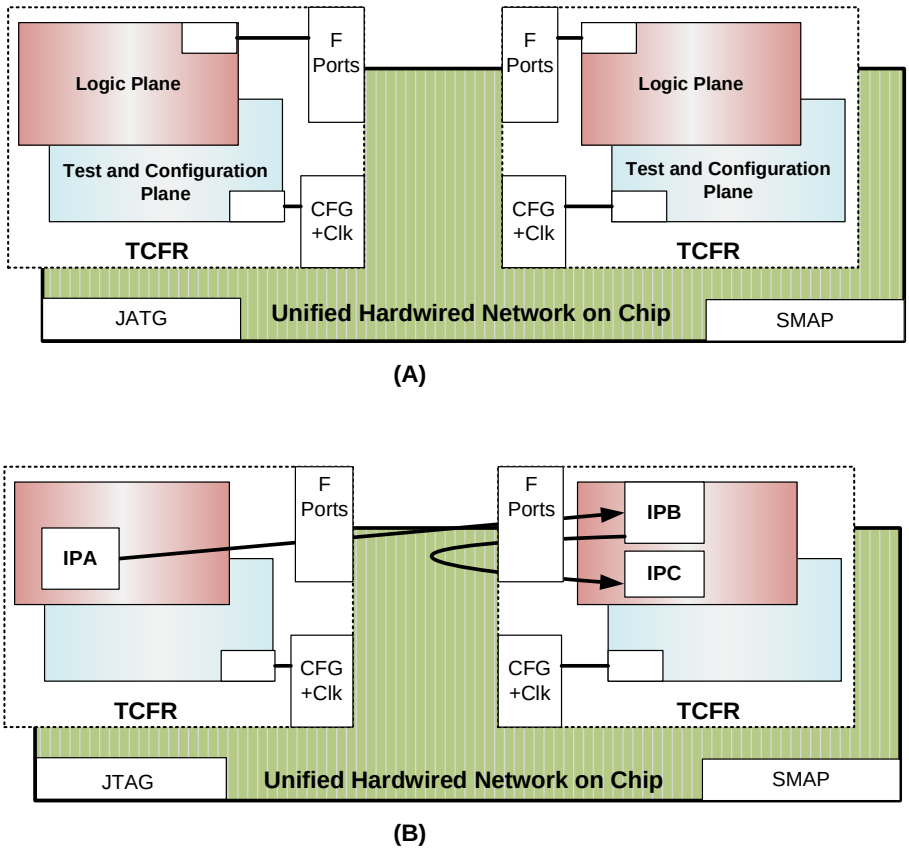


Figure 3.3: (A) Abstract View of the Proposed FPGA Architecture, and (B) Application on the Proposed FPGA.

plane(s) of TCFR(s). However, unlike the conventional FPGA, the associated functional interconnect does not exist in the logic plane because the same hardwired network on chip is used for inter-IP communication, i.e., to transport control and functional data for application IPs, as shown in Figure 3.3B.

3.1.2 Proposed Design Flow

Figure 3.4 shows that our design flow is comprised of three phases, which are explained below.

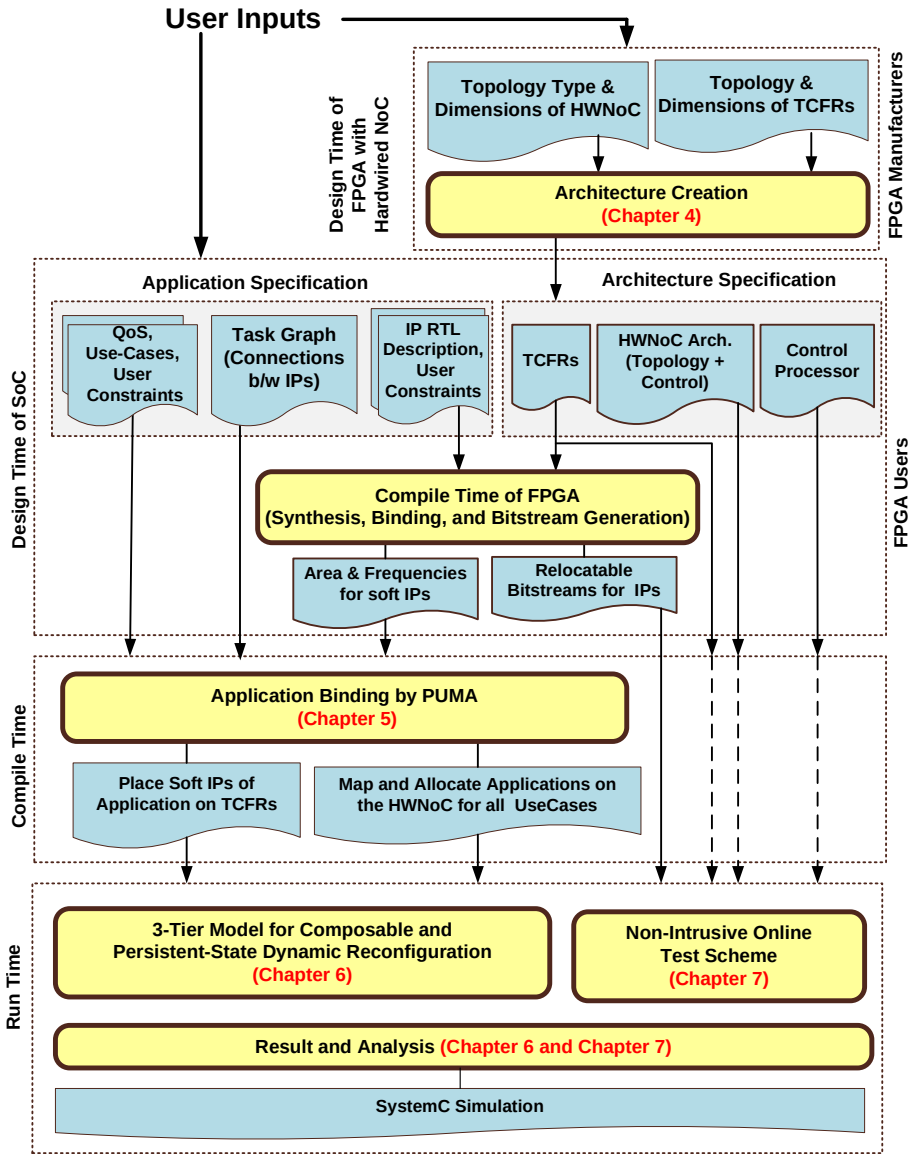


Figure 3.4: Our Design Flow for the Proposed FPGA Architecture.

Design Time

The design time specifications of FPGA with hardwired NoC can be viewed from two point of views: (a) FPGA manufacturers, and (b) FPGA users.

The design time input specifications for FPGA *manufacturers* include type and dimensions of hardwired NoC topology, size of each TCFR, total TCFRs, hardwired NoC to TCFR mapping (i.e., how many NIs are attached to a TCFR), and control processor. The inputs are fed to an architecture creation tool, which generates the architectures of TCFRs and hardwired NoC. The control processor contains different ports to configure and program the TCFRs (Chapter 6), and test the TCFRs (Chapter 7).

The design time input specifications for FPGA *users* include RTL description of IPs and user constraints. It is important that the *specifications of each IP* consists of: (a) RTL description that indicates IP functionality, and (b) RTL description of IP network interface shells (usually from a standard protocol lib, e.g., for AXI, DTL). The user constraints define timing constraints on intra-IP level and not on inter-IP, because inter-IP communication is taken care of by the hardwired NoC. The inputs are fed to synthesis, binding, and generate bitstream tools. As a result of which the relocatable bitstreams for the target FPGA architecture are generated. Therefore, in our solution, the compile time flow of the conventional FPGA has become a part of design time flow.

Compile Time

At compile time the outputs of design time are fed to *an application to* FPGA *binding* tool. The tool is named PUMA that unifies the placement, mapping, and allocation while binding an application to FPGA, Chapter 5. For this purpose, the architecture and application specifications are provided. The architecture specifications include TCFRs and hardwired NoC architecture. Application specifications are provided for all the applications in SoC. For each application the specifications include: (a) area and frequencies of soft IPs, (b) connections between the IPs, (c) Quality-of-Service constraints for each connection, (d) use-cases of application, and (e) (optional) user-constraints. The PUMA tool tries to bind SoC applications on FPGA by: (a) placing IPs in TCFRs, (b) mapping IP ports to NI kernels, (c) allocating TDM slots and paths through the network. In case the binding fulfills QoS constraints for all input applications, then the output is generated in the form of application to architecture binding.

Run Time

The run time part of the design flow can be used to dynamically (re)configure and program an FPGA to execute applications, Chapter 6. The hardwired NoC and TCFR architecture takes care of composability at inter-application level and keeps persistent-state at intra-application level during the dynamic reconfiguration process. We can test an FPGA architecture at run time, as described in Chapter 7. The run-time design flow ensures that the test process does not produce intrusiveness with other running applications. At run time, we can also verify and analyse the procedure and results by running SystemC simulations.

Our proposed design flow provides:

1. For FPGA manufacturers, a design-time flow to generate an FPGA architecture with a hardwired NoC.
2. For FPGA users, a compile-time flow to bind input applications to FPGA architecture (Chapter 5).
3. For FPGA users, a run-time flow for;
    - composable and persistent-state dynamic reconfiguration of applications (Chapter 6),
    - a non-intrusive online test scheme to ensure that applications execute on a reliable FPGA architecture (Chapter 7).

In the remainder of this chapter, we discuss our techniques (define in Section 1.4) that use the proposed solution (FPGA with hardwired NoC) to fulfill the requirements of Section 1.3.

3.2 Technique: Hardwired Network on Chip

In this section, we present the overview and motivation for the hardwired NoC. Afterwards, we provide the related work, and at the end position our technique with respect to the existing state of the art.

3.2.1 Overview

The HWNOC, due to its embedded nature, has a *fixed topology*, i.e., its dimensions can not be changed at run time unlike a soft NoC. The HWNOC makes use of *connections* that are programmable at run time, to transport data (control and functional data) in between the IPs. In addition, by using a hardwired NoC we present the *single-platform-serves-all* concept. This means, along with transporting the functional and control data of applications, the hardwired NoC can transport time-multiplexed configuration data (Chapter 6), and test data (Chapter 7) as well. Application IPs that are placed in TCFRs, however, still make use of switch-boxes and wires for intra-IP communication.

3.2.2 Motivation

We use a motivational case study to explain how a soft functional interconnect, which is stretched in FPGA logic plane, can impose restrictions in the placement of IPs. For the motivational case study, we consider a SOC with 2 applications and an FPGA with 24 CLBs (or 6 CLB columns), Figure 3.5A and Figure 3.5B respectively.

The *soft* functional interconnect is placed in the center columns of FPGA and all application IPs are placed except IP1, which has an area requirement of 6 CLBs, Figure 3.5C. The FPGA meets the area requirements of IP1, because it has free area equals to 6 CLBs. However, the FPGA area is fragmented into two regions, due to which IP1 can not be placed in the FPGA. Alternatively, IP1 can be placed but after being partitioned into two smaller IPs (IP1a and IP1b), as shown in Figure 3.5D. Another option is to redesign the soft functional interconnect to place IP1, as shown in Figure 3.5E. However, this involves the reconfiguration of the soft NoC to accommodate IP1. This disrupts the execution of other running application, i.e., A2, because during the reconfiguration time the soft NoC is unavailable for inter-IP communication.

In contrast, the hardwired NoC does not exist in the FPGA logic plane. This means that a hardwired NoC decouples inter-IP communication from computation. The IPs, therefore, do not face placement restrictions because of the functional interconnect. In addition, the hardwired NoC can be used to transport unified data, i.e., test, configuration, programming, and functional data.

In the following section, first we describe the related work, then we classify and compare our our work with respect to the related work.

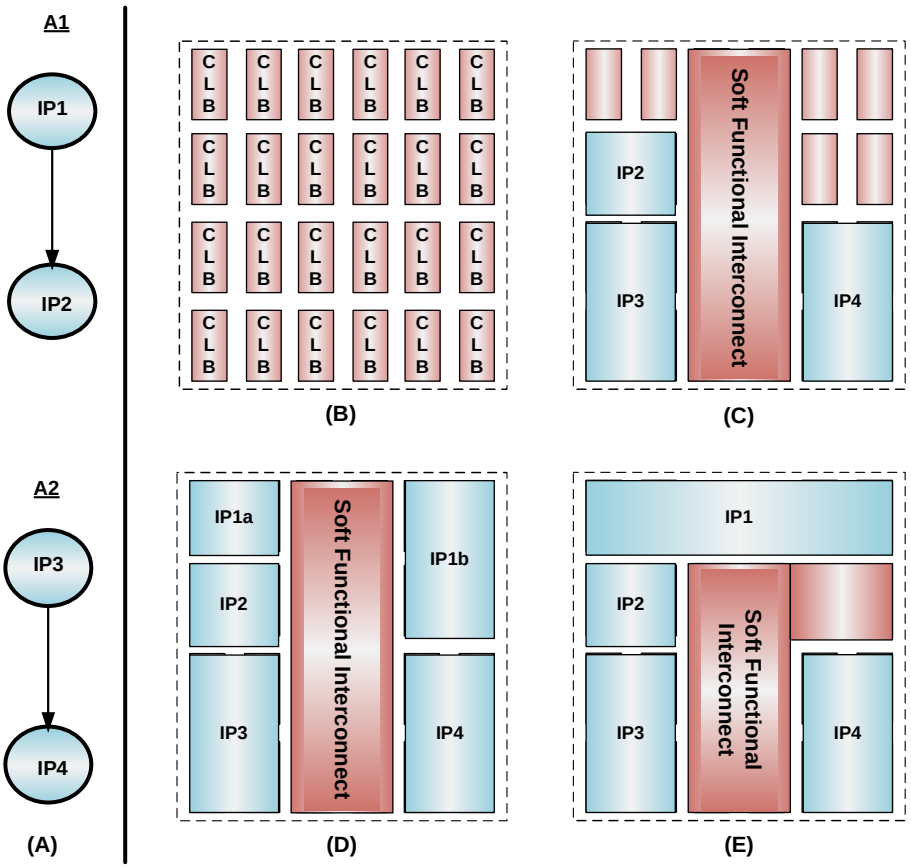


Figure 3.5: Restricted IP Placement due to the Presence of Soft Functional Interconnect.

3.2.3 Related Work on Conventional FPGA with Soft & Hard Interconnect

Our FPGA architecture comprises a hardwired NoC and multiple TCFRs, by using which we can have: (i) decoupled communication and computation to achieve scalable IP integration, (ii) unified data transportation, and (iii) multiplexed operations, i.e., configuration and test data for multiple regions is transported, simultaneously. Therefore, we will investigate the related reconfigurable architectures from these three above-mentioned aspects. The literature survey is performed for schemes that use conventional FPGAs (e.g., Altera, Xilinx) as their reconfigurable logic plane.

Conventional FPGA with Soft Interconnect

Scores of soft interconnects ranging from point-to-point [63], busses [16, 125], cross-bars [21, 62], indirect interconnection networks ,e.g., fat-tree based approaches [29,30], to NoCs [56,78,88–90,92,106,111,149] have been presented in the literature.

These implement different cost:performance trade-offs. Larger and faster FPGAs [162] can contain many IPs and multi-hop network solutions are sure to gain popularity. [16] presents four communication schemes, including bus macros, shared memories, linear array multiple bus, and external crossbars that used for different communication modes. In [61], a reprogrammable interconnect is implemented based on a LUT-based bus macro and is used to dynamically reconfigure the attached IPs. In [21, 164], a large (928 928 bits) crossbar using native programmable interconnects and LUTs is presented.

The authors in provide an innovative way to implement a flexible interconnection architecture. They name it DRAFT, i.e., Dynamic Reconfiguration Adapted Fat-Tree, architecture that is implemented as a central column into the FPGA. A number of routers are used to connect the various processing elements in DRAFT. The authors claim that DRAFT needs fewer resources, e.g., communication links, than a mesh and a fat-tree. In addition it also experiences lower average latency than mesh and a fat-tree topology.

Integration of IPs at run time is achieved by using dynamic partial reconfiguration. For a scalable IP integration, the functional interconnect and IPs must be disjoint, otherwise they must be reconfigured simultaneously. By reprogramming the functional interconnect, IPs can be dynamically added and removed through partial reconfiguration, e.g. [16, 61, 89, 90, 125]. Only few works reconfigure the soft interconnect itself. [63] dynamically reconfigures a point-to-point soft interconnect and [111] shows how a soft NOC can be partially reconfigured by adding or removing router modules at run time.

Conventional FPGA with Hard Interconnect

Only two groups have reported on hard NOCs in FPGAs [25, 39, 53]. The authors in [53] make use of a packet-switched hardwired NoC to reconfigure an FPGA, i.e., only as configuration interconnect. The hardwired NoC serves as an additional high-level routing resource. The authors in [25, 39] propose a hierarchical architecture for future FPGAs, consisting of two types of regions connected by a NoC: (1) Configurable Regions (CR) consisting of

Table 3.1: Our Work Positioning with respect to the State of the Art on Traditional FPGAs.
Abbreviations: Comm. = Communication, Comp. = Computation, Ops. = Operations, xbar = Crossbar, NA = Not Applicable, T = Test, O = cOnfiguration, C = Control, F = Functional. For example T, O, CF means three separate interconnects are used: one for T, one for O, and one for C & F.

Scheme	Comm. Inter-connect	Separate Comm. & Comp.	Unified Data	Parallel Test & Config.
Hur [63]	*pt-to-pt*	No	T, O, CF	No
Bobda [16]	*bus*	No	T, O, CF	No
Sedcole [125]	*bus*	No	T, O, CF	No
Brebner [21]	*xbar*	No	T, O, CF	No
Devaux [29, 30]	*Fat-Tree*	No	T, O, CF	No
Hur [62]	*xbar*	No	T, O, CF	No
Young [164]	*NA*	No	NA	No
Huebner [61]	*NA*	No	NA	No
Marescaux [88]	*NoC*	No	T, O, CF	No
Nikolov [106]	*NoC*	No	T, O, CF	No
Wee [149]	*NoC*	No	T, O, CF	No
Hecht [53]	*hard NoC*	Yes	T, O, CF	No
Cidon [25]	*hard NoC*	Yes	T, O, CF	No
Gindin [39]	*hard NoC*	Yes	T, O, CF	No
Our Technique [41, 145]	*hard NoC*	Yes	TOCF	Yes

programmable logic, (2) and Functional Regions (FR) to perform a predefined task, e.g., general-purpose processors, DSP units, fast external interfaces, etc. The regions are interconnected using a NoC. Given the hierarchical chip organisation, the design methodology that programs such FPGA consists of following phases: (i) division into high-level modules of roughly the size of CRs; (ii) placement of high-level modules; (iii) implementation of each module within a region; (iv) and inter-region routing.

3.2.4 Positioning with the State of the Art

In this section, we position our work with respect earlier presented related work.

Traditionally, point-to-point [63], (non-) segmented buses [16, 125], crossbars [21, 62], and indirect interconnection fat-tree based network [29, 30] have been used for data path interconnection because of their simplicity. However, these posses a non-scalable nature and therefore, *unable to meet QoS requirements*, as the system sizes grows. For example in [21, 164], even a large (928 x 928 bits) crossbar switch can connect with few IPs, using standard IP communication protocol such as AXI [9] which requires several hundred of wires per IP. Though soft network-on-chips [88, 106, 149] present a scalable solution to integrate IPs, but they consume a *large amount of* FPGA *area*. Hence leaving less logic area for the IPs. Additionally, their presence *restricts the placement of IPs*. In contrast to above solutions, we propose to use a hardwired NoC for inter-IP communication.

The concept of hardwired NoC for inter-IP communication has also been proposed by [25, 39]. The work of [25, 39] proposes to use a hard NoC as the functional interconnect. Although the basic idea of a hard NoC is introduced, no architecture details are provided. Our work differs from [25, 39] in the following. Foremost, we combine test, configuration, control, and functional data interconnects in the same hard NoC, which has not been proposed by any prior work. As illustrated in Table 3.1 (Column 4), the prior works can use the same communication architecture to transport control and functional data, i.e., support unified control and functional data transportation. However, test and configuration data is not transported by using the same communication architecture. As we shall explain in Chapter 7 that by using the hardwired NoC we can implement parallel operations of configuration and test for multiple TCFRs. However, this has not been exercised by any of the prior works as shown in Column 5 of Table 3.1.

Furthermore, our partitioning of the network interface (NI) in hard and soft regions draws the distinction more clearly at the network versus transport layer (see Section 4.5). For example, we firmly place routing in the hard NI kernel domain. Moreover, in [25] the architecture is based on tiles (functional regions), which can communicate only through the NoC. As we shall explain in Section 4.3, we do not partition an FPGA in distinct functional regions and also keep the functional regions orthogonal to the configuration regions. Next, we define the (re)configuration and (re)programming steps required to boot a system in Chapter 6. The requirement for guaranteed communication services (GS) to support real-time streaming of bitstreams is not met by their NoC (Chapter 6).

3.2.5 Related Work on Custom Reconfigurable Architectures

In this section, the literature survey is performed for schemes that use custom reconfigurable architecture, and whose basic logic element is different from the conventional FPGAs. A number of custom based reconfigurable architecture have been proposed. We cover the coarse grained reconfigurable architecture to seek a comparison with our proposed approach in terms of abovementioned requirements, i.e., separate communication and computation, unified data communication, and parallel test and configuration approach. In the following discussion, we will briefly outline each of these architectures, individually.

Reconfigurable architecture workstation (RAW) [137] is a *coarse grained architecture*. The RAW architecture is aimed at exploiting different kinds of parallelism at instruction and at data levels. The RAW architecture is comprised of multiple tiles arranged in a 2-D mesh topology. Each tile contains a MIPS R2000 microprocessor, ALU, different registers, a dynamic router, and a programmable switch. A *network on chip is used to provide inter-tile routing*, which connects each tile to its four neighbors. Tilera could be another similar example that reaches 100 processing cores on a single chip [72, 139].

The *Colt machine* [14] is a coarse grained architecture, and uses a distributed reconfiguration mechanism called *Wormhole Run Time Reconfiguration* (RTR). By using Wormhole Run Time Reconfiguration *multiple data ports can be used to program different sections of the chip*, simultaneously. In Wormhole RTR system a stream, used for reconfiguration, is an independent self-steering concatenation of programming information. On the other hand, the operand data, which interacts with other streams within the architecture, performs a given computational problem. From an architecture viewpoint, the work uses *stream controllers to allocate and then program a path through the chip* by using one of the data ports. The stream of data proceeds through the *crossbar*, and then to either the mesh of Functional Units (FUs) or to the multiplier. The stream can then go back through the crossbar to any part of the chip or off chip via a data port.

Morphing System (MorphoSys) [131] is a coarse grained architecture. The main component of MorphoSys is the *Reconfigurable Cell* (RC) array. It has 64 RCs arranged in a 2D mesh of 8 by 8 dimensions. The RC array is further divided into quadrants of 4 by 4 16 bit RCs. Each RC features an ALU, multiplier, a register file, and a 32 bit context register to store the configuration word. The RC *interconnection network is comprised of three layers*. 1) The underlying network throughout the array is a 2D-mesh. This provides nearest-

neighbor connectivity. 2) Then comes Intra-quadrant connectivity by using which each cell can access the output of any other cell in its row (column). 3) At the highest or global level, there are *buses to route connections* between adjacent quadrants. These buses are called as express lanes and run through rows as well as columns.

PipeRench [40] is a coarse grained architecture that is aimed to stream multimedia applications. It is comprised of a number of 8-bit processing elements (PEs), where each PE consists of an 8-bit ALU, a pass register file, and interconnection resources to communicate with other PEs. A *row of PE's create a strip* within the architecture and strips are stacked on top of each other. Each strip consists of 16 PEs. The interconnection scheme of PipeRench features *local interconnect inside a strip*, as well as *local and global interconnect between strips and four global buses*.

The GARP [52] architecture is a fine grained architecture. It combines a standard MIPSII processor with a dynamically *reconfigurable array*. GARP reconfigurable array is composed of entities called *blocks*. One block on each row is known as a control block. The rest of the blocks in the array are logic blocks, which roughly corresponds to the CLBs of Xilinx 4000 series. The GARP architectures fixes the number of columns of blocks at 24. The number of rows is implementation-specific, but can expected to be at least 32. The 24th column of control blocks is dedicated to managing communication outside the array. For fast reconfigurations, the RA features a distributed cache with depth 4, which stores the least recently used configurations. The architecture proposes a *direct connection between the reconfigurable array and the memory*.

TABULA's [140] Time Machine is a fine grained architecture, which gives the concept of a *three-dimensional chip*. It appears to be an FPGA/PLD with upto eight stacked layers of physical logic, memory, and interconnects. However, in reality there is only one physical layer. Unlike FPGA device that can perform operations once per use clock cycle, the Tabula device can *reconfigure and perform eight operations in the same user clock cycle*. Beginning at fold 0, the logic operations ar performed on the data. At the end of fold 0, resources are modified for the next fold (fold 1), according to settings stored in configuration memory, and logic operations are performed. This process continues through all the folds, each time with the resource being modified according to settings stored in the configuration memory. At the completion of fold 7, the entire user function has been performed, and the configuration of fold 0 is used again to modify the resources for the next user clock cycle.

The Plastic Cell Architecture (PCA) [66] is a fined grained architecture, which *unifies the transportation of data and bitstream*. It is composed of cells, where each cell consists of two parts: a variable part as the place holder of new configuration (*Plastic*), and a fixed part which is responsible to set the former (*Built-in*). Built-in part configures plastic part according to given configuration data. Plastic part has memory units, and Built-in part writes/read the configuration data to/from these memory cells. For this purpose, configure-in and configure-out instructions are used. This architecture does not have support for parallel reconfigurations.

Triptych [19], is proposed as a new FPGA architecture, for those circuits that *include both data-path elements and control logic*. Triptych treats the functional and the interconnect elements as a single unit. Hence it *blends logic and routing resources in a routing logic block* (RLB). A Triptych RLB is capable of performing both function calculation and routing tasks simultaneously. The RLB array is structured to match the inherent fanin/fanout tree structure of circuit graphs. It allows the physical layout of a mapped circuit to follow its logical structure, reducing the need for extensive routing resources. Triptych falls somewhere between general-purpose and domain-specific FPGAs. For applications with local communication, RLBs are used primarily for logic, allowing implementations that are competitive with domain-specific FPGAs. For more general applications such as FSMs, a higher percentage of RLBs are used to route signals.

3-D FPGA [148] is a fine grained architecture. It is a dynamically reconfigurable FPGA that is *comprised of three layers*: i) the routing and logic block (RLB) layer, ii) the routing layer (RL), iii) and memory layer (ML). The RLB layer is responsible tto implement logic functions and to perform limited routing. The remaining part of the routing structures is implemented in the RL that is formed by connecting multiple switch boxes in a mesh array structure. The memory layer is used to stored configuration bits for both the RLB and RL.

3.2.6 Positioning with the State of the Art

In this section, we position our work with respect to earlier presented related work.

The works of [14, 66, 148] *physically decouple computation and communication planes*. Therefore, can implement scalable IP integration. However, the remaining schemes [19,40,52,131,137,140] implement computation and communication in the same plane. Hence inducing restrictions in the integration /

placement of IPs.

As far as *unified data transportation* is considered, none of the earlier mentioned schemes [14,19,40,52,66,131,137,140,148] can transport all four types of communication traffic (test, configuration, control, and functional data) by making use of the same communication interconnect, see Table 3.2 (Column 4). Only two schemes [14, 66] support unified transportation of three types of communication traffic, i.e., configuration, control, and functional data. However, our architecture, in addition to configuration, control, and functional data, can transport test data as well. The architecture of [66] proposes unification of configuration, control, and functional data by introducing cells, but no framework is presented to implement it. Moreover, to exercise the above concepts on modern FPGAs, a logic cell architecture would have to be altered. In comparison, we do not propose to alter it and can still implement unified data transportation for all of test, configuration, control, and functional data.

The schemes of [14, 19, 40, 52, 66, 131, 137, 148] do not provide support for *parallel test and configuration operations*. Although Tabula's [140] time machine can provide multiple reconfigurations at one time. However, to enable this ultra-rapid reconfiguration, configuration data is stored locally to the required resources. This local configuration memory is made to look like a stack. As each configuration is read from the top of the stack, the next configuration rises to the top. The current configuration goes to the bottom of the stack. This process repeats continuously. In contrast, we just need one configuration memory. The control processor uses multiple ports to perform multiplexed operations, i.e. configuration and test data are transported to multiple regions (TCFRs) simultaneously.

3.3 Technique: Binding of Applications to FPGA

In this section, we provide the overview and motivation for our technique to achieve the required solution of application to FPGA binding. We then provide the related work, and at the end position our technique with respect to the state of the art.

3.3.1 Overview

Our proposed scheme performs a unified placement, mapping, and allocation (PUMA) of applications to FPGA by:

Table 3.2: Our Work Positioning with respect to the State of the Art on Custom Reconfigurable Architectures.
Abbreviations: Comm. = Communication, Comp. = Computation, Ops. = Operations, xbar = Crossbar, NA = Not Applicable, T = Test, O = cOnfiguration, C = Control, F = Functional. For example T, O, CF means three separate interconnects are used: one for T, one for O, and one for C & F.

Scheme	Comm. Inter-connect	Separate Comm. & Comp.	Unified Data	Parallel Test & Config.
RAW [137]	*network*	No	T, O, CF	No
Colt Machine [14]	*xbar*	Yes	T, OCF	Yes
MorphoSys [131]	*buses*	No	T, O, CF	No
PipeRench [40]	*buses*	No	T, O, CF	Yes
GARP [52]	*buses*	No	T, O, CF	No
TABULA's [140]	*NA*	No	T, O, CF	Yes
Plastic Cell [66]	*NA*	Yes	T, OCF	No
Triptych [19]	*custom*	No	T, O, CF	No
3-D FPGA [148]	*custom*	Yes	T, O, CF	No
Our Technique [41, 145]	*hard NoC*	Yes	TOCF	Yes

1. using a hardwired NoC as the communication plane,
2. minimizing application dependencies by decomposing it into fully / partially independent clusters,
3. transforming binding (i.e., placement, mapping, and allocation) into a single problem, while selecting an FPGA region.

1) HWNoC as communication plane: Our proposed PUMA scheme ensures that for a successful application to FPGA binding, application QoS constraints are fulfilled. PUMA uses a hardwired NoC as the communication plane. In comparison with a soft NoC, the hardwired NoC (i) does not occupy FPGA logic area and exhibits higher scalability, and (ii) poses 148 times better cost:performance ratio (Section 4.9.6).

2) Application decomposition: Prior to performing the binding of an application, the application is decomposed into multiple *clusters*. The clusters are created by exploiting inter-IP communication dependencies, because inputs of an IP can be dependent on the output of some other IP(s). In other words, each cluster represents inter-communication dependencies across a group of IPs.

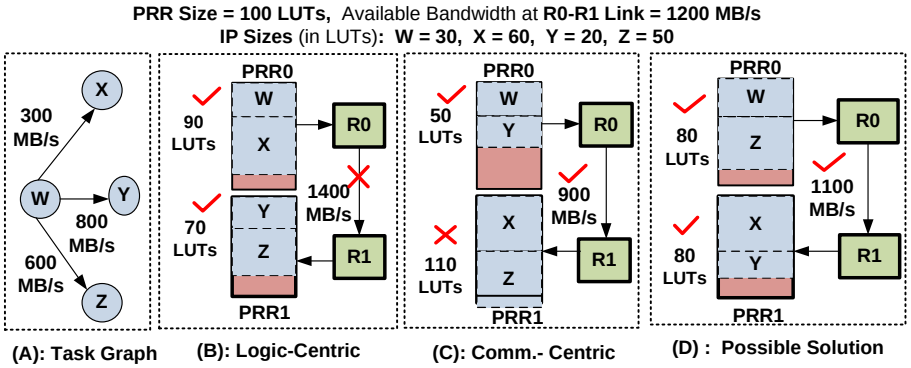


Figure 3.6: Motivational Case Study for Unified Placement, Mapping, and Allocation.

This is done because applications are becoming increasingly complex [127]. Therefore, picking an application as a whole and try every possible binding possibility becomes highly time consuming. Alternatively, there are different schemes [112] that can allow cluster or level-based traversal of an application task graph. Our PUMA scheme goes for the *cluster-wise decomposition* of applications.

3) Transforming binding into a single problem: For each cluster, PUMA takes into account the required application resources and available FPGA resources in both the logic and communication planes, simultaneously. Hence the temporal (time-slots) and the spatial (logic CLBs, communication links) constraints of an application are considered simultaneously, while selecting an FPGA region.

3.3.2 Motivation

Let us motivate the need for a unified placement, mapping, and allocation scheme by means of a simple case study. We have an application to be bound to two partial reconfigurable regions (PRRs), Figure 3.6A. The PRRs are connected to a NoC. Single or multiple IPs can coexist in a single PRR, provided these do not exceed a PRR area, i.e., 100 LUTs. An NoC link, which provides a maximum of 2000 MB/s bandwidth, can be shared by multiple applications. In our case study, 40% of the R0-R1 link is utilized by other existing applications (not shown for the simplicity) of the system. Therefore, the R0-R1 link has a spare bandwidth of 1200 MB/s for our *test* application.

As shown in Figure 3.6A, the test application comprises an IP *W*, which serves

as the source for the remaining three IPs, i.e., *X*, *Y*, and *Z*. As a starting point, we place *W* in PRR0. As shown in Figure 3.6 (B, C, and D), three different solutions are possible to select the next IP. Figure 3.6B shows a logic-centric binding solution, which opts for *W* and *X* in the same PRR0 to produce the least possible fragmentation in it. Afterwards, *Y*, and *Z* are placed in PRR1. However, it can cause an application to under perform, because *W* does not get the required throughput on R0-R1 link to communicate with *Y* and *Z*. Figure 3.6B shows a communication-centric binding solution, which opts for *W* and *Y* in the same PRR0 to restrict the highest throughput connection from going to network, i.e., to shared link R0-R1. However, due to excessive fragmentation in PRR0, binding of the complete test application is not possible in this case.

Figure 3.6D shows a successful solution for the test application binding. Before selecting an IP that is placed next to *W*, the temporal and the spatial constraints (of both application and FPGA) are evaluated. However, doing so is complex, since that the logic-centric binding alone is an NP-hard problem [99]. The problem becomes more complex, when an application exhibits; a) more IPs with more dependencies, b) and stringent QoS constraints.

In the following sections, first we describe the related work, then we classify and compare our our work with respect to the related work.

3.3.3 Related Work

In existing state of the art schemes, the authors have performed application to architecture binding such that certain metrics of interest (e.g., energy, execution time, area, throughput, latency etc.) are optimised/fullfilled. To perform an application binding, the existing schemes either unify the processes of mapping and placement, unify the processes of mapping and allocation, or do not unify any of the processes (i.e., placement, mapping, and allocation). Hence the techniques of existing schemes can be classified into: a) unified place and map, b) unified map and allocate, c) and non unified. In the remaining discussion, we will illustrate the existing schemes. Then, we position our PUMA scheme with respect to the techniques applied by the existing schemes to achieve a successful application binding.

Unified Place and Map

There are multiple schemes [4, 17, 99, 133] that are based on unified placement and mapping. In [4], a physical planner is used during topology design to reduce power consumption on wires. However, the work does not consider the area and power consumption of switches in the design. Also, the number and size of network partitions are determined manually. Scheme [17] on the contrary, treats each IP as a component. The IP encapsulates a circuit implemented with the resources in a given area (routers logic and IP). After an IP is placed, the IP coordinates are set to that of its router. The work in [99] takes into account the physical planning issues, while mapping an application IP on the communication plane, i.e., a network on chip. For this purpose, a floor-planner is used during the mapping process to get area and wire-length estimates. For a given mapping, the relative position of the cores with respect to each other is obtained from the tabu search, but the relative position of the switches is unknown. The authors applied Mixed Linear Integer Program [79] based physical plan algorithm, to place the IPs. Ideally, a switch should be placed around the core to which it is connected. This is achieved by inserting the switches into the vacant regions, which were left out after placing the application IPs. Once the switches are in place, the IPs are mapped. It is followed by routing and allocating the communication channels among the application IPs. In [133], a slicing tree based floor-planner is used during the topology design process. This work assumes that the switches are located at the corners of the cores, and it does not consider the network components (switches, network interfaces) during the floor-planning process. Also, deadlock free routing, which is critical for custom NoC designs is not supported in the work.

Unified Map and Allocate

The works in [50, 83, 100, 104] apply unified mapping and allocation to ensure QoS guarantees. The authors in [50, 83, 100] incorporate application binding into path selection, while aiming to minimize the over-allocation of the network. These schemes use a network on chip as the communication plane. In these works, mapping and allocation are tightly integrated. This is done on a per communication flow basis. Once the resource allocation for the flow is ensured across the communication plane, the ports of IPs, which are associated with that communication flow, are mapped at respective network interfaces of the NoC simultaneously. The authors in [104] analysed the average of heterogeneous NoC network latency in terms of the queuing latency. They ap-

plied the branch-and-bound algorithm to find out the QoS aware mapping that automatically maps IPs onto NoC architecture, while compromising between throughput maximization and latency minimization.

Non Unified Schemes

The works in [58,70,87,101,102,128] are based on non unified place, map, and allocate processes. In [58] authors propose a branch-and-bound based scheme to bind a given set of application IPs on a generic regular NoC architecture. The binding process not only satisfies the required bandwidth constraints, but also minimises the total communication energy for the resultant application binding. The work in [70] uses two heuristics, a fast successive relaxation and a genetic algorithm to find mapping over network with irregular topology. Authors in [87] map an application on a regular mesh NoC by using a genetic algorithm, with an objective of minimized execution time. Similarly, authors in [102] uses a heuristic algorithm (NMAP) under different routing functions. This algorithm maps the core onto NoC architecture after fulfilling bandwidth constraints and at the same time optimising the average communication delay. This technique uses the average packet hop value as a cost function and relates it to the communication energy consumption. In [101], the authors also propose an mapping and deterministic routing algorithm called SUNMAP that is similar to the NMAP. The difference is that this system can automatically choose the NoC topology from the ones embedded inside it. In [128] a Binomial Mapping (BMAP) method is introduced, which aims to minimize total traffic on network, the number of hops, and hardware costs.

In *non unified schemes* [70, 87, 101] real-time guarantees on an application QoS requirements are not ensured. Moreover, authors in [101] perceived application to architecture binding as NP-complete quadratic assignment problem (QAP). The scheme binds the tasks onto a regular mesh network under bandwidth, area or power constraints. It is followed by routing the communication by using a predefined routing function. Although in [58, 102] QoS constraints are fulfilled, but after iteratively refining the mapping and routing of application IPs.

3.3.4 Positioning with the State of the Art

Table 3.6 positions our proposed PUMA scheme with respect to the existing state of the art. In Table 3.6, the comma is used to show the non unified nature of a specific binding decision from the rest.

Table 3.3: Our Work Positioning with respect to Existing Application Binding Approaches.
Abbreviations: P = Placement, M = Mapping, A = Allocation, + = With, - = Without. For example PM, A means: P & M are unified, and A is non-unified from both P and M.

Scheme	Application Binding Approach
Ahonen [4]	(PM , A) + QoS
Bobda [17]	(PM , A) + QoS
Murali [99]	(PM , A) + QoS
Srinivasan [133]	(PM , A) + QoS
Hansson [50]	(P, MA) + QoS
Kumar [83]	(P, MA) + QoS
Murali [100]	(P, MA) + QoS
Nguyen [104]	(P, MA) + QoS
Jang [70]	(P, M, A) - QoS
Lukovic [87]	(P, M, A) - QoS
Murali [101]	(P, M, A) - QoS
Hu [58]	(P, M, A) + QoS
Murali [102]	(P, M, A) + QoS
Our Technique [147]	(PMA) + QoS

Unified Place and Map Schemes

In *unified place and map schemes* [4, 17, 99, 133], the placement of IPs due to physical planning, will induce lower logic fragmentation. In addition, the IPs and switches will be placed next to each other, which will reduce the length of connecting wires. As a whole, the unified place and map will result in efficient logic plane resource utilizations. However, on the negative side, the schemes [4, 17, 99, 133] could suffer through long routes with no analytical bounds over the QoS guarantees. This is mainly because of the allocation phase, which is not integrated with the place and map. Moreover, the mapping is performed in prior to routing. An optimal allocation that can satisfy the QoS requirements of the placed application is largely dependent upon the routing algorithm. This could cause IPs to get blocked, due to unavailability of resources on the NoC. Additionally, the scheme of requires a fully-connected network, which costs high in terms of area.

Unified Map and Allocate Schemes

In *unified map and allocate schemes* [50, 83, 100, 104], IP ports are mapped only and only if resources for communication flows are guaranteed over the NoC. However, on the negative side, placement can only be performed once mapping and allocation are done. It is important that during mapping and allocation, the available logic area against a communication node is not taken into account during an application binding process. In other words, these schemes can be applied to an FPGA, after assuming that sufficient logic resources are available next to the communication node, which could result in placing an application IP in far regions of the chip. It can be unaffordable in an FPGA, when application to FPGA area constraints are stringent.

Non Unified Schemes

In [58, 70, 87, 101, 102, 128] the placement, mapping, and allocation processes are non unified. In contrast, we unify all the three process of placement, mapping, and allocation during an application binding. Our PUMA scheme ensures that, for a successful application to FPGA binding, the application QoS constraints are fulfilled. PUMA, for this purpose, takes into account the required application resources to available FPGA resources in both the logic and communication planes, simultaneously. This means the temporal and spatial constraints of an application are considered simultaneously, while selecting an FPGA region. Our FPGA communication plane is a hardwired Network on Chip (HWNoC), which does not compete with logic resources and is scalable.

3.4 Technique: Composable and Persistent-State Dynamic Reconfiguration

In this section, we provide the overview and motivation for our technique, i.e., the 3-tier model, to achieve composability and persistent-state during the dynamic reconfiguration process. We then provide the related work, and at the end position our technique with respect to the existing state of the arts.

3.4.1 Overview

We propose a 3-tier reconfiguration model that consists of the system manager, an application manager per application, and application(s). The role of the

system manager is to manage applications, i.e., test, reconfigure, and start / stop the applications. The role of an application manager is to time-multiplex parts of of a single application, by swapping in / out parts of the application and taking care of the persistent-state between the sub-applications. The 3-tier reconfiguration model:

1. uses a hardwired NoC for functional and configuration data transportation,
2. enforces composability, i.e. the dynamically inserted (sub)application does not interfere with the execution of other running (sub)applications, as long as their allocation remains unchanged,
3. enforces persistent-state, i.e. the state-information (spread at multiple places in the system) of the sub application must be saved, when it is swapped out.
4. enforces predictable application behavior, i.e., throughput and latency demands for the dynamically inserted (sub)applications on an FPGA are fulfilled.

The system manager initially configures application IPs and programs the connections among the IPs. Afterwards, the respective application manager, which has already been configured by the system manager, is programmed with parameters such as: address and quantity of input / output application data. An application manager then programs the associated IPs with appropriate data set. An application after receiving input data from its associated application manager, executes on it, and forwards it back to its application manager. An application manager, then stores the received data on the appropriate memory addresses. Meanwhile, an application manager observes the progress of its client application. Once, application has processed required amount of input data, and all output data has been produced. Then, an application manager stops from sending / receiving data to / from its client application. An application manager afterwards signals reconfiguration request to the system manager. It is important that the 3-tier model uses hardwired network on chip to transport configuration, programming, and functional data.

3.4.2 Motivation

We use a motivational case study to explain how the presence of a soft functional interconnect can affect composable dynamic reconfiguration. For the

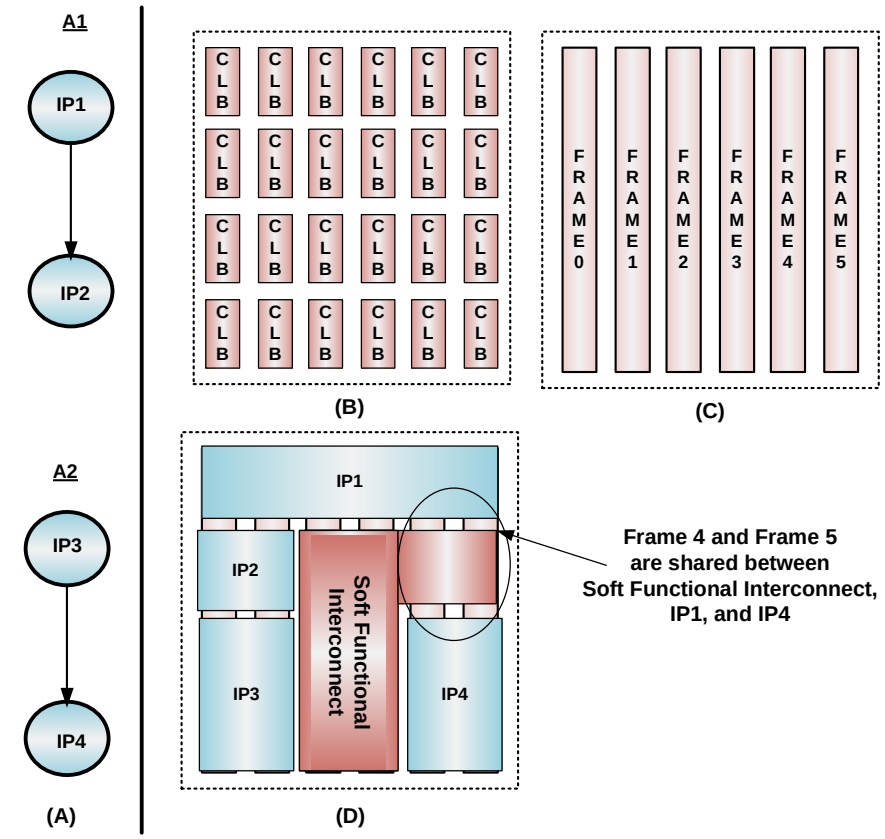


Figure 3.7: (A) Two Applications, (B) FPGA Architecture, (C) Bitstream Frames in FPGA Architecture, (D) Sharing of Frames between the Soft Functional Interconnect and IPs.

motivational case study, we consider SoC with 2 applications and FPGA with 24 CLBs (or 6 CLB columns), Figure 3.7A and Figure 3.7B, respectively.

The FPGA is configured in a frame-wise manner, where each frame belongs to multiple CLBs, Figure 3.7C. It is important that when an application is started / stopped, dynamic reconfiguration is performed and as a requirement the functional interconnect must be updated, i.e. reconfigured and/or reprogrammed. However, in case of traditional FPGA architecture, the application IPs and the functional interconnect can share single /multiple bitstream frames, Figure 3.7D. In such a situation the dynamic reconfiguration performed for the application can interfere with other applications, because the interconnect would be stopped during the reconfiguration process.

In contrast, by using hardwired NoC, we ensure a composable system behavior during the dynamic run time reconfiguration. The reason is that the IPs and interconnect are decoupled in several senses: physically (e.g. to avoid glitches), in placement (reconfiguration and functional regions of IP and interconnect are disjoint), and logically (there is no communication to / from IPs that are reconfigured, and communication between other IPs is therefore, not affected). Also, the fact that one TCFR never contains IPs of greater than one applications is helpful in achieving the composable behavior of system during the dynamic run time reconfiguration.

In the following sections, first we describe the related work, then we classify and compare our our work with respect to the related work.

3.4.3 Related Work

We present the technique to perform dynamic application reconfiguration with (i) composable behavior during inter-application reconfiguration, and (ii) persistent-state assurance during intra-application reconfiguration. Additionally, our technique applies a scalable (3-tier) reconfiguration model and guarantees over QoS constraints. In the following discussion, we present the related work that targets any of the above requirements while performing dynamic application reconfiguration.

Composable Dynamic Reconfiguration

In literature, a number of efforts [11, 15–17, 20, 60, 71, 125, 141] have been performed for the dynamic *inter-application* reconfiguration. Authors in [15–17, 60] place dynamically inserted module in vertical slots, which allow the modules to be attached at any location. On the other hand, authors in [125] provide a way to place hardware modules of predetermined size and positions, above each other. To connect the modules; work in [16] uses a reconfigurable multiple bus (RMB), work in [17] uses an NOC, work in [60] uses lookup tables, and work in [125] uses bus macros. Additionally, in [11] a dynamic instruction set architecture-based approach is used, where the authors make use of dynamically rotating instructions for runtime swapping of reconfigurable modules. The work in [20] uses a reconfigurable system that is based on square-shaped and arbitrary-sized *swappable logic units* (SLUs). The SLUs are arranged in mesh, and communicate with each other through a small communication buffer. Work in [71] allocates FPGA resources at run time by making use of a centralized resource manager. The research in [141] achieves

dynamic on-demand reconfiguration by making use of a run time system software on MicroBlaze, which controls reconfiguration and message handling.

Persistent-State Dynamic Reconfiguration

In literature, a number of efforts [77,94,107,122,130] have been performed for the dynamic *intra-application* reconfiguration. The works assume that tasks can only migrate at predefined execution points, like we do.

The authors in [122] collects all unprocessed messages into a special buffer when a migration point is reached. After the actual migration, all communication peers are notified and their task lookup table is updated to reflect the new location of the migrated task.

In [107], when the task on the source tile reaches a migration point, it signals this event to the Operating System (OS). In turn, the OS instructs the producers to send one last tagged message and then to stop sending. The OS consequently sets up, initializes and starts the migrating task on the destination tile. The next step is to forward all buffered and unprocessed messages to the new location of the migrated task. To this end, the OS initializes a so-called destination lookup table (DLT) (containing the new destination for the messages) on the source tile and instructs it to orderly forward all incoming messages. When all tagged messages have been sent, the migration process is finished and the OS can free the resources of origin tile.

In [94] when task reaches a migration point, it goes into the interrupted state. In this interrupted state all the relevant state information of the migration point is transferred to the Operating System (OS). Consequently, the OS re-initiates the task on the new destination position using the received state information. The task resumes on the second processor, by continuing to execute in the corresponding migration point.

In [130], authors make use of bitstream readback facilities of the FPGAs configuration port. All the configuration data is read back and state extraction is performed after reading the configuration data. State extraction is achieved by getting all status information bits out of the read back bitstream.

Unlike Simmler, the authors of [77] do not read back all configuration data. Only those data is read back that include state information and belong to the task to be suspended. Furthermore, the actual state information extraction is not done after but during reading the configuration.

3.4.4 Positioning with the State of the Art

In the following discussion, we position our 3-tier model against the schemes that perform composable dynamic reconfiguration and persistent-state dynamic reconfiguration.

Composable Dynamic Reconfiguration

In contrast to our inter-application level composability, the above-mentioned approaches [11, 141] take into account single application, and implement task level composable behavior of the system. However, due to the possible resource fragmentation, it becomes difficult to ensure application guarantees. Work in [71] takes into account the concurrent execution of multiple applications, but the mechanism to implement persistent-state and QoS guarantees is missing. The remaining approaches [15–17, 60, 125] do not explicitly state the level of composability. These schemes [15–17, 60, 125] are more concerned about providing the communication among the dynamically placed modules rather than handling the important issues of *stability of prior services*, mechanism to assure *safe-state transition*, and *QoS guaranteed resource allocation* with the addition/removal of application/modules.

In contrast to our layered approach (Section 3.4), the existing works of [11, 15, 20, 71, 141] make use of a centralised resource manager for dynamic reconfiguration. Unlike the works of [11, 125] that assume an FPGA system with a single application, our technique performs dynamic reconfiguration for a system with multiple applications. Additionally, the above approaches face certain limitations, which are not an issue with our technique. For instance: a) the possibility of execution time being dominated by the reconfiguration time due to the small communication buffer [20], is avoided by an application manager that ensures the memory allocation for its client application till the required execution stage. b) A high ratio of area of processing to network element [17] is avoided by hard-wiring the underlying functional architecture. c) The possibility of run- time reconfiguration of the functional architecture [60] that in turn could disrupt the executing applications, is avoided because our functional architecture only needs to be programmed with each adding IP / application. d) We offer QoS guarantees for dynamically adding applications by providing a virtual platform for each (sub)application, and reserving the required resources across that platform in contrast to [11, 71, 141] that provide no guarantees on Quality-of-Service requirements.

Table 3.4: Our Work Positioning with respect to Composable Dynamic Reconfiguration Approaches.

Scheme	**Composability Abstraction**	**Guarantees on QoS**	**Functional Interconnect**
Bauer [11]	Task level	No	Soft Interconnect
Bobda [16]	Task level	No	Soft Interconnect
Brebner [20]	Task level	No	Soft Interconnect
Huebner [60]	Task level	No	Soft Interconnect
Jean [71]	App. level	No	Soft Interconnect
Sedcole [125]	Task level	No	Soft Interconnect
Ullmann [141]	Task level	No	Soft Interconnect
Our Technique [143, 144]	Application Level	Yes	Hard Interconnect

Persistent-State Dynamic Reconfiguration

The works of [94, 107, 130] provide a mechanism to achieve persistent-state during intra-application dynamic reconfiguration. In these works, the state information of a task is distributed: a) within the tasks, b) and in between the tasks. The state preservation within the task/IP causes complex issues with respect to the register states, and clock phase to preserve data and timings. *Our 3-tier model*, in contrast, uses stateless IPs and an application manager is the one that triggers the reconfiguration request. However, an application manager triggers the reconfiguration request only when the sub application IPs achieve the required execution granularity. Importantly, the inter-IP state is preserved by an application manager by providing the persistent storage in between the sub application swapping. We do not allow cycles during sub application swapping. Therefore, the existing sub application's pipeline is completely flushed and preserved in an application manager before starting the next sub application.

The schemes of [94, 107, 130] make use of input queues to collect all unprocessed data, or destination lookup tables (DLTs) to forward unprocessed data to new location. In both the situations, extra / special hardware in the form of queues and DTL is required to implement persistent-state transitions. The read back methods do not require extra hardware, because they use inherent access structures of the configuration circuitry and the configuration port. However, they suffer from poor data efficiency, which means the proportion of useless data in the readback stream is rather high. *Our 3-tier model* also induces extra

Table 3.5: Our Work Positioning with respect to Persistent-State Dynamic Reconfiguration Approaches.

Scheme	**Reconfiguration Decision**	**Hardware Overhead**	**Resource Management**
Kalte [77]	Statefull IPs	No	Centralized
Mignolet [94]	Statefull IPs	Yes	Centralized
Nollet [107]	Statefull IPs	Yes	Centralized
Russ [122]	Statefull IPs	Yes	Centralized
Simmler [130]	Statefull IPs	No	Centralized
Our Technique [143, 144]	Stateless IPs	Yes	(Frequent) intra-application distributed, (Infrequent) inter-application centralised

hardware in the form of an application manager per application.

All the schemes implement a centralized way of implementing the persistent-state transitions of tasks / applications. This means that a single resource manager (e.g. OS) takes care of implementing persistent-state transitions for all the applications in SoC. This can be a performance bottleneck in several ways: (a) frequent transitions for a single application, (b) multiple applications are going through state transition, simultaneously. *Our 3-tier model*, in contrast, implements a distributed way of implementing persistent-state transitions for the applications of SoC. This is achieve with dedicated application managers for each application.

3.5 Technique: Online Testing

In this section, we provide the overview and motivation for our online test technique. We then provide the related work, and at the end position our technique with respect to the existing state of the arts. The choice of state of the arts is based on different requirements, i.e., i) test access mechanism they use, 2) the nature of test they perform, and 3) the intrusiveness level they produce with the already executing applications.

3.5.1 Overview

Our proposed online test scheme performs a structural test at the startup of an application by using the hardwired NoC. The structural test is performed for FPGA TCFRs that are associated with the application that has been started. More specifically, our online test scheme:

1. uses a hardwired NoC as a test access mechanism (TAM),
2. tests undisrupted in parallel with other application(s) configuration, programming, and execution,
3. tests at an application startup, i.e., before the configuration of an application,
4. uses structural test for the correctness of an FPGA,
5. reduces spatial and temporal overheads with respect to conventional, online test schemes.

In the following discussion, we explain each of the above-mentioned points.

1 & 2) HWNoC as TAM: Our FPGA is comprised of multiple test configuration functional regions (TCFRs). The online testing is performed at the granularity of TCFR. For verifying, the (structural) correctness of a TCFR, our online test scheme uses a HWNoC [41] as a TAM. The HWNoC can transport four types of data, i.e., test, configuration, and functional (programming and execution) data. For this purpose, it uses connections. The connections are allocated at compile time, but are created and terminated at run time. Moreover, the HWNoC architecture can ensure non-intrusive data transportation, which means all types of data (test, configuration, programming, and execution) that flows through the same HWNoC does not produce interference with each other. This means while testing, it is possible to achieve: (a) un-disrupted execution for already running application(s), (b) the operations of configuration, programming, and execution for new applications.

3) Test at application startup: The test procedure is triggered at application startup time. We assume that multiple applications can not occupy the same TCFR(s), simultaneously. However, an application can execute in multiple TCFRs. In our online test scheme, an application TCFRs are tested in prior to configure an application. However, before an application is executed, its associated TCFRs are tested. By doing so, the disruption in application execution,

which could by caused by the test procedure, is avoided. On the negative side, the application configuration is delayed until the application TCFRs are tested. Additionally, a fault that occurs during an application execution time, can only be detected after an application finishes its execution.

4) Structural test: Our online scheme performs the structural test and currently detects the permanent stuck-at[1] faults. The structural test provides us with the increased reusability, which can be exploited in multiple ways. A single test suite (bitstreams and stimuli for an FPGA TCFR) irrespective of the intended set of applications that run on the TCFR. This mean the test bitstream and expected results for a single TCFR can be reused for multiple TCFRs.

5) Reduced Spatiotemporal Overheads: Moreover, the analysis of structural faults in a TCFR is performed by making use of HWNoC connections, and the system manager [146]. The same HWNoC is used to transport functional data, whereas the system manager can execute on a programmable hardware processor, e.g., PowerPc. The *additional resources* are in the form of test connections and the code to perform the analysis. In our scheme, no specialised test hardware (e.g., TPGs and ORAs) are imported to FPGA logic plane. This in turn, eliminates the spatiotemporal overheads that are required to place and configure the test hardware on FPGA logic plane. However, towards downside, the software implementation of test is more constrained. Additionally, the system manager has more work to do which can result in slower test.

In the next section we build the motivation behind using these parameters for our technique.

3.5.2 Motivation

For motivation purpose, we consider SoC that comprises 2 applications, Figure 3.8A. Figure 3.8B shows the binding of these applications on the logical regions of an FPGA. IPs of both the applications communicate with each other by using a functional interconnect (e.g., bus or NoC). In other words, in conventional test schemes application IPs and the associated functional interconnect both coexist in the same reconfigurable plane [38, 142], as shown in Figure 3.8B. To build the motivation of our scheme, we examine the possible shortcomings that can arise while testing the SoC by using the conventional test schemes, such as [38, 142].

A region under test (RUT) in existing schemes, e.g., [142] can be considered

[1] A logic block or wire is said to experience a stuck-at fault when its logic value always stays at `1` or `0` and can not be reversed.

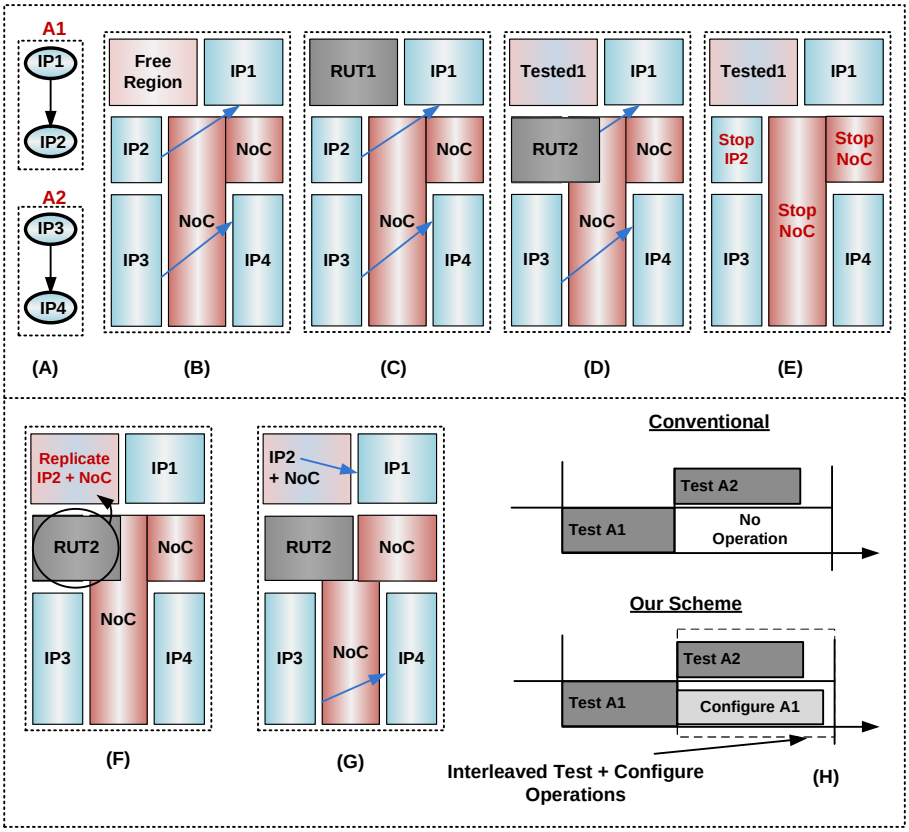


Figure 3.8: (A) Applications, (B) Application to FPGA binding, (C) Testing of an FPGA region, i.e., RUT1, (D) Start to Test another region of FPGA, i.e., RUT2, (E) Stopping IP and Functional Interconnect that is affected by RUT2, (F) Replicate IP and Functional Interconnect in a Previously Tested Region, (G) Start Test for RUT2 and Execution for IP2 and Functional Interconnect in New Region, (H) Abstract Comparison of our Scheme with Existing Conventional Schemes.

as a set of wires and CLBs. A region under test can be a free region, i.e., not in use by any application as shown with RUT1, or an RUT can belong to IP plus interconnection as shown with RUT2. The conventional schemes scan through the FPGA to find out faults [38]. This means testing is performed by roving an RUT across the chip. RUT1 in Figure 3.8C and RUT2 in Figure 3.8D indicate two such roving instances. In Figure 3.8D, *Tested1* indicates the region that was earlier tested by RUT1.

Conventionally, the region under test is taken off-line, while allowing the rest of FPGA to continue its normal operation. However, in case an application is already executing on such a region. Then the application is first stopped before being replicated to already tested region. Additionally, the FPGA interconnection is en-routed accordingly for intra-application execution.

For instance, Figure 3.8D shows that RUT2 test the region that is taken by IP2 of A1 and NoC. This means, before performing the test, the IP2 and NoC both are stopped as shown in Figure 3.8E. Then, IP2 and the respective portion of NoC are replicated in region that is free and previously passed test, e.g., Tested1 is one such regions as shown in Figure 3.8F. After the replication process, IP2 and associated NoC start working and the test process for RUT2 starts as well, as shown in Figure 3.8G. The above discussion motivates us to *trigger the online test at an application startup time*. By doing so, we can *avoid intrusiveness* by ensuring that an application always executes at the pretested regions.

More importantly, application A2, whose area is not under test, stops executing because the NoC does not transport any data during the stop and replication process. In Figure 3.8E and Figure 3.8F, this is shown be removing the blue arrows that indicate the communication between the IPs of both the application, i.e., A1 and A2. Hence it motivates us to have applications and the functional interconnect in *disjoint planes*, which is ensured by hardwiring the communication architecture (i.e., HWNoC) in FPGA.

Additionally, the existing approaches [6, 38] use boundary scan infrastructure (BSI) [64], which occupies the FPGA configuration circuit while testing. This means the inherent nature of the TAM in these schemes does not allow the configuration (of another application) in parallel with the ongoing test process. Therefore, restricting the parallel operations of test and configuration for multiple applications. For instance in Figure 3.8H, the parallel operations of A1 testing and A2 configuration (on a pretested region) are not possible with existing schemes, such as [6, 38]. Hence imposing a delay on either, test or configuration, process. This motivates us for a *TAM which does not inter-*

fere with the bitstream loading of an application, because it is implemented by HWNOC that is parallel, i.e., has greater than one connection.

Moreover, FPGA based SOCs can comprise multiple time-multiplexed applications, where developing and exercising a test suit for each application is prohibitive due to economic and time constraints. Therefore, it motivates us to perform the *structural test*, which not only posses an application independent nature but also ensures maximum percentage of fault detection.

In the following sections, first we describe the related work, then we classify and compare our our work with respect to the related work.

3.5.3 Related Work

In the literature, a number of online test schemes [1,6,32,37,38,115,129,142] have been proposed to detect faults in FPGA architecture. In the following discussion, we explain each of these individually.

In [1], the authors introduce the concept of roving STARs, where a STAR is a self testing area that is comprised of TPG, ORA, and circuit under test. The online testing is structural which roves the STARs periodically to test every section (that can comprise multiple CLB columns) of the FPGA architecture. The TAM mechanism used by the scheme is the conventional boundary scan infrastructure [64].

The authors in [6] make use of built in self test (BIST) to detect operational faults in the system. The scheme in [6] applies the test sequence periodically to the circuit under test (CUT), and checks the CUT responses to detect the existence of operational faults. The online testing is functional, which can test multiple CLBs simultaneously. The authors, however, have not mentioned the TAM mechanism for the proposed method.

In [37, 38], the authors apply the concept of active replication for the online testing of CLBs. This active replication method enables the relocation of each CLB functionality without halting the system, even if the CLB is part of an executing application. In [38] the authors have applied the structural testing, whereas in [37] the authors have performed functional testing of the target FPGA. However, in both the schemes the test is performed at a single CLB level by making use of the conventional boundary scan infrastructure [64].

The authors in [115] propose a new CLB architecture for FPGAs and associated online testing, and reconfiguration techniques that detect configuration faults in the CLBs of FPGAs. The test scheme is structural, which can detect

single or multiple faults in an FPGA. The authors, however, have not mentioned the TAM mechanism for the proposed method.

The authors in [86] provide an error mitigation technique that is based on modular redundancy and time redundancy. It uses duplication with comparison (DWC) and concurrent error detection (CED). The test scheme is functional, i.e. application dependent, which requires suitable encoding and decoding functions to test the CLBs. The authors, however, have not provided any details about the TAM.

The testing in [129], like the conventional online test schemes, is based on fault-scanning method. The scheme is applicable to bus-based FPGA architectures, and it assumes that certain parts of the FPGA are fault-free. The online testing is structural, which is performed at the granularity of multiple CLBs. The authors, however, have not provided any details about the TAM.

The authors in [142] claim to provide a faster online test scheme as compared to [1]. The scheme is based on roving tester (ROTE), which tests parts of the circuit by duplication and comparison manner. The testing is intended to detect possible circuit functions (applications) with a test granularity that can range from single CLB column to multiple CLB columns. The authors, however, do not specify the TAM for their approach.

3.5.4 Positioning with the State of the Art

From the above discussion, we can conclude that the existing schemes scan through the FPGA chip to find out the perspective faults [1,6,37,38,115,129, 142]. In these schemes, a relatively small portion of FPGA chip is taken off-line, while allowing the rest of FPGA to continue its normal operation. The region to be tested is replicated on an already verified portion of the device, before being taken off-line and tested. Testing in these schemes is accomplished by roving the test functions, i.e., test pattern generator, output analyser, and region under test bitstream, across the entire FPGA. These schemes, as illustrated through Table 3.6, use a conventional boundary scan infrastructure (BSI) as their TAM [1, 37, 38]. The nature of online testing can be structural [1, 37, 38] or functional [6, 32, 38, 142]. The schemes can have different levels of test granularity, ranging from a single CLB [37, 38, 86] to multiple CLB columns [1,129,142]. The main focus of the existing online test schemes has been to maximise the percentage of fault detection with the minimum fault detection latency. Additionally, none of these schemes, due to the limitations of the existing TAM architecture, can achieve the interleaved test and configu-

Table 3.6: Our Work Positioning with respect to Existing Online Schemes. Abbreviations, NA = Not Applicable, Col: Column, St: Structural, Fu: Functional, P/C: Partial/Complete.

Scheme	**TAM**	**Test Type**	**Test Granularity**	**Test& Load**	**Intrusiveness Level**
Abramovici [1]	BSI	St	CLB Col	No	(P/C) App.
Al-Asaad [6]	NA	Fu	CLB Col	No	(P/C) App.
Gericota [37]	BSI	St	CLB	No	(P/C) App.
Gericota [38]	BSI	Fu	CLB	No	P. App.
Lima [86]	NA	St	CLB	No	(P/C) App.
Reddy [115]	NA	St	CLB	No	(P/C) App.
Shnidman [129]	NA	St	CLB Col	No	(P/C) App.
Dutt [32]	NA	Fu	CLB Col	No	(P/C) App.
Verma [142]	NA	Fu	CLB Col	No	P. App.
Our Online Test Methodology	**HW-NoC**	**St**	**TCFR**	**Yes**	**None**

ration operations for multiple applications, Table 3.6 (Column 4). This means that each one of these induces a level of intrusiveness, which could be a partial or a complete application, Table 3.6 (Column 5).

In contrast, our proposed methodology uses a *hardwired network on chip as test access mechanism*, and conducts *tests on a region-wise basis*, see Chapter 7. A region in our methodology is termed as test configuration functional region (TCFR), as we shall explain in Section 4.3. The proposed test methodology exhibits *a non-intrusive behavior*, which means it allows the test process in parallel with the configuration, programming, and execution of applications in regions that are not under test. Moreover, our online test methodology performs *test when an application is invoked*, which ensures that application always execute on a reliable architecture. The nature of the *test is structural*, which ensures a high percentage of fault detection for the target FPGA architecture. In addition, the proposed scheme has *reduced spatiotemporal overheads*, because it does not make use of TPGs and ORAs to generate and analyse the test sequences. Instead, the proposed scheme uses the connections through the hardwired NoC to access and analyse the architecture of a particular region in FPGA.

3.6 Conclusions

In this chapter, we explained the architecture and the design flow of the proposed solution (Section 3.1). Afterwards, we explained the techniques that use the architecture and the design flow of the proposed solution to fulfill the requirements, as mentioned earlier in Section 1.3 and Table 1.1. For each technique (i.e., hardwiredNoC, PUMA, 3-tier reconfiguration model, and online testing), we provided the methodology, motivation, related work, and positioning with the state of the art. (1) We provided hardwired NoC to fulfill the requirements of *scalable IP integration*, and *separate communication and computation* (Section 3.2). (2) We provided PUMA scheme to fulfill the requirement of *automation* to overcome the problem of high time to market (Section 3.3). (3) We provided the 3-tier reconfiguration model to fulfill the requirements of *composable and persistent-state dynamic reconfiguration* to overcome the problems of interference and data-loss during dynamic run time reconfiguration (Section 3.4). (4) We provided the online test scheme to fulfill the requirement of reliable architecture to overcome the problem of run time faults in an FPGA architecture (Section 3.5).

4

FPGA Architecture with a Hardwired Network on Chip

In this chapter, we provide the details of the proposed FPGA architecture. Initially, we present the overview of our FPGA architecture in Section 4.1. Afterwards, we explain the architecture of the hardwired NoC that serves as the communication plane in the proposed FPGA, Section 4.2. We then illustrate the architecture of test, configuration, and logic plane of the proposed FPGA in Section 4.3. Thereafter, we elaborate the control processor architecture that uses the HWNOC to transport data to the test, configuration, and logic plane of our FPGA, Section 4.4. We continue to elaborate the reasoning of hard and soft partitioning of components in the proposed architecture, Section 4.5. Thereafter, we provide the implementation details of the proposed architecture in Section 4.6. We then discuss the possible extensions of hardwired NoC in Section 4.7. The architectural limitations are provided afterwards in Section 4.8. Thereafter, we present the results and evaluation in Section 4.9. Lastly, we end this chapter with conclusions in Section 4.10.

4.1 Overview

Our proposed FPGA architecture consists of a *control processor* and multiple *test configuration functional regions* (TCFRs) that are connected to a *hardwired NoC*, as shown in Figure 4.1. The control processor uses the HWNOC to transport test, configuration, functional, and control data to a TCFR, where each TCFR represents the unified test, configuration, and logic region. In other words, application IPs are tested, configured, and executed on TCFRs.

Each TCFR consists of an array of *minimum test configuration regions* (MTCR), where each MTCR defines the minimum region that can be tested or

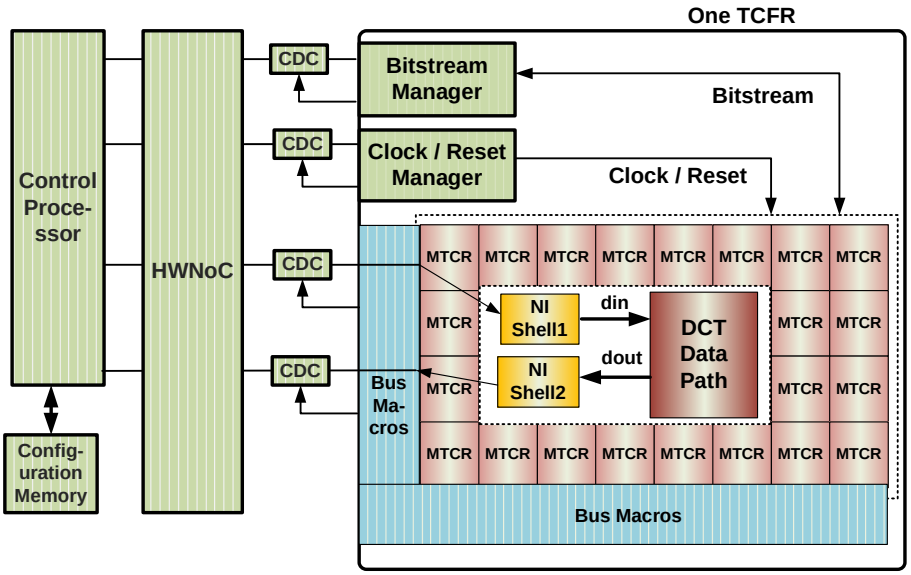


Figure 4.1: Overview Diagram of the Proposed FPGA Architecture.

configured to load a specific application[1]. A TCFR connects to the HWNOC by using *clock domain crossing* (CDC) FIFOs and *Bus Macros*, as shown in Figure 4.1. The CDC FIFOs and Bus Macros are used to implement the transportation of (configuration, test, functional, and control) data between the HWNOC and a TCFR. A TCFR has its own *bitstream manager* that operates on the received test and configuration bitstreams and loads it to the respective MTCR. There is a local *clock / reset manager* that is memory-mapped. This means programmable clock frequency and set / reset signals can be generated for the required MTCR by writing to the registers of clock / reset manager that are accessible through the HWNOC.

The control processor makes use of the hardwired NoC to test, configure, program, and execute application IPs on a TCFR. The control processor uses streaming ports to test and configure a TCFR, and memory-mapped IO (MMIO) ports to program and execute IPs on a TCFR[2].

From a high-level perspective, to start a single soft IP operating, the control processor takes following steps. (1) Read the bitstream packets (of IP) from the bitstream memory. (2) Send the bitstream frames to the respective TCFR, where each frame programs an MTCR. The control processor uses NoC normal

[1]The detailed architecture of MTCR is explained in Section 4.3.
[2]The control processor to TCFR communication is detailed later in Section 4.4.

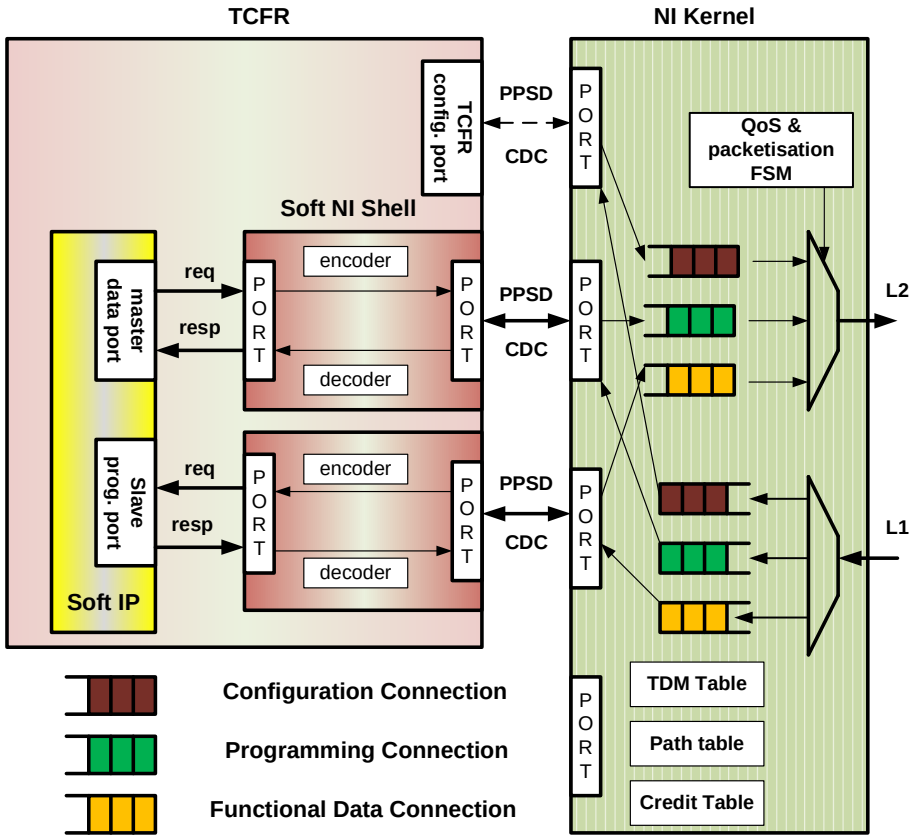


Figure 4.2: IP with one Master and one Slave Port, without Reprogramming and Reconfiguration Privileges, and its NI Shell and Kernel.

connections to stream the bitstreams. It does so by first setting up a connection from a port on its NI to the configuration port on the NI of a TCFR. This entails programming the HWNoC using memory-mapped IO (MMIO), as described in detail in [41, 46]. For our purposes, this is performed by an abstract open_connection function. After the bitstream has been sent to the TCFR, the bitstream connection is removed. The control processor then (3) switches on the clock of the IP and (4) resets the IP. (3) and (4) also use connections from control processor to MMIO port on the clock / reset manager of the TCFR.

4.2 Hardwired NoC Architecture

A hardwired Network on Chip (HWNoC) [41] is used for the transportation of unified (test, configuration, functional, and control) data in between the test configuration functional regions. It is a system level interconnect that is embedded in the proposed FPGA architecture, and consists of routers (R) and network interfaces (NIs).

A router in our design is used to send / receive data from / to a network interface. We do not propose any changes in the router for our HWNoC, and details of the router architecture can be found in Section 2.2.1. However, we propose a new port (used to load bitstreams) for our HWNoC network interfaces, as shown in Figure 4.2. We will explain the nature of the port later in this section.

The NIs in our HWNoC, like NIs in a conventional soft NoC, are split into two parts: the kernel (performing network layer functions) and the shell (for transport layer functions) [121]. Two IPs communicate by using the *connections* to transport data from a source NI to a destination NI. A connection consists of two channels, i.e., a request channel and a response channel. Each channel has its own quality of service (QoS), i.e., the allocated bandwidth and latency. An NI kernel is responsible to receive packets from the router on link L1 and depacketise them, Figure 4.2. Conversely, it packetises data and sends them to the router on link L2, according to the channel's QoS. Our example NOC, ÆTHEREAL [43] uses a virtual-circuit TDMA scheme to offer strict guarantees on bandwidth and latency requirements. The kernel therefore, contains a programmable TDMA table, credit counters for end-to-end flow control, and per channel programmable path that its packets take through the NOC, Figure 4.2.

The ports on the NI kernel are point to point streaming data ports (PPSD), i.e., a fixed word-width with valid / ready handshake, a CDC (not shown in Figure 4.2) is used to cross from NOC to IP clock domains. Streaming IP can connect directly to these ports. The encoder and decoder in an NI shell, implement the valid / ready handshakes per command, read / write data groups of, e.g., AXI and DTL, and their serialisation to / from the NI kernel ports, etc.

The soft IP resides on a test configuration functional region (TCFR), as shown in Figure 4.2. The IP can have a master data port on which it sends read / write requests to a slave somewhere on the HWNOC, and a slave MMIO programming port over which read / write requests are received.

The *configuration port* on a TCFR is the new addition, proposed in this thesis. Because configuration data is streaming data and not shared memory communication, the configuration port of a TCFR is connected directly to the NI

kernel. From the HWNOC perspective, it is just another port, over which data is communicated. This is achieved by programming sufficient bandwidth allocation in the kernel's TDMA slot table, and sizing the buffers in the NI kernel to absorb any jitter from unevenly-spaced TDMA slots [26].

An IP is placed in the reconfigurable logic plane of our FPGA, as shown in Figure 4.2. The reconfigurable plane is comprised of multiple test configuration functional regions (TCFRs), whose architecture is explained in the next Section 4.3.

4.3 Test Configuration Functional Region Architecture

The architecture of a test configuration functional region [145] is illustrated with Figure 4.1. It shows that each TCFR constitutes a number of minimum test configuration regions (MTCRs), Bus Macros, clock domain crossing FIFOs, a local bitstream manager, and a *clock / reset manager*. Importantly, a TCFR is split in more than one MTCR to amortise the cost of the bitstream and clock / reset managers. In the following discussion, we provide the details of each architectural component that is part of a TCFR.

4.3.1 Minimum Test Configuration Regions

A minimum test configuration region (MTCR) in our TCFR architecture is comprised of 16 configurable logic tiles, as shown in Figure 4.3. Each logic tile consists of a configurable logic block (CLB) and a configurable switch matrix (SM) [158]. The CLBs[3] are connected to each other by using the associated switch matrix. For this purpose, the switch matrices use the routing wires, which surround each logic tile and are used to connect two tiles with each other. Moreover, the tiles in each MTCR exhibit column-wise organisation, as shown in Figure 4.3.

Importantly, a TCFR is not isolated at functional level. This means that the logic plane of an FPGA is not divided into multiple disjoint regions. This means a soft IP can be placed in MTCRs that belong to different TCFRs, and hence removes the restrictions on an IP placement, although, for dynamic partial reconfiguration, some restrictions persist. However, the span of an IP in

[3]The architecture of a CLB is presented earlier in Section 2.1.1.

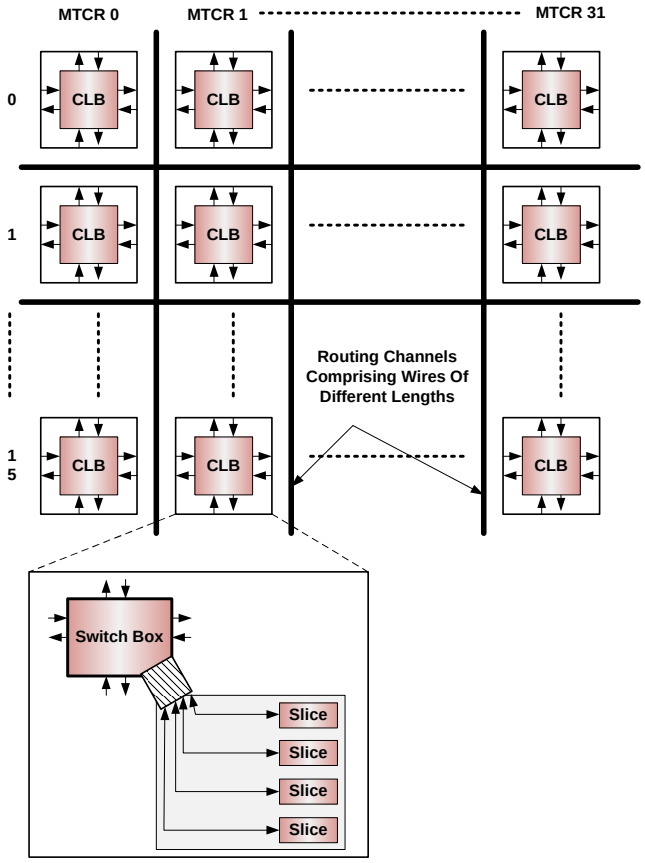


Figure 4.3: Detailed Functional Architecture of a Minimum Test Configuration Region.

multiple TCFRs is achieved by making use of BUS Macro units as explained in the following section.

4.3.2 Bus Macros

The connection between the HWNoC and TCFRs is persistent. This means that the physical link in a TCFR, which is made up of multiple wire-segments and connects to the hard link coming out of an NI, is not altered during dynamic application reconfiguration. To have persistent connections we make use of Bus Macros [61] at the boundary of a TCFR, where a Bus Macro is a relatively placed and routed IP core that provides a dedicated communication point. We

call it relative because all components within the IP core are placed and routed with reference to each other, but the IP core itself can be placed at different places on the FPGA without affecting the IP-core internal composition.

In our architecture the Bus Macros provide fixed point connections in between an NI kernel (for hard links) and a TCFR. In other words Bus Macros serve to isolate the HWNOC links from glitches or invalid data when the TCFR is being reconfigured. Moreover, Bus Macros are used at the boundaries of each TCFR, so as to connect multiple TCFRs. In our architecture, the Bus Macros that connect two TCFRs provide the interface for intra-IP communication for an IP that spans multiple TCFRs. By using BUS Macros, an application IP and its protocol shell, synthesized individually or together, that is placed in a TCFR can be connected to a HWNoC, Figure 4.1 and 4.9.

4.3.3 Clock Domain Crossing FIFOs

In our architecture, we make use of bi-synchronous FIFOs [150] to provide the clock domain crossings (CDC), as shown in Figure 4.1. The reason of using CDC FIFOs is that hardwired NoC (that provide inter-IP communication) and application IPs that execute on TCFRs can operate at different frequencies. In this case, due to the different clock frequencies of HWNOC and IPs, data loss can occur during inter-IP communication. The clock domain crossing FIFOs provide a mechanism to ensure reliable data transportation across the HWNOC boundary, i.e., where TCFRs connect to the HWNOC. Note that each IP uses a single clock, and all inter-IP communication takes place via the HWNOC. Hence no other clock domain crossing FIFOs (in a TCFR) are required.

The CDC FIFOs are gray-coded and pointer-based, which is compatible with standard CAD tools. Each bi-synchronous FIFO is connected to the HWNoC clock at one end, and at the other end to the local clock manager of a TCFR. This way the FIFOs provide clock domain crossings and let the IPs (or shells) robustly interface with the HWNoC. In our architecture these FIFOs used for clock domain crossings, provide one read / write transfer per clock cycle and possess a small latency overhead of two cycles.

4.3.4 Bitstream Manager

The architecture of a bitstream manager that processes the incoming test and configuration bitstreams in a TCFR is shown in Figure 4.4. The bitstream manager comprises a newly introduced port and logic, e.g., address decoder, dedi-

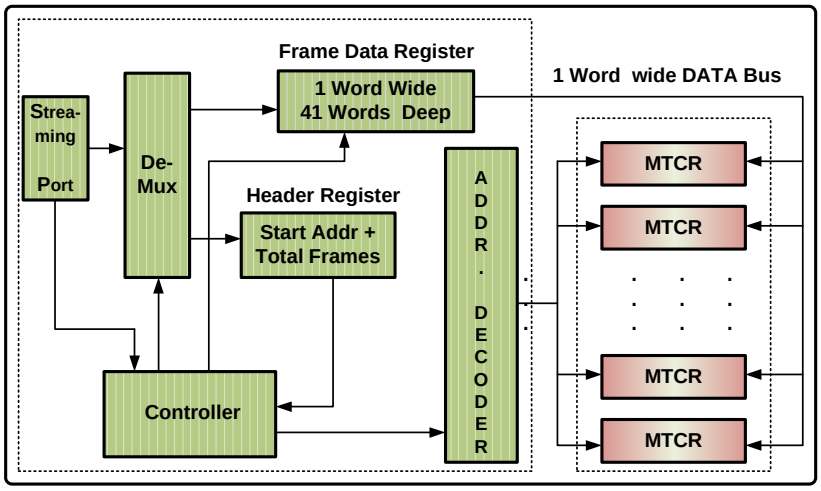


Figure 4.4: Bitstream Manager to Write Bitstreams in a TCFR.

cated registers and a 1 word wide *data bus* to write the incoming bitstream in the desired minimum test configuration region (MTCR). We use the HWNOC to send a bitstream to this port and for this purpose, a real-time connection with fixed latency and guaranteed bandwidth is used. A header register is used to store the bitstream header, whereas the frame data register is used to store the bitstream frames, as shown in Figure 4.4. The frame data register is large enough to store a bitstream frame of 41 words [154]. The job of the address decoder, which is connected to all MTCRs, is to enable the respective MTCR to receive the bitstream. The controller is a finite state machine, and controls the specific operations of: (i) bitstream reception from the streaming port, (ii) activation of the demultiplexer to direct the received data either to the header register or to the frame data register, and (iii) word-by-word right shifting of data in the frame register to load words in the *data bus*. Further details of the bitstream writing process are explained in Chapter 6.

4.3.5 Clock / Reset Manager

In our architecture, a programmable clock tree is used to distribute the clock signals in a TCFR, see Figure 4.5. The clock distribution is performed in spine-and-branch manner, which resembles Xilinx Virtex-4 and Xilinx Virtex-5 FPGA distribution [126, 160]. However, in our architecture the clock signals of a TCFR clock tree are driven from the output of the clock / reset manager, which is memory-mapped. This means the programmable clock frequency for

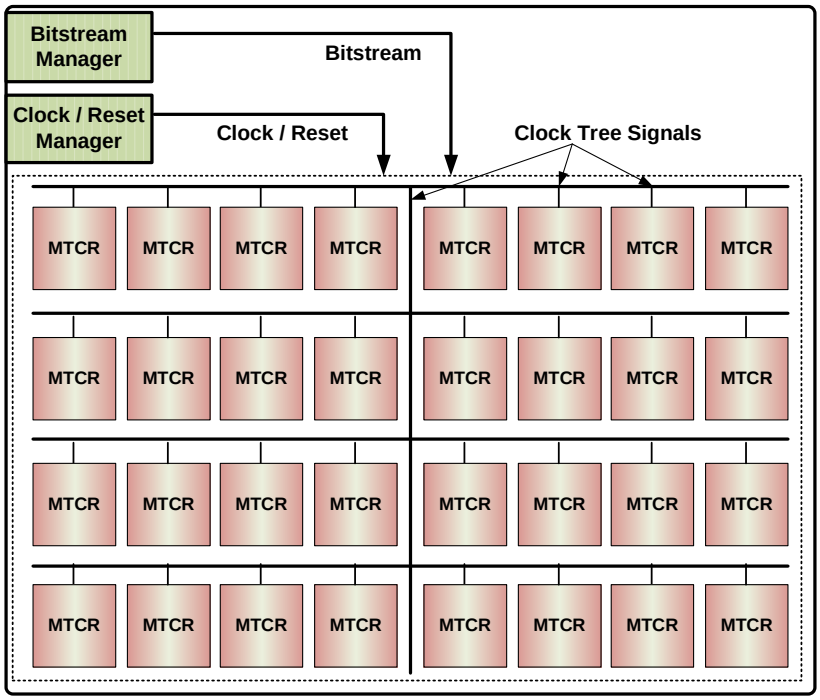


Figure 4.5: Clock Tree in a Test Configuration Functional Region.

the required MTCR is generated by writing to the specific register of clock / reset manager that is accessible over the HWNOC. In our architecture, we have a single clock per TCFR that goes to all the MTCRs.

Similarly the clock / reset manager can enable and disable the soft MTCRs from processing the input data. For this purpose, it uses a 32-bit register,whose each bit line connects to an MTCR. In other words, each MTCR has a separate reset signal. However, in our architecture the reset is performed on an IP basis that resides on top of MTCRs. This means when a soft IP must be reset, then the resets of the associated MTCRs are triggered accordingly.

There must be at least one IP that can program the system. This can be a CPU that bootstraps the system by programming the NoC, or a hard (secure) boot module [31, 145]. In our FPGA system, we make use of a control processor to bootstrap the system. The details of the control processor architecture are explained in the next Section 4.4.

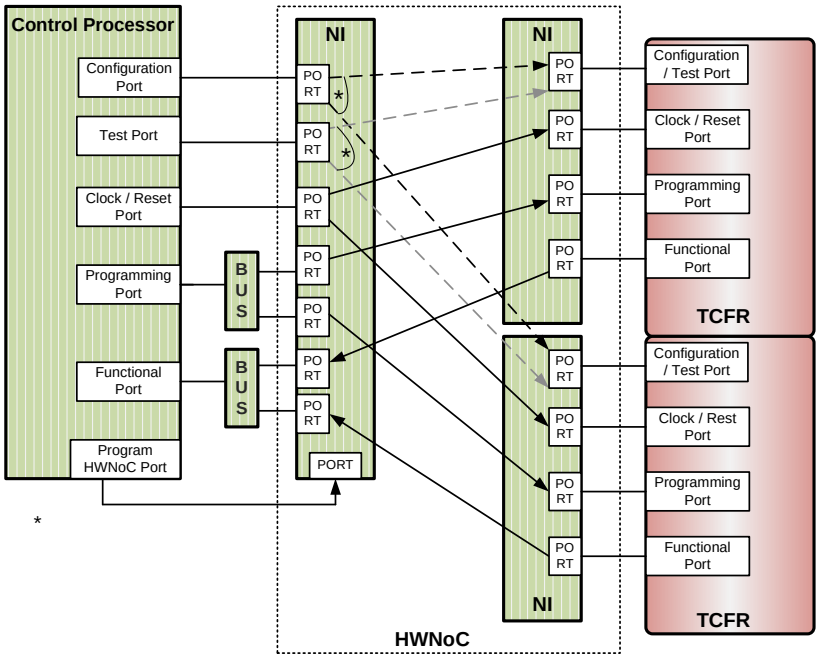


Figure 4.6: Control Processor Communication with TCFRs.

4.4 Control Processor Architecture

An FPGA can be used by multiple applications that have diverse resource requirements from both the FPGA planes, i.e., the HWNoC and TCFRs. Additionally, these applications can start and stop dynamically at run time. Therefore, we make use of a control processor that not only bootstraps the system, but (as we shall see in Chapter 6 and Chapter 7) tests, configures, and programs an FPGA system at run time. The control processor uses different ports and registers to to transport test, configuration, functional and control data, as shown in Figure 4.6 and Figure 4.7.

Figure 4.6 shows a situation where the control processor has connections to two test configuration functional regions (TCFRs). The control processor uses a streaming port to configure a TCFR. The streaming port connects directly to the HWNoC without needing a network interface shell, see Figure 4.7. We make use of a single streaming port for configuration purpose, as our design choice. This means, the configuration of only one TCFR is possible at one point in time, i.e., either the configuration port is connected to $TCFR_1$ or $TCFR_2$ as indicated by (*) in Figure 4.6, and the same applied for test port. Im-

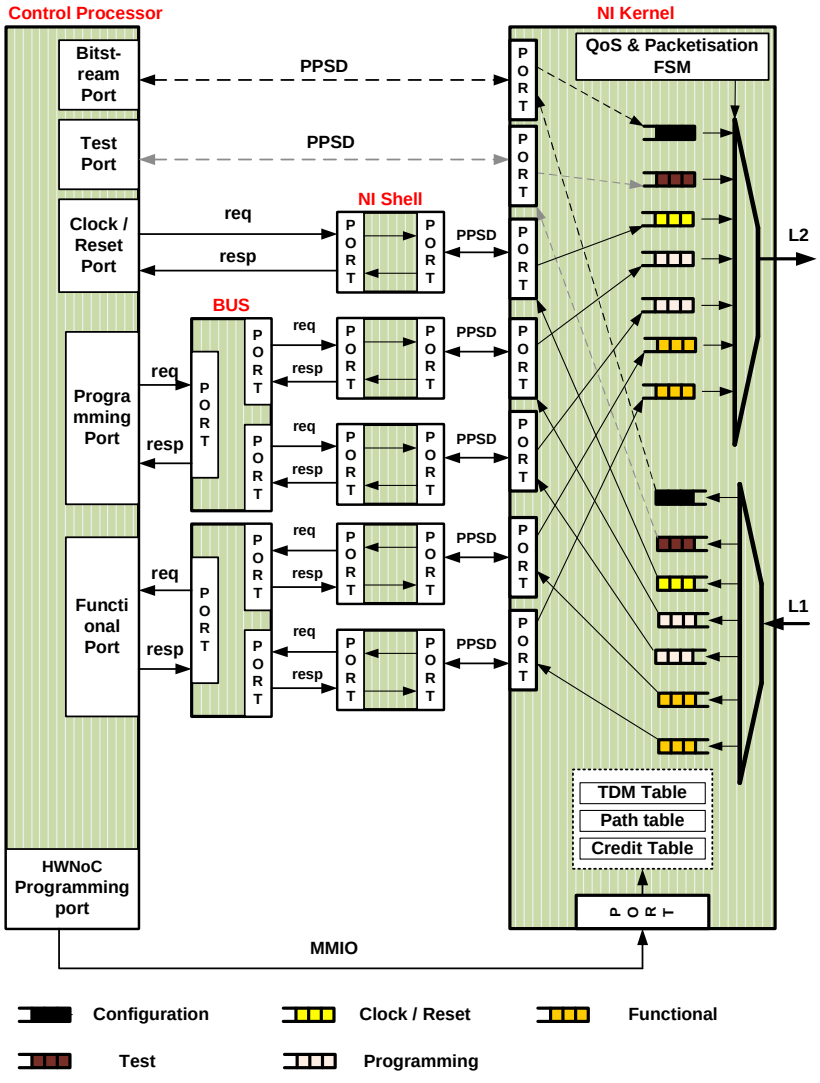


Figure 4.7: Details Architecture of the Control Processor.

portantly, the HWNOC is (re)programmed for each connection that is setup in between the control processor and the destination TCFR. Same is the case with testing, which also makes use of a streaming port to test a particular TCFR.

Figure 4.7 shows that in addition to two streaming ports, the control processor has three memory mapped input output (MMIO) ports that are used to clock / reset, program, and execute application IPs on TCFRs by using the hardwired NoC. However, the control processor first establishes connections, through the hardwired NoC, to the required TCFR. The setting up of a connection is achieved by using a MMIO port, which is the *HWNoC programming* port in Figure 4.6 and 4.7, with a loop-back connection to program the slot, path, and credit tables of the network interface of the control processor. Note that the control processor can program any network interface of the HWNOC by writing to its slots, path, and credit tables. However, the HWNOC is programmed in a similar way as explained in [48]. Therefore, in Figure 4.6 and 4.7, we limit ourself to show the architectural details that are required to configure, test, and program the TCFRs.

The control processor clocks and resets one TCFR at a time. Therefore, it makes use of a single MMIO port (i.e., the *Clock / Reset* port in Figure 4.6 and 4.7) that connects to the HWNOC by using a protocol shell. The HWNOC is reprogrammed each time whenever a TCFR is clocked / reset.

However, due to multiple (sub)applications running concurrently, the control processor requires multiple concurrent programming connections. Therefore, the master bus is attached to the programming port that connects the initiator *Programming* port (on the control processor) to the target MMIO ports of both the TCFRs, see Figure 4.6 and 4.7). The bus attached to the initiator *Programming* port of the control processor directs the request to the appropriate target port, based on the address of the request. For this purpose, the local bus makes use of a demultiplexer that based on the address (received from the sending IP) selects the appropriate network interface shell.

Similarly, the control processor can receive requests from multiple (sub)applications, e.g., request of data from a specific memory location or notification about the completion of an application execution. In this case, the control processor makes use of a slave bus that has multiple ports to receive data from multiple initiators. For instance, the slave bus that is attached to the *Functional* port of the control processor in Figure 4.6 and 4.7), can receive requests from two TCFRs simultaneously.

4.5 Hard Soft Partitioning

In this section, we discuss which parts of the proposed FPGA should be *hard* and which should be *soft*, see Table 4.1.

4.5.1 Hardwired NoC Partitioning

In the following discussion, we provide the hard soft partitioning of different components of a hardwired NoC.

Local Buses of Control Processor

The architecture of (master and slave) local buses of the control processor, as shown in Figure 4.7, is *not dependent on applications* that execute on an FPGA. In our architecture a local bus connects one memory-mapped initiator / target port to multiple target / initiator ports. A local bus is *not configurable* because its architecture / dimensions are fixed at design time. However, the local buses are *programmable* to access any target / initiator port in the network. For this reason, a local bus is *hardwired* in the proposed architecture.

Network Interface Kernel

The architecture of our HWNoC NI kernel is explained in Section 2.2.1. As far as an NI kernel is concerned, it can have a *mix of hard and soft blocks*. The hard soft partitioning is performed with respect to *NI kernel FIFOs* and *NI kernel control* as explained below.

NI kernel FIFOs are the request / response channel FIFOs in the input and output sections. The FIFOs are *dependent on the nature of an application* and are *configurable*. The channel FIFOs should be big enough to support the maximum throughput demands that can be required by an application of a system. However, at design time of a HWNoC, it is hard to predict the worst-case throughput values. The reason is that an FPGA can be used for a number of SoC applications, which can not be determined at an FPGA fabrication time. Hence the channel FIFOs position themselves as a soft candidate, i.e., to be built by using FPGA logic resources. However, as we shall explain in Section 4.9.6, the cost of look up table (LUT) based FIFOs is prohibitive, compared to hardwired FIFO. Hence given that the ratio of *soft* to *hard* is a quite high 35 [84]. Therefore, we can reasonably over-dimensioned the channel FI-

Table 4.1: Hard Soft Partitioning of FPGA with Hardwired NoC.
Abbreviations: Ap = Application, In = Independent, De = Dependent, Mgr = Manager.

FPGA Architecture	**Module**	**Nature of Function**	**Config-urable**	**Progra-mmable**	**Implem-entation**
HWNoC	Local Buses	Ap In	No	Yes	Hard
	NI Kernel FIFO	Ap De	Yes	No	Hard at Hard IP
	NI Kernel Control	Ap In	No	Yes	Hard
	NI Shell	Ap De	Yes	No	Hard at Hard IP
	NI Shell	Ap De	Yes	No	Soft at TCFR
	Router	Ap In	No	No	Hard
TCFR	MTCR	Ap In	Yes	No	Hard
	Bus Macro	Ap In	Yes	No	Hard
	CDC FIFO	Ap In	No	No	Hard
	Bitstream Mgr	Ap In	No	Yes	Hard
	Clock / Reset Mgr	Ap In	No	Yes	Hard
Control Processor	- -	Ap In	No	Yes	Hard

FOs, after considering a worst-case value for them. In an exceptional case where an over-dimensioned channel FIFO still proves insufficient, the remainder of the channel FIFO can be implemented in the corresponding network interface shell, which is explained in the following section.

Next comes the *NI kernel control* that exhibits an *application independent* nature. The *NI kernel control* is *not configurable, but programmable* at run time. More specifically, the (de)packetisation unit in an NI control can be hardwired because of a fixed and predetermined packet format. The scheduler unit takes inputs from a fixed number of control tables and, therefore can also be hardwired. The channel table and the space table, both of which have one entry per network interface kernel port, are dimensioned equal to the number of ports of an NI kernel. Hence both of these can also be hardwired, provided the NI kernel has a fixed number of ports. For these reasons, the control of an NI kernel is implemented as *hard*.

Network Interface Shell

The NI shell, however, is *soft* for the following reasons. First, the port protocol depends on the IP and a single system often contains a number of different port protocols. Second, the depth of a channel FIFO depends on the required bandwidth and latency, which *depends on an application*. The channel FIFO is for a small part in the NI kernel, where it has a fixed size, and for the remainder in the NI shell. Figure 4.8 shows an example of soft NI shells that are connected to DCT and IDCT IPs, and exist in the FPGA reconfigurable plane.

The only NI shells that are *hard* are the following. First, the NI shells that connect to the MMIO programming port of NI kernels, because it always requires and uses a fixed protocol. Those NI shells that connect to hard IPs (whose protocol is design time specific) such as the control processor as shown in Figure 4.7.

Router

The function of a router is to forward a packet, received at its input, to a specific output port. A router in our architecture has nothing to do with applications that have variable QoS requirements. In other words, a router has an *application independent* nature. Therefore a router is *neither configured nor programmed* at run time. A router in our proposed architecture is best implemented as *hard*.

4.5.2 TCFR Partitioning

In the following discussion, we provide the hard soft partitioning of each TCFR component.

MTCR

Starting with the minimum test configuration region, we see that it has *application independent* nature. An MTCR is *configurable* to configure a new functionality on an FPGA, and is *not programmable*. The architecture of an MTCR is implemented as *hard* by the FPGA manufacturers.

Bus Macro

Similarly, a Bus Macro exhibits an *application independent* nature. A Bus Macro can be *configured* on minimum test configuration regions, because it can be placed at different locations of a TCFR. In our architecture, as Bus Macro is placed at the boundaries of a TCFR, which are fixed. For this reason, a Bus Macro is *hardwired* in our architecture.

CDC FIFO

The clock domain crossing FIFOs exhibit an *application independent* nature, because these are meant for data synchronisation across the HWNoC and TCFR boundary. In other words, unlike the channel FIFO, the CDC FIFOs have nothing to do with fulfilling the throughput demands of an application. The CDC FIFOs are *not configurable*, and for this reason the CDC FIFOs are *hardwired* in our architecture.

Bitstream Manager

The bitstream manager exhibits an *application independent* nature. The reason is that for a specific FPGA architecture the bitstream structure, to configure and test the logic block and the programmable interconnect, is always the same. Therefore, the bitstream manager is *not configurable*. However, the bitstream manager is *programmable* to write the bitstreams in different parts of a test configuration functional region. In short, we can say that the bitstream manager is best implemented as *hardwired*.

Clock / Reset Manager

Similarly, the clock / reset manager is *not dependent on the nature of applications* that execute on an FPGA chip, therefore, is *not configurable*. However, clock and reset manager is *programmable* to program the clock of an IP or set / reset an IP that reside on a TCFR. For these reasons, each clock / reset manager is implemented as *hard*.

4.5.3 Control Processor Partitioning

The control processor architecture is *application independent*, because it is used to test, configure, and program multiple applications on test configuration functional regions of the proposed FPGA architecture. In the proposed architecture it is a *hardwired* IP that is programmable and not configurable.

Here, we feel it important to describe the partitioning of the system manager and application managers of our 3-tier model, as described earlier in Section 3.4. The system manager is mapped to the control processor since it is always there; and there exists only one system manager in our architecture. The application managers, of which there are as many as there are applications, could be mapped to the control processor. However, this would introduce composability and scalability issues. Therefore, we map application managers (comprising memories, address generations units, and DMAs, etc.) as soft IPs.

We next explain the implementation details of the proposed FPGA architecture.

4.6 Implementation versus Modeling

In this section, we explain the implementation and modeling details of our proposed architecture. In the following discussion, we will explain how the components of our FPGA architecture (HWNOC, TCFRs, and the control processor) are implemented, see Table 4.2.

4.6.1 Hardwired NoC Implementation versus Modeling

The HWNoC architecture is comprised of multiple network interfaces and routers [43, 44, 48]. Each network interface is further subdivided into a network interface kernel, and a network interface shell.

The network interface kernel architecture is implemented both in systemC and VHDL languages [48, 121]. It is a cycle-accurate implementation, which means in each cycle, we can keep track of the NI kernel functionality. A network interface shell is implemented both in systemC and VHDL languages [48]. The router architecture is implemented both in systemC and VHDL languages [48]. The NI kernel, NI shell and router have a cycle-accurate implementation, which means in each cycle, we can keep track of their functionality.

Table 4.2: Modeling Vs Implementation of the Proposed Architecture.

FPGA Architecture	Module	Modeled	Implemented
HWNoC	Local Buses	System C (Cycle-Accurate)	VHDL
	NI Kernel FIFO	System C (Cycle-Accurate)	VHDL
	NI Kernel Control	System C (Cycle-Accurate)	VHDL
	NI l Shel	System C (Cycle-Accurate)	VHDL
	Router	System C (Cycle-Accurate)	VHDL
TCFR	MTCR	-	-
	Bus Macro	-	-
	CDC FIFO	-	-
	Soft IPs	System C (Trans-Accurate)	VHDL -
	Bitstream Manager	System C (Cycle-Accurate)	- -
	Clock / Reset Manager	System C (Trans-Accurate)	- -
Control Processor	- -	System C (Trans-Accurate)	- -

4.6.2 TCFR Implementation versus Modeling

The test configuration functional region (TCFR) architecture comprises multiple minimum test configuration regions (MTCRs), Bus Macros, CDC FIFOs, a bitstream manager, and clock / reset manager. The TCFR architecture is modeled in a cycle-accurate SystemC model[4], and is shown in Figure 4.8.

The minimum test configuration regions (MTCRs) of a TCFR are configured to implement a certain functionality / IP. Therefore, instead of modeling the architecture of an MTCR, we model the functional model of a soft IP that is configured on the MTCRs. This is done by modeling the loading of bitstreams, inspecting the bitstream, and based on this selecting the behavioral model of

[4]The modeling of a TCFR is not performed at a bit-accurate SystemC level.

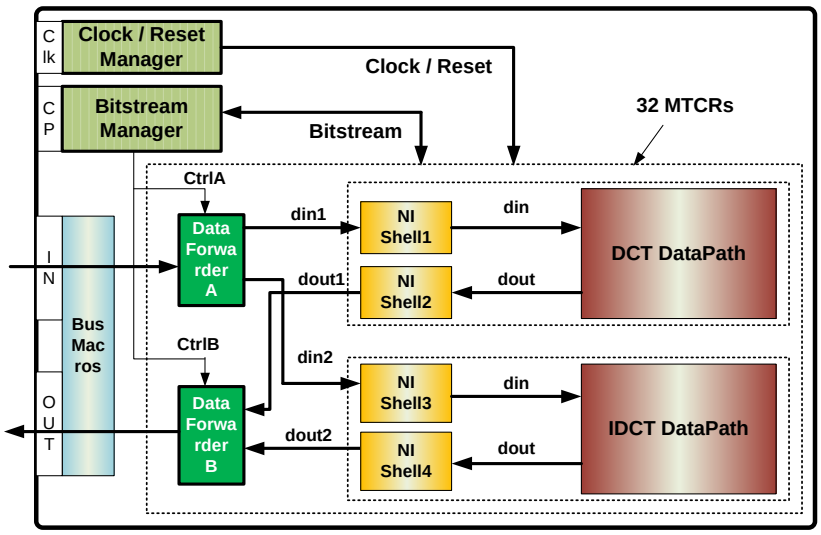


Figure 4.8: Data Forwarders in the SystemC Model of a Test Configuration Functional Region.

the soft IP that has been configured [145]. It is important that multiple application IPs can be dynamically swapped in or swapped out of a TCFR at run time. In our systemC simulation, we make use of data forwarders to decide which IP is swapped in / out. The control inputs of data forwarders are triggered by the bitstream manager. The bitstream manager after loading the bitstream of an IP (e.g. DCT IP in Figure 4.8), sends the control signal to data forwarders to enable a a specific IP (in this case DCT) to send / receive data to / from the HWNOC. In our architecture data forwarders and the TCFR controlling logic are the modeling artifacts. The BUS Macros are are part of the proposed architecture. However, these are yet to be implemented, which means the cost of Bus Macro in terms of timing is not known, and therefore is not part of our calculations. Similarly, clock domain crossing FIFOs are also part of the proposed architecture, but yet to be implemented.

The architecture of the bitstream manager is modeled in cycle-accurate SystemC, which means in each cycle we can keep track of: i) receiving of a bitstream, ii) inspecting the bitstream, and (iii) storing of the bitstream in the respective MTCR memories. The architecture of the clock / reset manager is modeled in transaction-accurate SystemC language.

4.6.3 Control Processor Implementation versus Modeling

The control processor architecture is a programmable hardwired IP that is connected to the first network interface of the HWNoC. A functional model of the control processor is implemented in transaction-accurate SystemC language. This means, like TCFR, the functional model of the control processor does not have any instruction set. The functionality of control processor is modeled that include: i) bootstrapping of the FPGA system, ii) programming of the HWNoC, iii) configuration, testing, and programming of the required functionality into TCFR(s) [145].

We next explain the possible extensions of hardwired network on chip in the proposed FPGA architecture.

4.7 Hardwired NoC Extensions

The proposed hardwired NoC in FPGA has the flexibility to be extended from the architecture point of view as well as from its applicability point of view. It is explained in the following sections.

4.7.1 Soft & Multi FPGA NoC

The architectural extension is to allow the hard NoC to be expanded by soft routers and NIs, as shown in Figure 4.9. This is useful when functional regions are large and more NI kernel ports are required than are present on the hard NI kernels near the region. However, because the hard NOC will be running at higher frequencies than can be achieved with a soft NoC, it passes through the clock domain crossing FIFOs that serve as the bridge between hard NoC and the soft NoC.

A related extension is the use of the functional IO to connect the NoCs on multiple FPGAs, to create a multi-FPGA NoC [24, 103, 109, 134]. The NI kernel on one NoC converts packets to streaming data, which is transported over the functional IO to the other FPGA, where it is re-packetised by the NI kernel on the other NoC, which can be soft or hard.

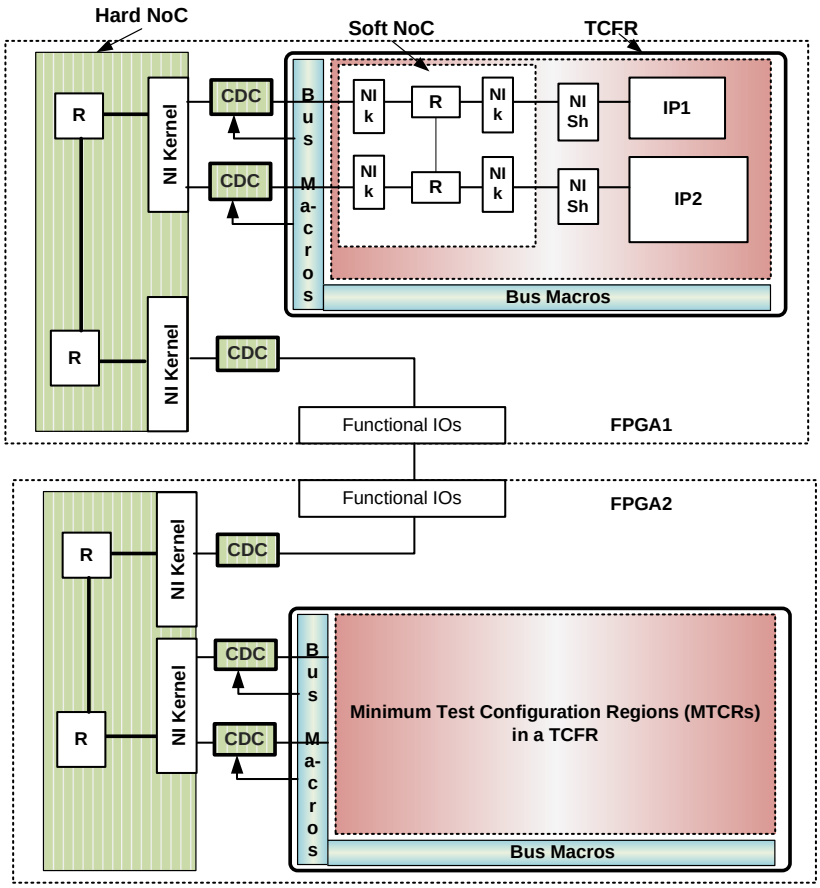


Figure 4.9: Architectural Extensions of the Hardwired NoC.

4.7.2 Applicability Extensions

In addition to using a hardwired NoC to provide inter-IP communication traffic for multiple on-FPGA applications, the hardwired NoC applicability can be extended for the following purposes:

1. *Secured communication:* The use of a functional interconnect for decrypted bitstreams is in general not safe, because it allows a malicious IP to capture secret information. However, first note that a HWNoC is a (virtual) point-to-point and not a broadcast interconnect. The connection between the two IPs is programmed at run time. Moreover, HWNoC with guaranteed communication services also decouples the

temporal behaviors of communicating IP [135]. This removes the possibility to obtain secret information either from the timing of communication (events), or from a malfunctioning system after injecting spurious data.

2. *Functional operations on bitstreams:*

   (a) By using the hardwired NoC and the ability to program connections between FPGA regions, the bitstreams can be dynamically reloaded and / or relocated at run time.

   (b) Functionally operating on bitstreams can be used to check, if IP configurations have been tampered with since they were installed. This can be achieved by streaming the bitstream to a (hard) encryption or cyclic redundancy check (CRC) IP and comparing the output with the original encrypted bitstream or a smaller checksum.

4.8 Architectural Limitations

Following are the limitations in our proposed architecture:

1. *Test Configuration Functional Region:* The test configuration functional regions in our FPGA architecture are equal sized, and poses identical number of logic resources. Hence currently we limit ourself to a rather homogenous distribution of logic resources. However, this might be an issue with a domain specific FPGA, where the distribution of resources can be uneven. Our TCFR architecture does not support a configuration granularity that is less than an MTCR unit, i.e., 16 configurable logic blocks or 128 LUTs. In our current implementation, a single clock is delivered to all the MTCRs of a TCFR. This means all the IPs in a TCFR are bound to run at a single frequency. Additionally, the memory-mapped set / reset register has 32 signal lines, where each signal is used to set / reset an IP block. This may limit the number of IPs in a TCFR to a maximum of 32. Currently, we make use of conventional BUS Macros at the boundaries of a TCFR. This in turn, is useful for intra-IP communication that spans multiple TCFR.

2. *Network Interface:* The architecture of our HWNOC NI has a design time fixed slot table size, number of ports, and FIFO depths. This means the resource dimensioning is not application-specific. This can give rise

to; (a) under-utilization of resources when the on-FPGA applications require less than the available ones, and (b) shortage of resources when on-FPGA application require more than the available ones.

3. *Single Host for Control Infrastructure:* We can add more hosts. However, we limit ourself to a single host for our programming the control infrastructure. This might impose scalability problem; (a) as the number of applications increases, (b) (re)configuration and programming is performed at a frequent basis.

4.9 Results and Analysis

We model our HWNOC architecture in SystemC using the design flow of [42, 45]. The flow output also provides us with the technology-independent RTL VHDL of the HWNoC components, which include network interfaces and routers. We synthesize, place, and route (to the pads) different instances of these HWNOC components onto a Virtex-4 XC4VLX200ff1513-11, for which we use Xilinx ISE 10.1. The Virtex-4 chip contains 178176 LUTs, and is fabricated in CMOS technology with a 90 nm Copper CMOS process [158]. The synthesis numbers give us the *soft* area overhead for the hardwired NoC components. At the moment, our proposed HWNoC is not implemented on an FPGA. Therefore, we use [84] to compute *hard* area in terms of *LUT-equivalent* from the synthesised *soft* area[5]. The authors in [84] report area cost ratio of 35 between ASIC and FPGA, which means we compute the area of *hard* router after dividing the area of *soft* router by a factor of 35. The reason to show the area of the hard components in LUT-equivalent rather than mm^2 is 1) to compare more easily and 2) conversely converting area in LUT to mm^2 is hard because the FPGA vendors do not provide any information (in the specifications) about the area of an LUTs in mm^2.

In the remaining section, for the new FPGA architecture, we will find out the area overheads for HWNoC and TCFR. To calculate the area overhead, we used different instances of NI (Section 4.9.1), router (Section 4.9.2), and TCFR (Section 4.9.3). Afterwards, we explore the design space for different dimensions of HWNoC and TCFR, (Section 4.9.4). At the end we compare our hardwired NoC with its soft variant, in terms of the area and functional performance (Section 4.9.6).

[5]For our convenience and due to the lack of tools, we do not synthesise the hard blocks and instead use the work of [84] to extract the area of hard blocks in terms of *LUT-equivalent*.

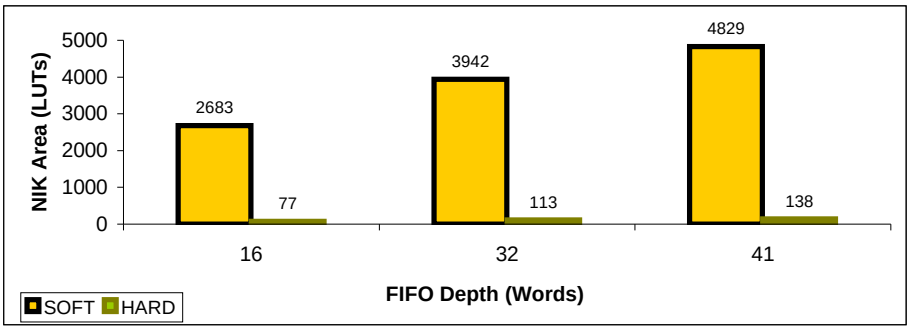


Figure 4.10: NI Kernel with Variable FIFO Depths (Slot Table = 166 time-slots, and Ports = 2).

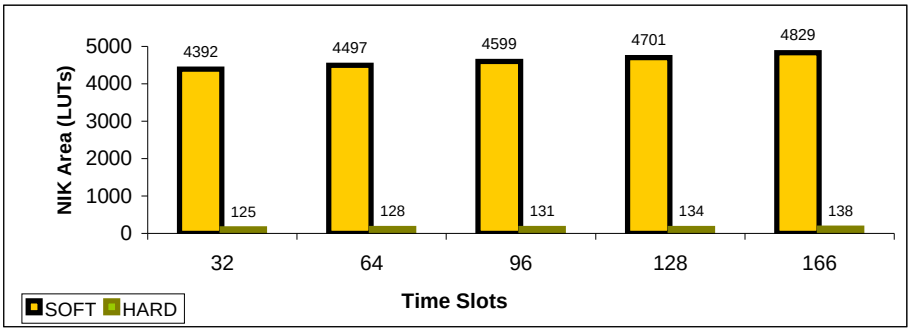


Figure 4.11: NI Kernel with Variable Time-Slots (Ports = 2, and FIFO depth = 41 words).

4.9.1 Network Interface Variations

A network interface is sub-divided into a network interface kernel, and multiple protocol shells that are attached to NI kernel ports. Therefore, we evaluate their area cost, independently.

We synthesize different instances of an NI kernel after varying (a) the depth of its FIFOs, (b) slot table size, and (b) the number of ports. The reason to perform the evaluation is because, as explained earlier in Section 4.5.1 that the slot table and FIFO depths should be dimensioned accordingly to fulfill the throughput requirements of an application. In addition, the number of ports are also varied because each port is associated with two channel FIFOs that incurs a significant area overhead. This means, an *increase or decrease in the count of ports* can significantly affect the area of an NI kernel. In our experiments, we vary one parameter (i.e., FIFO depth, slot table size, or port number) at one

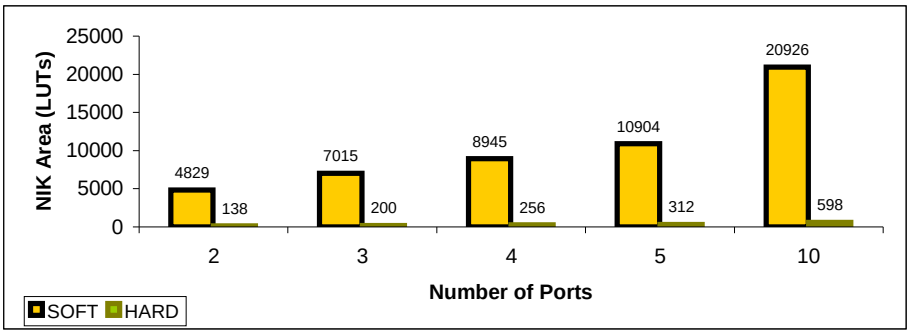


Figure 4.12: NI Kernel with Variable Ports (Slot Table = 166 time-slots, and FIFO depth = 41 words).

time to perform the evaluation.

First, we vary the depth of channel FIFOs, but keep the slot table size and NI kernel ports constant at 166 time-slots and 2 ports respectively. Figure 4.10, shows that by increasing the FIFO size by 16 words, the area of *hard NI kernel* increases by 36 LUT-equivalent.

Similarly, we changed the slot table size to analyse its impact on the area of an NI kernel. Figure 4.11 shows that the area of a *hard NI kernel* is increased by approximately 4 LUT-equivalent, when the slot table size is increased by 32 time-slots.

Next, we vary the number of ports in the NI kernel. However, we keep the slot table constant at a maximum of 166 slots, and the FIFOs are deep enough (i.e., 41 words) to store a bitstream frame. Figure 4.12 illustrates the impact of variable number of ports on the area of a *hard NI kernel*. It shows that the area increases linearly as we increase the number of ports. Figure 4.12 also shows that each additive port in a *hard NI kernel* induces an area overhead of 60 LUT-equivalent.

We make use of soft NI shells. However, hard NI shells might be required for some embedded processing units, e.g., PowerPC, an encryption unit, an embedded memory controller, etc. Therefore, we evaluate the cost of *hard and soft NI shells*. In our architecture, we limit our self to *NI shells* for DTL protocol, the area cost of which is found to be 90 LUTs for a soft shell and 3 LUT-equivalent for a hard NI shell.

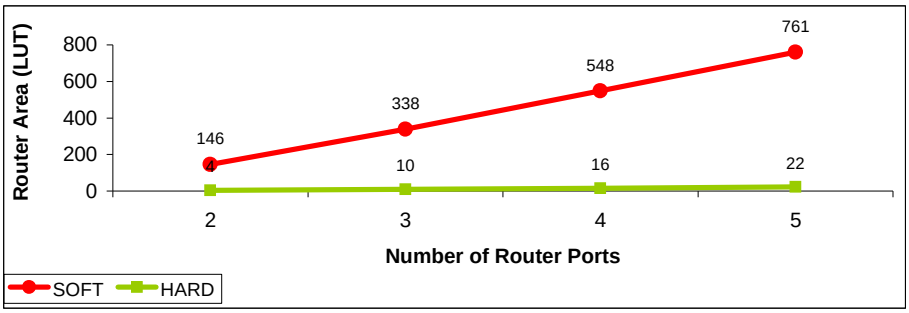


Figure 4.13: Router Area Overhead with Variable Number of Ports.

4.9.2 Router Variations

To determine the area of a *hard* router, we synthesized a *soft* router with 32 bit data width. We then varied the router ports to evaluate its area overhead. Figure 4.13 illustrates that as we add a port to a *hard router*, its area is increased by 6 LUT-equivalent. In our experiments, we use mesh topology for the HWNOC that can have a router with 5 ports in maximum. As shown in Figure 4.13, the area of a hard router with 5 ports is 22 LUT-equivalent.

4.9.3 Test Configuration Functional Region Variations

The logic plane in our architecture is comprised of multiple TCFRs, each of which holds an isolated bitstream manager and configuration architecture. Therefore, in contrast to the conventional FPGA that contains a single bitstream manager, our proposed FPGA has multiple ones. Hence we need to evaluate the area overhead that could be incurred by multiple instances of these bitstream managers. To find out the area overhead of the configuration architecture in a TCFR, we synthesized its components, individually. This accounts for the frame header register, frame data registers, the address decoder, associated streaming port, and the de-multiplexer unit.

Figure 4.14 illustrates the cost of the bitstream manager for different TCFR areas that can be found in our FPGA. For our experiments, we change the size of a TCFR from 1 k LUTs to 32 k LUTs. Note that an MTCR in our architecture equals to 128 LUTs. As shown in Figure 4.14, the area of the bitstream manager changes by approximately 10 LUT-equivalent when the size of a TCFR is changed from 1 k LUTs to 32 k LUTs. The change in area is mainly due to the address decoder, whereas the remaining hardware, i.e., frame

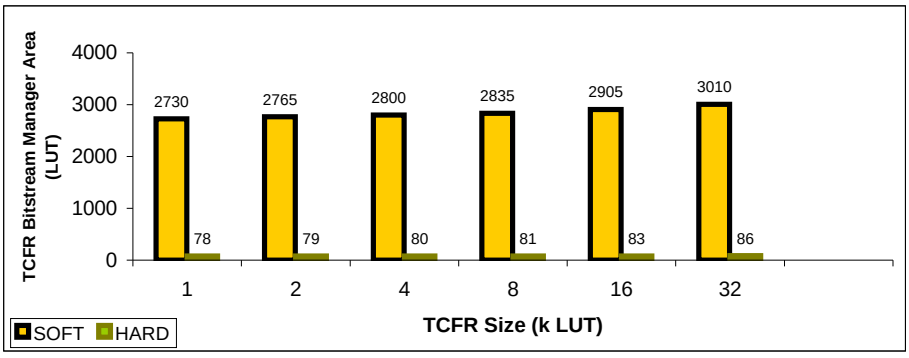


Figure 4.14: Bitstream Manager Area Overhead with Variable Sizes of a TCFR.

registers, de-multiplexer are not affected by changing the TCFR size.

4.9.4 Design Space Exploration with Constant TCFR Size

In this section, we explore the design space for the proposed FPGA architecture, and evaluate the cost and benefit of HWNoC for the given FPGA chip. We first present the specifications of the target architecture in Table 4.3 that are used to explore the design space.

Table 4.3: Specifications of the Target FPGA Architecture.

1	FPGA Area in LUTs = 178176
2	TCFR Size = 32 MTCR $\simeq$ 4000 LUTs
3	NI kernel with 2 ports, 166 time-slots, and 41 word FIFOs
4	Router with 5 ports
6	1 NI per Router
7	1 NI-Router pair per TCFR
8	Bandwidth per Hard NI = 2 GB/s
9	Bandwidth per Soft NI $\simeq$ 0.5 GB/s

For the convenience of the reader, we present the (soft & hard) area numbers of above-mentioned NoC modules, and TCFR bitstream manager area overhead in Table 4.4 that have earlier been discussed in the previous Sections. However, the last two rows of the Table are not explained earlier, as we will describe these later in Section 4.9.6, that hard NoC run at 500 MHz and soft NoC run at 118 MHz. This means, the bandwidth that is available per hard NI is 32 bits x 500 MHz = 16000 Mb/s or 2 GB/s. Similarly, the bandwidth available for the

soft NI is 32 bits x 118 MHz = 3776 Mb/s or approximately 0.5 GB/s.

Table 4.4: Soft and Hard Values of Different Components in FPGA.

Module	Soft (LUTs)	Hard (LUTs Equivalents)	Figure Reference
NI kernel (2 ports)	4829	138	Figure 4.12
Router (5 ports)	761	22	Figure 4.13
TCFR Overhead	-	80	Figure 4.14

TCFRs per FPGA with Soft & Hard NoC

In this section, we find out the maximum number of TCFRs that are present in an FPGA with soft NoC and hard NoC. A TCFR in an FPGA with soft NoC represents a partial reconfiguration region without a bitstream manager and configuration architecture, which the HWNOC FPGA does have.

When a soft NoC is used, then we can calculate the maximum number of TCFRs by using Equation 4.1 and Equation 4.2. Similarly with a HWNOC, we can calculate the maximum number of TCFRs by using Equation 4.3 and Equation 4.4.

$$\#\ of\ TCFRs\ per\ FPGA\ = FPGA\ LUTs\ /(TCFR\ size + Soft\ Cost\ per\ TCFR) \tag{4.1}$$

$$\begin{aligned} Soft\ Cost\ per\ TCFR\ &= Soft\ Cost\ of\ NoC\ per\ TCFR \\ &= Soft\ (Area)\ Cost\ of\ NI\ kernels\ and\ Routers\ per\ TCFR \end{aligned} \tag{4.2}$$

$$\#\ of\ TCFRs\ per\ FPGA\ = FPGA\ LUTs\ /\ (TCFR\ size + Hard\ Cost\ per\ TCFR) \tag{4.3}$$

$$Hard\ Cost\ per\ TCFR\ = (Cost\ of\ R + NI + bitstream\ manager\ per\ TCFR)\ /35 \tag{4.4}$$

For instance, to calculate the number of TCFRs in an FPGA with soft NoC, we obtain the soft cost per TCFR by using Equation 4.2, and Table 4.3 and

Table 4.4, which is found to be 4829 + 761 = 5590 LUTs. Afterwards, the soft cost per TCFR is used in Equation 4.1 to obtain the total number of TCFRs in an FPGA, which is found to be 178176 / (4096 + 5590) = 18 TCFRs. Here 4096 represents the size of a single TCFR in LUTs.

When a HWNOC is used, a maximum of 41 TCFRs are possible for an FPGA with specifications of Table 4.3 and Table 4.4. The cost per TCFR is obtained by dividing the soft cost of a TCFR by 35, where 35 is the area cost area of soft to hard [84]. In addition, we take into account the area cost of the bitstream manager which is found to be 80 LUT-equivalent for a TCFR of 4 k LUTs. Hence the hard cost of TCFR is (5590 / 35) + 80 = 240 LUT-equivalent. This accounts for a total of 178176 / (4096 + 240) = 41 TCFRs.

Cost and Benefit of Soft & Hard NoC per FPGA

In this section, we obtain the (area) cost and (bandwidth) benefit that is incurred by soft and hard NoC in an FPGA. We use Equation 4.5 to calculate the area cost, where *Cost* represents the soft cost and hard cost for soft and hard NoC respectively. Similarly, we calculate the benefit in terms of bandwidth that is available to FPGA application due to the presence of soft and hard NoC. This is obtained by using Equation 4.6 and Equation 4.7. This involves multiplication of total number of TCFRs with the bandwidth available per TCFR, i.e., 2 GB/s (see Table 4.3).

$$Cost\ per\ FPGA = \#\ of\ TCFRs * Cost\ per\ TCFR \tag{4.5}$$

$$Bandwidth\ per\ FPGA = \#\ of\ TCFRs * Bandwidth\ per\ TCFR \tag{4.6}$$

$$Bandwidth\ per\ TCFR = \#\ of\ NIs\ per\ TCFR * Bandwidth\ per\ NI \tag{4.7}$$

For instance, when we have TCFRs of 4 k LUTs in the FPGA chip of Table 4.3, then the area cost of soft NoC is approximately 18 * 5590 = 101 k LUTs. An FPGA with soft NoC can support a maximum bandwidth of 18 * 0.5 = 9 GB/s. However, an FPGA with HWNOC incurs an area cost of approximately 41 * 240 = 10 k LUTs and provides a bandwidth benefit of 41 * 2 = 82 GB/s.

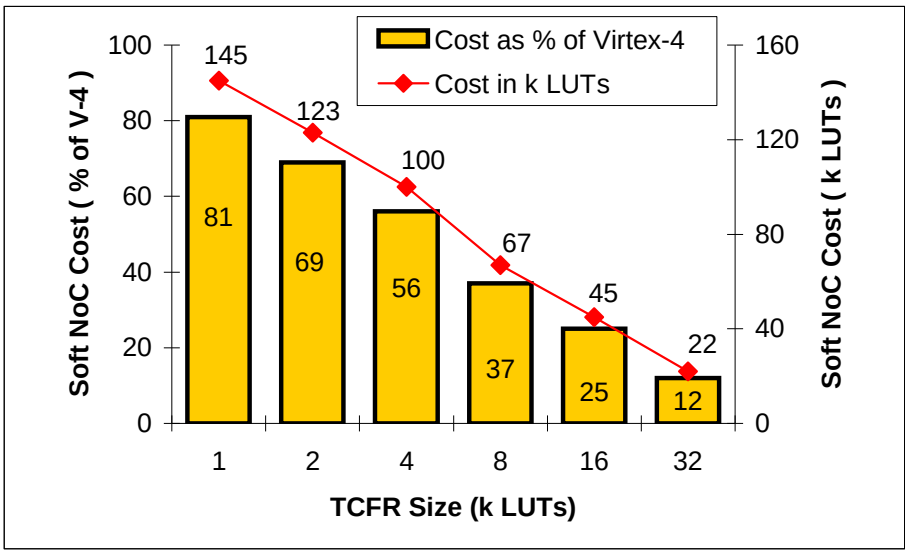


Figure 4.15: Soft NoC Cost for a Virtex-4 FPGA with Variable TCFR Sizes.

4.9.5 Design Space Exploration with Variable TCFR Size

In this section, we explore the design space for the proposed FPGA architecture, by varying the sizes of TCFRs. The specifications of Table 4.3 are still valid except that a particular FPGA chip can have TCFRs, whose size can range from 1 k LUTs to 32 k LUTs. The evaluation is performed to see the impact of TCFR size on 1) the number of TCFRs per FPGA, b) the cost and benefit of soft & hard NoC per FPGA, and c) the impact on area and bandwidth of an IP in an FPGA.

The cost of the soft NoC is approximately 150 k LUTs when an FPGA consists of TCFRs of 1 k LUTs, see Figure 4.15. At this point the cost of the soft NoC accounts to approximately 80% of Virtex-4 chip area. Importantly, in contrast to an exponential increase in the area of a TCFR, the cost of soft NoC per FPGA decreases in a linear fashion, as shown in Figure 4.15. The reason can be found by observing the pattern of values in Col 4 and 5 of Table 4.5, as the size of a TCFR increases. For each row, the combined values of Col 4 and 5 of Table 4.5 is approximated to FPGA total area. Though, the size of a TCFR is increased in an exponential fashion, the aggregated area of all the FPGA TCFRs increases linearly by approximately 30 k LUTs, and at the same time the total area cost of soft NoC decreases linearly by the same proportion, i.e., decreases by 30 k LUTs, see Col 5 of Table 4.5.

Table 4.5: Results of Design Space Exploration with Variable TCFR Size. Total LUTs in V4 FPGA = 178 k LUTs, Soft NoC Cost per TCFR = 5.8 k LUTs.

TCFR Size	# of TCFRs	Total TCFR Area	Soft NoC Cost	
k LUTs		k LUTs	k LUTs	%V4 Area
1	26	26	147	84
2	22	44	125	71.2
4	18	72	101	58.2
8	12	96	68	38.8
16	8	128	45	25.9
32	4	128	22	12.9

As far as throughput benefit of soft NoC is considered, this also varies by changing the size of TCFRs in an FPGA. An FPGA with TCFRs of 1 k LUTs and specifications of Table 4.3 (i.e., one NI-Router pair per TCFR) can have a maximum of 26 TCFRs. This in turn makes it possible to provide a maximum bandwidth of 13 GB/s for an FPGA applications, see Figure 4.16.

The area cost of soft NoC can be reduced to a reasonable 12% of Virtex-4, if we use TCFRs of 32 k LUTs. However, it reduces the throughput achieved to 4 GB/s, as shown in Figure 4.16.

The cost of the HWNOC is approximately 34.4 k LUT-equivalent when an FPGA consists of TCFRs of 1 k LUTs, see Figure 4.17. Moreover, at this point the cost of the HWNOC accounts to approximately 20% of Virtex-4 chip area.

As far as throughput benefit of HWNOC is considered, this varies by changing the size of TCFRs in an FPGA. An FPGA with TCFRs of 1 k LUTs and specifications of Table 4.3 (i.e., one NI-Router pair per TCFR) can have a maximum of 140 TCFRs, see Figure 4.18. This in turn makes it possible to provide a maximum bandwidth of 280 GB/s for an FPGA applications.

The area cost of HWNoC can be reduced to a reasonable 10% of Virtex-4, if we use TCFRs of 4 k LUTs. Consequently, the benefit also fall to 82 GB/s, as shown in Figure 4.18. However, this is approximately 21 times (82 / 4) higher than the throughput of an FPGA with soft NoC, which costs approximately 10% of FPGA chip.

From these trends we draw the following conclusions. A soft NoC is feasible (defined as $\leq 10\%$ area cost) only for a small number (4) of large (32 k LUTs) IP, exactly when it is not required. A hard NoC is feasible for small IPs (≥ 4 k LUTs), allowing a maximum of 41 such IPs on a Virtex-4.

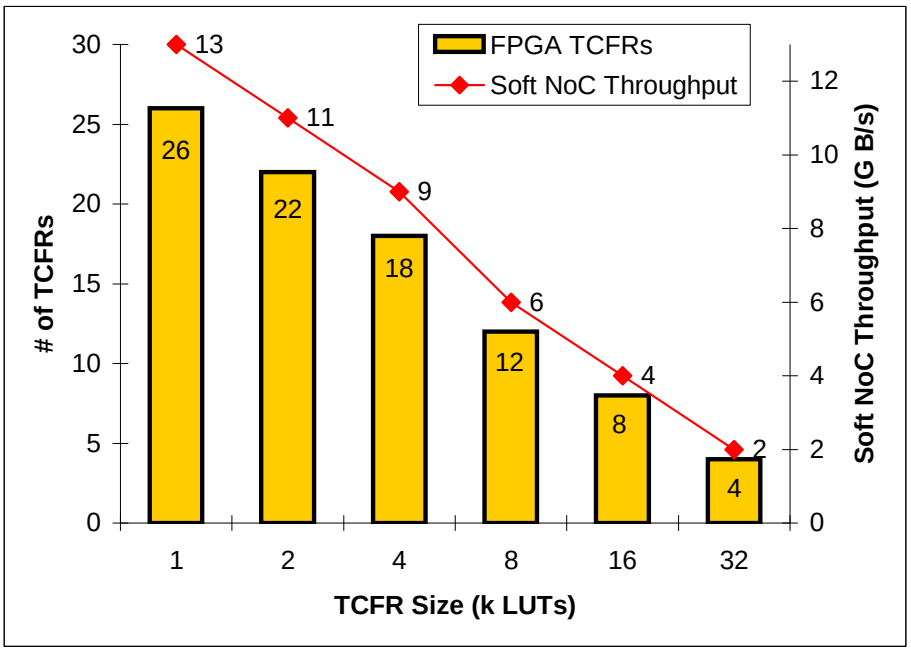


Figure 4.16: Soft NoC Benefit for a Virtex-4 FPGA with Variable TCFR Sizes.

4.9.6 Area & Functional Performance Comparison of Soft & Hard NoC

In this section we compare the area in mm^2, and functional performance of soft and hard NoC.

Area

We synthesised and mapped a NoC onto a Virtex-4 4vlx200ff1513-11, for which we used Xilinx ISE 10.1. The Virtex-4 contains 178176 LUTs. It is fabricated in CMOS technology with a 90 nm Copper CMOS process.

Table 4.6 shows the area results for soft and hard modules for the example NoC. The router and NI instances have been implemented to timing back-annotated layout (including scan chains) [43, 120, 121]. Based on these instances the design flow accurately estimates the area of any generated module (i.e., network interface, router) instance in 130 nm CMOS technology (Table 4.6 Column 4). Note that for Column 5, we refer to the work of [48] that has synthesis results for different instances of routers and network interfaces

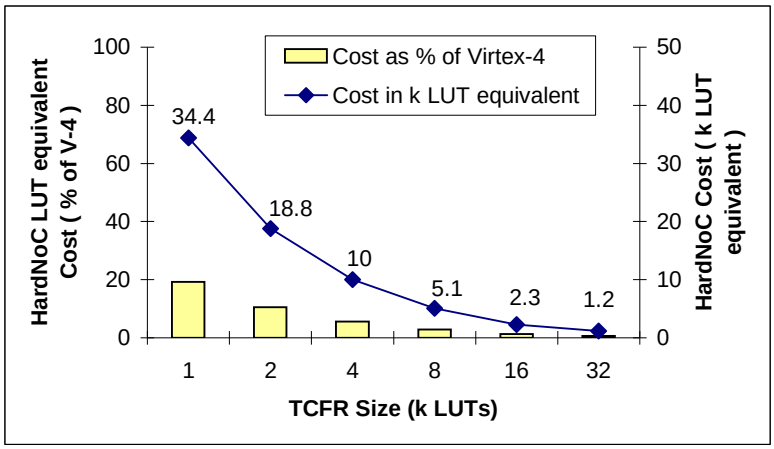


Figure 4.17: HWNoC Cost for a Virtex-4 FPGA with Variable TCFR Sizes.

Table 4.6: Area of Network on Chip Components.

Module Nature	Module Name	# LUTs	mm^2 (130 nm)	mm^2 (90 nm)	mm^2 (90 nm)
Hard	NI	-	0.13	**0.064**	-
	Router	-	0.33	**0.016**	-
Soft	NI Kernel	**4392**	-	2.24	5.7
	NI Shell	**180**	-	-	0.234
	Router	**761**	-	0.56	0.99

of our example NoC in 90 nm CMOS technology (Table 4.6 Column 5).

In Table 4.6, the bold numbers are used to derive the remaining data in the following manner. For the fifth column, we derive the area of soft components from the 90nm-equivalent hard components [48]. For instance, the area (in mm^2) of a soft router is computed by multiplying the area of the hard router by a factor 35, reported to be the cost ratio between ASIC and FPGA in [84]. This means the soft variant of a hard router, which occupies $0.016mm^2$ in 90 nm, would occupy 0.56 mm^2 area in a 90 nm FPGA. The hard NI area is similarly scaled and divided over the soft NI kernel and shell in the ratio of their number of LUTs.

We ensured that all calculations, here and below, are conservative, i.e. underestimate cost of the soft NOCs, and overestimate cost of hard NOCs. As an example, an alternative calculation of router and NI area, shown in italics in the right-most column of Table 4.6, uses the area per LUT. The area is com-

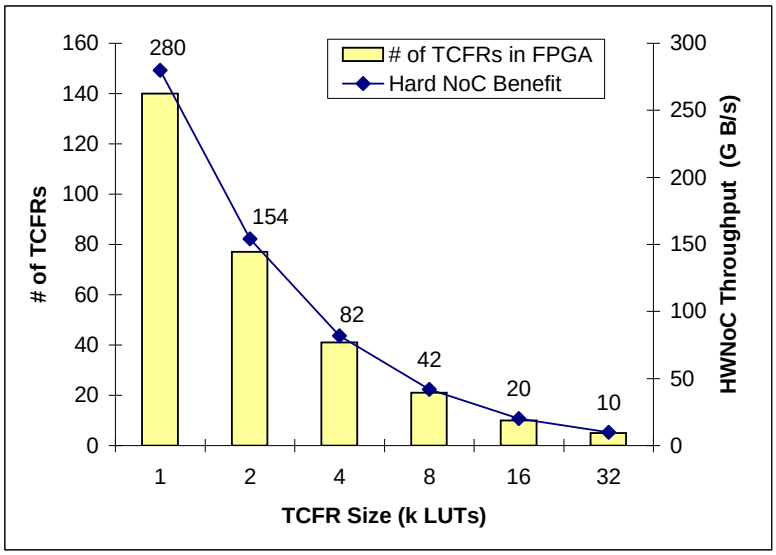


Figure 4.18: HWNoC Benefit for a Virtex-4 FPGA with Variable TCFR Sizes.

puted by dividing the number of required LUTs by the die size of the Virtex-4 device (735 mm^2, estimated from www.fpga-faq.org). We divide the die size by 3, considering that the device contains embedded blocks (such as I/O pins, DSP, memories). In this case, the area ratio of soft:hard is approximately 70. Our analyses remain within the same order of magnitude.

Functional Performance

We now turn our attention to the speed of the NoCs. The ASIC 130 nm hard implementation of ÆTHEREAL operates at 500 MHz (worst-case military) [43]. The example soft NoC described above operates at 118 MHz (NI) and 124 MHz (router) in the Virtex-4 (90 nm technology). We provide a conservative estimate of functional performance, because we ignore the speed increase of hard NoC from ASIC 130 nm to FPGA 90 nm. For a fair benchmark of soft and hard NOCs, the performance:cost ratio ($bit/sec/mm^2$) should be compared in 90 nm technology. With equal topologies and architectures, we compare the raw link bandwidths (operating frequency times 32 bits) per area (of R+NI):

$$\begin{array}{lll} \texttt{soft} & 32\,bit * 118\,MHz/2.8\,mm^2 = 1348.5 & 1 \\ \texttt{hard} & 32\,bit * 500\,MHz/0.08\,mm^2 = 200000 & 148 \end{array}$$

Thus, the performance of a hard NOC is 148 times times better than a soft NOC.

4.10 Conclusions

In this chapter we presented the architecture of an FPGA that was comprised of two planes, a communication plane and a logic plane. The HWNoC served as the communication plane to transport the unified data, i.e., test, configuration, functional, and control in between the test configuration functional regions (TCFR). The TCFRs, on the contrary, served as the logic plane to provide the computing and storage resources for application IPs.

The HWNoC provided scalable IP integration by taking no share from the logic plane. Hence leaving the full FPGA logic plane (i.e. TCFRs) for an application IPs. Each TCFR had its own test and configuration architecture. However, the TCFR of the proposed FPGA architecture were not isolated at functional level. This means that the logic plane of an FPGA was not divided into multiple disjoint regions. Therefore, a soft IP can be placed in MTCRs that belong to different test configuration functional regions. The interaction between the HWNOC and a TCFR was made possible by using clock domain crossing FIFOs and BUS Macros. The CDC FIFOs ensured that data is not lost at the boundary of HWNOC and a TCFR, which could operate at different frequencies. The BUS Macros, on the other hand, provided persistent connections between the HWNOC and a TCFR, even during the dynamic run time reconfiguration.

Our results and analysis show that for a Virtex-4 chip, we had 140 TCFR of 1 k LUT each and a HWNOC that costed approximately 19.6% of the Virtex-4 chip. However, the HWNOC benefit came in the form of 280 GBytes/sec bandwidth that could be used to transport test, configuration, functional, and control data in between the TCFRs. Moreover, when the performance of our HWNOC was compared to a soft NoC (normalised to equivalent area), it was found to be 148 times better.

5

Preparing the FPGA System at Compile Time

In this chapter, first we provide architecture and application specifications in terms of definitions in Section 5.1. Then, we apply our PUMA scheme to perform the binding of input applications on the target FPGA architecture, see Section 5.2. PUMA limitations are provided afterwards in Section 5.3. Thereafter, we present the results and evaluations of the proposed PUMA scheme in Section 5.4. In this section, the performance and scalability of our PUMA scheme is evaluated for multiple combinations of applications and target FPGA architecture. Lastly, we end this chapter with conclusions in Section 5.5.

5.1 Architecture and Application Specifications

In this section, we provide the specifications of the target FPGA architecture and input applications. Then, we define the objectives that are required to achieve a successful binding of input applications on the target architecture.

5.1.1 Architecture Specifications

An FPGA with a hardwired NoC [41] is shown in Figure 5.1A. An FPGA consists of multiple FPGA nodes (Fnodes), and physical links that FPGA nodes use to transport data.

Each Fnode is further decomposed into three nodes, i.e., a test configuration functional region (TCFR), router, and network interface (NI). A test configuration functional region (TCFR) is used to place application computational resources (i.e., IPs). A TCFR consists of multiple configurable logic blocks

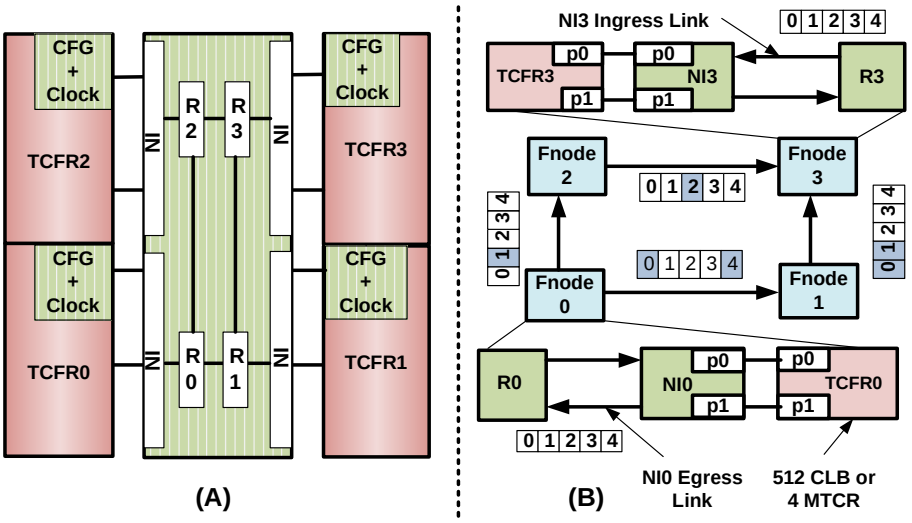


Figure 5.1: FPGA with Hardwired NoC: (A) High level Architecture, (B) Architecture Resource Details.

(CLBs), as described earlier in Section 4.3. The remaining two nodes of an Fnode, i.e., a router and a network interface are used to provide communication among an application IPs.

Each physical link has a slot table of fixed number of time-slots. The router and network interface are connected through exactly one egress and one ingress link. Two Fnodes share data by using their routers.

We illustrate the above discussion by using *an example architecture*, Figure 5.1B. It shows an FPGA with 4 Fnodes. Here $Fnode_0$ and $Fnode_3$ are expanded for the illustration. Each Fnode comprises a TCFR with 512 CLBs, an NI with 2 ports, and a router. Figure 5.1B shows that each FPGA link has a slot table 5 time-slots, where the allocated slots are shown with dark colors. Moreover, Figure 5.1B shows the empty slot tables for the egress link of NI0 and ingress link of NI3. We have not shown all the links in between the Fnodes to keep the things simple, e.g., $Fnode_1$ to $Fnode_0$ link is missing in Figure 5.1B.

5.1.2 Application Specifications

An application task graph is a directed graph, which consists of IP cores that compute and store functional data of an application, and the communication

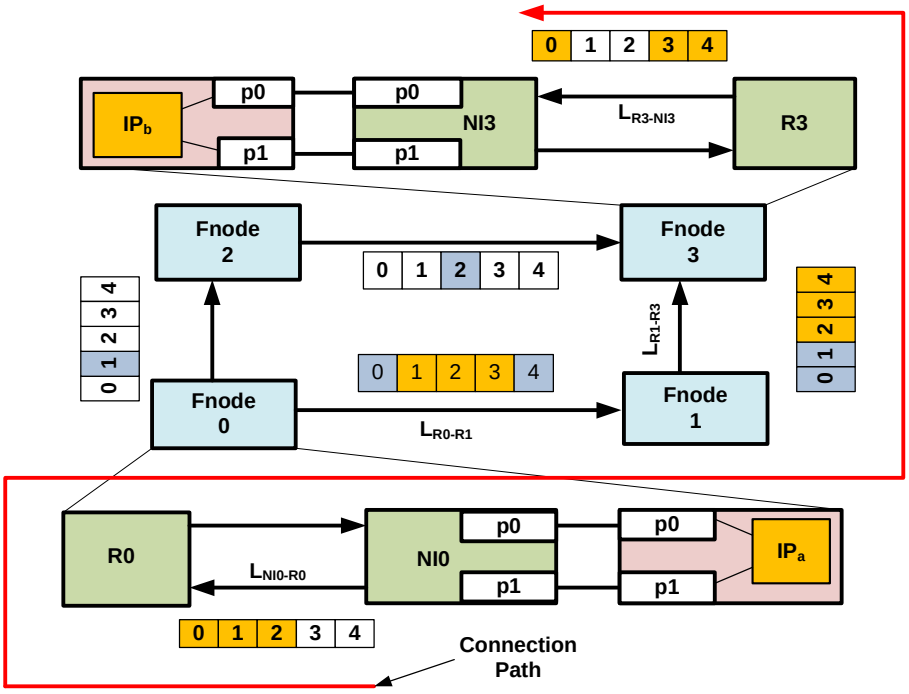


Figure 5.2: An Example Instance of Two IPs on FPGA Nodes and Connection Path in Between Them.

connections to transport data among the IPs. The nature of our application is streaming, i.e., it is throughput sensitive and does not have strict latency constraints.

Each IP is annotated by its area and ports. An application connection has a minimum throughput demand that can be converted into discretised time-slots. Additionally, each connection requires a path to exchange data in between the IPs that belong to it. The connection path from source IP to destination IP is defined as the nonempty sequence of links in between the source and destination Fnodes.

5.1.3 Required Objectives

The successful application binding on the target FPGA architecture ensures that for each of its IPs there exist at least one FPGA node that can fulfill the placement, mapping, and allocation requirements as explained below.

1. An IP is said to be *placed* on a FPGA node if its area constraint is met. IP area must be less than or equal to a single TCFR area. In other words, the available area of an Fnode TCFR is greater than or equal to the required area of IP.

2. An IP is said to be *mapped* to FPGA node, when the port, ingress connection, and egress connection constraints of IP are met. The Fnode NI, at the time of mapping decision of an IP, has: (a) at least the required number of ports, (b) enough bandwidth (without taking time-slots into account) at its ingress link to meet the aggregated bandwidth requirement of all the input connections of IP, and (c) enough bandwidth at its egress link to meet the aggregated bandwidth requirement of all the output connections of IP.

3. An IP is said to be *allocated*, when the connections of IP are allocated according to its QoS constraints. For instance, if IP_a and IP_b are two IPs that are placed on $Fnode_0$ and $Fnode_3$ respectively, see Figure 5.2. Then, for IP_a to IP_b connection that requires 3 time-slots as its throughput QoS constraint, and follows the path that consists of $L_{NI0\text{-}R0}$, $L_{R0\text{-}R1}$, $L_{R1\text{-}R3}$, $L_{R3\text{-}NI3}$ links, as shown in Figure 5.2. The allocation of the connection is successful, if and only if, all the links that are part of the path from IP_a to IP_b have at least 2 time-slots that can be reserved in a pipelined fashion.

In the next section, we explain our PUMA scheme that performs the binding of an application on the target FPGA architecture.

5.2 PUMA: (Road to) Unified Placement, Mapping, and Allocation

Figure 5.3 illustrates the working principle of our PUMA scheme. Initially, a database is created that reflects the availability of FPGA residual resources on both the logic and communication nodes, see Section 5.2.1. The database is updated during the binding process of an application. The PUMA flow performs a cluster-wise binding of an application (Section 5.2.2), where each *cluster* represents the inter-communication dependencies among a group of IPs, e.g., CL0 represents one such cluster in Figure 5.4C. Then, based on the area and communication demands of a cluster, the solution space of a cluster is obtained, see Section 5.2.3. The solution space is obtained to figure out

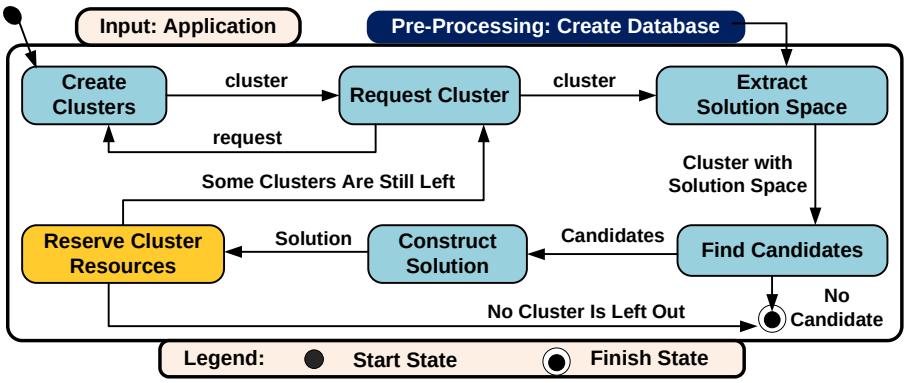


Figure 5.3: High Level Flow of our PUMA Scheme.

the possible candidate solutions, Section 5.2.4. The construction of the best possible solution comes next, which is evaluated in accordance with a design time specified cost matrix, see Section 5.2.5. The cost matrix takes into account the impact of binding of an application on the target FPGA architecture in terms of area fragmentation, allocation of communication resources (i.e., time-slots), and contention produced. In case of a successfully constructed solution, the cluster binding is performed by reserving the (constructed solution) resources on both the FPGA planes, Section 5.2.6. Our PUMA flow then, depending upon the remaining clusters, either requests the next cluster of current application, or starts the binding process for the next application.

Our PUMA scheme performs the binding of applications on a one-by-one basis. This means, when there are multi-applications present in a system, our PUMA scheme selects one application and tries to bind it on the target FPGA, and then goes for the next application in the queue. At run time when there are multiple applications running they do not interfere with each other and each application gets resources as per required to meet its QoS constraints[1]. This is due to: 1) PUMA scheme that makes sure that an application becomes candidate for execution only if its QoS constraints are fulfilled, and 2) the 3-tier reconfiguration model that makes use of the system manager and composable HWNOC to avoid interference among the running applications (see Section 6.2).

We now explain the different steps of our PUMA scheme, individually.

[1]Note that we do not take into account the reconfiguration time when we schedule and bind the application.

Algorithm 5.1: Calculation of Effective Throughput between two FPGA Nodes.

```
Input: All Fnodes
Output: Effective PathSlots in between any two Fnodes
for All Fnodes do
    Select an Fnode and name it as srcFnode;
    Select netFnodes of srcFnode;
    for All netFnodes do
        Select a netFnode;
        Find the path between srcFnode and netFnode;
        Select first path-link;
        Find out all the available / free time-slots on the first path-link;
        for (i = 0; i ≤ free time-slots; i++) do
            Set FreeSlot = i;
            while all path-links are not traversed do
                Select a path-link;
                Set NextSlot = FreeSlot % SlotTableSize;
                if NextSlot is not free in the selected path-link then
                    Exit the while loop because the selected FreeSlot can not be
                    allocated to the path-links in pipelined fashion;
                end
            end
            if all path-links are traversed then
                Push FreeSlot in the PathSlots vector;
            end
        end
        Map PathSlots with the path that is in between the srcFnode and netFnode;
    end
end
```

5.2.1 Preprocessing: Database Creation

In our PUMA scheme, initially a database for the available FPGA resources is constructed. To explain the database creation process, Figure 5.1B is used as the reference example. Initially, the available resources on all the FPGA nodes (Fnodes) are extracted. For each Fnode, this accounts for the available: (i) CLBs in its TCFR, (ii) ports in its NI, and (iii) bandwidth on the ingress and egress links of the NI. Afterwards, the paths among the Fnodes are constructed. Currently, PUMA constructs the possible shortest path after applying *XY* routing in between the two Fnodes. PUMA then evaluates the effective throughput (in terms of time-slots) on each path that is used to determine the allocation

of a cluster connections in Section 5.2.4. The effective throughput of a path is a term used to represent the number of time-slots that can be assigned in a pipelined fashion on the path. For instance, the effective throughput for connection path that exists in between $Fnode_0$ and $Fnode_3$ in Figure 5.2 equals to 3 time-slots.

Algorithm 5.1 shows the process to obtain effective number of time-slots that are available on the path between two FPGA nodes. Prior to explaining the process, we explain the data structures that are used as inputs, i.e., FPGANodes and PathSlots. FPGANodes represent all the FPGA nodes, where each FPGA node is a structure to store the details as follows: (i) TCFR id, area, and the record of IPs that reside on it, (ii) the number of NIs and their Ids, and for each NI the number of free ports, bandwidth at its ingress and egress links, (iii) the paths to remaining FPGA nodes. The PathSlots is a vector to store times-slots, where each time-slot is an integer number.

The first for loop iterates over all the FPGA nodes of the target FPGA, line 1 - 24 of Algorithm 5.1. In each iteration a srcFnode (i.e., the first / start Fnode of the path) is selected and its network Fnodes (i.e., all the Fnodes except the srcFnode) are obtained, line 2 - 3 of Algorithm 5.1. To obtain the effective number of time-slots on the path in between srcFnode and netFnode, the following procedure is used.

First, the path from the srcFnode to netFnode is obtained, line 6 of Algorithm 5.1. Afterwards, the time-slots that are not allocated to any connection are obtained on the first link of the path that exist in between the srcFnode and netFnode, line 8 of Algorithm 5.1. Then, each of the free time-slot on the first link is selected to find out its availability in the following links in a pipelined fashion, line 9 - 21 of Algorithm 5.1. In case of success the selected time-slot is pushed into the effective PathSlot database against that specific path.

After creating the database, the clusters of the input application are created, as explained below.

5.2.2 Traversing the Application and Creating Clusters

The applications are becoming increasingly complex. Therefore, picking an application and an FPGA as a whole, and try every possible binding possibility would be highly time consuming. Alternatively, there are different schemes [112], which can allow cluster or level-based traversal of an application task graph. Our PUMA scheme, performs the binding of an application on a cluster-wise basis. As we shall explain in the following discussion, while

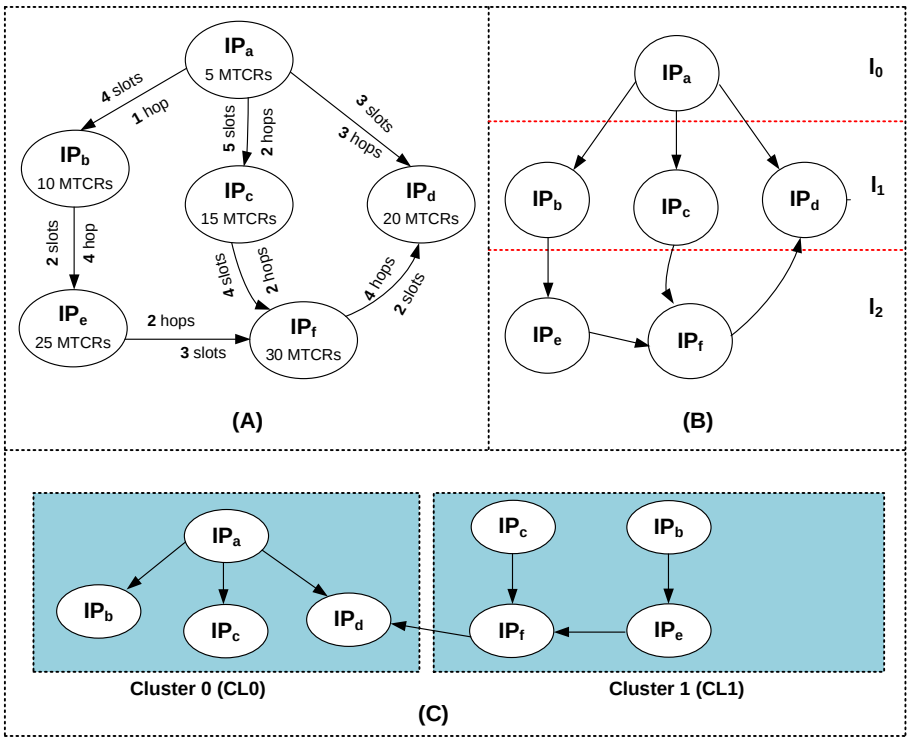


Figure 5.4: (A) Application Task Graph with Communication Requirements in Terms of Time-Slots, (B) Communication Levels, and (C) Clusters.

binding a cluster, its inter-cluster dependencies are taken care of. The clusters are created by exploiting inter-IP communication dependencies, because inputs of an IP can be dependent on the output of some other IP(s). To explain the cluster creation process, we use an application in Figure 5.4A, which exhibits all types of inter-IP communication, i.e., single IP to single IP, single IP to many IPs, and many IPs to single IP communication. For example in Figure 5.4A, IP_a communication with IP_b, IP_c, IP_d stands for the single IP to many IPs situation. The numbers in Figure 5.4A indicate inter-IP communication demands in terms of number of time-slots. PUMA creates clusters based on the following rules:

Rule 0: Pick a start IP, i.e., (IP_a).

Rule 1: Identify the communication levels (hereinafter termed as levels) among application IPs. For this purpose, we place IP_a in the zeroth level (l_0), see Figure 5.4B. As the inputs of IP_b, IP_c, and IP_d, are driven by IP_a outputs,

Algorithm 5.2: Cluster Creation Process.

```
Input: Application IPs
Output: Clusters of Application
while IPList is not empty do
    if First IP of Application then
        Mark First IP as srcIP;
        Get ConnectedIPs of the srcIP;
        Map srcIP and ConnectedIPs and Push in CurCluster;
    end
    else
        Clear ConnectedIPs;
        for all NextLevelIPs do
            Select an IP and Mark it as srcIP;
            Get ConnectedIPs of the srcIP;
            Map srcIP and ConnectedIPs and Push in CurCluster;
        end
    end
    Clear NextLevelIPs;
    Push CurCluster in AppClusters;
    Push ConnectedIPs in NextLevelIPs;
    Update IPList by Removing NextLevelIPs;
end
```

these are placed in the next level, i.e., l_1. This process is repeated until all IPs are placed in a level hierarchical way.

Rule 2: A cluster indicates communication among the IPs that belong to adjacent levels. For example, if there are three levels, say l_i, l_{i+1}, and l_{i+2}. Then, the first cluster (i.e., cluster i) includes all IPs in level l_i that communicate to IPs in level l_{i+1}. Similarly, the second cluster, (i.e, cluster i+1) includes IPs in level l_{i+1} that communicate to IPs in level l_{i+2}. For instance, Figure 5.4C shows a cluster (CL_0) that indicates communication between l_1 and l_2 IPs.

Rule 3: The inputs of an IP can be driven by multiple IPs, which exist in different hierarchies. For instance, IP_d inputs are driven by IP_a and IP_f, which are placed in l_0 and l_2 respectively. The placement of IP_d is performed as part of CL_0 binding, which is the first application cluster. However, at the time of placing IP_d care must be taken that the resultant placement of IP_d should not block the allocation of a future IP_f-IP_d connection. This inter-dependency is detailed in Section 5.2.5.

Algorithm 5.2 shows the process to create application clusters. In Algo-

rithm 5.2 we use different data structures where: (i) *srcIP* represents the reference IP to create a (partial) cluster, (ii) *IPList* is the vector of application IPs, (iii) *ConnectedPs* is the vector of IPs that are connected to a srcIP and are placed in the next communication level to a srcIP, (iv) *NextLevelIPs* is the vector to store the ConnectedIPs of all the srcIPs that are part of a specific cluster, and (v) *CurCluster* is the vector to store the maps of all the srcIPs and their associated ConnectedPs in the two adjacent hierarchies, (vi) *AppClusters* is the vector to store all the clusters of an application.

Initially, the first IP of an application is selected as the srcIP and the IPs that are connected (i.e., ConnectedPs) to it are obtained, line 2 - 4 of Algorithm 5.2. As a next step, the srcIP and the ConnectedIPs are mapped and marked as the first cluster. The CurCluster and ConnectedIPs are then pushed in AppClusters and NextLevelIPs respectively and IPList is updated by removing the NextLevelIPs, line 16 - 18 of Algorithm 5.2. Next time the same process repeats for the IPs that are placed in NextLevelIPs vector, but with the difference that now multiple srcIPs can be the part of a cluster, line 9 - 13 of Algorithm 5.2.

After creating a cluster, PUMA extracts the solution space for it as explained in Section 5.2.3.

5.2.3 Solution Space Extraction

After a cluster is created, its *solution space* is extracted. *The solution space of a cluster is the set of FPGA nodes that can fulfill the combined logic and communication demands of the cluster*. A cluster can have more than one source IP that communicates with IPs in the next level, e.g., CL_1 in Figure 5.4C is one such example.

Algorithm 5.3 shows the process to extract the solution space for a cluster. Initially, for each source IP (srcIP) the ConnectedIPs and Connections are obtained, line 2 - 4 of Algorithm 5.3. Next, if a srcIP is already placed on FPGA (i.e., on any of the Fnodes) then its Fnode is obtained and the solution space is constructed by calling CreateSolSpace function, line 15 - 18 of Algorithm 5.3. The Fnode of a srcIP determines the starting point to create the solution space. However, when a srcIP is not present on the FPGA logic plane, then the search for a suitable Fnode that can place the srcIP, is made by calling the PlaceSrcIP function, line 6 of Algorithm 5.3. In case the PlaceSrcIP function finds a suitable Fnode to place the srcIP then the CreateSolSpace function is called, line 7 - 10 of Algorithm 5.3. Otherwise, a failed binding instance for the input

Algorithm 5.3: Finding the Solution Space for a Cluster.

```
Input: An application cluster
Output: Solution space for an application cluster
1  for All srcIPs do
2      Pick a srcIP;
3      Get ConnectedIPs of srcIP;
4      Get Connections of srcIP;
5      if srcIP is not already placed on FPGA then
6          Call PlaceSrcIP Function;
7          if PlaceSrcIP Succeeded then
8              Get FPGA Node of srcIP;
9              Call CreateSolutionSpace Function;
10         end
11         else
12             Announce Failed Binding and Exit;
13         end
14     end
15     else
16         Get FPGA Node of srcIP;
17         Call CreateSolutionSpace Function;
18     end
19 end
```

application is announced back, line 11 - 13 of Algorithm 5.3. In the following discussion, we explain the PlaceSrcIP and CreateSolSpace functions.

Algorithm 5.4 shows the process to place a source IP on a suitable FPGA node. Initially, the current FPGA node is obtained, which can be the first Fnode, i.e., $Fnode_0$, or the FPGA node where the last IP of the previous cluster was bound. The srcIP resources are obtained afterwards, line 3 of Algorithm 5.4. These include the area and ports of a srcIP, and the required time-slots of connections that are input and output to / from a srcIP. Similarly, the available resources of the selected FPGA node are obtained, line 4 of Algorithm 5.4. These include the available area at its TCFR, available ports at its NI, and available time-slots at its ingress and egress links. The resources of srcIP and FPGA node are compared afterwards. If the selected FPGA node fails to meet the srcIP requirements then the next FPGA node is chosen as the current FPGA node, line 6 - 8 of Algorithm 5.4. The process repeats until the srcIP is placed or all the FPGA nodes are traversed.

After the FPGA node of srcIP is obtained the CreateSolSpace function is called. To extract the solution space for the selected cluster, the source Fn-

Algorithm 5.4: Determining the Placement of Source IP of a Cluster.

Input: srcIP
Output: FPGA Node on which srcIP is Placed

```
1  while (srcIP is not placed) OR (All FPGA nodes are not traversed) do
2      Pick current FPGA node;
3      Get srcIP resources;
4      Get FPGA node resources;
5      Compare FPGA node and srcIP resources;
6      if current FPGA node fails to place srcIP then
7          Mark next FPGA node as the current FPGA node;
8      end
9      else
10         Mark srcIP as placed;
11         Update current FPGA node resources;
12     end
13 end
```

ode (where the source IP is placed) is obtained. Then, the most strict QoS constraints of the cluster are obtained, i.e., area, throughput, and latency requirements. This means the cluster task graph is traversed to find out: (a) its IP with maximum area requirement, (b) connection with maximum throughput, and (c) connection with minimum latency requirements. The information is then input to Algorithm 5.5.

Algorithm 5.5 then finds out the Fnodes that are in the network of source Fnodes[2], Algorithm 5.5 (line 1). Then, the residual area of a network Fnode is obtained, Algorithm 5.5 (line 3). Afterwards, the path slots and hop delay are also obtained between the source Fnode and a network Fnode, Algorithm 5.5 (line 4-5). In Algorithm 5.5, `0.006` indicates a per hop delay in μs. The obtained information is then evaluated against the inputs of the Algorithm 5.5, i.e., the maximum area demand, the maximum required throughput, and the minimum permitted latency, Algorithm 5.5 (line 7). In case of success, the selected network Fnode is added in the solution space of the source Fnode.

Example

For example, in our application of Figure 5.4A, IP_a is the source IP of the first cluster, i.e., CL_0 . PUMA first looks for the feasible Fnode that can fulfill area,

[2]The network Fnodes of a source Fnode accounts for all the Fnodes of FPGA minus the source Fnode.

Algorithm 5.5: Determining the Solution Space for an IP of a Cluster.

Input: minlatency, maxslots, maxarea, srcFnode
Output: SolSpace for srcFnode

```
1  dstFnodes = getDstFnodes(srcFnode);
2  while ALL dstFnodes are not traversed do
3      Pick a dstFnode;
4      Get dstFnodearea;
5      Get Pathslots between the srcFnode and destFnode;
6      Get hop count between the srcFnode and destFnode;
7      Calculate Hopdelay;
8      if (Hopdelay < minlatency) and (Pathslots > maxslots) and (dstFnodearea <
       maxarea) then
9          Push dstFnode in the solution space of srcFnode;
10     end
11 end
```

ports, and communication demands for IP_a. Once such an Fnode is obtained, the process to extract the solution space is started. If IP_a of CL0 (shown in Figure 5.5A) is placed in $Fnode_0$, then the dotted rectangle in Figure 5.5B stands for the solution space of CL0.

According to the most relaxed conditions, IP_b has the least area requirements, i.e., 5 MTCRs , see Figure 5.4A. Connection IP_a-IP_d requires the least number of time-slots, i.e., 3 time-slots. Connection IP_a-IP_c has the maximum permissible latency, i.e., 3 hops. On the other hand Figure 5.5B shows that no IP is placed on any of the FPGA nodes, and therefore the available area of each Fnode stays at a maximum of 32 MTCRs. This means, all the Fnodes meet the area requirement. Next comes in the throughput requirements, which include all the Fnodes except $Fnode_2$ and $Fnode_5$. The reason is that one of the link to both the Fnodes contain 1 time-slot which is less than the required number of 3 time-slots. As the next step, $Fnode_8$ is taken away from the earlier selected set of Fnodes. The reason is that the hop count from $Fnode_0$ where is placed to $Fnode_8$ is more than 3 hops, which is the maximum permissible latency constraint. As a result, the solution space of CL0 is the one which is marked with dotted rectangle in Figure 5.5B.

5.2.4 Candidate Solution Finding

The extracted solution space is now used to obtain the *candidate solutions* to bind the input cluster, as shown in Figure 5.6. PUMA generates a number

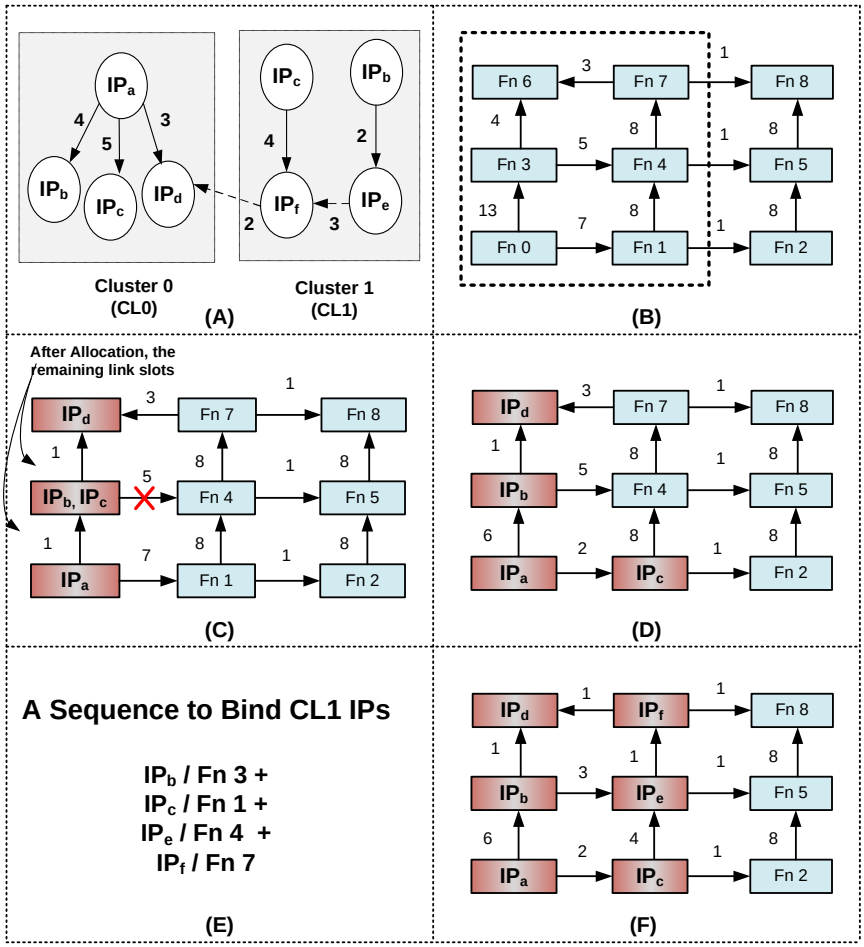


Figure 5.5: Example that Shows the Binding of Clusters on our FPGA. (A) CL0 and CL1 IPs with Time-Slot Requirements, (B) FPGA Architecture with Available Time-Slots. Rectangle Indicates CL0 Solution Space, (C) Failed Binding for CL0 IPs, (D) Successful Binding of CL0 IPs, (E) A sequence for CL1, (F) Binding of CL1 IPs in Accordance with the Sequence.

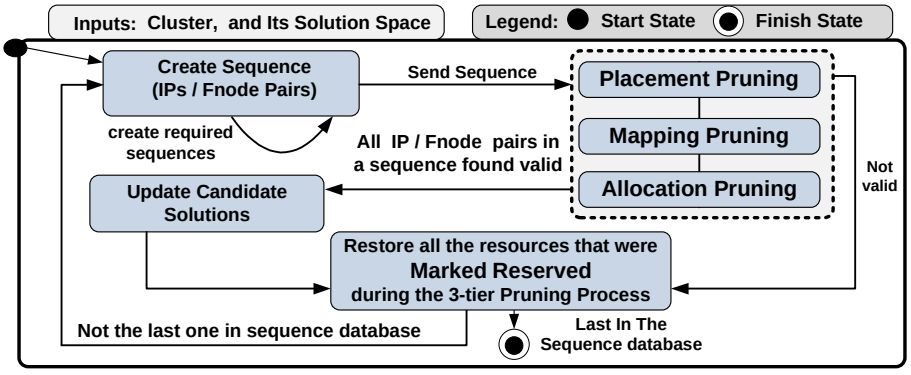


Figure 5.6: Finding the Candidates Solutions.

of sequences to find out the solution candidates of a cluster, Figure 5.6. *A sequence consists of all the cluster IPs that are paired with some / all Fnodes of the solution space*. Figure 5.5E shows one such generated sequence, where each term represents an IP /Fnode pair.

Algorithm 5.6 shows the process to find the candidate solutions of a cluster. Initially, a total number of sequences are created after obtaining the IPs and Fnodes of the cluster, line 1 - 3 of Algorithm 5.6. Then for each selected sequence, PUMA examines its validity by using the 3-tier pruning process, line 5 - 12 of Algorithm 5.6. First place and map pruning is performed, which if successful then triggers the allocation pruning process for the selected sequence.

The 3-tier pruning process is performed for all the IP / Fnode pairs, and compares the residual resources of an Fnode against the required resources of the paired IP(s). In case of success of allocation pruning the selected sequence is pushed in the database of candidate solutions, line 10 of Algorithm 5.6. However, in case of failure the selected sequence is marked as failure and the next appropriate sequence is selected, line 16 - 18 of Algorithm 5.6. Once, the validity / failure of a sequence is concluded. The resources that were marked reserved, during the sequence evaluation process, are restored back, line 19 - 20 of Algorithm 5.6. This way the remaining sequences are examined to create a set of possible candidate solutions. We next explain the 3-tier pruning process that decides the validity of a sequence.

Algorithm 5.6: The Process to Find Candidate Solutions.

```
Input: Cluster, Cluster Solution Space
Output: Candidate Solutions for the Cluster
Get IPs in the cluster;
Get FPGA Nodes in the solution space;
Create Total sequences;
while Last sequence is not reached do
    Select a sequence;
    Perform Place and Map Prune for the sequence;
    if Place and Map Prune is succeeded then
        Perform Allocation Prune for the sequence;
        if Allocation Prune is succeeded then
            Add sequence in the CandSolutions database;
        end
        else
            Mark current sequence as failed;
        end
    end
    else
        Mark current sequence as failed;
    end
    Restore the resources on FPGA Nodes and path-links to values that were at the
    start of sequence 3-tier pruning process;
    Jump to the next sequence;
end
```

Placement Pruning

A successful placement pruning requires that objective 1 in Section 5.1.3 is satisfied for all the IP / Fnode pairs of the input sequence. In other words the placement pruning ensures that an Fnode TCFR in the selected sequence meets the area requirements of all the IPs that are associated with it.

For example, in Figure 5.5E the generated sequence creates the pair of IP_f and $Fnode_7$. In this case $Fnode_7$ passes placement pruning for IP_f, if and only if, the residual area of TCFR (of $Fnode_7$) is less than the required area of IP_f. In this way, the placement pruning process is conducted for the remaining IP / Fnode pairs of the sequence. After a successful placement pruning, the mapping of a sequence is evaluated as explained below.

Mapping Pruning

A successful mapping pruning process requires that the objective 2 in Section 5.1.3 is satisfied for all the IP / Fnode pairs of the input sequence. In other words a successful IP to Fnode mapping ensures that the respective NI (in the Fnode) has enough ports to connect the IP ports. In addition, the NI should have enough bandwidth on its ingress and egress links to fulfill the throughput demands of incoming and outgoing connections of the mapped IP, respectively.

For example, for the sequence in Figure 5.5E, the IP_f / $Fnode_7$ passes mapping pruning, if and only if, $Fnode_7$ associated NI meets the following conditions. (1) The NI has at least 3 ports that are required to connect the IP_f ports. (2) The ingress link of the NI has at least 7 time-slots that are required by the input connections of IP_f, i.e., connection IP_c-IP_f and connection IP_e-IP_f, as shown in Figure 5.5A. (3) The egress link of the NI should have a minimum of 2 time-slots that are required by the output connection of IP_f, i.e., connection IP_f-IP_d, as shown in Figure 5.5A.

Once the mapping pruning process is passed for an IP / Fnode pair of the input sequence, the Fnode resources are updated and the mapping pruning process is repeated for the next IP / Fnode pair. Note that PUMA, during the mapping pruning process, takes into accounts the sum of time-slots that are available on the NI ingress and egress links, but that the positioning of the time-slots is not taken into account during the mapping pruning process. However, the actual positioning of the time-slots on the links is evaluated during the allocation pruning process, which is explained below.

Allocation Pruning

During the allocation pruning, it is ensured that the paths between the sequence Fnodes have sufficient time-slots to *allocate the connections that exist among the IPs in the sequence*. In other words, a (cluster) sequence passes allocation pruning, if and only if, objective 3 in Section 5.1.3 is satisfied for the cluster connections.

We explain the allocation pruning by using our example application in Figure 5.4. We assume that at the time of CL_1 allocation pruning, CL_0 IPs IP_a, IP_b IP_c and IP_d are bound to $Fnode_0$, $Fnode_3$, $Fnode_1$, and $Fnode_6$, respectively (Figure 5.5D). Now, if the generated sequence in Figure 5.5E binds CL_1 cluster IPs, i.e., IP_e and IP_f to $Fnode_4$ and $Fnode_7$, respectively. Then, the allocation pruning ensures that this sequence is valid, if and only if, all its four

Algorithm 5.7: Allocation Pruning Process.

Input: Cluster Connections
Output: Allocation Prune Result

```
1  while (Allocation Prune is not Failed) AND (All Connections are not traversed) do
2      Select a connection;
3      Get source IP (srcIP) and its FPGA node (srcFnode);
4      Get destination IP (dstIP) and its FPGA node (dstFnode);
5      Get time-slots and permissible latency of the connection;
6      Get xyPath between srcFnode and dstFnode;
7      Get time-slots (i.e., effective throughput) and hop-delay of the xyPath;
8      if (xyPath time-slots > connection time-slots) AND (xyPath hop-dalay <
       connection latency) then
9          Subtract connection time-slots from the time-slots of links that are part of
           xyPath;
10     end
11     else
12         Allocation Prune is Failed;
13     end
14 end
15 if (Allocation Prune is not Failed) AND (All Connections are traversed) then
16     Allocation Prune is succeeded;
17 end
```

connections, i.e., IP_b-IP_e, IP_c-IP_f, IP_e-IP_f, and IP_f-IP_d are allocated according to their QoS constraints[3].

This means: (i) $Fnode_3$-$Fnode_4$ path has at least 2 effective time-slots[4] to allocate the IP_b-IP_e connection, (ii) the $Fnode_1$-$Fnode_7$ path has at least 4 effective time-slots to allocate the IP_c-IP_f connection, (iii) the $Fnode_4$-$Fnode_7$ path has at least 3 effective time-slots to allocate the IP_e-IP_f connection, and (iv) the $Fnode_7$-$Fnode_6$ path has at least 2 effective time-slots to allocate the IP_f-IP_d connection.

Algorithm 5.7 explains the process of allocation pruning, which receives all the connections of the cluster as input. For each selected connection, the source IP and source Fnode, and destination IP and destination Fnode are obtained, line 3 - 4 of Algorithm 5.7. Then, the Quality-of-Service (QoS) constraints of each connection, which are provided in terms of minimum required time-

[3]For the convenience of the reader, in our example we are using time-slots / throughput as the only QoS constraint and latency is not included in the example to keep things simpler.

[4]As stated earlier in Section 5.2.1, we term effective throughput or effective time-slots as the number of time-slots that can be assigned to path-links in a pipelined fashion.

slots and maximum permissible latency demands, are obtained. It is followed by obtaining the *xy path* between the source and destination Fnode, line 6 - 7 of Algorithm 5.7. As a next step, the available time-slots on the xy path, and hop delay of the xy path are obtained. The hop delay is calculated by multiplying the number of hops in xy path and delay per hop. Afterwards, the comparison between the connection resources and xy path resources is performed, line 8 of Algorithm 5.7. In case of a successful comparison, the resources of appropriate links are updated, line 9 of Algorithm 5.7. However, in case of failure the 3-tier process announces its failure for the input sequence, line 11 - 13 of Algorithm 5.7.

Our PUMA scheme, after a successful 3-tier pruning, marks the current sequence as a possible candidate solution. Then, depending upon the sequence generation logic, either apply the 3-tier pruning on the next generated sequence or start the construction of the best possible solution as explained below.

5.2.5 Solution Construction

After obtaining the candidate solutions where each solution binds the cluster with guarantees on placement, mapping, and allocation, PUMA starts constructing the best possible solution, for which it takes into account two important factors as discussed below.

Inter-Cluster Dependencies

First, in the constructed solution an IP (now associated with an Fnode), which has dependencies with IPs in the clusters that are still to be bound, should not be blocked at the router of its Fnode.

For example, Figure 5.4C shows that CL_0 IPs IP_b and IP_c have dependencies with yet to be bound CL_1 cluster. Now, if IP_b and IP_c are paired to the same $Fnode_3$, as shown in Figure 5.5C, then from Figure 5.5C we can conclude that the throughput requirements (5 time-slots) of connection IP_a-IP_b and throughput requirements (4 time-slots) of connection IP_a-IP_c are fulfilled. Meanwhile, the produced logic fragmentation is at minimum. So, from the current cluster point of view, IP_b / $Fnode_3$ and IP_c / $Fnode_3$ pairs are valid.

However, in this case, IP_b or IP_c gets blocked at $Fnode_3$ router, as shown in Figure 5.5C. Because the maximum sum of time-slots that are available on one of the outgoing links of $Fnode_3$ router is 5. This is less than the required 6 time-slots by IP_b and IP_c output connections. This in turn can block either IP_b or IP_c

Algorithm 5.8: Construction of the Best Solution.

Input: Candidate solutions of the cluster
Output: Best solution of the cluster

```
1  Set BestCost equals to a very high value;
2  while All candidates solutions are not traversed do
3      Select a solution and name it as TempSolution;
4      Evaluate AreaCost of TempoSolution;
5      Evaluate CommunicationCost of TempoSolution;
6      Evaluate CongestionCost of TempoSolution;
7      Add AreaCost, CommunicationCost, and CongestionCost and name it as
       TempCost;
8      if TempCost < the BestCost then
9          Set TempCost as BestCost;
10         Set TempSolution as BestSolution;
11     end
12 end
```

from sending data into the network. Figure 5.5D shows the alternative solution. Here, IP_b / $Fnode_3$ and IP_c / $Fnode_1$ pairs not only fulfill CL0 allocation, but the result binding also does not block the connection IP_b and IP_c from sending data into the network.

The Optimisation Criteria

Next, the optimisation criteria is used that decides the overall cost of the constructed solution. It consists of three factors as described below:

$$cost = \lambda * LogicI + \beta * CommI + \theta * ContI \tag{5.1}$$

Here LogicI indicates the fragmentation produced in Fnodes by the selected candidate solution. For example, if a candidate solution IPs are placed in two TCFRs (e.g., TCFR0 and TCFR1), then the remaining area of both the TCFRs, after placing indicates the fragmented area. It is designed such that the best solution opts for the candidate solution with minimum fragmentation in the least number of Fnodes. This not only reduces the number of fragmented Fnodes, but also increases the (untouched) Fnodes for the remaining clusters / applications.

The second part of Equation 5.1 accounts for the network allocation, which is caused by the selected candidate solution. It is designed such that the best

Algorithm 5.9: Calculating Area Cost Matrix to Determine the Best Solution.

```
   Input: Candidate solution of the cluster
   Output: Area cost of a solution of the cluster
 1 Initialize TCFRVector;
 2 Initialize TotalTCFRs to Zero;
 3 Initialize FragmentedArea to Zero;
 4 while All IP / Fnode pairs are not traversed do
 5 |   Select an IP / Fnode pair;
 6 |   Get IP area;
 7 |   Get TCFR of the respective Fnode;
 8 |   Subtract IP area from TCFR area;
 9 |   Update TCFR area after subtracting IP area;
10 end
11 while All IP / Fnode pairs are not traversed do
12 |   Get TCFR of the respective Fnode and name it as TempTCFR;
13 |   if TempTCFR is not present in TCFRVector then
14 |   |   Get FreeArea of TempTCFR;
15 |   |   Add FreeArea in the FragmentedArea;
16 |   |   Push TempTCFR in TCFRVector;
17 |   |   Increase TotalTCFRs by 1;
18 |   end
19 end
20 AreaCost = FragmentedArea / TotalTCFRs;
21 PercentAreaCost = (AreaCost / DesignTimeTCFRArea) * 100;
```

solution opts for the candidate solution with minimum allocated resources over the network. The third part of Equation 5.1 accounts for the mean of contention over the network links. It is calculated by averaging the residual slots across the links of the selected paths in the candidate solution.

Algorithm 5.8 shows the process to obtain the best solution, which iterates over all the candidate solutions. The best solution is the one with minimum cost. For each candidate solution its area, communication, and congestion costs are evaluated, line 3 - 6 of Algorithm 5.8. Next the total cost of solution is obtained by adding all the three costs multiplied by appropriate constant values, line 7 of Algorithm 5.8. If the solution cost is less than the current best cost then the current solution is set as the best solution, line 8 - 11 of Algorithm 5.8.

Algorithm 5.9 shows the process to calculate the area cost, which is obtained by looping over all the IP / Fnode pairs of the selected candidate solution. For each IP / Fnode pair, the area of IP is subtracted from the area of Fnode TCFR, line 4 - 10 of Algorithm 5.9. The TCFR area is then updated afterwards. The

Algorithm 5.10: Resource Reservation Process for the Best Solution.

Input: Solution, CurClstrCon, PrvClstCon
Output: For all Cluster IPs: place, map , allocate

```
1  Get all IP / Fnode pairs of Solution;
2  while All IP Fnode pairs are not traversed do
3      Select an IP / Fnode pair;
4      Get TCFR of IP / Fnode pair;
5      Get NI of IP / Fnode pair;
6      Get Ingress link of IP / Fnode pair;
7      Get Egress link of IP / Fnode pair;
8      Get Connections of IP and Insert it in ClusterConArray;
9      Place IP in TCFR;
10     Map all ports of IP to NI port;
11     Map all input connection of IP to Ingress link;
12     Map all output connection of IP to Egress link;
13 end
14 while All Connections in ClusterConArray are not allocated do
15     Pick a Connection;
16     Get source NI of connection;
17     Get destination NI of connection;
18     Find path between source NI and destination NI;
19     Allocate Connection on the path;
20 end
```

second loop that iterates over IP / Fnode pairs is used to obtain the total number of TCFRs, line 11 - 19 of Algorithm 5.9. The area cost is then calculated as the aggregated fragmentation that is produced by the candidate solution in all the TCFRs, line 20 of Algorithm 5.9.

In Equation 5.1 λ, β and θ are constant that vary from 0 to 1.0, and indicate the importance of the decision with respect to logic and / or communication resource optimisation. It is because the $S_i \in Sl$ can belong to an application with variable area and communication demands with respect to the available area and communication resources of FPGA. Therefore, the optimisation criterion are not the same every time. For example, an application with low area and high communication demands can require a solution, which comprises shorter paths and (possibly) uniform contention. On the contrary, an application with high area and low communication demands would prefer a solution with low logic fragmentation. Note that values of λ, β and θ are selected at design time. However, the selection of λ, β and θ constant values is part of future work.

After constructing the best solution, resources are reserved for the solution as

explained below.

5.2.6 Cluster Resource Reservation

Once the best solution is obtained, the resource reservation is performed. Initially, all the IP / Fnode pairs of the solution are extracted, Algorithm 5.10 (line 1). Then, necessary information for each IP / Fnode pair is obtained that include: (i) the IP and its ports, (ii) TCFR and NI of the Fnode, (iii) ingress and egress NI links, and (iv) the connections that the IP uses to communicate with IPs in the existing and already bound clusters, Algorithm 5.10 (line 3-8). After placing an IP in the TCFR, the IP is mapped to the respective Fnode NI. This is achieved by connecting the IP ports to the NI ports. Then reserving time-slots of input and output connections of the IP on the NI links, Algorithm 5.10 (line 9-12).

Allocation comes next to placement and mapping. The connections, which need to be allocated, represent the communication of cluster IPs in the existing and previously bound clusters. The allocation process is executed by first extracting the required paths in between the Fnodes. Then, reserving the resources across the paths, Algorithm 5.10 (line 14-20). Meanwhile, the database is updated to reflect the new residual resources of the respective Fnodes, and new slot tables of the paths that were used during the allocation of the best solution.

5.3 Limitations

Following are the limitations with the proposed PUMA scheme:

1. *Input Application*: The nature of the application is streaming. The task graph of our input application is connected. Moreover, during the cluster creation process, our PUMA assume that there is always one IP that serves as the starting point to traverse through the application task graph.

2. *Placement*: The placement of an IP is restricted to a single TCFR. This means, PUMA does not accounts the span of an IP in multiple TCFRs. The placement decision is based on the required area value of an IP and the residual area of TCFR, i.e., our PUMA scheme does not account the xy dimensions of an IP, while deciding the placement of an IP. In other words the residual area of a TCFR is the metric to compute the fragmentation in our proposed PUMA scheme.

3. *Mapping*: PUMA scheme assumes a single NI per TCFR, where each NI can provide a maximum of 2 GB/s throughput. This means, an IP that has connections with more 2 GB/s can not be bound by using the current implementation of our PUMA scheme.

4. *Allocation*: Our PUMA scheme only takes into account the hop count as the metrics for latency calculations. We also assume that NI buffers are large enough to not lose bandwidth due to flow control. The measures, e.g., FIFO depths at network interfaces, and slot table sizes are not taken into account while verifying the maximum latency constraints for a yet to be allocated connection. This means the latency calculations are rather optimistic. As our input applications are not latency critical, an optimistically allowed latency value for a connection, might not be problematic in fulfilling its Quality-of-Service constraints. However, with latency critical applications, PUMA needs to account for all the three metrics (i.e., hop count, FIFO depths at NIs, and slot table size) while allocating a connection.

5.4 Results And Analysis

We implemented the PUMA scheme in SystemC using the design flow of [42]. We evaluated the performance, and scalability of our PUMA scheme, as described in the following sections.

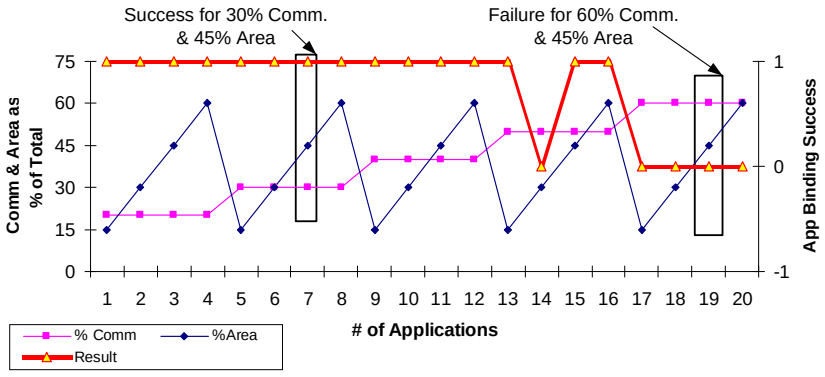


Figure 5.7: PUMA Success Rate with Variable Communication and Area Demands.

5.4.1 Performance: Success Rate

PUMA success rate, i.e., *binding of application with QoS guarantees*, was evaluated by generating a number of synthetic applications generated using SDF3 [136]. We discuss our results in terms of success-rate to evaluate that how good the PUMA scheme is in providing a binding solution over a range of applications that can vary in terms of area communication requirements. In Figure 5.7, Figure 5.8, and Figure 5.9, we indicate a successful binding instance with 1, and a failed binding instance with 0.

Figure 5.7 illustrates PUMA binding results for 20 applications, each of which comprises 15 IP cores, but contained different area and communication requirements against the target FPGA architecture. Meanwhile, for a selected application all the connections were kept at the same throughput demand, i.e., there was no deviation as far as throughput demands of an application connections are considered.

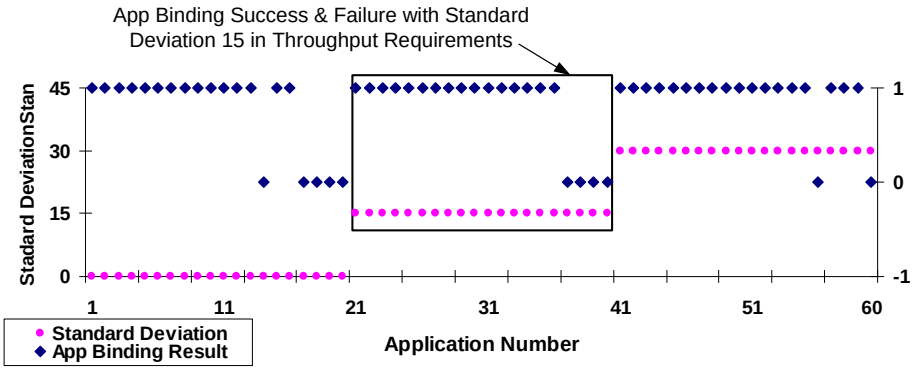


Figure 5.8: Binding Results of Applications with Variable Standard Deviations w.r.t. the Communication Throughput Demands.

Next for the same set of 20 applications of Figure 5.7, we introduced randomness in the throughput demands by changing the standard deviation by 0, 15, and 30, as shown in Figure 5.8.

Once the throughput requirements reach 60% of the FPGA resources, the failed binding instances are prominent with 0 SD as compared to 15 SD or 30 SD. This is because the throughput demand of *each application connection* stands at a higher 900 MB/s, when the communication requirement of an application is 60% of the FPGA communication resources and connection values deviates with each other by 0 value. In this situation, any two connections that share a link can induce a failed binding instance. Because the combined

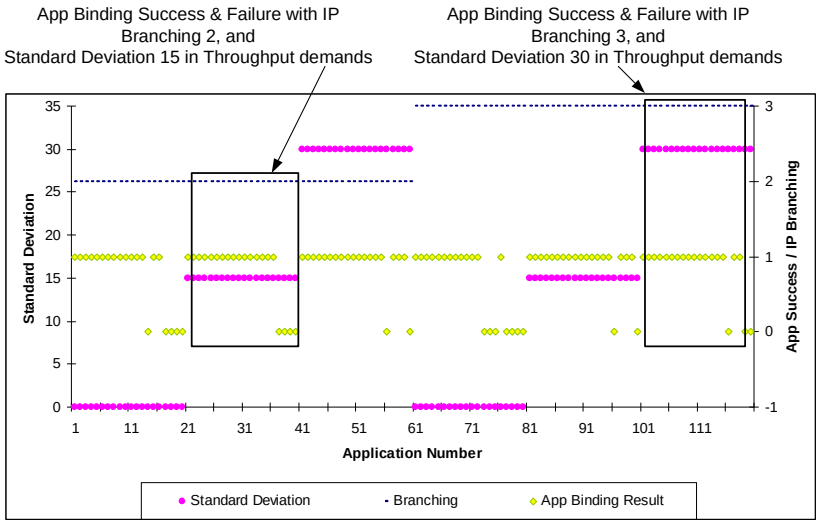


Figure 5.9: Impact on the Binding Success of Applications by Increasing Inter-IP Dependencies.

(raw) throughput demand of any two connections (with standard deviation 0 and 60% communication) is approximately 1800 MB/s, which approaches the maximum throughput capacity of a link in our HWNoC architecture. In other words, PUMA fails to fit two connections (with standard deviation 0 and 60% communication) on the same link. However, PUMA success-rate (for the same 60% communication) increases as the standard deviation between the connections reaches to 15 or 30, as shown in Figure 5.8.

Next for the same set of 60 applications of Figure 5.8, we changed IP *branching*, i.e., average number of connections per IP. This means an IP can now communicate with more IPs. Therefore, interdependencies among the application IPs increases. Figure 5.9 shows application success rate with an average branching of 2 and 3. Interestingly, the results for each branch hold a similar pattern, mainly due to the assurance of enough resources on the NIs of Fnodes at the time of IP mapping. This enables the future connections for that IP to get allocated.

Next we averaged the success rate for a particular area / communication combination that exists in 120 applications of Figure 5.9. Figure 5.10 reflects success rate with high area (i.e., 50% and 70% area requirement of the available FPGA area) requirement and variable communication requirements. However, Figure 5.11 reflects success rate with low area (i.e., less than 50% area requirement of the available FPGA area) requirement and variable commu-

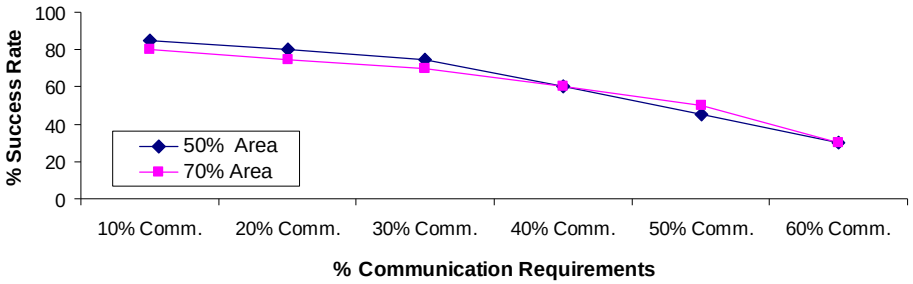


Figure 5.10: PUMA Success Rate with High Area (i.e., 50% and 70% Area of FPGA) and Variable Communication Requirements.

nication demands. The figures Figure 5.10 and Figure 5.11 indicate approximately 50% success rate when the communication requirements are 30% of the available communication resources of the target FPGA. However, as the communication requirement reaches the 60% mark, the success rate decreases to a low 20% value.

In Figure 5.10, this is mainly due to the increased randomness in the demands of area and communication requirements, which makes it difficult to achieve successful binding. Though PUMA takes into account the inter-cluster dependencies, i.e, while binding a cluster PUMA makes sure that IPs that have connections with IPs in yet to be bound clusters do not get blocked at their respective network interfaces. However, the calculations take into account the availability of required number of time-slots at network interface links, and not the exact positioning of time-slots that can only be known when the remaining IPs are bound in the following clusters. For low area and variable communication situation as shown in Figure 5.11, the decreased success rate is due to the area saving objective that concentrates the resources in a smaller number of Fnodes. Here, the objective is to reduce the cost that is paid over QoS guarantees, i.e., to avoid producing high fragmentation over the logic plane, which can prohibit the binding of future applications.

5.4.2 PUMA Scalability

To find out the the scalability of our PUMA scheme, we varied the number of application / architecture combinations as shown in Table 5.1. For instance, the first row of Table 5.1 shows an FPGA with 9 TCFRs and hops that are arranged 3x3 dimensions. During the binding process we: (a) pick up an application task graph and an FPGA architecture, (b) change the application area demands

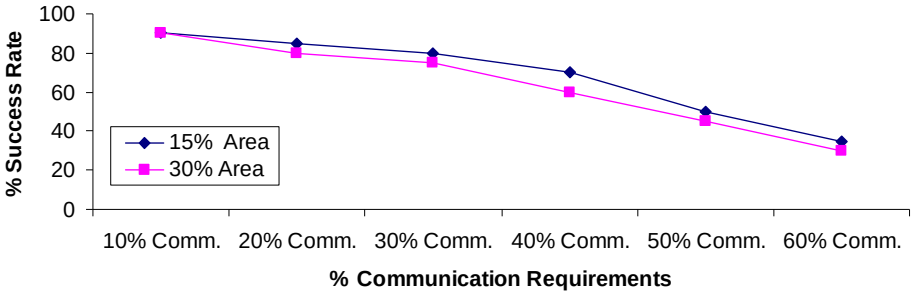


Figure 5.11: PUMA Success Rate with Low Area (i.e., 15% and 30% Area of FPGA) and Variable Communication Requirements.

Table 5.1: Success Rate over Multiple Applications and FPGA Dimensions.

FPGA Dimensions	App IPs	App Connections	%Success Rate
3x3	**5**	**6**	**68**
	7	**8**	**60**
	9	**14**	**56**
4x3	**11**	**18**	**65**
	13	**21**	**53**

from 15% to 60% of the target FPGA architecture, (c) for each area percentage, we change the throughput demands of the application from 10% to 60% of the available throughput, (d) record the result for each combination, and (e) average out the results to find out the success rate of the application binding on the architecture. As could be seen in Table 5.1, PUMA scheme holds a success rate in between 70% and 50%.

5.5 Conclusions

In this chapter, we presented a scheme to bind applications on FPGA architecture, which unifies all the three processes of placement, mapping, and allocation. The PUMA scheme ensures the QoS guarantees, whenever the binding of an application is successful. In the end, we presented the mechanism to traverse through the application to perform the binding, evaluated the success rate and scalability over multiple synthetic applications.

6

Run-Time FPGA System Adaptation

Earlier we explained the proposed FPGA architecture, which comprises multiple TCFRs, a hardwired NoC, and a control processor, Chapter 4. In this chapter, we make use of the HWNoC to transport functional communication (data and control) as well as *configuration* (bitstreams for soft IP). We start with listing the sequence of steps that are required to configure and program an FPGA system, Section 6.1. These will be discussed for the conventional as well as the newly proposed FPGA architecture. Next, we discuss a 3-tier model that we proposed for the dynamic run time reconfiguration of applications, Section 6.2. The 3-tier reconfiguration model performs composable inter-application and persistent-state intra-application dynamic reconfiguration. Afterwards, we list the limitations of the 3-tier model in Section 6.3. Thereafter, we present the results and evaluation of the run time reconfiguration process, Section 6.4. We end this chapter with conclusions in Section 6.5.

6.1 System Configuration & Programming: Overview

The control processor, as explained earlier in Chapter 4, is used to bootstrap, configure, and program the FPGA system. It does so by using its local NI to transport configuration and programming data over the HWNoC to TCFRs. First, the local NI is programmed with a channel to a remote NI. The new channel is used to program the remote NI. In this manner the whole HWNoC can be programmed, i.e. programming and data channels are set up. Conventionally in an ASIC [45, 46, 121], following this, the IPs are programmed (initialized and started) on their MMIO ports. However, in an FPGA, loading of bitstream is required first. At this point, the application is running. We now explain the configuration and programming of an FPGA with soft (conventional) and hard (our) interconnects.

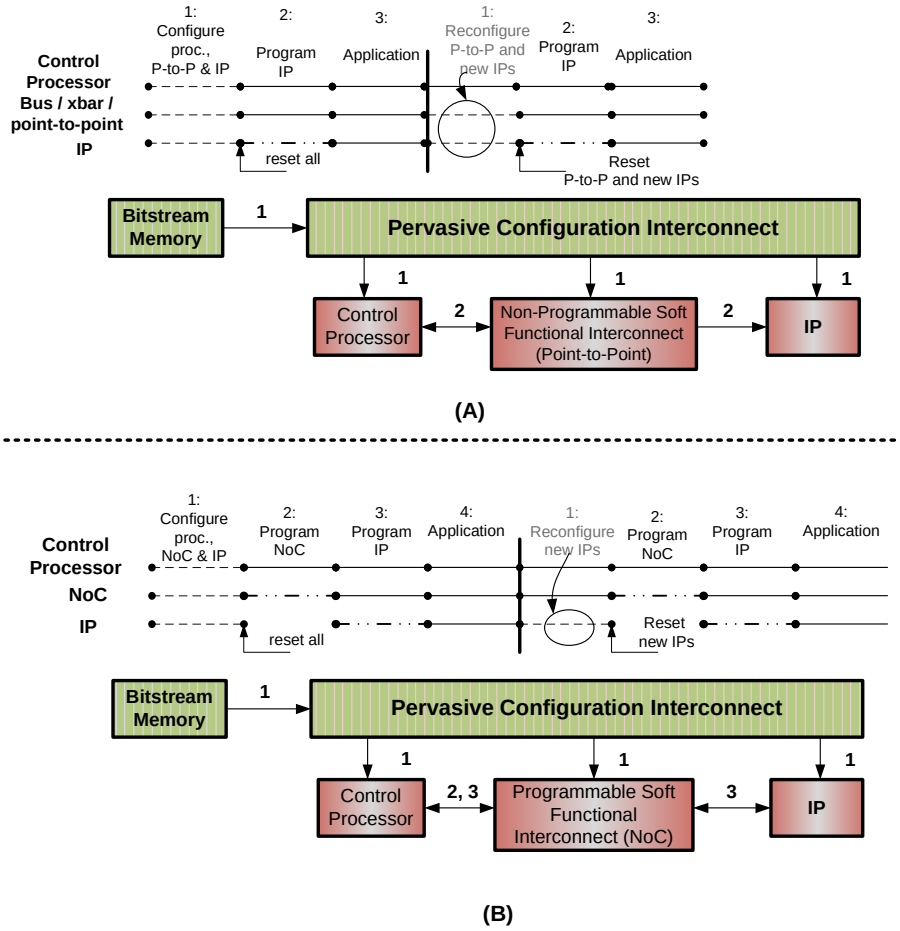


Figure 6.1: Conventional Configuration and Programming with (A) Non-Programmable Soft Functional Interconnect, and (B) Programmable Soft Functional Interconnect.

Note that in Figure 6.1, we use different types of lines to indicate different processes, i.e., configuration, programming, and execution. A *dashed line* indicates the configuration process, a *dot-dashed* line that it is being programmed, and a *solid line* indicates that the component is executing (i.e., functionally active). In addition, the labels on the lines show the sequence in which the processes take place.

6.1.1 FPGA With Soft Interconnect

For chips with configurable components (FPGAs, but also ASICs with embedded FPGA) a configuration phase is required. Figure 6.1 shows how an FPGA application without HWNOC is bootstrapped conventionally by using a non-programmable functional interconnect, and by using a programmable NoC functional interconnect.

To bootstrap an application on a conventional FPGA, first (shown by dashed lines and label 1 in Figure 6.1), the control processor (if not a hard processor, such as a PowerPC), interconnect (e.g., not programmable such as crossbar or point-to-point, or programmable such as NoC), and IP are all configured by copying a bitstream from a configuration memory (e.g., flash) using the conventional configuration IO (e.g., ICAP). After a functional reset, the control processor programs the IP only when non-programmable functional interconnect is in place, as shown with label 2 in Figure 6.1A. However, with programmable functional interconnect (e.g., NoC), the control processor programs the NoC and then the IP, as shown with labels 2 and 3 in Figure 6.1B respectively. Finally, the application is executing, i.e., processor, functional interconnect (NoC and point-to-point), and IP are all in functional mode, as shown with label 3 in Figure 6.1A and label 4 in Figure 6.1B.

(Partial) reconfiguration of the system, shown after the vertical bar, operates identically. However, care must be taken that those parts of the system that continue to operate are shielded from parts that are reconfigured [111]. As shown in the *grey text and circle* in Figure 6.1A, when IPs are reconfigured the non-programmable interconnect must probably be reconfigured too. If both non-programmable interconnect and IPs are reconfigured they are allowed to store the same (partial) reconfiguration regions, i.e., CLB columns. However, it is beneficial to leave the interconnect in place and not reconfigure it when reconfiguring the IPs. This is accomplished by making it programmable and placing it in separate CLB columns from the soft IP. It can then be left in place while IPs are reconfigured, and it is reprogrammed afterwards. The configuration interconnect is marked as pervasive because it reaches all configurable elements in the FPGA from the configuration IO connected to the (off-chip) bitstream memory. (As in conventional FPGAs.)

6.1.2 FPGA With Hard Interconnect

Although a soft programmable interconnect has advantage over a non-programmable soft interconnect; our proposed programmable hard NoC has

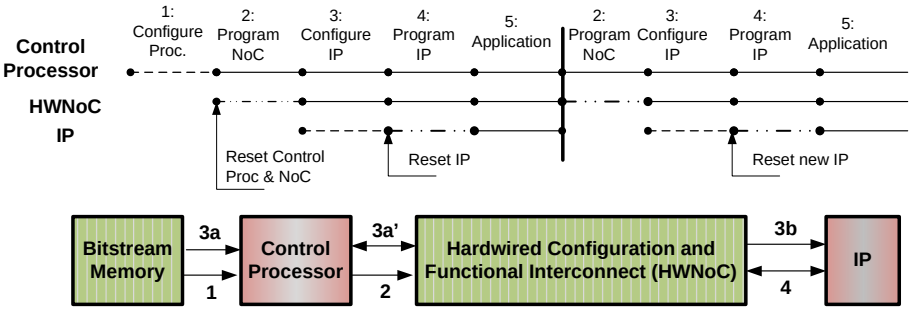


Figure 6.2: New Configuration and Programming with Programmable Hardwired Network on Chip.
Note that in Figure 6.2, we use different types of lines to indicate different processes, i.e., configuration, programming, and execution. A *dashed line* indicates the configuration process, a *dot-dashed* line that it is being programmed, and a *solid line* indicates that the component is executing (i.e., functionally active). In addition, the labels on the lines show the sequence in which the processes take place.

more advantages. Figure 6.2 shows how to configure a system when a hard NoC is used as the configuration interconnect. First, notice that the NoC is no longer configured (no dashed line for NoC). Second, IPs are configured only after the NoC has been programmed (dot-dashed line and label 2) because the bitstreams are transported using the HWNOC in the functional mode.

The configuration interconnect is now split in two parts 3a' and 3b, and Figure 4.1. First, the minimal top-level part that connects the configuration IO (with a DMA engine) to a master configuration port on an NI kernel, as shown with label 3a and 3a' in Figure 6.2. This is comparable to a streaming request channel from the control processor to the configuration port of an FPGA TCFR. Second, once the bitstream arrives on the other side of the NI (e.g., the IP of Figure 4.2) it enters the local (conventional) configuration interconnect of a TCFR to configure a specific IP, as shown with label 3b in Figure 6.2.

For a (partial) reconfiguration, the following steps are required. First, the IPs / TCFRs to be reconfigured are stopped by programming their MMIO ports. Then, via the HWNOC they receive new bitstreams and are reset on their MMIO ports. The HWNOC may be reprogrammed with the new application mode as well. Recall that bitstreams are streaming data and are not interrupted during their transport from bitstream memory to the IP (reconfiguration region). The original ICAP achieved this by transporting a single bitstream at a time. But a NoC transports many streams (data, control, bitstream at the same time), and communication in general does not achieve this. But the fixed-

latency GS communication service of the HWNOC is essential to avoid any (temporal) interference, because during the partial reconfiguration other IPs continue to operate and communicate using the HWNOC. Thus, our hard NoC offers two indispensable qualities: reprogramming instead of reconfiguration, and guaranteed (fixed-latency) communication.

6.1.3 Summary

We explained the sequence of steps that are required for a running FPGA system. The steps illustrated the configuration and programming of an FPGA system with a soft interconnect, and a hard interconnect e.g. a HWNoC. It shows that our HWNoC needs to be programmed, in contrast to a (conventional) soft functional interconnect configuration and at higher frequency. As we see later programming involves less data and it is faster than an ICAP. Therefore, with our architecture the pervasive configuration architecture can be replaced by a hardwired NoC.

6.2 3-Tier Model for Composable & Persistent-State Run-Time Reconfiguration

In this section we introduce the three tiers of the reconfiguration model in Section 6.2.1. We discuss the steps to enforce composability at inter-application level, while an application is dynamically reconfigured, Section 6.2.2. Afterwards, the dynamic reconfiguration process of an application is explained in Section 6.2.3. In the next section, we explain how persistent state is ensured at intra-application level, Section 6.2.4. Lastly, we provide the summary of the complete discussion in Section 6.2.5.

6.2.1 Responsibilities Across the 3 Tiers

In this section we explain the responsibility of each tier of our dynamic reconfiguration model, as shown in Figure 6.3.

System Manager

The system manager is executed on the control processor and executes always. The role of the system manager is to manage (allocate, deallocate) resources

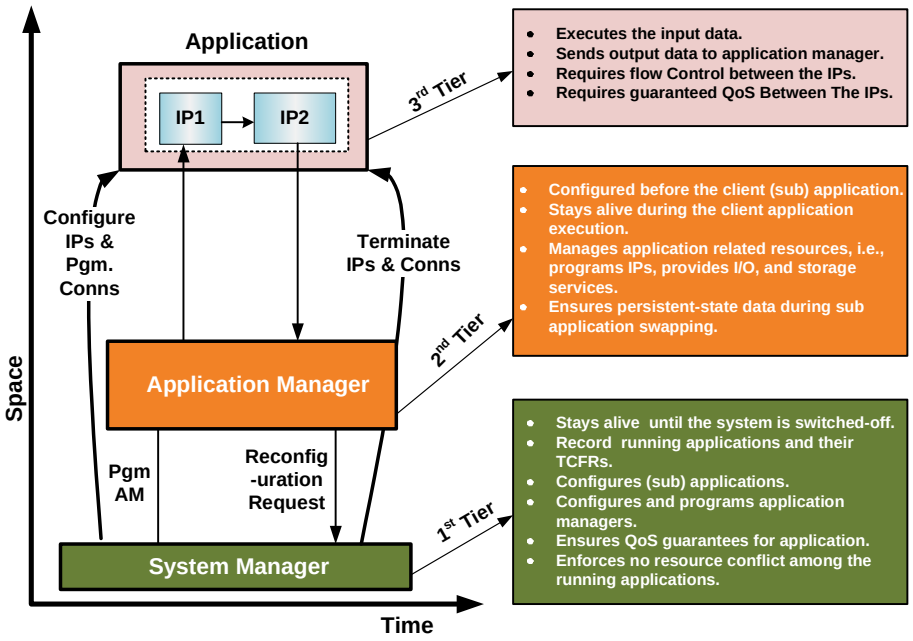


Figure 6.3: 3-Tier Reconfiguration Model with an Overview of Responsibilities of each Tier.

of applications. Applications have no access to resource budgets, schedulers, etc. to avoid interference, and to make the system more robust.

For each application, the system manager records the application identifier, application manager, configuration region and use-cases of the application, etc. The system manager also keeps record of resource budget (obtained at compile time) for all the applications. This includes the reservation of the paths for application IPs to communicate with each other, number and positioning of time-slots on the paths, and credit counter (over the network) between the source and destination application IPs [49] and TCFR regions.

Application Manager

Each application has its own application manager is deployed on per application basis. An application manager is a soft IP using FPGA resources, and that manages the resources of an application, i.e., programs IPs, loads / stores RAM data from / to IPs, saving application persistent-state (see Section 6.2.4) etc. As the system manager is unaware of an application's execution status,

an application manager is the one who notifies the system manager about its client application termination. An application manager must be stopped, if its application no longer exists

Application IPs operate on streaming data that arrives in a FIFO channel. At the start and end of the pipeline the data is sent / received by an application manager. The data comes from / goes to (e.g., a file in) memory or hardwired peripherals (e.g., ADC, DAC, USB, UART). This may take place through a direct connection from IP to peripheral, or via the application manager that adds addresses to receive / store data at appropriate memory location. However, an entire application might be too large to completely fit in an FPGA (or the part of FPGA that is reserved for it). In this case, an application is split into multiple sub-applications that are dynamically swapped in and out, as data flows through the system. This in turn requires that the state between the sub-applications is not lost during the reconfiguration process. An application manager, as discussed later in Section 6.2.4, provides a mechanism for dynamic swapping of sub-applications, taking care of persistent-state.

Application

An application forms the third tier of our reconfiguration model. The nature of our application is *streaming* and it consists of a number of *soft* IPs. The application IPs operate, i.e., consume and produce addressless data. The execution of application IPs on data is performed in a pipelined fashion, and the data is distributed in spatial and temporal domains. The (soft) IPs of an application are configured before being initialised, programmed, and executed, as explained in Section 6.2.3.

The 3-tier reconfiguration model enforces; (a) composability across the applications while they are configured and while they run, and (b) at the end of (sub)application execution, ensures persistent-state transitions while swapping a (sub)applications in / out of the system. We will expound on each of these functions in the following sub sections.

6.2.2 Enforcing the Inter-Application Composability

In a real scenario, each use-case represents a different combination of application(s). Therefore, each application will have its own QoS requirements, e.g., bandwidth and latency constraints that the communication infrastructure must efficiently fulfill to meet the required performance constraints.

Composable starting / stopping of applications ensures that the other running applications are not disrupted running. This means the composable swapping of an application avoids any conflict of resources on both the FPGA planes, i.e., the logic and communication planes. The logic plane resources include minimum configuration regions in a TCFR. The communication plane resources include; data connections and resources associated with each data connection, e.g., FIFO, and time division multiplex (TDM) slots.

To cope with the above challenges, the proposed methodology allocates a virtual platform for each application [5]. In this way the swapping of applications does not interfere with existing applications. To implement such a platform, the dynamic reconfiguration methodology calculates the communication plane resources (at compile time) for each application in accordance with its QoS requirements [147]. In case of multiple use-cases the principles of [45] are followed as well. The authors in [45] propose to consider all the use-cases and allocate for each application the required resources for its connections, such that not only its QoS requirements are fulfilled but also the use-case transitions do not impact its execution.

At run time, the reconfiguration methodology: 1) checks if new applications can be started, i.e., enough resources are available, 2) starts applications (as explained in Section 6.2.3 and Figure 6.2.).

6.2.3 Run Time Application Reconfiguration

Run time reconfiguration of an application is illustrated in Figure 6.4, which shows that the system manager instantiates (configures and programs) all the IPs of an application and the NoC. Figure 6.5 illustrates the process of instantiating a soft IP in its TCFR.

The system manager manages all resources in the system, i.e., keeps track of NoC connection, TCFR regions, and frames, etc. It ensures that an application is only started when all its required resources are available.

The system manager loads the bitstreams of an IP from the bitstream memory to the appropriate TCFRs. It does so by first setting up a connection from a port on its NI to the configuration port on the NI of the TCFR. This entails programming the HWNOC using memory-mapped IO (MMIO), as described in detail in [41,46]. For our purposes, this is performed by an abstract open_connection function. After the bitstream has been sent to the TCFR, the bitstream connection is removed. Bitstreams are modeled accurately, with registers such as system frame length (FLR), start frame address (SFAR), total bitstream frames

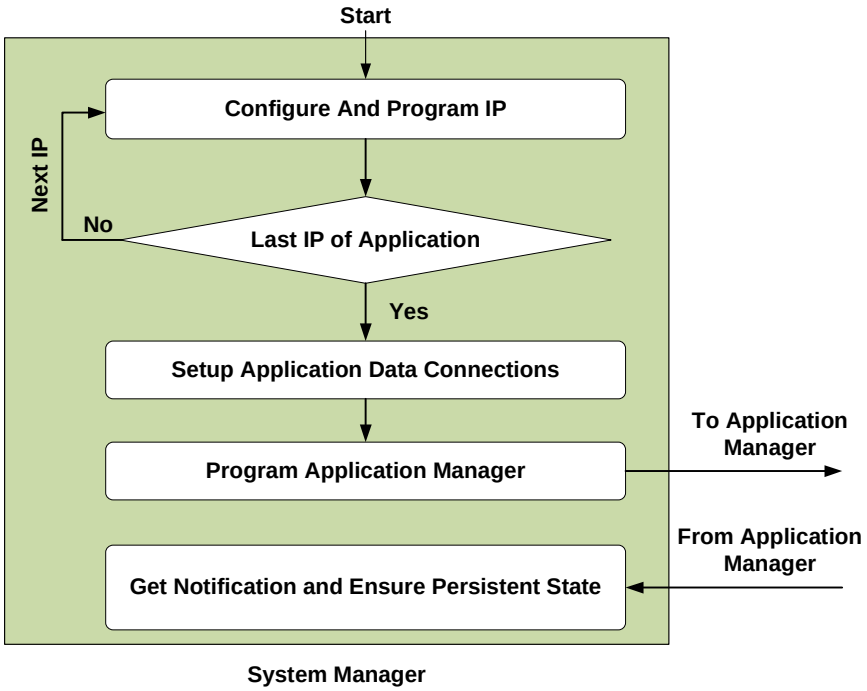


Figure 6.4: Application Configuration by Using the System Manager.

(TFR) and data frame register (DFR).

Initially, the system manager retrieves the bitstream for an IP from external configuration memory. The bitstream is a combination of frame headers and data. The header contains information about the location of the start frame, number of frames, and TCFR identifier. By keeping track of all the frames of other running applications in the system, the system manager can spot if an active frame would be overwritten. After this check, the respective TCFR is informed of the start frame location (SFAR), the number of frames (TFR), and the actual bitstream frames. Afterwards the bitstream frames are stored in the data frame registers, and are transported to the TCFR. The process repeats until the last set of configuration frames of an IP bitstream is transported to the destination TCFR.

At the destination TCFR, the configuration controller (see Figure 4.1) handles the incoming bitstream headers and frames. The configuration controller places the incoming bitstream at the correct locations as elaborated in Section 4.5.2. This way an IP is configured and the bitstream loading process iterates for all the IPs that can be placed in a TCFR. Afterwards, the initialisation

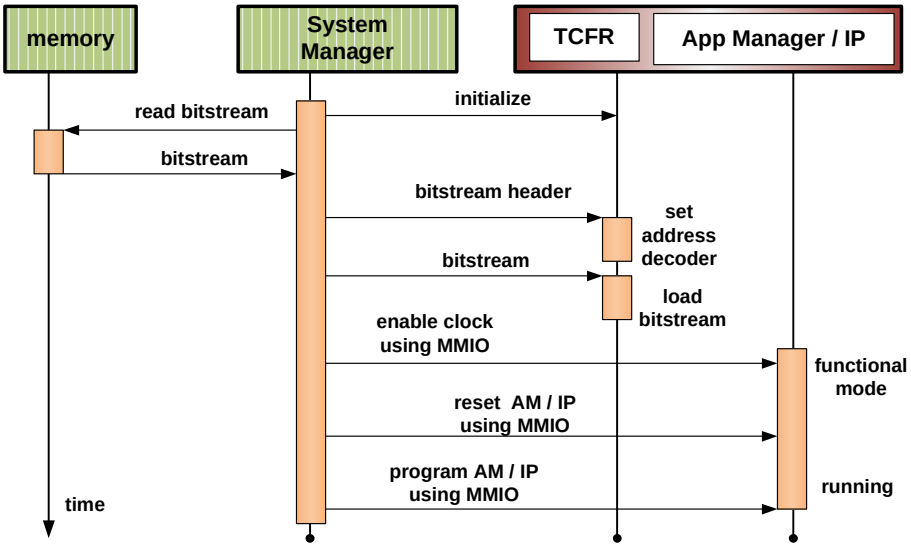


Figure 6.5: Starting a Soft IP.

is carried out per IP, i.e., setting up the clock and reset of an IP. The clock of an IP can be programmed and switched on/off by writing to memory-mapped (MMIO) registers via the NoC. Similarly, the soft IP can be reset by MMIO. On the TCFR end, there the clock and reset managers are used to enable / disable the clock and reset signals, respectively. At the end, the initialisation connection is closed and the network resources are released. This way a soft IP is physically placed on the reconfigurable plane of a TCFR.

As the next step, i.e., after placing the IPs on TCFRs, the system manager establishes the communication connections among the IPs, see Figure 6.4. It uses the resource allocation that is computed at compile time.

After the connections are established the system manager programs an application manager that interacts with an application during its execution time. An application manager plays a critical role to assure the persistent-state of application data while the (sub)applications are swapped in and out. In the next section, we illustrate the mechanism by using which an application manager ensure the persistent-state during dynamic reconfiguration.

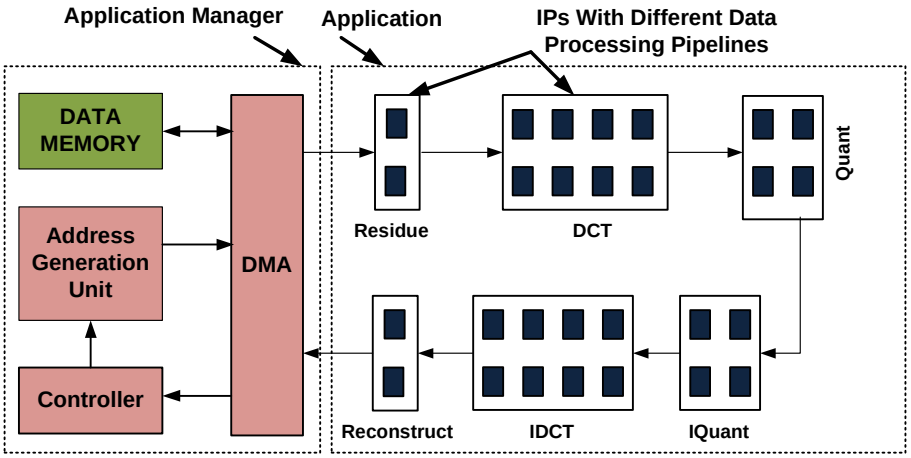


Figure 6.6: Interaction between an Application Manager and its Application.

6.2.4 Assuring the Intra-Application Persistent-State Transition

In this section, we explain the architecture of an application manager, and the procedure an application manager uses to ensure persistent-state transition of sub-applications. We also provide the details of the protocol that the system manager uses to program an application manager.

Application Manager Architectural Description

The architecture of an application manager is illustrated in Figure 6.6. It consists of a direct memory access (DMA) unit, a memory unit, address generation unit (AGU), and a controller. The DMA engine uses an address generator unit, which is programmed by the controller to generate input / output memory addresses. The input/output address are then used to fetch / store application data from / to data memory. An application manager in our 3-tier reconfiguration model is a soft IP. However, we have implemented an application manager in SystemC, where the different architecture blocks, e.g., DMA and memory unit interact by using function calls instead of using the ports. However, in a real architecture the communication between a DMA and memory unit can be performed by using the memory-mapped ports.

An application manager has a number of responsibilities. First, it acts as a *source and sink* for the processing pipeline, i.e., supplies the first IP in the pipeline with data, and receives the results from the final IP in the pipeline,

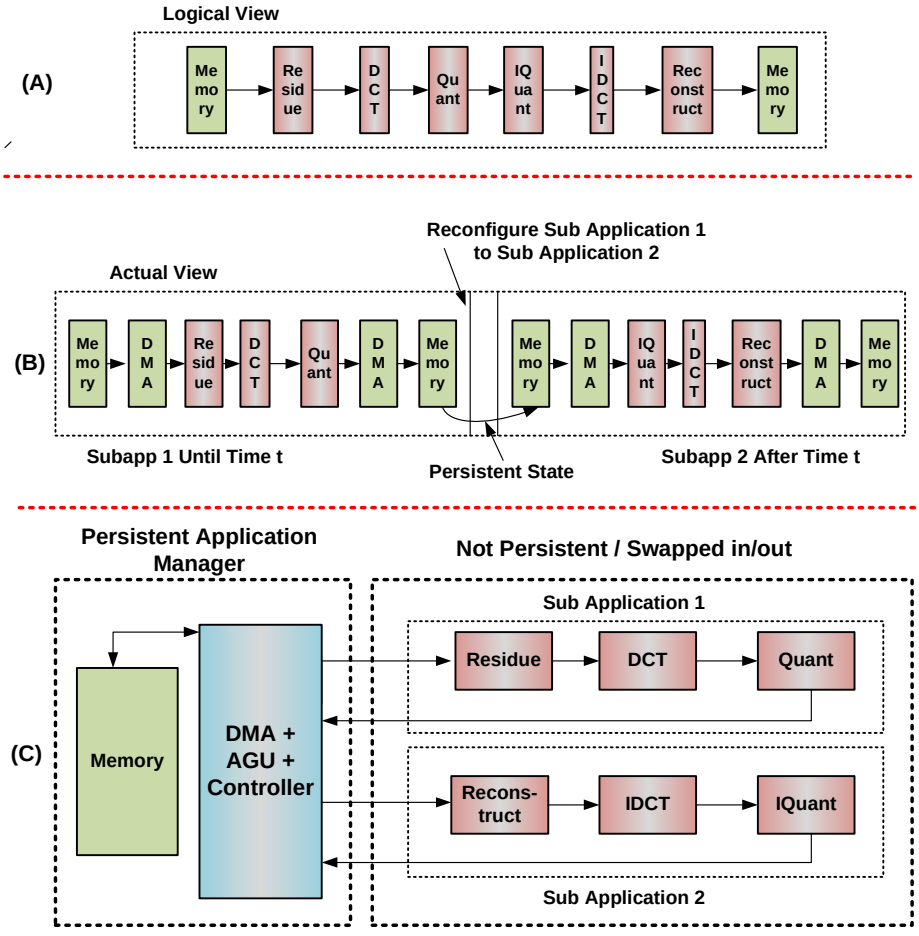


Figure 6.7: (A) Logical View of an Application, (B) Actual View after the application is divided into Two Sub Applications, (C) Interaction of Application Manager with Both the Sub Applications and Persistent-State Data during Reconfiguration.

see Figure 6.6. The DMA of an application manager sends / receives data to / from the client application by using the streaming ports. This data is read from and stored in memory, which can be internal or external to an application manager. In our experiments, it is internal as shown in see Figure 6.6. In a real system, the input / output data could come from the FPGA I/O, and would be handled in the same manner. Second, an application manager is aware of the *progress* of the pipeline, i.e., knows when all the source data has been processed (if ever), and when all the result data has been received (and hence if the pipeline is empty). As mentioned, an application manager contains one or more memories that act as sources and sinks to the rest of the pipeline.

Similarly, the address generation of output data from the respective memory locations is performed by using the address generation unit. Importantly, the controller has information about the application pipeline such as the number of input/output channels of the application, where the data of these channels is stored in the memories, and how much information must be sent or received. This information is stored in the local memory and has been received from the system manager as explain later. It also allows an application manager to know when the application pipeline is empty and hence finished.

An application that is dynamically swapped in and out can be split into multiple sub-applications, as the data flows through the system, as shown in Figure 6.7A, B. It incorporates that the state between the sub-applications is not lost during reconfiguration, i.e., the state of data persists during the swapping of applications, as shown in Figure 6.7 C. An application manager achieves this by keeping the record memory addresses of output data of previously swapped application. An application manager then can extract the data from there and provide as input to the next sub-application in the loop. The procedure by using which an application manager ensures a persistent-state during sub-application swapping is explained below.

Procedure to Assure Persistent State

Figure 6.8 explains a series of actions that an application manager performs to provide input to an application or after getting an output from an application. The numbers on the arrows provide the sequence in which the different functions are performed in an application manager.

On the input path (comprised of light blue rectangles in Figure 6.8), (1) an application manager receives application information from the system manager. (2) Based on the received application information, an application manager se-

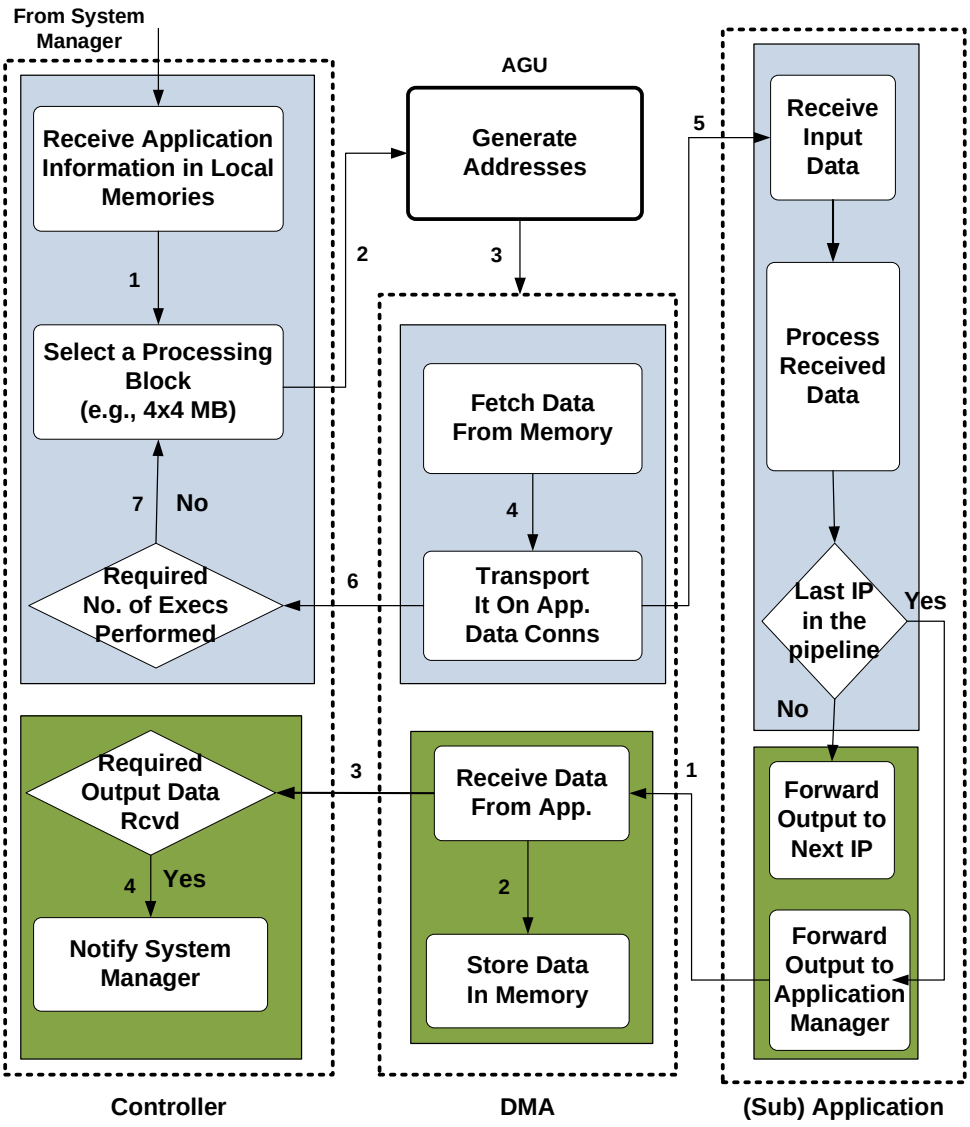


Figure 6.8: Procedural Description to Assure Persistent State by Using Application Manager.

lects the required processing block, and sends its information to the address generation unit. (3) An address generation unit then generates the required input and output memory addresses for the processing block, and sends the memory addresses to the DMA unit. (4) The DMA unit contacts memory and uses *input memory addresses* to fetch the required data. (5) Afterwards, the DMA unit transports data on a connection that connects the DMA to first IP of an application, as shown in Figure 6.7A. (6) The DMA unit then informs the controller about the transportation of data. (7) In case the required number of executions are not performed, the controller selects the next processing block and sends its information to the address generation unit. Alternatively, the controller stops sending the information of processing block to AGU, and waits for the required output data to receive.

On the output path (comprised of dark green rectangles in Figure 6.8), (1) the DMA unit receives processed data for the block whose data was earlier input to the first IP. The DMA receives the processed data from the last IP of the application, as shown in Figure 6.7A. (2) Afterwards, the DMA unit contacts the memory and stores the data on the required output addresses that were earlier received from the address generation unit. (3) The DMA unit then informs the controller about receiving the data for a particular execution. The controller verifies if data for the required number of executions has been received. (4) If so, the controller of an application manager notifies the system manager that the application has completed the execution of the required data. This means that at the end of a sub-application execution, the application manager triggers the reconfiguration request for the next sub-application by sending a notification to the system manager. The system manager then ensures that transition from one sub-application to another sub-application is performed as explained below.

Persistent-state reconfiguration of sub-applications is achieved first by tearing down the application data connections by using a systematic procedure, proposed in [46]. Disabling the application computational resources comes next, which starts with opening a reset connection to NI(s) associated with application IPs and afterwards sending a 32-bit reset signal to disable IPs from processing further. The system manager then configures the next sub-application and programs an application manager with the number of executions to performed. However, the system manager does not send the information about the memory locations to load and store the data. The reason is that the system manager is unaware of the memory locations, i.e., where the output of the previously executed sub-application was stored. It is the responsibility of the respective application manager to provide input data that is saved as persistent-

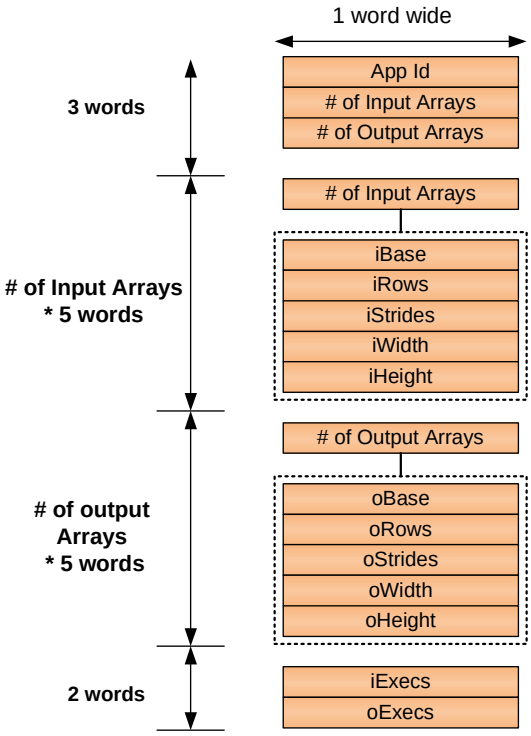


Figure 6.9: Programming Protocol Structure.

storage, to the next sub-application.

Notably during the swapping of a sub-application all the existing soft IPs of an application are reconfigured, except its application manager, see Figure 6.7. The reason as explained before that, an application manager sees consistent view over multiple sub-applications through persistent storage in its local memory. The next sub-application that is scheduled then operates on this data.

Programming an Application Manager

The system manager programs an application manager with the (1) identification of (sub) application, (2) number of different places from where data needs to be fetched from or to be stored (we call these arrays), (3) the pattern in which data should be fetched / stored for each array, and (4) the number of executions to be performed on the data.

The data structure by using which the programming information is sent is shown in Figure 6.9. The first three words of the data structure are used to represent the (sub)application id, the number of arrays to fetch the input data, and the number of arrays to store the output data respectively. Afterwards for each input array a 5 word information is embedded in the data structure in the following sequence. (i) The base address, i.e., the start location to pick up the input data. (ii) The number of rows, i.e., the rows that are going to be traversed each time data is picked up for the respective input array. (iii) The strides, i.e., the range of data to be picked in each row. (iv) The x-axis width of input array. (v) The y-axis height of input array, see Figure 6.9. Similarly, the information of each output arrays is embedded in the data structure, as shown in Figure 6.9. The last two words in the data structure indicate the (a) input executions, i.e., the number of times data is sent to an IP that receives input from an application manager, (ii) output execution, i.e., the number of times an application manager receives data from an IP that sends data to an application manager. It is important that in our experiments an IP that receives data from an application manager is the first IP of a (sub) application, and the one that sends data to an application manager is the last IP of a (sub) application as shown in Figure 6.6 where they are Residue and Reconstruction respectively.

It is important that the system manager is provided with the above information at design time, and it sends the programming information after the IPs of an application are configured and programmed to the application manager. After getting the programming information an application manager interacts with its (sub)application and ensures that a sub-application is replaced only when it finishes the required execution. Meanwhile, an application manager keeps the data in a persistent state while swapping in / out the sub applications. The architecture and procedure that are used to hold persistent-state data during the swapping of sub-applications is explained in the following discussion.

Example

In this example, we explain the process of generating input addresses for a single array. Figure 6.10A shows an example array, i.e., a frame with dimensions of 32x16 pixels, and that consists of two 16x16 macroblocks (MBs). Figure 6.10B shows the programming information that is input from the system manager to an application manager. This programming information is for the input data that is processed by the application. To keep things simple, we omit the process to store the output data that follows the similar procedure.

Figure 6.10B indicates the base address in a frame, i.e., the start execution

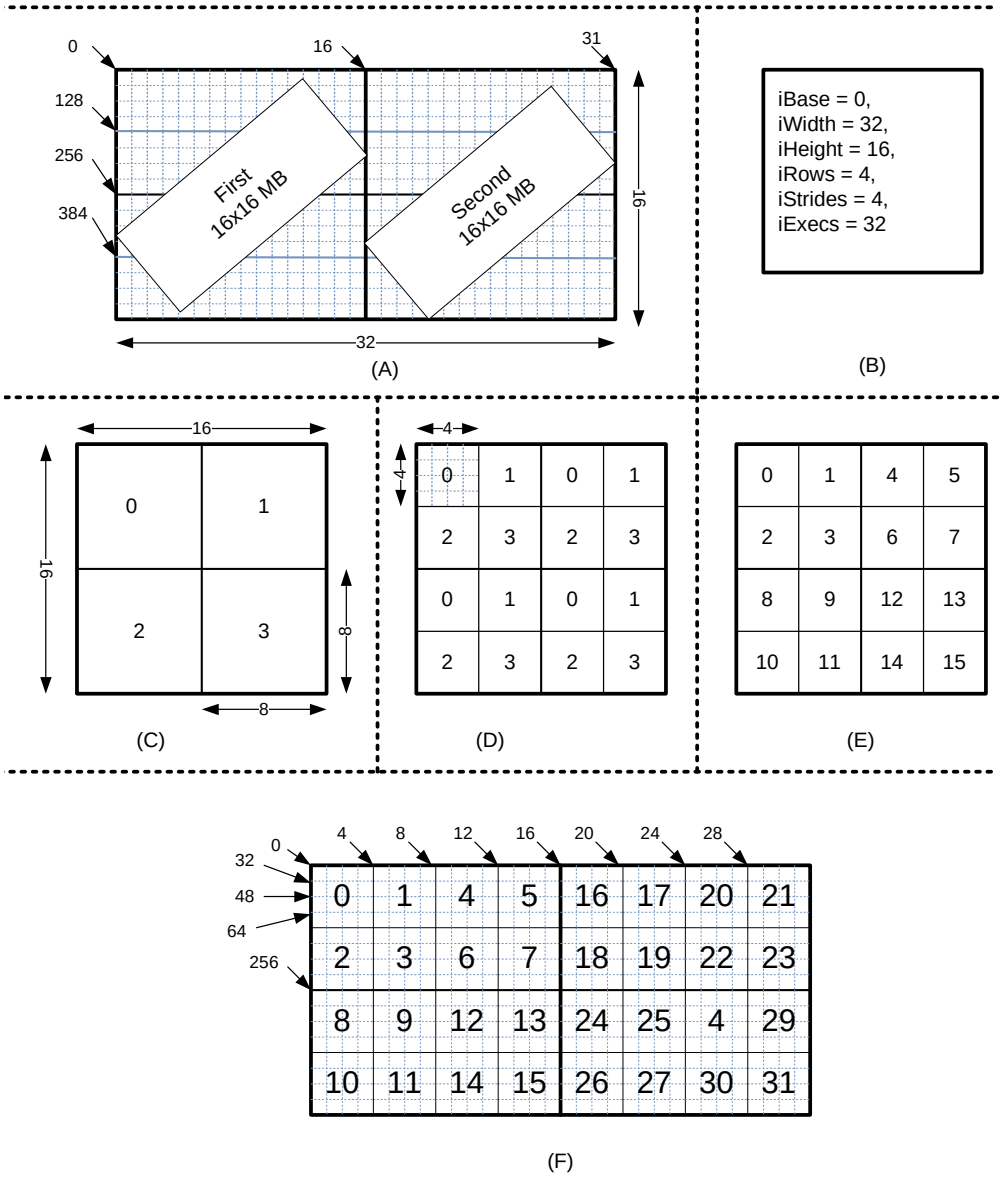


Figure 6.10: An Example Case Study of Application Manager Operating on Input Data.

point for an application manager. In this case it is the address of first 16x16 MB in a frame. Similarly, iExec = 32, iRows = 4, and iStrides = 4 indicate that an application manager will perform 32 executions on the input data, and in each execution an application manager will pick data from 4 rows, and from each row strides through 4 data units (or pixels).

Figure 6.10C shows a 16x16 Macroblock (MB), where each 16x16 MB consists of 16x16 = 256 pixels. Each 16x16 MB is then divided into 4 8x8 MBs. The order in which each 8x8 MB is picked-up for processing in a 16x16 MB is shown in Figure 6.10C. Similarly, each 8x8 MB is further subdivided into 4 4x4 MBs, where each 4x4 MB is the smallest execution unit of an application manager. This means, an application manager requires 16 executions to process a 16x16 MB, the order of executions of 4x4 blocks at 16x16 MB level is shown in Figure 6.10D.

Figure 6.10F shows the order of 32 executions, i.e., an application manager first goes through the first MB and then to the next one. Figure 6.10F also shows that to perform the 0th execution of first 16x16 MB, the row data is picked up from the logical frame addresses 0, 32, 48, and 64. The mapping of logical frame addresses to physical memory addresses is an internal function of address generation unit and is not the focus of the discussion.

6.2.5 Summary

Our reconfiguration model uses three tiers: system manager, application manager, and application to enforce composable and persistent-state dynamic reconfiguration. At run time, the *system manager* checks and configures the resources globally allocated (at compile time) to an application. Resources include the reconfigurable resources of TCFRs that are used to implement the soft IPs of an application, connections on the HWNoC that are used for communication between the soft IPs, and the memories used by the application. While there is only one persistent system manager, each application is accompanied by its own *application manager* that exist only for the duration of the application. It programs an application, and manages the application's IO and (persistent) storage. The application itself is unaware of these aspects and focuses on computation with IPs consuming and producing data on FIFOs. However, application IO and large FIFOs are implemented by RAMs with DMAs and address generators that are programmed by the application manager.

If an application is too large to fit then it is split up in sub-applications that are loaded by the system manager, but started / stopped by the application man-

ager. The application manager also ensures that data between sub-application reconfiguration is properly stored (persistent). In short, services which are related to application loading and resource allocation (i.e., HWNOC programming) are always provided by the system manager to ensure that one application can never affect another. On the other hand, an application manager provides intra-applications services which include I/O and storage to clients and enforcement of data integrity between the sub-applications that are dynamically swapped in and swapped out.

6.3 Limitations

Following are the limitations for run time application reconfiguration in the proposed FPGA architecture:

1. *System and Application Managers*: In our calculations we do not take into account the area of the system manager and of an application manager. Also no calculations are performed to indicate the power computation required by the system manager and an application manager.

2. *Composability Enforcement*: The resource allocation for applications is fixed, i.e., an application always receives the same resources independent of other applications it could be running with. Only one system manager is used, for all the applications of an FPGA, to ensure starting and stopping of applications in a composable way. This in turn can be a performance bottleneck as the number of application increases.

3. *Persistent State*: Our current implementation to achieve persistent state during intra-application swapping is rather limited. We allow no cycles in the application, and completely empty the sub-application's pipeline before starting the next sub-application. Thus soft IPs may be pipelined, but must be able to empty their pipeline when no new data arrives.

6.4 Evaluation and Results

In the result section, we provide the timings for system configuration and programming, by using conventional and proposed FPGA architectures.

6.4.1 Configuration, Programming, & Functional: Comparison

In the following discussion, we compare the configuration, programming, and functional phases of conventional FPGA and proposed FPGA.

Configuration

We first provide the details of the conventional configuration process. The configuration unit of a Virtex-4 device is a frame, containing 41 32-bit configuration words [154]. Our Virtex-4 device contains 39120 configuration frames. SelectMAP provides the fastest configuration, at a rate of 1.9 Gb/s (32-bit interface at 60 MHz). Therefore, the entire FPGA can ideally be configured in (39120 x 41 x 32 / (32 x 60 x 10^6)) = 26 ms or 0.7 μs per frame. A CLB column is the smallest coherent reconfiguration unit that contains 22 frames [132] and takes 15 μs to configure.

In the conventional FPGA the control processor, soft NoC, and IP are all configured, as shown earlier in Figure 6.1. In our experiments we do not perform configuration of the control processor. Therefore, only the timings of soft NoC and IP are obtained for analysis purpose. Note that for our experiments, we have taken an IP of 1000 LUT or 176 configuration frames. The number of configuration frames are estimated by using the following equation 6.1.

$$(IP\ LUTs * frames\ per\ column)/(LUTs\ per\ CLB * CLB\ per\ column) \tag{6.1}$$

In the *conventional* FPGA, the soft NoC requires 8100 LUTs per router-NI pair (Table 6.1). Here, the router consists of 5 ports, whereas the NI contains consists of 4 ports, 166 slots, and FIFOs of 41 words. To configure a single router-NI pair, at least (8100 LUTs x 22 frames per column) / (8 LUTs per CLB x 16 CLBs per column) = 1392 frames are required. This is equivalent to a bitstream of 1392 frames x 41 words x 32 b = 1.8 Mb. It takes at least 1392 frames x 0.7 μs = 975 μs to configure a single router-NI pair. Moreover, the soft NoC is distributed over the FPGA and is likely to occupy more frames. In addition, the soft NoC frames must be disjoint from *soft* IP frames, otherwise they cannot be reconfigured independently. Both increase the NoC configuration time. Therefore, both the configuration time and bitstream size are optimistic estimates. Similarly, an IP of 1000 LUTs is configured by using the same configuration interconnect and it takes 176 frames x 0.7 μs = 123 μs.

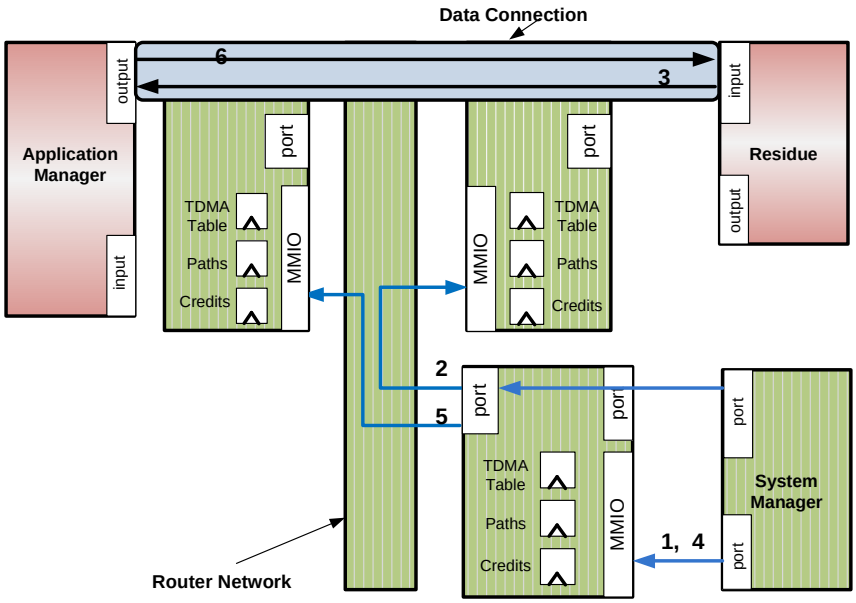


Figure 6.11: Procedure to Program NoC Connection.

In the *proposed* FPGA, the NoC is not configured due to its hardwired nature. However, the control processor and soft IP are configured both, as shown earlier in Figure 6.2 (Step 1 and Step 3). As stated earlier, we do not perform the configuration of the control processor in our experiments. However, an IP is configured by using the hardwired NoC that runs at 500 MHz, as earlier explained in Section 4.9.6. Note that we assume a 30% packetisation overhead, which reduces the efficiency of NoC to 70%. We have reserved 50% of NoC resources for the configuration connection, which means (176 frames x 41 words x 32 bits) / (0.7 x 0.5 x 32 bits x 500 x 10^6) = 40.4 μs to configure an IP (of 1000 LUTs or 176 frames) to the required FPGA region.

Programming

In the conventional and proposed architectures, the programming of NoC and IP is performed. In other words an NoC, whether soft or hard, must be programmed, as described in Section 6.1 and [46]. An NoC is programmed to setup data connections for an application, so that IPs can communicate with each other. The programming of a connection requires a systematic procedure as proposed in [46] and shown in Figure 6.11.

Figure 6.11 explains the procedure that the system manager uses to set up a connection between an application manager and Residue IP. A connection consists of request and response channels. The system manager programs the NI of application manager and NI at Residue each contains information (e.g., TDM slots, path, and credits) to set up a connection between application manager and Residue. Each connection (a pair of request-response channels) requires between 832 and 2096 bits. It takes 2.5 μs to program a single connection [46] assuming a hardwired NoC running at 500 MHz. The system manager uses MMIO read and write transactions to configure the NIs (routers are not programmed). The boot time of the control processor is not taken into account here.

In the proposed FPGA architecture, an IP is programmed via hardwired NoC. Programming of IP entails changing / resetting the state of its registers, whenever required. However to program an IP, the hardwired NoC must be programmed so as to reach the specific IP. This means for each IP, two connections must be programmed; first to program the NI for setting up data connection between IPs, and then to program an IP itself. This means 2 x 2.5 = 5 μs are required to program an IP.

In the conventional FPGA architecture, the soft NoC takes 2.5 μs x 500 MHz / 118 MHz = 10.6 μs to program a data connection between the two IPs. Here, 500 MHz and 118 MHz stand for hard NoC frequency and soft NoC frequency, respectively. These frequencies were obtained earlier in Section 4.9.6. Note that an IP itself needs to be programmed as well, and a separate connection is required to program an IP that also takes 10.6 μs.

Functional Mode

In the conventional architecture, the soft NoC runs at 118 MHz, which means an NI can provide a raw bandwidth of (32 bits x 118 x 10^6) = 3.8 Gb/s. On the contrary the hard NoC that provides a raw bandwidth of (32 bits x 500 x 10^6) = 16 Gb/s is a factor 4 faster than the conventional FPGA. The IPs in both the conventional and proposed architecture are configured in the reconfigurable logic plane. Therefore execute at the same frequency in both the cases.

However, as discussed above with 30% packetisation overhead the efficiency of NoC to transport actual data reduces to 70% of the raw bandwidth. This means, the soft NoC to provide a net bandwidth of (3.8 Gb/s) x 0.7 = 2.6 Gb/s, and hard NoC to provide a net bandwidth of (16 Gb/s) x 0.7 = 11.2 Gb/s.

Table 6.1: Configuration, Programming, and Functional Comparison of Conventional and Proposed Architectures.

Phases	Modules	Conventional Soft	Proposed Hard
Configuration	Ctrl Proc.	-	-
	NoC (per R-NI pair)	975 μs	0
	IP (1000 LUTs)	123 μs	40.4 μs
	Overall (IP+R+NI)	1098 μs	40.4 μs
Programming	NoC (1 Conn.)	10.6 μs	2.5 μs
	IP (1 Conn.)	10.6 μs	2.5 μs
	Overall	21.2 μs	5 μs
Functional	NoC (Raw)	3.8 Gb/s	16 Gb/s
	NoC (Net)	2.6 Gb/s	11.2 Gb/s

Overall

Recall Figures 6.1 and 6.2 where the phases of booting a system were described. The Table 6.1 shows the time that is spent on each of following phases for each of the two scenarios: configuring the functional interconnect, programming the functional interconnect, and configuring and programming the IP.

A hard NoC requires programming per NI, and the soft NoC requires configuration per NI. The programming requires fewer bits and is faster too, a system with a hard NoC is ready for functional operation 1098 μs / 40.4 μs = 27.1 times faster. The additional gain of (21.2-5) μs to program each functional connection does not significantly improve this number.

Note that our analysis is mostly independent of particular NoC because it essentially depends on three factors: the ASIC:FPGA area ratio for LUTs, the ASIC:FPGA operating frequency ratio, and the configuration:programming footprint (bitstream versus number of MMIO bits) ratio.

6.4.2 Conventional and Proposed Architecture Comparison for Larger Systems

The analysis in the previous section is based on unit values, i.e., single IP, single router-NI pair, and single connection / IP. We extend our configuration, programming, and functional comparison to larger systems (i.e., larger networks, more IPs, and more connections), as shown in Figure 6.12 and Figure 6.13

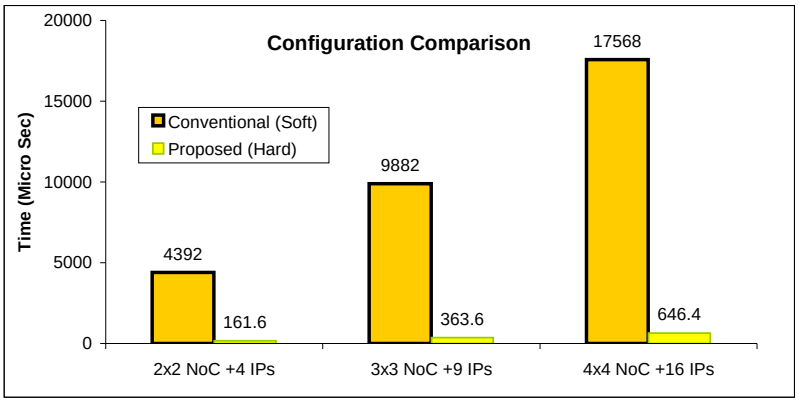


Figure 6.12: Configuration Time Comparison Between the Soft and Hard Architectures.

respectively.

For instance, Figure 6.12 shows that the conventional FPGA architecture takes 9882 μs to configure a system with 3x3 NoC and 9 IPs of 1000 LUTs each. However, the same system is configured in 363.6 μs by using the proposed FPGA with hard NoC.

Similarly, the programming of 9 connections, assuming one connection / IP, requires 190.8 μs in a conventional FPGA with a soft programmable NoC, see Figure 6.13. However, the proposed FPGA can program the same number of connections in less time (45 μs).

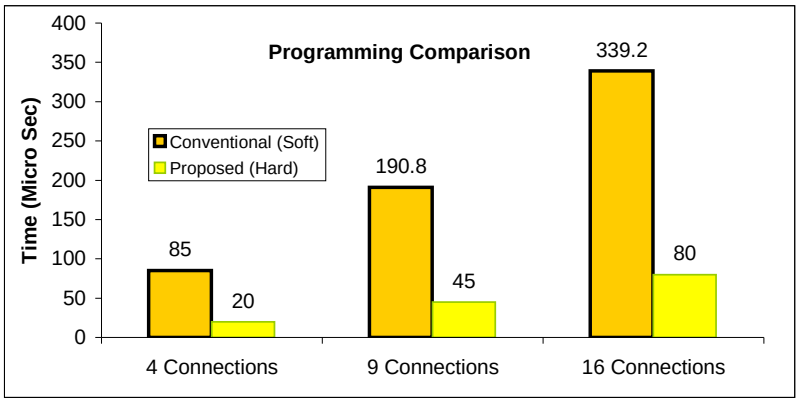


Figure 6.13: Programming Time Comparison Between the Soft and Hard Architectures.

6.5 Conclusions

In this chapter, we modeled a FPGA architecture that uses a hardwired NoC (HWNOC) to transport configuration (bitstream) and functional data. We describe the run-time procedures to configure, program, and run soft IPs. We modeled the platform in cycle-accurate transaction-level SystemC, together with soft IP blocks. In particular, we model bitstream loading and frame placement of soft IPs, soft IP clock management and reset, the programming of HWNOC and IPs, and their functional operation. We compared the performance of a conventional FPGA with a soft NoC and dedicated configuration interconnect with our HWNOC platform. Bitstream loading is faster in our platform.

This basic infrastructure is then used to dynamically start and stop entire applications, and also sub-applications. The latter is useful when an application does not fit in the resources (configuration regions) allocated to it. Dynamically swapping sub-applications then enables the entire application to execute anyway, although at a lower speed. Our platform is composable, which means that starting and stopping of applications does not affect concurrently operating applications (and vice versa).

7

Online Testing of FPGA Architecture

In Chapter 6, we unified the transport of bitstream and functional data by using the proposed HWNOC architecture. In this chapter, we make use of Hardwired Network on Chip (HWNoC) to transport *test* data. We propose to use the HWNoC as Test Access Mechanism (TAM) and conduct *test on a region-wise basis*. A region in our methodology stands for a test configuration functional region (TCFR), as explained earlier in Section 4.3. Our online test scheme is *non-intrusive*, because our test scheme allows an interleaved test process in parallel with configuration, programming, and execution of other applications. This means the testing of a TCFR is not stopped / delayed because of ongoing configuration, programming, and execution processes in the remaining TCFRs of FPGA. Moreover, our online test process is triggered at an application startup time, so as to make sure that an application always executes on a tested region. However, the nature of testing is application independent, i.e., it is structural. This means the test process evaluates the structure of FPGA to find out the prospective faults. A structural test ensures a high percentage of fault detection for the target FPGA architecture. In addition, the proposed scheme has *reduced spatiotemporal overheads*, because it does not make use of FPGA reconfigurable resources for creating test pattern generators (TPGs) and output response analysers (ORAs) that generate and analyse test sequences, respectively. Instead, the proposed scheme uses the system manager that makes use of connections through the hardwired NoC to provide test stimulai and analyse the architecture of a particular region in FPGA.

To ensure a non-intrusive test behavior, we take into account design, compile, and run time phases described earlier in Section 3.1.2. At design time we provide FPGA architecture and application specifications as shown in Figure 3.4. From the *architecture point of view*, the specifications are (i) the dimensions and architecture of hardwired NoC and test configuration functional regions (TCFRs), and (ii) the control processor, which is a programmable hardwired IP.

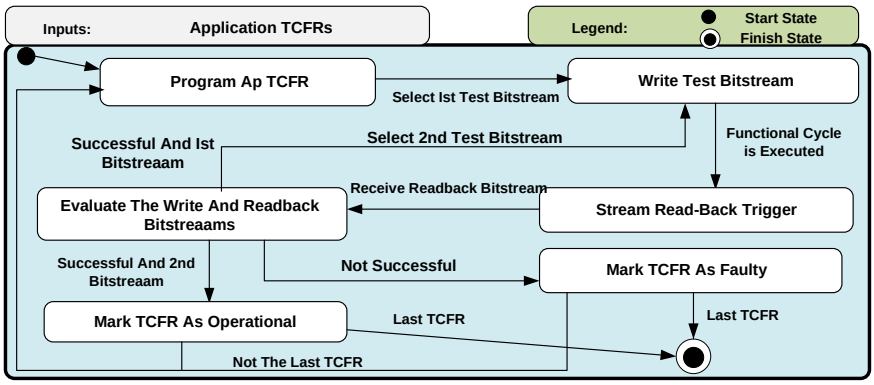


Figure 7.1: Run Time Flow for the Test Process.

From the *application point of view*, the specifications about the system application are provided. The *system application* performs the online testing of the target FPGA architecture and, also the configuration of the user-application in the desired FPGA region(s). The system application consists of the system manager that is mapped to programmable control processor, and TCFRs of FPGA. A test connection is used to transport test data between the system manager and a TCFR. The test connections are allocated at compile time, but are created and terminated at run time. The PUMA scheme, as discussed in Chapter 5, is used to reserve the resources of connections at compile time.

In the following discussion, we explain the run time behavior of our test methodology in Section 7.1. We provide the limitations of our test approach in Section 7.2. Afterwards, we evaluate the non-intrusive nature of the test scheme, and present timings details to test an application TCFRs, see Section 7.3. We end this chapter with conclusions in Section 7.4.

7.1 The Test Methodology

In this section we provide the details of testing test configuration functional regions (TCFRs) in Section 7.1.1. Although the testing of TCFRs is the focus of the chapter, we also provide the details to test HWNOC in Section 7.1.2.

7.1.1 TCFR Testing

At run time, the online testing of an application TCFRs is performed. However, before explaining the process we discuss the nature of faults that our methodology targets.

A TCFR, as detailed earlier in Section 4.3, consists of multiple configurable logic blocks (CLBs) (in our case it is 512 CLBs). The online test methodology targets stuck-at faults that can occur in the combinational part a CLB (i.e., look-up-tables (LUTs)), and configuration memory of a CLB. Note that the configuration memory cells of a CLB contain the bits that are related to the: (i) combinational part of a CLB, (ii) sequential part of a CLB, and (iii) associated interconnection network[1].

In a faulty FPGA, the configuration memory cell can stuck-at a particular value, i.e., 0 or 1. To find out stuck-at faults in the configuration memory, we make use of two complementary test bitstreams. One test bitstream, configures each LUT as exclusive-OR (XOR) gate. The other test bitstream configures the LUTs as exclusive-NOR (XNOR) gate [117–119].

Similarly, the functional input(s) of an LUT can get short circuited. The output of LUT can then permanently point (i.e., stuck-at) to the same configuration memory address irrespective of the input patterns applied to an LUT. To figure out a defective input port to configuration memory address mapping, we provide an exhaustive set of $2^4 = 16$ test vectors of 4-bit each at the functional inputs of an LUT. This way we can verify that the output of an LUT is as per desired or not, i.e., the output is not stuck-at a particular configuration memory address.

A 5 phase (program, write, execute functional cycle, read, and evaluate) test process is carried out for each test bitstream, (i.e., XOR and XNOR configurations), see Figure 7.1. The test process starts with programming a test connection for the required TCFR. This is followed by writing the test bitstream to the TCFR. Once writing is finished, a functional cycle phase is executed to find out the prospective defects in the combinational part of CLBs. Afterwards, a read-trigger is sent to the TCFR to read-back the test bitstream. To find out the faults in the configuration memory, the read-back bitstream is then evaluated against the originally written test bitstream. In the following discussion, we explain each of the phases.

[1]Each of 8 LUTs in a CLB has 16 1-bit configuration memory cells, but the configuration memory that is associated wit a complete CLB is approximately 1800 bits [125, 154].

Program

Programming requires the usual open-connection sequence (e.g., see Chapter 6). Two bidirectional connections are required: (1) for writing test bitstream and the other is for writing functional data from the system manager to TCFR. The system manager programs its NI and the NI of the required TCFR to setup a test connection.

Write

The system manager after programming a connection retrieves the test bitstream from an off-chip memory. The bitstream is a combination of packet headers and frame data to configure a complete TCFR. The test bitstream is sent in the frame-wise manner over an established test connection. A frame of 41 words arrives over the NI and is received by a TCFR[2]. Then it is shifted byte by byte into the address decoder and to the appropriate MTCR. This is repeated for the appropriate number of frames. The bitstream loads a test IP (XOR, or XNOR).

Execute Functional Cycle

At the end of the bitstream writing process, an XOR or XNOR test IP has been placed on the logic plane of a TCFR. The logic circuit of test IP spans complete TCFR, i.e., the bitstream of test IP is loaded in all the minimum test configuration regions (MTCRs) of a TCFR, as shown in Figure 7.2A. It is important that each MTCR unit in a TCFR is configured as a set of four chains of CLBs, as shown in Figure 7.2B. Moreover, each of the eight (4-bit input to 1-bit output) look-up-tables in a CLB is configured as an independent exclusive-OR or exclusive-NOR circuit, as shown in Figure 7.2D and Figure 7.2E respectively. Hence 32 bits are required to derive the inputs of all the LUTs in a CLB, which in turn produces an 8-bit output, see Figure 7.2B.

Figure 7.2B shows that for each CLB chain, the 32-bit inputs of the first CLB are derived from a 32-bit register. This can be a hardwired (Block RAM) register, which is programmed through the system manager. The 8-bit output of the first CLB is then input to the next cascaded CLB and so on, see Figure 7.2B. As a result of the arrangement, each CLB chains has an 8-bit output that is fed into a 32-bit programmable register. The data is then read back by the system

[2]We refer to Figure 4.4 while explaining the write process.

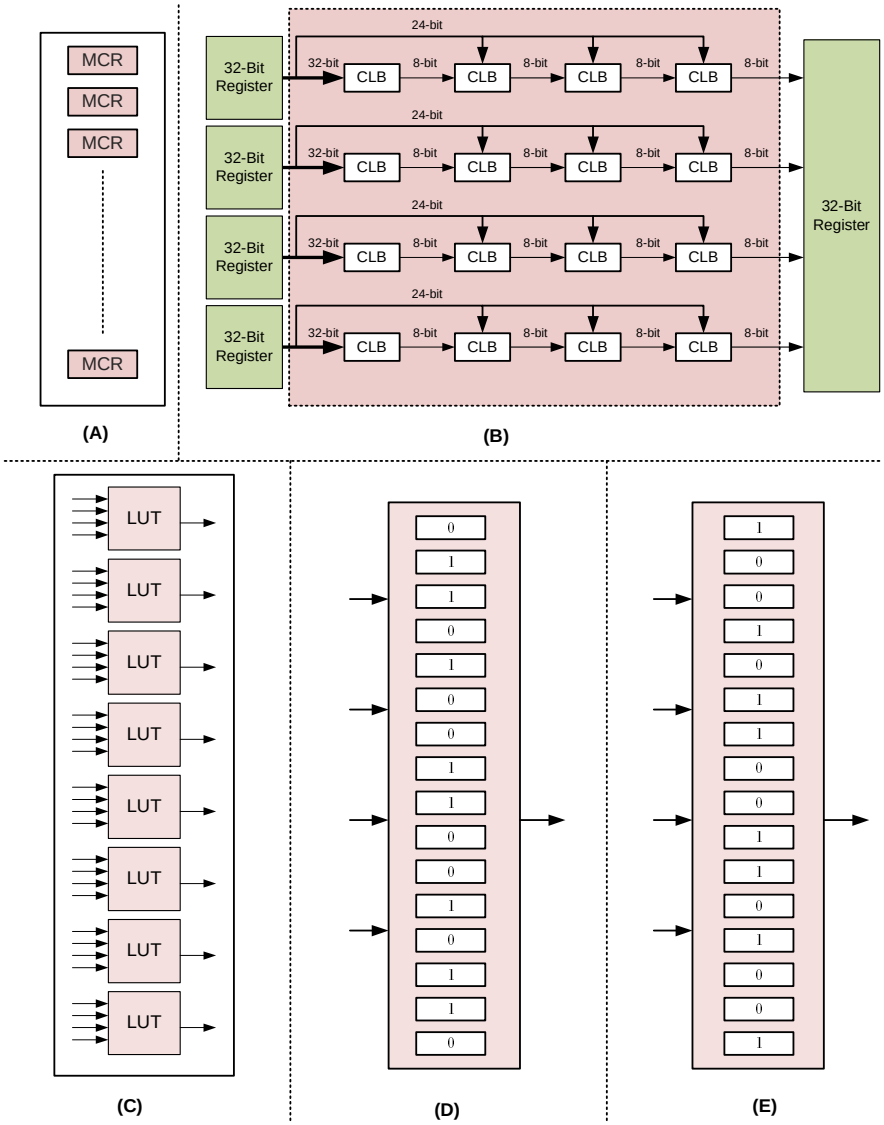


Figure 7.2: Test IP placed in our FPGA with Different Abstract Level Details. (A) Test IP Circuit in terms of MTCRs, (B) An MTCR Configured as a Set of Multiple CLB chains, (C) 8 LUTs in a CLB that are not cascaded within a CLB, (D,E) An LUT Configured as XOR and XNOR Gate Respectively.

manager to analyze the response of an MTCR for a particular set of input data. At this point it is important to mention that the use of hardwired registers for input and output of CLB chains is a design time choice. The hardwired registers are used, because it eliminates the need to configure additional hardware on the reconfigurable plane that is already under test.

Once the test IP is placed, the system manager executes the test application for multiple functional cycles to find out the defects in the mapping of functional ports of an LUT. The system manager generates and transports test vectors for all the CLB chains by using the HWNOC. To test each CLB chain the system manager sends 16 test vectors of 32-bit each. Each test vector is then further decomposed into 8 groups of 4-bit each that changes from 0000 to 1111. The reason is that each of the eight (4-bit input to 1-bit output) look-up-tables in a CLB is configured as an independent exclusive-OR or exclusive-NOR circuit. It is important that the whole process is pipelined and it takes 4 cycles to reach from the input to output of a CLB chain.

After evaluating defects in the combinational part of a TCFR, the system manager triggers the read-back bitstream process. This is performed to evaluate defects in the configuration memory of a TCFR.

Read-back

After executing the test application, the bitstream is read back for comparison to the original bitstream that was uploaded. This checks for stuck-at faults in the bitstream registers. The system manager sends a read command with start address and size (Number of frames). The TCFR performs the reverse actions of what is required for writing the bitstream.

Evaluate

In this phase the system manager evaluates the read-back bitstream by comparing it with the originally written bitstream. As stated before, two complementary bitstreams are required to identify possible stuck-at 0 and stuck-at 1 faults in a TCFR. However, the second bitstream is sent only if the evaluation process is successful for the first test bitstream, Figure 7.1. The reason is that we currently focuses on fault detection, and no fault tolerance mechanism is applied to replace the faulty portion of a TCFR. This allows us to save the time that can be spent in testing a TCFR, which has already been detected faulty.

7.1.2 Perform HWNoC Test

The HWNOC testing is associated with testing of network interfaces and routers. We do not perform the testing of hardwired NoC as it is not the focus of thesis. However, the network interfaces can be functionally tested by using the IP cores attached with them as suggested in [93]. The method proposed in [93] tests the functionality of NIs, because a fault-free NI can then produce the expected functionality for the attached core. The routers can be tested online by using the approaches of [57] and [36].

Once the test is finished, the dynamic configuration of the application is started, see Section 6.2.3. Additionally, to allow interleaved test and configure operations, we impose a restriction that no two applications share the same TCFR(s) simultaneously.

7.2 Limitations

Following are the limitations of our online test methodology:

1. In the combinatorial part of a TCFR, the contents of LUTs are tested and the testing of flip flops is not aimed.
2. The testing of HWNoC and other hardwired components, e.g., control processor is not targeted.
3. The clock tree of the architecture is also not part of the proposed test scheme.
4. The test scheme target only the stuck-at permanent faults and other types of faults, e.g., transient faults, coupling faults etc. are not meant to be tested.

7.3 Results And Analysis

We exercise our online test methodology in SystemC, and use ÆTHEREAL [44, 48] NoC as the hardwired NoC. The HWNOC platform runs at 500 MHz, and consists of routers and NI kernels with FIFO sizes of 3 and 41 words, respectively. We use Virtex-4 XC4VLX200 chip to synthesis the applications, and then embed the synthesis results in the SystemC model of our FPGA.

Table 7.1: IP Synthesized Area, Frequency, and Bitstream Frames.

IP	Area ($kLUTs$)	Frequency *(MHz)*	Bitstream *(Frames)*
Residue	**1.68**	**100**	**285**
DCT	**2.36**	**66**	**396**
Quantizer	**2.21**	**75**	**370**

The FPGA architecture consists of multiple TCFRs, whose combined area is equivalent to 178176 LUTs, i.e., equals to the *Virtex-4 XC4VLX200* chip. We use two complementary test bitstreams to *verify a TCFR for stuck-at fault model*. The size of the each test bitstream is estimated from the following equation:

$$(IP_{LUT} * MTCR_{frame}) / (CLB_{LUT} * MTCR_{CLB}) \quad (7.1)$$

In equation 7.1 the first term, i.e., IP_{LUT} refers to LUT area of an IP, CLB_{LUT} refers to LUTs in a single CLB, and $MTCR_{frame}$ and $MTCR_{CLB}$ stand for frames and CLBs in a single MTCR respectively. Each CLB consists of 8 LUTs, and an MTCR consists of 16 CLB units[3]. This means that our FPGA architecture consists of (178176 / 8) = 22272 CLBs and 1392 MTCRs. Moreover, an *MTCR is configured by using 23 41-words frames* [154].

For our online test methodology, we (i) provide an evidence of its non-intrusive nature (Section 7.3.1), (ii) evaluate the performance in terms of fault detection latency (Section 7.3.2), (iii) evaluate the cost in terms of spatiotemporal overheads (Section 7.3.3), (iv) analyse the impact of TCFR area on the performance and cost of our methodology (Section 7.3.4), and (v) compare the performance and cost with the two state of the art schemes [37, 142] (Section 7.3.5).

7.3.1 A Non-Intrusive Test Methodology

In this section we analyse the non-intrusive nature of our test methodology. First, we describe the experimental setup in the following discussion.

Experimental Setup Discussion

We use the behavioral models of H.264 IPs, i.e., (Residue, DCT, and Quantiser). Synthesis of the VHDL implementations of these IPs, on a Virtex-4

[3] In Virtex-4 the minimum test configuration region, i.e., an MTCR consists of a column of 16 CLBs and the associated programmable interconnect [154].

XC4VLX200 chip using Xilinx ISE 10.1, provide their MHz frequencies that are used in the SystemC simulations, Table 7.1. The size of their bitstreams is estimated from their kLUT areas by using the Equation 7.1.

To verify the non-intrusive nature of our online test methodology, we use a *system with three applications*, i.e., A0 (Residue + DCT), A1 (DCT only), and A2 (Quantiser only) that execute in two use-cases. A0 and A1 execute in parallel and belong to use-case 0 (UC0), whereas A1 and A2 execute in parallel and belong to use-case 1 (UC1). Importantly, A0 and A2 are the sub-applications of one larger application, and are time-multiplexed on the same set of TCFRs, i.e., TCFR0 and TCFR1. A1 executes on TCFR3, as shown in Figure 7.3.

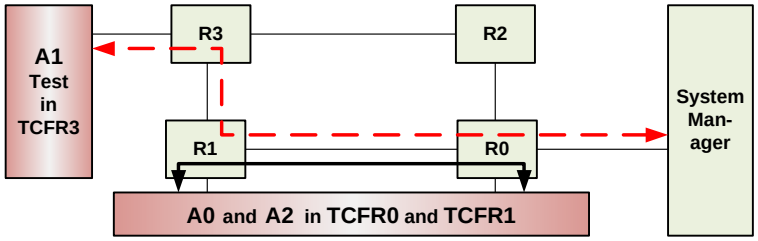


Figure 7.3: Applications in Different TCFRs.

Non-Intrusive Behavior Analysis

To evaluate the non-intrusive nature of our online test methodology, we *analyse the system in different modes*. This means that while testing the system goes through different operations, i.e., execution, configuration, programming, and use-case transitions. For this purpose, we (a) *start the test of A1* while A0 is executing, and (b) *start loading of A2* (at the end of A0) while A1 is testing. By doing so, we evaluate (i) the impact of (if any) dynamic configuration (A2) on the online testing (A1), and (ii) the ability to test and configure at the same time. Figure 7.4A shows the timing details of test, load, and execution operations of the system applications in different use-cases.

We analyse the behavior of our methodology for above two scenarios, and *show a small time window of 300 μs* as illustrated with Figure 7.4B. The time interval shows *the presence of all the three operations* of different applications, i.e., execute (A0), test (A1), and load (A2). Important observations can be drawn from the above scenarios. For instance, as shown in Figure 7.4A, the *loading of A2 triggers the use-case transition*, which should be transparent to

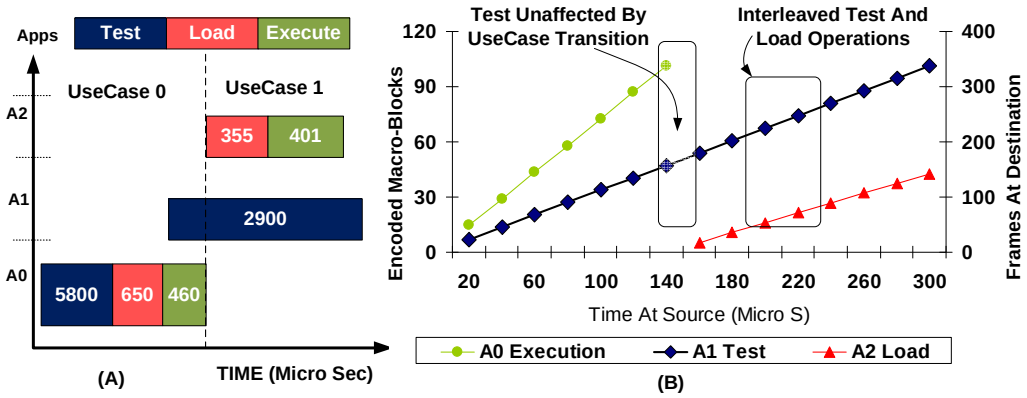


Figure 7.4: (A) Application Timing Details in Different Phases, and (B) Interleaved Test and Load for A1 and A2. MB encoded for A0, frames arrived for A1 and A2.

the ongoing test process of A1. In Figure 7.4B, the first rectangle highlights the unaffected behavior of A1 test, while the use-case transition is made due to the A2 invocation. Importantly, as shown in Figure 7.4B, our methodology supports *the interleaved test and load operations* of multiple applications. It is important that *A1 test and A0 execution both share a network path (R0-R1)*, as shown in Figure 7.3. The shared path can lead to intrusiveness in between the test and execution operations. However, the composable HWNOC takes care of this.

7.3.2 Performance: Fault Detection Latency

We determine the fault detection latency of the whole FPGA, which contains 1392 MTCRs (i.e. equivalent to Virtex-4 XC4VLX200). For experiment purpose, we divide the FPGA into 4 TCFRs of 348 MTCRs each. First, we determine the fault detection latency of a TCFR, and then scale the calculations to the whole FPGA chip.

The following discussion illustrates the latency to detect a fault in a TCFR that could belong to any application. The process to detect a faults in a TCFR is explained earlier in Section 7.1.1, and initiates with programming the test connection in 21 μs. Afterwards, the writing of the test bitstream, which consists of 8004 frames (348 MTCR x 23 frames / MTCR), starts.

Each HWNOC link can transport data at a maximum of 16 Gb/s. However, to test a TCFR, we reserve approx. 10% of link resources of the HWNoC

architecture[4]. This means, test data is allocated a maximum of 16 x 0.1 = 1.6 Gb/s on a link, out of which we assume a 30% packet overhead. In other words each link provides an effective bandwidth of 1.1 Gb/s to transport test data. A TCFR requires 10.5 Mb (i.e., 8004 frames x 41 words / frames x 32 bits / word). This means that it takes 9.5 ms or 9500 μs to write a test bitstream in a TCFR of 348 MTCRs.

Read-back process is the inverse of the writing process. It takes approximately 9500 μs to read-back the complete test bitstream. It is important that the evaluation process is performed on a frame-wise basis, and in parallel with the read-back process. The system manager, after receiving a test bitstream frame, evaluates a test frame for the possible stuck-at faults. However, the evaluation process does not add to the fault detection latency, because it is performed in parallel with the read-back process. This means, while reading the next frame, the previous frame is evaluated for the possible faults.

In short the total fault detection latency of a single test bitstream, for a TCFR of 348 MTCRs, equals to 2.5 + 9500 + 9500 = 19002.5 μs. With two test bitstreams, the worst case fault detection latency of a TCFR is approx. 2 x 19002 = 38005 μs or approximately 38 ms. For an application that uses the complete FPGA chip, which consists of 4 TCFRs of 348 MTCRs each, the worst-case fault detection latency accounts to 4 x 38005 = 152020 μs or 152 ms.

Next, we discuss the cost in terms of spatiotemporal overhead to test a TCFR in the current FPGA architecture.

7.3.3 Spatiotemporal Cost

We use a test connection to write and read-back the test data. Therefore, the resources acquired by the connection are accounted as the spatiotemporal overheads of our methodology. The temporal overhead of the test connection is found to be approx. 21 μs, which is the time required to setup a connection.

The spatial overhead of a test connection, between the source SM and destination TCFR, accounts for a number of hardware resources. (1) Two NI-Shells to serialise and deserialise the test frame in between the SM and the destination TCFR. (2) Two 41 words FIFOs at the NI kernels of the system manager and destination TCFR to send and receive a test frame, respectively. (3) Additionally, the FIFOs (3 words deep) of each router that are used during the con-

[4]Reserving more than 10% of resources for test purpose can significantly affect the success rate of applications to FPGA binding. For details see Chapter 6.

Table 7.2: Cost Evaluated for **the Complete** FPGA after Varying TCFR Area

TCFR Area (MTCRs)	**Temporal Overhead** μs	**Spatial Overhead** *CLB Equivalent*					
		NISh	NIK	Routers	TCFR	**Overall**	**% FPGA**
348	84	4	68	4	51	127	0.57
174	168	7	123	8	51	189	0.85
88	336	13	232	16	51	312	1.4
44	672	25	451	32	51	559	2.5

nection time also contribute towards the spatial overhead. Note that the router resources are reserved only 10% of the time as test connection is allocated 10% of time-slots. The test connection, at worst-case, can utilize the FIFOs of all the 4 routers of the HWNOC architecture. The *connection resources are hardwired*, and to obtain the (ASIC) area numbers, we refer to [84].

The authors in [84], illustrate that an ASIC implementation can take approximately 35 times lower area than its equivalent FPGA implementation. Therefore, we first synthesis (on Virtex-4 XC4VLX200 chip and by using Xilinx ISE 10.1) the connection resources, then reduce these by (conservative) 30 times to obtain the ASIC equivalents. The synthesis numbers for each component are: (1) 23 CLBs for an NI-Shell, (2) 390 CLBs for a 41-word FIFO, and (3) 29 CLBs for a 3-word FIFO. The total spatial overhead of a synthesised test connection then accounts to 2 x 23 + 2 x 390 + 4 x 29 = 952 CLBs. After hardwiring the connection resources, *the actual spatial overhead to test a TCFR is approximately 32 CLB equivalent*.

7.3.4 TCFR Area Impact on Performance & Cost

In this section, we analyse the impact of TCFR area on the performance and cost of our test methodology. For this purpose, we select 4 different archtiectures of the FPGA, each of which has a different TCFR area as shown in Figure 7.5. For instance T-174 in Figure 7.5 represents an FPGA architecture with 8 TCFRs of 174 MTCRs each. Figure 7.5 also shows that the worst-case fault detection latency per TCFR decreases from approximately 38 ms to 4.8 ms when TCFR area is varied from 348 MTCRs to 44 MTCRs respectively.

The high performance, in terms of reduced fault-detection latency, induces high costs in terms of spatiotemporal overheads. The second and third columns

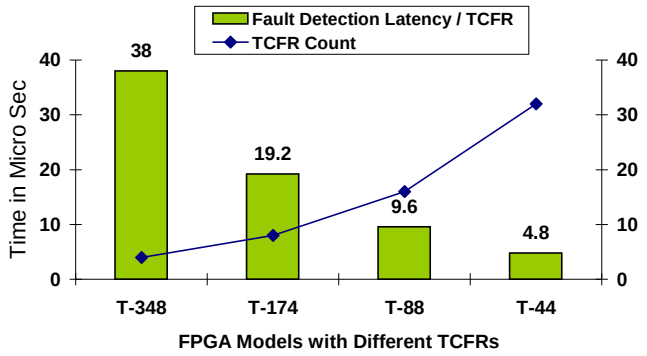


Figure 7.5: Different FPGA Architectures with Variable TCFR Area and Count. Also Showing Fault Detection Latency Per TCFR.

of Table 7.2 illustrate the temporal and spatial overheads of each FPGA architecture, respectively. For instance, the spatial overhead of T-174 FPGA architecture is found to be 138 CLB equivalent. In T-174, the test connection can access any of the 8 TCFRs. Therefore, the worst-case spatial overhead is obtained by accounting the area costs of all the 8 NI-Shell, 41 words FIFOs (one in each NI kernel), and 3 words FIFOs in the routers. In addition, the area cost of hardware registers, which are used for input and output of test IP CLB chains, in a TCFR is also included. The area cost of hardwired register is found to be 51 CLB equivalent.

For the whole FPGA, decreasing the TCFR area from 348 MTCRs to 44 MTCRs, the temporal overhead increases from approx. 84 to 672 μs, whereas spatial overhead increases from approximately 76 to 508 CLB equivalent.

7.3.5 Comparison with the State of the Art

We compare the performance and cost of our methodology with the existing state-of-the-arts [37, 142]. The comparison is made for the compile and run time phases, and as a *fairness of the comparison*, we use the same platform of Virtex-4 for each of these schemes.

It is important that the range of faults that we cover in the paper closely matches with the ones that authors in [37, 142] have addressed. For instance, the work of [37] performs the online test at the granularity of a single CLB. Similarly, the authors in [142] test CLBs in group of 4. The nature of faults that are covered in [37, 142] is stuck-at faults in the sequential and combinational logic of CLBs. In comparison, we perform the test at the granularity

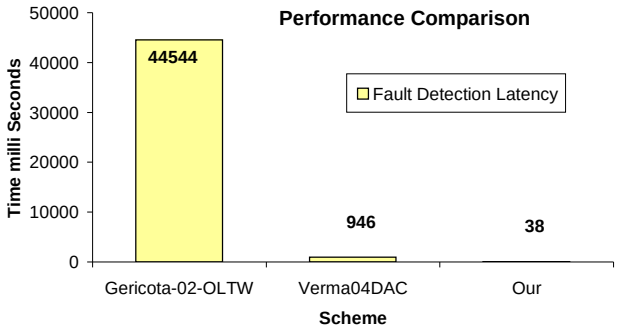


Figure 7.6: Per TCFR: Fault Detection Latency (*mili sec*).

of a TCFR, i.e., 512 CLBs or 32 MCRs. However, we test the combinational part of CLBs (i.e., LUTs), but in addition we perform the test for configuration memory cells as well.

At the same time it is important to mention that our test scheme ensures a non-intrusive nature of online testing with respect to already running applications. In comparison, though the approach of [37] also ensures un-disrupted application execution by using active replication method, but no such claim is found for the other state-of-the-art, i.e. [142].

Run Time (Fault Detection Latency)

The authors in [37] first replicate a CLB, which is to be tested, on an earlier tested CLB. The test-access-mechanism runs at 20 MHz to replicate and test a CLB in 24000 μs. However, *as a fairness of the comparison*, we use the same Virtex-4 features for [37], which runs BSI TAM at 66 MHz. Therefore in [37], the fault detection latency of a single CLB reduces from 24000 μs to 8000 μs. On a scaled level of TCFR with 348 MTCRs (i.e., 5568 CLBs), the fault detection latency of the scheme [37] is approximately 5568 x 8000 = 44544000 μs or 44544 ms, as shown in Figure 7.6.

The scheme [142], on the contrary, tests 4 CLBs at one time and uses 6 test sessions to test the CLBs. In each test session 2 out of 4 CLBs serve as test-stimuli and response analysis blocks. This means, to find the fault in the set of 4 CLBs, the authors configure 4 x 6 = 24 CLBs. We calculate the optimistic fault detection latency by simply accounting the time to configure the CLBs. For this purpose, we use equation 7.1 that *gives 59 words or 236 bytes of data to configure 1 CLB*. This means 236 x 24 = 5664 bytes are required to configure 24 CLBs. The one-bit wide BSI TAM, running at 66 MHz, takes

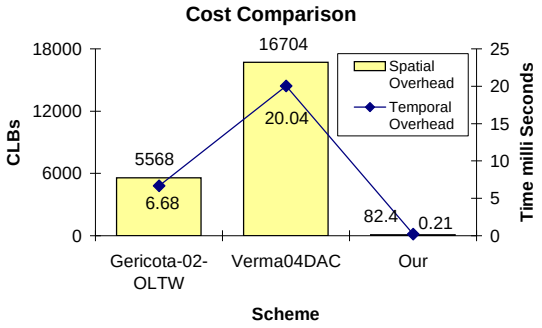


Figure 7.7: Per TCFR: Spatiotemporal Overheads.

approximately 680 μs to detect a fault in the set of 4 CLBs. On a scaled level of TCFR with 348 MTCRs, the fault detection latency of the scheme [37] accounts to 680 x (5568 / 4) = 946560 μs or 946.5 ms.

Similarly, we obtain the fault detection latencies of [37] and [142] for the smallest TCFR area that we have used for our architecture, i.e., TCFR with 44 MTCRs. The fault detection latencies for [37] and [142] are found to be approximtely 5568 ms and 118.3 ms respectively.

Run Time (Spatiotemporal Overheads)

The work of [37] requires 1 additional CLB to replicate the CLB that is going to be tested. This accounts for 100% spatial overhead, i.e., additional 5568 CLBs are required to test a TCFR of 348 MTCRs. The one-bit wide BSI TAM, running at 66 MHz, takes 28.3 μs to configure 236 bytes of CLB. The temporal overhead, i.e., time to configure the spatial overhead of 5568 CLB is 5568 x 28.3 = 157574.4 μs or 157.6 ms, as shown in Figure 7.7.

The scheme [142], on the contrary, tests 4 CLBs at one time and uses 6 test sessions for their verification. Each test incurs a spatial overhead of 2 CLBs, because 2 out of 4 CLBs serve as test-stimuli and response analysis blocks. This means, an overhead of 12 CLBs is incurred in 6 test sessions to test 4 CLBs. This is 3 times the CLBs that are going to be tested. On a scaled level, i.e., for the TCFR of 348 MTCRs, the total spatial overhead then accounts to 3 x 5568 = 16704 CLBs. The temporal overhead is, therefore, approx. 16704 x 28.3 = 472723.2 μs or 472.7 ms, as shown in Figure 7.6.

Similarly, we obtain the spatiotemporal overheads of [37] and [142] for the smallest TCFR area that we have used for our architecture, i.e., TCFR with 44

MTCRs. For scheme [37], the spatial overhead is 704 CLBs and the temporal overhead is approximately 19.7 ms. For the scheme [142], the spatial overhead is 2088 CLBs and the temporal overhead is 59 ms.

Compile Time (Impact on User Application)

From a compile time perspective, we can qualitatively compare our methodology with [37, 142] in a sense, that an additional time is required to reserve and allocate the test resources across the HWNoC. In addition, as explained earlier in Section 7.3.2, we reserved approx. 10% of our HWNoC resources to conduct the online test. Therefore, a user application would be allocated from the remaining 90% resources.

7.4 Conclusion

We presented an online test methodology for FPGAs that used a HWNoC as the test access mechanism. Our online test scheme ensured the non-intrusive behavior by: (a) invoking test at application startup time, (b) allowing undisrupted execution for already existing applications, and (c) not restricting the parallel operations of dynamic reconfiguration and test for multiple applications. To achieve the objectives, our methodology took into account the design, compile, and run time phases.

We analysed the performance and cost of our test methodology for different test functional regions (TCFRs) of an FPGA architecture. The largest TCFR area was 348 MTCRs (5568 CLBs) and the smallest one was with 44 MTCRs (704 CLBs). Our methodology detected faults in the largest TCFR in 38 ms at the cost of temporal overhead of 0.021 ms and spatial overhead of 82.4 CLBs. Similarly, for the smallest TCFR, the fault detection latency was found to be 4.8 ms and at the cost of temporal overhead of 0.21 ms and spatial overhead of 65 CLBs.

We then compared the performance and cost of our scheme with two other schemes [37] and [142] for the same set of TCFRs. *For the smallest TCFR* (44 MTCRs), the fault detection latency of [37] was approximately 5568 ms with the spatiotemporal overheads of 5568 CLBs and 157.6 ms respectively. Similarly, for [142], the fault detection latency was approximately 118.3 ms and at the spatial cost of 2088 CLBs and temporal cost of 59 ms. Our scheme, therefore, in comparison to [142] possess approximately 24.8 times better fault

detection latency. Moreover, it comes at lower spatiotemporal costs which are found to be 67 and 280 times lower, respectively.

8

H.264 Encoder Case Study

We selected the state of the art H.264 encoder as the case study. Through this case study, we provide the complete picture of our system, i.e., (1) design time specifications to (2) compile time binding to (3) run time (composable and persistent-state) dynamic reconfiguration of H.264 on the target FPGA architecture.

We start with the design time specifications that include application (H.264) and architecture (our FPGA model) specifications, see Section 8.1. Afterwards, we explain the H.264 binding process on the given FPGA platform, see Section 8.2. It is followed by illustrating the run time composable and persistent-state reconfiguration process (for a simplified H.264 encoder), see Section 8.3. In Section 8.4, we provide the concluding remarks for the chapter.

8.1 Design Time Specifications

In this section, we provide application and architecture specifications that are used for the case study.

8.1.1 H.264 Specifications

The task graph of H.264 encoder application (excluding CAVLC) is shown in Figure 8.1. We synthesized RES, (I)DCT, (I)QT, and Reconstruction blocks on a Virtex-4 XC4VLX200 chip using Xilinx ISE 10.1, whereas the area of remaining IPs is estimated from [127]. Table 8.1 illustrates the area and frequencies for H.264 IPs. The Column 2 of Table 8.1 lists the area in k LUTs, whereas the Column 3 of Table 8.1, shows the area in terms of MTCRs. The throughput requirements as shown in Figure 8.1 were calculated for Quarter

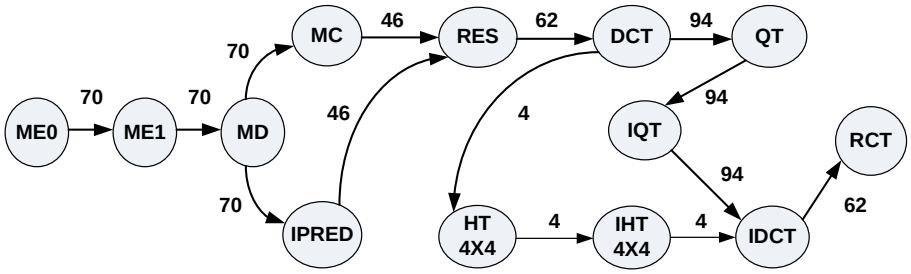


Figure 8.1: H.264 Task Graph with Communication Demands.

Table 8.1: Application IP Synthesized Area, Frequency and Reconfiguration Time

IP	Area (k *LUTs*)	Area (*MCR*)
ME	7.8	60
IPRED	1.4	11
MC	2.2	17
RES	1.6	13
(I)DCT	2.3	**19**
(I)Quant	2.2	**18**
(I)HT	2.3	**18**
RCT	1.6	**13**

Common Intermediate Format (QCIF) resolution video frames.

8.1.2 FPGA Specifications

To perform H.264 binding, our selected FPGA model comprises 9 TCFRs, that are attached in 3x3 dimensions. Each TCFR consists of 4 k LUTs or 32 minimum test configuration regions (MTCRs). Figure 8.2 illustrates that there are two links for a TCFR pair, and each link consists of 10 time-multiplexed slots. We assume that at the time of H.264 application binding all the resources of target FPGA are available, i.e., our application is the first application that is going to execute on FPGA platform.

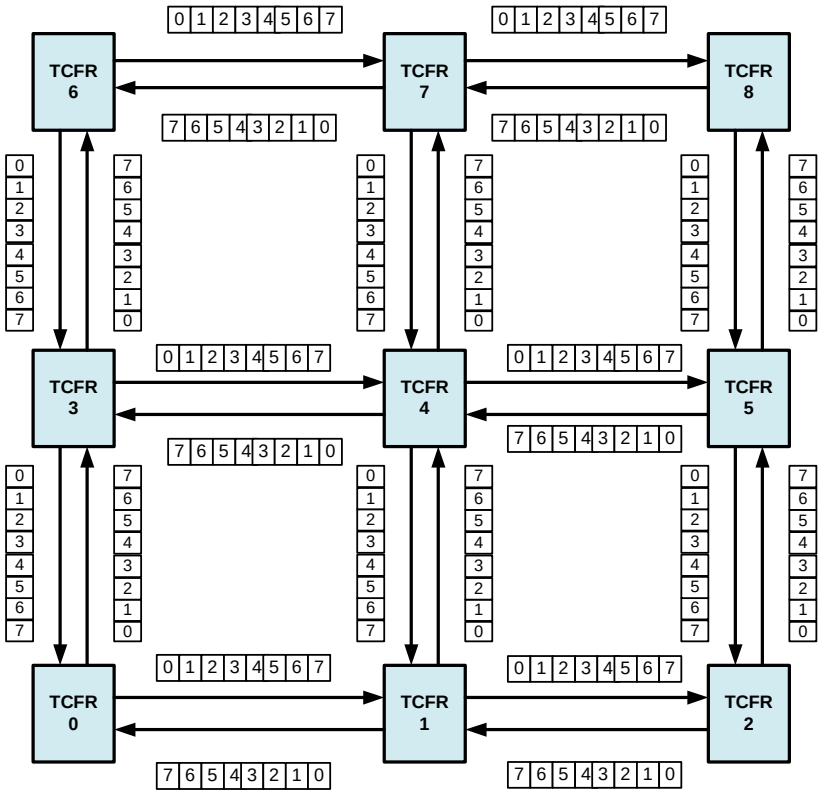


Figure 8.2: Specification of the Target FPGA Architecture.

8.2 Compile Time Binding of H.264 to FPGA

In the following sections we provide the details about the binding of H.264 on the target FPGA architecture.

8.2.1 Cluster Creation

The binding of H.264 encoder on the target architecture was performed after creating its clusters. Our PUMA scheme created 9 clusters for the input H.264 application, see Figure 8.3. The H.264 IPs have different types of inter-IP dependencies, i.e., an IP input is dependent on a single but another IP output (we call it one-to-one), etc. This is shown in Figure 8.3, where cluster 0 shows one-to-one inter-IP communication, cluster 2 shows one-to-many inter-IP communication, cluster 3 shows many-to-one inter-IP communication, and

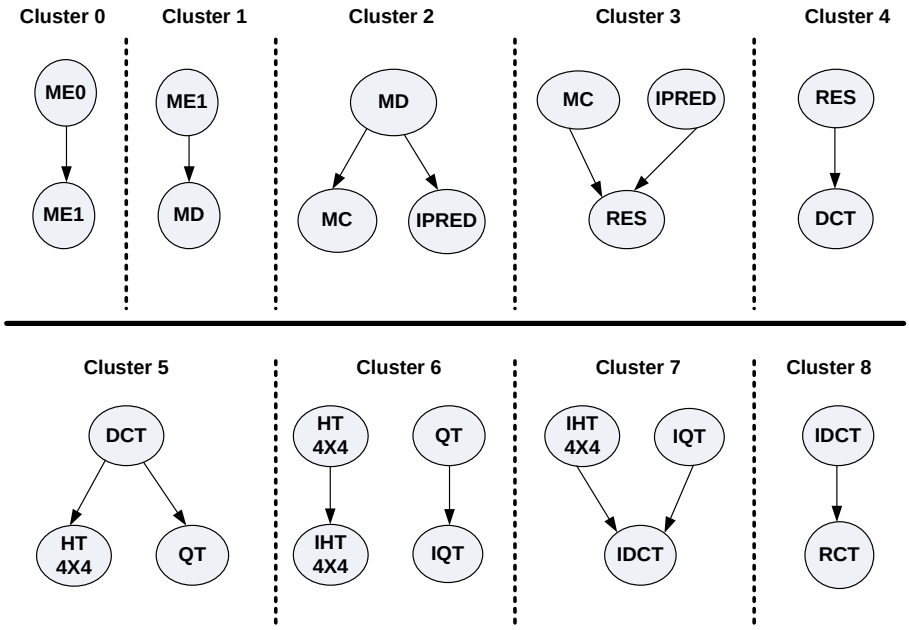


Figure 8.3: Showing H264 Clusters Created by using PUMA.

cluster 6 shows multiple one-to-one inter-IP communication.

8.2.2 QoS Ensured Cluster Binding

Figure 8.4 shows the resultant binding of all the clusters (of H.264) on the target FPGA architecture. In Figure 8.4, the necessary links and their allocated resources are shown. As an example, the connection between MD and MC, requires 2 slots, and traverses the path that consists of TCFR2, TCFR1, and TCFR4. In Figure 8.4, the respective time-slots for the MD-MC connection are shown, i.e., slots 3,4 on TCFR2-TCFR1 link, and slots 4, 5 on TCFR1-TCFR4 link. Similarly, the TCFR and network resources are assigned. The slots are allocated in a pipelined fashion, across the links of a connection. Additionally, as shown in the Figure 8.4, the slots are allocated in such a way that these are used to transport contention-free data over the network.

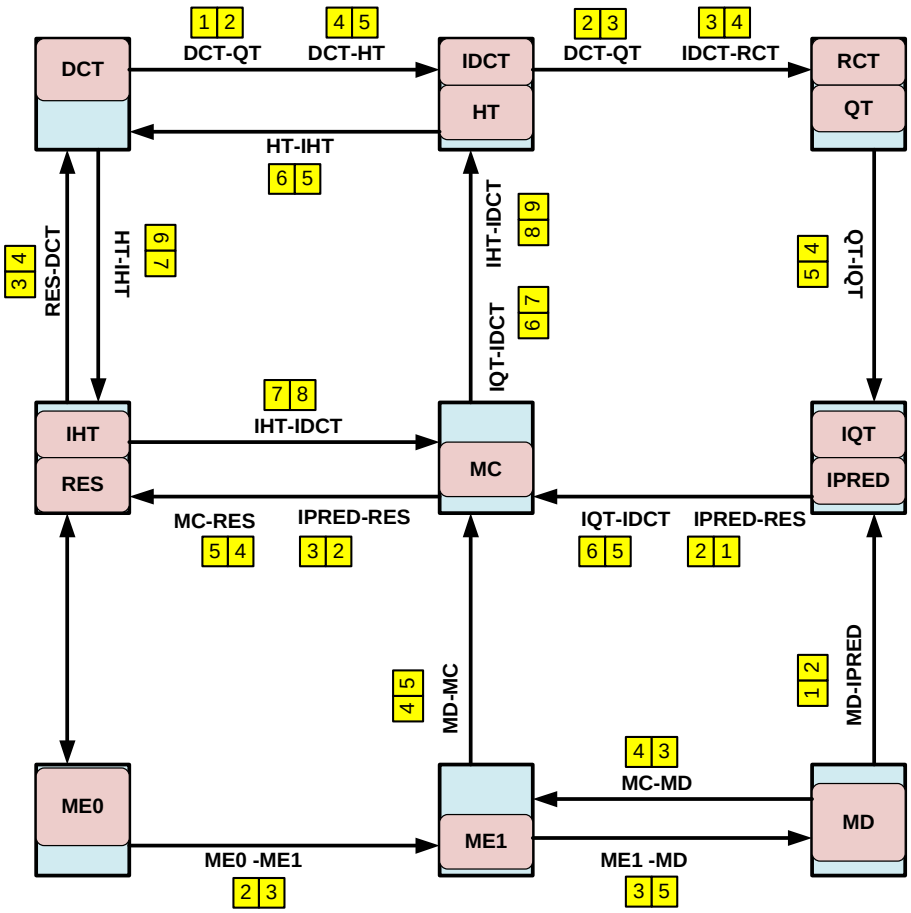


Figure 8.4: Compile Time Binding of H.264 to the Target FPGA Architecture.

8.2.3 Cost of QoS Guarantees

After performing the successful binding for our case study application. We evaluated the PUMA cost that was paid to fulfill the QoS guarantees for our H.264 video encoder application. The reason is that ensuring QoS guarantees is important but it should not be at the cost of too high resource reservation. As shown in Figure 8.5, our PUMA fulfills the QoS requirements for each of the H.264 connection. Importantly, the allocated throughput is kept around 1.2 times of the required ones. The allocation is always more than the asked bandwidth by application only due to distribution TDM slots. However, the allocated slots remain within a reasonable 20% extra of the required one. More-

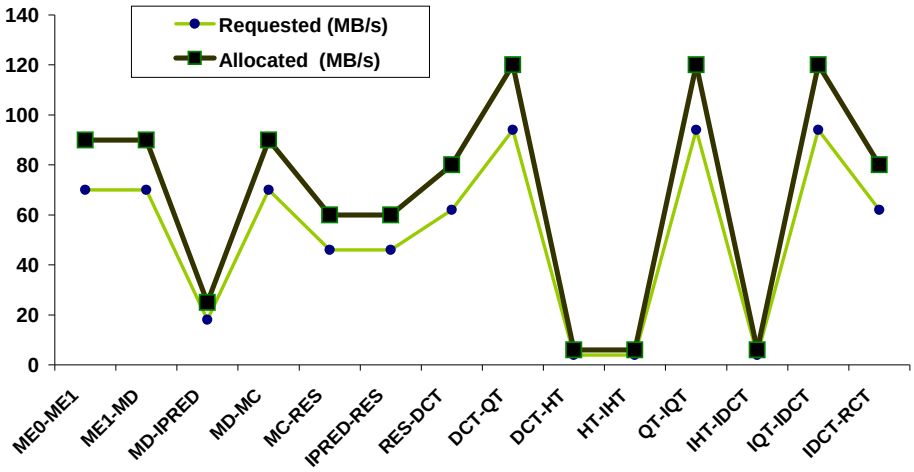


Figure 8.5: Showing Communication Cost that is Paid for the H.264 Binding.

over, the average hop count among the communicating IPs is fairly reasonable as shown in Figure 8.6.

In the next section, we provide we perform run time dynamic reconfiguration for H.264 application IPs, and evaluate the composable and persistent-state behavior of the system.

8.3 Run Time H.264 Dynamic Reconfiguration

We exercise the dynamic reconfiguration in SystemC using the 3-tier reconfiguration model, as explained in Chapter 6. Additionally, the dynamic reconfiguration is performed by using the proposed design flow of Section 3.1. To evaluate composable and persistent-state dynamic run-time reconfiguration, we used a system with two applications that constitute the IPs of *H.264*[1]. For this purpose, three IPs (Residue, DCT, and Quantiser) were used. Our implementation not only take into account the communication requirement of applications, (e.g.. QCIF resolution for our case study), but the computational requirements as well. For calculating the computational requirements, we synthesise the VHDL implementations of the *Residue*, *DCT*, and *Quantiser* IPs on a Virtex-4 XC4VLX200 chip using Xilinx ISE 10.1 provide their MHz frequencies, which were used in SystemC, Table 8.1. The bitstream size for all

[1]Note that in this case the cost of implementing dynamic reconfiguration by using our 3-tier reconfiguration model can come in the form of an application manager per application.

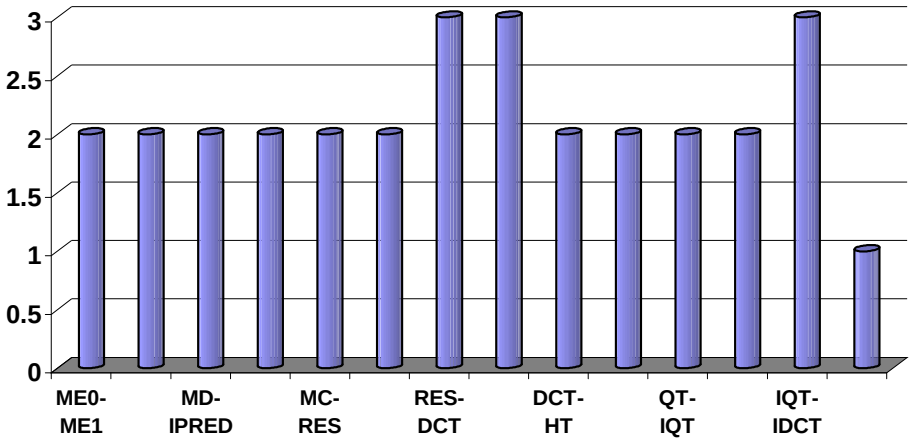


Figure 8.6: Showing Hop Count between the IP that Communicate with Each Other.

Table 8.2: Application IP Frequency and Reconfiguration Time

IP	Frequency *(MHz)*	Bitstream *(Frames)*	(Re)config Time *(μs)*
RESIDUE	**100**	**285**	**273.6**
DCT	**66**	**396**	**380.16**
Quantizer	**75**	**370**	**355.2**

the three IPs is estimated from their kLUT areas using Equation 8.1.

$$(IP\ LUTs * frames\ per\ column)/(LUTs\ per\ CLB * CLB\ per\ column) \tag{8.1}$$

The two applications that consists of H.264 IPs are *A1* (Residue + DCT +Quant), and *A2* (DCT only). Additionally, both the application execute in parallel. However, it is assumed that A1 does not meet the required area constraints, and is therefore sub-divided into two SubApps (*SA1* and *SA2*), which are configured and executed in separate use-cases. In addition, each use-case comprises the system manager and an application manager per application.

8.3.1 Temporal Analysis of Application Binding

In this section, we provide the temporal analysis for the application A2, which comprises two sub-applications (*SA1* and *SA2*). In Figure 8.7 both SA1 and

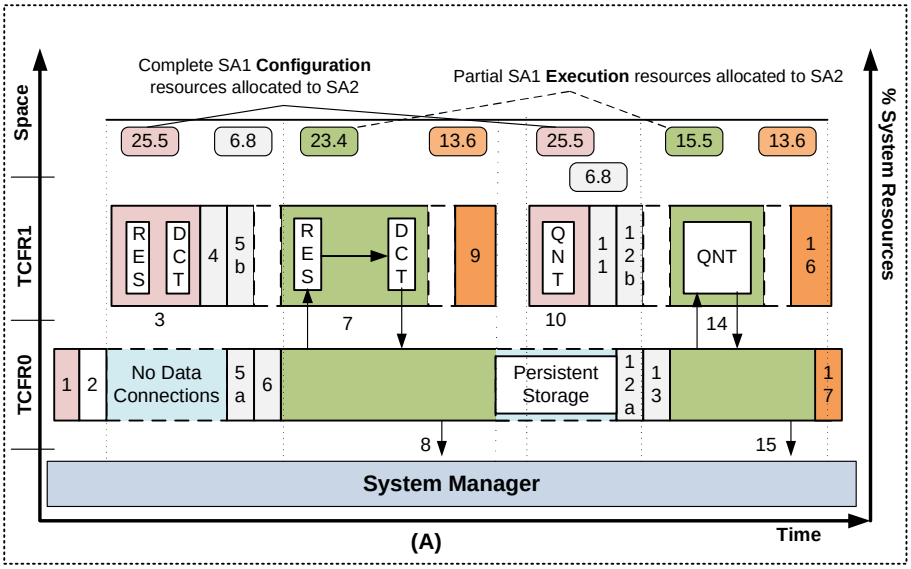
SA2 go through load, program, execute, and stop phases, as shown with steps (3, 5, 7 , 9) for SA1 and with steps (10, 12, 14, 16) for SA2 respectively. Each phase is preceded by programming the NoC, as illustrated in [46]. It takes approx. $2.5\,\mu s$, to program an NI. This additional phase delay has been included in the preceding discussions.

SA1 comprises 681 frames, where each frame accounts for 41 words. The system manager uses a fixed and low latency connection to load the SA1 bitstream in 653 μs, as shown in Figure 8.7 (step 3). Figure 8.8 shows the fixed latency bitstream transportation from the control processor to destination TCFR. To load the bitstream for SA1 IPs, the system manager follows the procedure of Section 6.2.3. After loading the SA1 IP bitstreams, the system manager initializes the IPs in 0.63 μs by programming memory mapped reset and clock generator in the destination TCFR, as shown in Figure 8.7 (step 4). It is followed by setting-up of three data connections for the SA1 IPs in approximately 2.9 μs, Figure 8.7 (step 5a, 5b). Once data connections are programmed and SA1 IPs are ready to receive input data, the system manager programs the respective application manager. It takes 0.97 μs to program the respective application manager with application parameters, which include full application's I/O addresses and ranges, Figure 8.7 (step 6). Afterwards, the SA1 executes to process 1 QCIF (99 Macro Blocks) video frame. The SA1 IPs process one 4x4 pixel block in a single execution, where 16 such 4x4 pixel blocks constitute a Macro Block (MB). This means, the SA1 IPs execute for 99 x 16 = 1584 times to process a single QCIF frame, which takes 460.3 μs in peer to peer streaming communication fashion, Figure 8.7 (step 7).

Sub-application 2 goes through the same phases but after achieving a persistent-state intra-application swapping, as shown in Figure 8.7 (step 8 to step 16). The persistent-state procedure by using which sub-application 2 is swapped is explained in the next section.

8.3.2 Persistent State Intra-Application

To achieve the persistent-state transition from SA1 to SA2 the application manager of SA1, notifies the system manager about that the SA1 has completed its execution, and needs to be reconfigured. Afterwards, the system manage shuts down the SA1 data connections from processing further input data, but after confirming that there are no ongoing transactions that are using those data connections. The clocks of SA1 IPs are lowered down as well, so as to prevent the IPs to compute data. In short the persistent-state transitional delay involve the factors, as mentioned in equation 8.2.



Application Manager and Initialization Timings

Steps	Details	Time (μ sec)
2, 4, 11	IP Initialization In TCFRs	0.63
6, 13	Application Manager Programming	0.97
8, 15	Application Manager Notification	0.24
17	Application Manager Shutdown	1.25

Steps	Details	Time (μ sec)
3	Bitstream Loading	653
5a, 5b	Data Connection Setup	2.9
7	Execution	460.3
9	Shutdown	5.6

Sub-Application 1

(B)

Steps	Details	Time (μ sec)
10	Bitstream Loading	355
12a, 12b	Data Connection Setup	1.02
14	Execution	401.5
16	Shutdown	3.9

Sub-Application 2

Figure 8.7: Showing: Temporal Analysis for SA1 and SA2

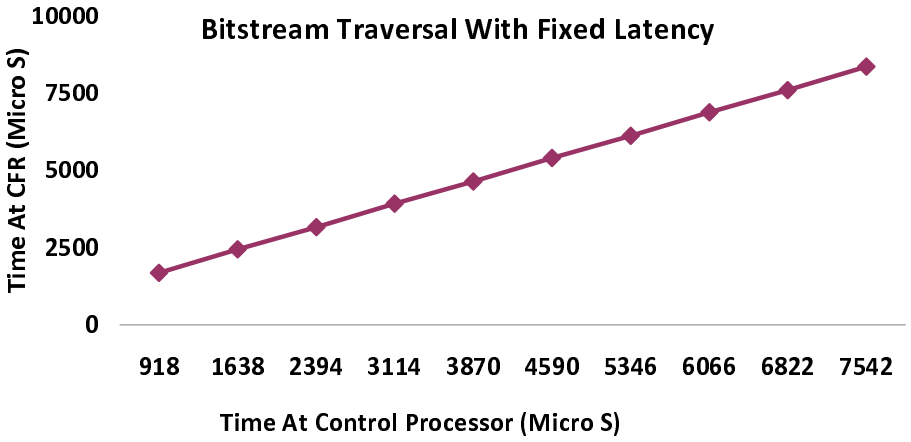


Figure 8.8: Bitstream Loading with Fixed Latency with Departure Time at Control Processor (X-axis) and Arrival Time at TCFR (Y-axis).

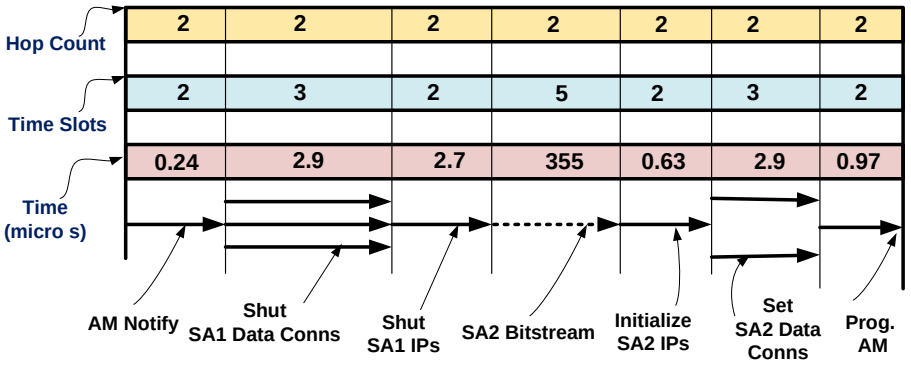


Figure 8.9: Persistent State Intra-Application Analysis.

$$T_{AM_{Notify}} + T_{SM_{Resp}} + \#ConnsxT_{Conn_{Shut}} + \#CFRsxT_{CFR_{Shut}} \quad (8.2)$$

To implement the above persistent-state transition, the application manager of SA1 notifies the system manager in 0.24 μs on an already established connection, Figure 8.7 (step 8). For our case, the system manager response time is negligible, because it is sitting idle and ready to serve at the time of an application manager notification. Achieving persistent-state is advanced by tearing down all the three SA1 connections in 2.9 μs, and followed by disabling the SA1 IPs in 2.7 μs. Figure 8.9 illustrates the resources reserved during safe switch from SA1 to SA2 with a transitional delay of 5.84 μs.

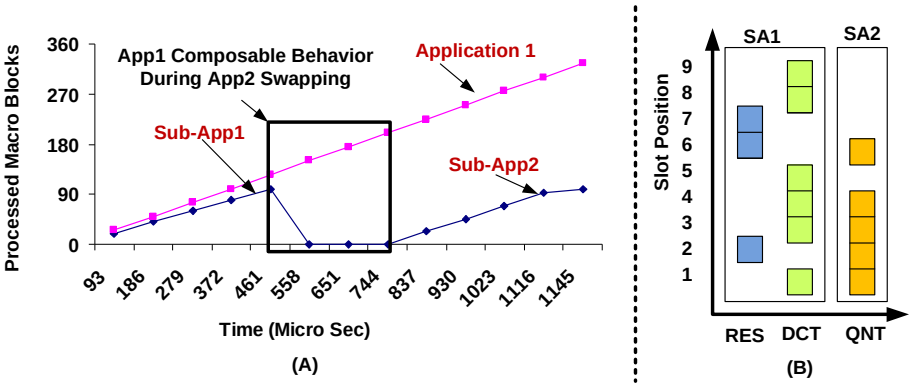


Figure 8.10: Showing: (A) Composable Inter-Application Reconfiguration, (B) Allocated Time Slots

8.3.3 Composable Inter-Application

This section provides the evidence of composable inter-application behavior by using the 3-tier reconfiguration model, see Figure 8.10A. In Figure 8.10A, an application1 execution is not affected when an existing application is: a) stopped (SA1 after processing the 99 MBs), b) or configured (SA2). This is achieved by our PUMA scheme [147], which reserves TDM slots over the communication channels in such a way that no two channel contend for network resources at the same point in time. The properly allocated TDM slots therefore, help in transporting interleaved yet non-interfering traffics for bitstream loading and execution over the same network path. The resources, which are freed after the removal of an application (SA1) are reallocated fully or partially Figure 8.7, as per required to the incoming application (SA2). During resource reallocation, the system manager takes into account the magnitude and positioning of the resources. For instance in Figure 8.10B, Quantiser is allocated with the resources that are released by Residue and DCT IP cores.

8.4 Conclusions

In this chapter, we expounded the binding of the case study application (H.264 video encoder) on the target FPGA platform. As a first step application and architecture specifications were provided. The application specifications included: a) H.264 task graph with communication demands for Common Intermediate Format resolution, b) area values of H.264 IPs (excluding CAVLC),

that were either synthesised or taken from [127]. The target FPGA architecture specifications included: a) the number and dimensions of FPGA-nodes, b) and the residual resources that are available at and across the FPGA-nodes. Afterwards, the binding of H.264 on the target FPGA was performed by using PUMA scheme. Importantly, PUMA ensured that the QoS constraints of H.264 are fulfilled as a result of the successful binding. At the same time, PUMA ensured that the cost of QoS guarantees should not exceed a reasonable value.

At run time, we evaluated the composable and persistent-state behavior of the system while the IPs of H.264 are dynamically reconfigured. To evaluate the composable and persistent-state reconfiguration, two multiple applications (i.e., A1 and A2) were created from the simplified H.264 vide encoder. We provided the temporal analysis for one of the application 2 (i.e., A2), which comprised two sub-applications (*SA1* and *SA2*). Both the SA1 and SA2 went through load, program, execute, and stop phases. The persistent-state between SA1 and SA2, i.e., during intra-application reconfiguration was achieved in approximately 5.84 μs. On the other hand the composable inter-application reconfiguration was implemented by reallocating the resources (TDM slots and CFRs) of SA1 to SA2.

9 Conclusions

In this chapter we first provide the summary of the thesis in Section 9.1. Afterwards, we list the contributions of the thesis in Section 9.2. Lastly, we discuss the open issues and future directions in Section 9.3.

9.1 Thesis Summary

The summary of the thesis can be summarized as follows.

In *Chapter 1* described the the trends of SoCs from architecture and application points of view. We discussed the problems that have emerged due to the trends, and elaborated the key requirements to overcome the problems. We then presented the techniques to fulfill the requirements. In the end we gave the problem statement that has been addressed in the thesis.

In *Chapter 2*, we provided the preliminary background information about the architecture and design flow of FPGAs and networks on chip.

In *Chapter 3*, we explained the architecture and the design flow of the proposed solution. We then explained the techniques (i.e., HWNoC, application to FPGA binding, composable and persistent-state dynamic run time reconfiguration, and online testing) that make up the proposed solution to fulfill the requirements of Chapter 1. For each technique, we provided the overview, motivation, related work, and positioning with respect to the state of the art.

In *Chapter 4*, we explained the architecture of the proposed FPGA that comprises HWNoC, test configuration functional regions (TCFRs), and a control processor. Initially, we provide the overview of the proposed architecture. Afterwards, we explained the architecture of the hardwired NoC that consists of local buses, network interfaces, and routers. A HWNoC is used to transport the unified (test, configuration, functional, and control) data in between the

test configuration functional regions (TCFRs). In other words, applications are tested, configured, and execute on the TCFRs, and a hardwired NoC provides a unified place to transport data between the TCFRs. The architecture of a TCFR consists of a number of components that include minimum test configuration regions (MTCRs), Bus Macros, clock domain crossing FIFOs, a local bitstream manager, and a clock / reset manager. We also explained the architecture of the control processor that uses the HWNOC to transport the unified data to any TCFR in the proposed FPGA architecture. Thereafter, we discussed which parts of the proposed FPGA should be *hard* and which should be *soft*. We then explored the possible extensions of a hardwired NoC in our proposed FPGA architecture. In the end we provided the results and analysis for the proposed architecture.

In *Chapter 5*, we presented a PUMA scheme to bind applications on the target FPGA architecture. The PUMA scheme unified all the three processes of placement, mapping, and allocation while binding application. Moreover, the proposed PUMA scheme ensured the QoS guarantees, whenever an application binding was successful. We presented the mechanism to perform the unification. The PUMA flow performed an application binding after creating multiple clusters out of it, where each cluster represented inter-communication dependencies across a group of IPs. PUMA limitations were provided afterwards. Thereafter, we presented the results and evaluations of the proposed PUMA scheme, where the performance and scalability of our PUMA scheme was evaluated for multiple combinations of applications and target FPGA architecture.

In *Chapter 6*, we described the run time procedures to configure, program, and run soft IPs. This basic infrastructure was then used to dynamically start and stop entire applications, and also sub-applications. The latter was useful when an application did not fit in the resources (configuration regions) allocated to it. Dynamically swapping sub-applications then enabled the entire application to execute anyway, although at a lower speed. Our platform was composable, which means that starting and stopping of applications did not affect concurrently operating applications (and vice versa). We modeled the platform in cycle-accurate transaction-level SystemC, together with soft IP blocks. In particular, we modeled bitstream loading and frame placement of soft IPs, soft IP clock management and reset, the programming of HWNOC and IPs, and their functional operation. We simulated the concurrent configuration and execution of two small applications, one of which was split in two sub-applications. We compared the performance of a conventional FPGA with a soft NOC and dedicated configuration interconnect with our HWNOC platform. Bitstream

loading was faster in our platform. In the end, we compared the performance of a conventional FPGA with a soft NOC and dedicated configuration interconnect with our HWNOC platform.

In *Chapter 7*, we presented an online test scheme for FPGAs that used a HWNoC as test access mechanism. Our online test scheme ensured a non-intrusive behavior to other communication traffic. In particular the proposed online test scheme ensured a non-intrusive behavior by: (a) invoking the test at an application startup time, (b) allowing undisrupted execution for already existing applications, and (c) implementing parallel operations of reconfiguration and test for multiple applications. To achieve these objectives, we took appropriate measures during the design, compile, and run time phases. We then provided results and analysis to verify the non-intrusive nature and presented the timings details to test a test configuration functional region.

In *Chapter 8*, we used a case study H.264 encoder application, and evaluated the compile and run time behavior of the system. PUMA was used to perform the compile time binding of the H.264 encoder, and the 3-tier model was used to perform composable and persistent-state dynamic reconfiguration of H.264 IPs at run time.

In the next section we provide the contributions of the thesis.

9.2 Thesis Contributions

The main contributions of this dissertation can be summarized as follows.

1. We presented the architecture of an FPGA with a hardwired NoC and multiple test configuration functional regions (TCFRs), where each TCFR is a unified test, configuration, and logic plane. The HWNoC was used as a system level interconnect to provide inter-IP communication. In our proposed FPGA architecture, a number of features were investigated that included: (a) concept and implementation of TCFRs, (b) the definition of the boundary that provided interaction between the HWNOC and TCFRs, and (c) the control architecture in the proposed FPGA.

2. We presented the concept of data unification by using the HWNoC. This means the same HWNoC architecture was used to transport all kinds of application data, i.e., bitstreams, test, functional, and control data. For

that we developed the FPGA architecture and the design flow that could ensure the unified data transportation by using the same hardwired NoC.

3. We presented a a binding scheme that unified the process of placement, mapping, and allocation while binding application to FPGA architecture.

4. We also presented a 3-tier model to perform composable and persistent-state dynamic run time reconfiguration for applications.

5. We presented an online test scheme that used a HWNoC as the test access mechanism. Our online test scheme performed real-time streaming of test data, and importantly without causing any intrusiveness to the other communication traffic.

9.3 Open Issues and Future Directions

In this section we explain the open issues for future research directions that are explained below.

The topology of our architecture is mesh, i.e., the way test configuration functional regions are connected to each other. The architecture can be extended to different types of regular (e.g., ring, hypercubes, and fat-trees) and irregular topologies. Test Configuration Functional Regions with soft (i.e., reconfigurable) resources are used for the thesis. Though different clock domains can exist at inter-TCFR level, but we considered a single clock domain at intra-TCFR level that means all the IPs in a TCFR were running at the same frequency. Extending TCFR architecture to a mix of soft and hard resources, e.g., hardwired memory units, and providing multiple clock domains can be a candidate for the future research.

We have evaluated the proposed solution for streaming applications with address-less communication. Latency critical applications, and address-based communication can be a good candidate for the future research and to extend the solution. Our solution makes off-line (i.e., at compile time) calculation of resources to ensures QoS constrains for applications. This means, the applications are served from their fixed quota of resources (reserved during the compile time) while they are executing. The proposed solution can be extended to make online (i.e. at run time) calculation of the resources for applications.

Bibliography

[1] M. Abramovici, J. Emmert, and C. Stroud. Roving STARs: An Integrated Approach to On-Line Testing, Diagnosis, and Fault Tolerance for FPGAs in Adaptive Computing Systems. In *NASA/DoD Workshop on Evolvable Hardware*, pages 73–92, 2001.

[2] M. Abramovici, C. Stroud, C. Hamilton, S. Wijesuriya, and V. Verma. Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications. In *IEEE International Test Conference (ITC)*, pages 973–982, 1999.

[3] Actel. Incredible shrinking medical devices. White paper, 2008.

[4] T. Ahonen, D. A. Sigüenza-Tortosa, H. Bin, and J. Nurmi. Topology optimization for application-specific networks-on-chip. In *ACM Workshop on System Level Interconnect Prediction (SLIP)*, pages 53–60, 2004.

[5] B. Akesson, A. Molnos, A. Hansson, J. A. Angelo, and K. Goossens. Composability and predictability for independent application development, verification, and execution. In M. Huebner and J. Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration, Circuits and Systems*, chapter 2, pages 25–56. Springer, 2010.

[6] Al-Asaad. On-line built-in self-test for operational faults. In *IEEE AUTOTESTCON*, pages 168–174, 2000.

[7] Altera Corporation. FPGAs Power High-Performance Computing. White Paper, 2007.

[8] Altera Inc. Stratix V Data Sheet, 2010.

[9] ARM Limited. *AMBA AXI Protocol Specification*.

[10] A. Artieri, V. DAlto, R. Chesson, M. Hopkins, and M. C. Rossi. Nomadik Open Multimedia Platform for Next Generation Mobile Devices. In *Proc. Industrial Electronics Society (IECON)*, 2003. Technical Article TA305.

[11] L. Bauer, M. Shafique, and J. Henkel. Efficient Resource Utilization for an Extensible Processor through Dynamic Instruction Set Adaptation. *IEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 16(10):1295–1308, 2008.

[12] M. Bekooij, A. Moonen, and J. van Meerbergen. Predictable and Composable Multiprocessor System Design: A Constructive Approach. In *Bits & Chips Symposium on Embedded Systems and Software*, 2007.

[13] L. Benini and G. D. Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, 35(1):70–80, 2002.

[14] R. Bittner, P. Athanas, and M. Musgrove. Colt: An Experiment in Wormhole Run-Time Reconfiguration. In *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic (SPIE)*, pages 187–194, 1996.

[15] C. Bobda and A. Ahmadinia. Dynamic interconnection of reconfigurable modules on reconfigurable devices. volume 22, pages 443–451, 2005.

[16] C. Bobda, A. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich. The Erlangen Slot Machine: Increasing Flexibility in FPGA-based Reconfigurable Platforms. In *Proc. Int'l Conference on Field-Programmable Technology (FPT)*, 2005.

[17] C. Bobda, M. Majera, D. Koch, A. Ahmadinia, and J. Teich. A dynamic NOC approach for communication in reconfigurable devices. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, pages 1032–1036, 2004.

[18] S. Borkar. Thousand core chips: a technology perspective. In *Proc. Design Automation Conference (DAC)*, pages 746–749, 2007.

[19] G. Borriello, C. Ebeling, S. Hauck, and S. Burns. The Triptych FPGA Architecture. *IEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 3(4):473–582, 1995.

[20] G. Brebner. The Swappable Logic Unit: a Paradigm for Virtual Hardware. In *Proc. Int'l Conference on Field-Programmable Custom Computing Machines (FCCM)*, pages 77–86, 1997.

[21] G. Brebner and D. Levi. Networking on chip with platform FPGAs. In *Proc. Int'l Conference on Field-Programmable Technology (FPT)*, pages 13–20, 2003.

[22] B. Burke. Re-Configurable SoC with Embedded FPGA: "Application Independent Standard Part". In *Samsung Electronics, Embedded Systems Conference*, 2006.

[23] Business Dictionary. Product life cycle. See http://www.businessdictionary.com/definition/product-life-cycle.html.

[24] C. Chang, J. Wawrzynek, and R. Brodersen. BEE2: a high-end reconfigurable computing system. *IEEE Design & Test of Computers*, 22(2):114–125, 2005.

[25] I. Cidon and K. Goossens. Network and transport layers in networks on chip. In G. De Micheli and L. Benini, editors, *Networks on Chips: Technology and Tools*, The Morgan Kaufmann Series in Systems on Silicon, chapter 5, pages 147–202. Morgan Kaufmann, 2006.

[26] M. Coenen, S. Murali, A. Rădulescu, K. Goossens, and G. De Micheli. A buffer-sizing Algorithm for Networks on Chip using TDMA and credit-based end-to-end Flow Control. In *Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 130–135, 2006.

[27] J. Collins, G. Kent, and J. Yardley. Using the Starbridge Systems FPGA-based Hypercomputer for Cancer Research. In *International Conference on Military and Aerospace Programmable Logic Devices*, pages 684–689, 2004.

[28] S. D. Craven and P. Athanas. Examining the Viability of FPGA Supercomputing. *EURASIP Journal on Embedded Systems*, 2007(1):1–24, 2007.

[29] L. Devaux, S. B. Sassi, S. Pillement, D. Chillet, and D. Demigny. DRAFT: Flexible interconnection network for dynamically reconfigurable architectures. In *Proc. Int'l Conference on Field-Programmable Technology (FPT)*, 2009.

[30] L. Devaux, S. B. Sassi, S. Pillement, D. Chillet, and D. Demigny. Flexible Interconnection Network for Dynamically and Partially Reconfigurable Architectures. *International Journal of Reconfigurable Computing (IJRC)*, 2010:15, 2010.

[31] J.-P. Diguet, G. Gogniat, S. Evain, R. Vaslin, and E. Juin. NOC-centric security of reconfigurable SoC. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, May 2007.

[32] S. Dutt, V. Verma, and V. Suthar. Built-in-Self-Test of FPGAs With Provable Diagnosabilities and High Diagnostic Coverage With Application to Online Testing. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(2):309–326, 2008.

[33] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design & Test of Computers*, (5):21–31, 2001.

[34] EE Times. Altera's new 40nm FPGAs - 2.5 billion transistors, 2008.

[35] C. Foudas, R. Bainbridge, D. Ballard, I. Church, E. Corrin, J. Coughlan, C. Day, E. Freeman, J. Fulcher, W. Gannon, G. Hall, R. Halsall, G. Iles, J. Jones, J. Leaver, M. Noy, M. Pearson, M. Raymond, I. Reid, G. Rogers, J. Salisbury, S. Taghavi, I. Tomalin, and O. Zorba. The CMS tracker readout front end driver. *IEEE Transactions on Nuclear Science*, 52(6):2836–2840, 2005.

[36] Gazaleh Nazarian. On line Testing of Routers in Networks-on-Chips. Master thesis, Computer Engineering Department, Technical University of Delft (TUDelft), The Netherlands, 2008.

[37] M. G. Gericota, G. R. Alves, M. L. Silva, and J. M. Ferreira. Active Replication: Towards a Truly SRAM-based FPGA On-Line Concurrent Testing. In *IEEE On-Line Testing Workshop*, pages 165 – 169, 2002.

[38] M. G. Gericota, G. R. Alves, M. L. Silva, and J. M. Ferreira. AR2T: Implementing a Truly SRAM-based FPGA On-Line Concurrent Testing. In *IEEE European Test Workshop*, pages 61 – 66, 2002.

[39] R. Gindin, I. Cidon, and I. Keidar. NoC-Based FPGA: Architecture and Routing. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 253–264, 2007.

[40] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor, and R. Reed. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–77, 2000.

[41] K. Goossens, M. Bennebroek, J. Y. Hur, and M. A. Wahlah. Hardwired networks on chip in FPGAs to unify data and configuration interconnects. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 45–54, 2008.

[42] K. Goossens, J. Dielissen, O. P. Gangwal, S. González Pestana, A. Rădulescu, and E. Rijpkema. A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1182–1187, 2005.

[43] K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.

[44] K. Goossens and A. Hansson. The Aethereal Network on Chip after Ten Years: Goals, Evolution, Lessons, and Future. In *Proc. Design Automation Conference (DAC)*, 2010.

[45] A. Hansson, M. Coenen, and K. Goossens. Undisrupted Quality-Of-Service during Reconfiguration of Multiple Applications in Networks on Chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 954–959, 2007.

[46] A. Hansson and K. Goossens. Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 233–242, 2007.

[47] A. Hansson and K. Goossens. An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, 2009.

[48] A. Hansson and K. Goossens. On-Chip Interconnect with aelite: Composable and Predictable Systems. In *Embedded Systems*. Springer, 2009.

[49] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–24, 2009.

[50] A. Hansson, K. Goossens, and A. Rădulescu. A Unified Approach to Mapping and Routing on a Network on Chip for both Best-Effort and Guaranteed Service Traffic. *VLSI Design*, 2007:Article ID 68432, 16 pages, 2007. Hindawi Publishing Corporation.

[51] A. Hansson, K. Goossens, and A. Rădulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007:Article ID 95859, 10 pages, 2007. Hindawi Publishing Corporation.

[52] J. R. Hauser and J. Wawrzynek. Garp: a MIPS Processor with a Reconfigurable Coprocessor. In *Proc. Int'l Conference on Field-Programmable Custom Computing Machines (FCCM)*, pages 12 – 21, 1997.

[53] R. Hecht, S. Kubisch, A. Herrholtz, and D. Timmermann. Dynamic Reconfiguration with hardwired Networks-on-Chip on future FPGAs. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, pages 527–530, 2005.

[54] J. Henkel. Closing the SoC design gap. *IEEE Transactions on Computers*, 36(9):119–121, 2003.

[55] J. Henkel, W. Wolf, and S. Chakradhar. On-chip networks: A scalable, communication-centric embedded system design paradigm. In *VLSI Design*, pages 845–851, 2004.

[56] C. Hilton and B. Nelson. PNoC: A flexible circuit-switched NoC for FPGA-based systems. *IEE Proceedings on Computers and Digital Techniques*, 153(3):181–188, 2006.

[57] M. Hosseinabadi, A. Banaiyan, M. N. Bojnordi, and Z. Navabi. A Concurrent Testing Method for NoC Switches. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Apr. 2006.

[58] J. Hu and R. Marculescu. Energy-aware mapping for tile based NoC architectures under performance constraints. In *Proc. Design Automation Conference. Asia and South Pacific (ASPDAC)*, pages 233–239, 2003.

[59] Y. W. Huang, B. Y. Hsieh, T. C. Chen, and L. G. Chen. Analysis, fast algorithm, and VLSI architecture design for H.264/AVC intra frame coder. *IEEE Transaction on Circuit and Systems for Video Technology (TCSVT)*, 15(3):378–401, 2005.

[60] M. Huebner, C. Schuck, M. Kihnle, and J. Becker. New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits. In *IEEE Symposium on Emerging VLSI Technologies and Architectures*, 2006.

[61] M. Huebner, M. Ullmann, L. Braun, A. Klausmann, and J. Becker. Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems. In *Field Programmable Logic and Application*, volume 3203 of *Lecture notes in computer science*, pages 1037–1041, 2004.

[62] J. Hur, T. Stefanov, S. Wong, and S. Vassiliadis. Customizing Reconfigurable On-Chip Crossbar Scheduler. In *Proc. Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 210 – 215, 2007.

[63] J. Hur, S. Wong, and S. Vassiliadis. Partially Reconfigurable Point-to-Point Interconnects in Virtex-II Pro FPGAs. In *Proc. of Int'l Workshop on Applied Reconfigurable Computing (ARC)*, 2007.

[64] IEEE Computer Society. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE Press, 1990.

[65] Intel Inc. Microprocessor Quick Reference Guide, 2009.

[66] H. Ito, K. Oguri, K. Nagami, R. Konishi, and T. Shiozawa. The Plastic Cell Architecture for Dynamic Reconfigurable Computing. In *Proc. Int'l Workshop on Rapid System Prototyping (RSP)*, pages 39 – 44, 1998.

[67] ITRS. *The International Technology Roadmap for Semiconductors, Interconnect*. 2005.

[68] ITRS. *The International Technology Roadmap for Semiconductors, Design*. 2009.

[69] ITRS. *The International Technology Roadmap for Semiconductors, System Drivers*. 2009.

[70] W. Jang and D. Z. Pan. A3MAP: Architecture-Aware Analytic Mapping for Networks-on-Chip. In *Proc. Design Automation Conference. Asia and South Pacific (ASPDAC)*, pages 523–528, 2010.

[71] J. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook. Dynamic Reconfiguration to Support Concurrent Applications. In *IEEE Transactions on Computers*, volume 48, pages 591 – 602.

[72] Jeffrey Burt. Tilera Talks 100-Core Processor, 2009.

[73] E. Jhonsa. FPGAs, ASICs, and the Xilinx-Altera Duopoly. *The Digital Pathfinder*, 1, 2004.

[74] Joint Photographic Experts Group. Motion JPEG, 2000.

[75] Joint Video Team (JVT). H.264 : Advanced video coding for generic audiovisual services, 2007.

[76] R. Joost and R. Salomon. Advantages of FPGA-based multiprocessor systems in industrial applications. In *Proc. Industrial Electronics Society (IECON)*, 2005.

[77] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, pages 223 – 228, 2005.

[78] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. Packet switched vs time multiplexed FPGA overlay networks. In *Proc. Int'l Conference on Field-Programmable Custom Computing Machines (FCCM)*, pages 205–216, 2006.

[79] J.-G. Kim and Y.-D. Kim. A linear programming-based algorithm for floorplanning in VLSI design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 22(5):584–592, 2003.

[80] H. Kopetz, C. E. Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *Proc. Int'l SOC Conference (SoCC)*, pages 87 – 90, 2008.

[81] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[82] M. Krstic, E. Grass, F. K. Grkaynak, and P. Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *IEEE Design and Test of Computers*, 24(5):430–441, 2007.

[83] A. Kumar, A. Hansson, J. Huisken, and H. Corporaal. An FPGA design flow for reconfigurable network-based multi-processor systems on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2007.

[84] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(2):203–215, 2007.

[85] P. H. Leong. Recent Trends in FPGA Architectures and Applications. In *International Symposium on Electronic Design, Test and Applications*, pages 137 – 141, 2008.

[86] F. Lima, L. Carro, and R. Reis. Designing Fault Tolerant Systems into SRAM-based FPGAs. In *Proc. Design Automation Conference (DAC)*, pages 650 – 655, 2003.

[87] S. Lukovic and L. Fiorin. An Automated Design Flow for NoC-based MPSoCs on FPGA. In *Proc. Int'l Workshop on Rapid System Prototyping (RSP)*, pages 58 – 64, 2008.

[88] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multitasking on FPGAs. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, 2002.

[89] T. Marescaux, J.-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, 2003.

[90] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. B. W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Run-time Support for Heterogeneous Multitasking on Reconfigurable SoCs. *Integration, The VLSI Journal*, 38(1):107–130, 2004.

[91] D. Marpe, V. George, H. L. Cycon, and K. U. Barthel. Performance evaluation of Motion-JPEG2000 in comparison with H.264/AVC operated in pure intra coding mode. In *SPIE Conference on Wavelet Applications in Industrial Processing*, 2003.

[92] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC. In *Proc. Symposium Integrated Circuits and Systems Design (SBCCI)*, 2005.

[93] G. D. Micheli, P. Pande, A. Ivanov, C. Grecu, and R. Saleh. Design, Synthesis, and Test of Networks on Chips. *IEEE Design & Test of Computers*, 22(5):404–413, 2005.

[94] J.-Y. Mignolet, V. Nollet, P. Coene, D.Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Mar. 2003.

[95] A. Molnos, J. A. Ambrose, A. Nelson, R. Stefan, S. Cotofana, and K. Goossens. A Composable, Energy-Managed, Real-Time MPSOC Platform. In *In Proc. Int'l Conference on Optimization of Electrical and Electronic Equipment (OPTIM)*, 2010.

[96] G. Moore. Progress in digital integrated electronics. *Electron Devices Meeting*, 21:11–13.

[97] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38:114–117, 1965.

[98] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard real-time jobs on a heterogeneous multiprocessor. In *Proc. Int'l Conference on Embedded software (EMSOFT)*, 2007.

[99] S. Murali, L. Benini, and G. de Micheli. Mapping and Physical Planning of Networks on Chip Architectures with Quality of Service Guarantees. In *Proc. Design Automation Conference. Asia and South Pacific (ASP-DAC)*, pages 27 – 32, 2005.

[100] S. Murali, M. Coenen, A. Rădulescu, K. Goossens, and G. De Micheli. A methodology for mapping multiple use-cases on to networks on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2006.

[101] S. Murali and G. De Micheli. SUNMAP: A tool for automatic topology selection and generation for NOCs. In *Proc. Design Automation Conference (DAC)*, pages 914 – 919, 2003.

[102] S. Murali and G. De Micheli. Bandwidth-constrained mapping of cores onto NoC architectures. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 896–901, 2004.

[103] A. B. Nejad, M. E. Martinez, and K. Goossens. An FPGA Bridge Preserving Traffic Quality of Service for On-Chip Network-Based Systems. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1 – 6, 2011.

[104] H. N. Nguyen, V.-D. Ngo, Y. Bae, H. Cho, and H.-W. Choi. An QoS Aware Mapping of Cores Onto NoC Architectures. pages 278–288, 2007.

[105] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *ACM Tansactions on Design Automation for Embedded Systems*, 7(3):233–270, 2002.

[106] H. Nikolov, T. Stefanov, and E. F. Deprettere. Efficient automated synthesis, programming, and implementation of multi-processor platforms on FPGA chips. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, pages 1–6, 2006.

[107] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 234–239, 2005.

[108] R. Obermaisser, C. E. Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(7):956 – 965, 2009.

[109] A. Patel, C. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow. A Scalable FPGA-based Multiprocessor. In *Proc. Int'l Conference on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–120, 2006.

[110] Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, July 2002.

[111] T. Pionteck, R. Koch, and C. Albrecht. Applying Partial Reconfiguration to Networks-on-Chips. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, pages 1–6, 2006.

[112] K. Purna and D. Bhatia. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. In *IEEE Transactions On Computers*, volume 48, pages 579 – 590, 1999.

[113] J. Rabaey. System-on-Chip-Challenges in the Deep-Sub-Micron Era A case for the network-on-a-Chip. In *INTERCONNECT-CENTRIC DESIGN FOR ADVANCED SOC AND NOC*, chapter 1, pages 3–24. Springer, 2005.

[114] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer. An FPGA-based soft multiprocessor system for ipv4 packet forwarding. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, pages 487 – 492, 2005.

[115] E. S. Reddy, V. Chandrasekhar, M. Sashikanth, V. Kamakoti, and N. Vijaykrishnan. Online Detection and Diagnosis of Multiple Configuration Upsets in LUTs of SRAM-based FPGAs. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[116] Reference Dictionary. Scalability. http://dictionary.reference.com/browse/scalability.

[117] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian. Test Pattern and Test Configuration Generation Methodology for the Logic of RAM-Based FPGA. In *Proceedings of the 6th Asian Test Symposium*, 1997.

[118] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian. RAM-based FPGA's: a test approach for the configurable logic. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 1998.

[119] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian. Testing the configurable interconnect/logic interface of SRAM-based FPGA's. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 1999.

[120] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip.

IEE Proceedings: Computers and Digital Techniques, 150(5):294–302, 2003.

[121] A. Rădulescu, J. Dielissen, S. González Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(1):4–17, 2005.

[122] S. H. Russ, J. Robinson, M. Gleeson, and J. Figueroa. Dynamic Communication Mechanism Switching in Hector. In *Mississippi State University*, 1997.

[123] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, 2006.

[124] D. P. Schultz, S. P. Young, and L. C. Hung. *Method and structure for reading, modifying and writing selected configuration memory cells of an FPGA*. Xilinx, Inc., Aug. 1999. Patent US 6255848.

[125] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in Virtex FPGAs. *IEE Proceedings on Computers and Digital Techniques*, 153(3):157–164, 2006.

[126] P. Sedcole, J. S. Wong, and P. Y. K. Cheung. Characterisation of FPGA Clock Variability. In *Proc. Symposium on VLSI*, pages 322 – 328, 2008.

[127] M. Shafique, L. Bauer, and J. Henkel. Optimizing the H.264/AVC Video Encoder Application Structure for Reconfigurable and Application-Specific Platforms. *Proc. Journal of Signal Processing Systems (JSPS)*, 60(2):183–210, 2010.

[128] W.-T. Shen, C.-H. Chao, Y.-K. Lien, and A.-Y. A. Wu. A new binomial mapping and optimization algorithm for reduced-complexity mesh-based on-chip network. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 317 – 322, 2007.

[129] N. R. Shnidman, W. H. Mangione-Smith, and M. Potkonjak. On-line Fault Detection for Bus-Based Field Programmable Gate Arrays. *TVLSI*, 6(4):656 – 666, 1998.

[130] H. Simmler, L. Levinson, and R. Mnner. Multitasking on FPGA Coprocessors. In *Proc. Int'l Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, pages 121–130, 2000.

[131] 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 and computation-intensive applications. In *IEEE Transactions on Computers*, volume 49, pages 465–481, 2000.

[132] L. Singhal and E. Bozorgzadeh. Physically-aware Exploitation of Component Reuse in a Partially Reconfigurable Architecture. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[133] K. Srinivasan, K. S. Chatha, and G. Konjevod. An automated technique for topology and route generation of application specific on-chip interconnection networks. In *Proc. of Int'l Conference on Computer Aided Design (ICCAD)*, pages 231 – 237, 2005.

[134] F. Steenhof, H. Duque, B. Nilsson, K. Goossens, and R. Peset Llopis. Networks on Chips for High-End Consumer-Electronics TV System Architectures. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2006.

[135] R. Stefan and K. Goossens. Enhancing the security of time-division-multiplexing networks-on-chip through the use of multipath routing. In *Int'l Workshop on Network on Chip Architectures (NOCARC)*, 2011.

[136] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, pages 777 – 782, 2007.

[137] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Int'l Symposium on Computer Architecture*, pages 2–13, 2004.

[138] Texas Instruments Inc. OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide, 2004.

[139] Tilera Corporation. Tilera: The cloud computer has arrived, 2011.

[140] Tom R. Halfhill. Tabulas Time machine: Rapidly Reconfigurable Chips Will Challenge Conventional FPGAs, 2010.

[141] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, april 2004.

[142] V. Verma, S. Dutt, and V. Suthar. Efficient on-line testing of FPGAs with provable diagnosabilities. In *Proc. Design Automation Conference (DAC)*, 2004.

[143] M. A. Wahlah and K. Goossens. 3-Tier Reconfiguration Model For FPGAs Using Hardwired Network on Chip. In *Proc. Int'l Conference on Field-Programmable Technology (FPT)*, Dec. 2009.

[144] M. A. Wahlah and K. Goossens. Composable And Persistent-State Application Swapping On FPGAs Using Hardwired Network on Chip. In *Proc. Reconfigurable Computing and FPGAs (ReConFig)*, 2009.

[145] M. A. Wahlah and K. Goossens. Modeling Reconfiguration in a FPGA with a Hardwired Network on Chip. In *Proc. Reconfigurable Architecture Workshop (RAW)*, May 2009.

[146] M. A. Wahlah and K. Goossens. A Non-Intrusive Online FPGA Test Scheme Using A Hardwired Network on Chip. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2011.

[147] M. A. Wahlah and K. Goossens. PUMA: Placement Unification with Mapping and guaranteed throughput Allocation on an FPGA Using A Hardwired NoC. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2011.

[148] P. Z. Waleed M. Meleis, Miriam Leeser and M. M. Vai. Architectural design of a three dimensional FPGA. In *Advanced Research in VLSI*, pages 256–268, 1997.

[149] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Kozyrakis, and K. Olukotun. A Practical FPGA based Framework for Novel CMP Research. In *Proc. Int'l International Symposium on Field Programmable Gate Arrays (FPGA)*, 2007.

[150] P. Wielage, E. J. Marinissen, M. Altheimer, and C. Wouters. Design and DFT of a high-speed area-efficient embedded asynchronous FIFO. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2007.

[151] P. Wijetunga. High-performance crossbar design for system-on-chip. In *System-on-Chip for Real-Time Applications*, 2003.

[152] W. Wolf. The Future of Multiprocessor Systems-on-Chips. In *Proc. Design Automation Conference (DAC)*, 2004.

[153] Xilinx Inc. Development System Reference Guide.

[154] Xilinx Inc. Virtex-4 Configuration Guide.

[155] Xilinx Inc. Virtex and Virtex-E FPGA Data Sheets, 2000.

[156] Xilinx Inc. Virtex-2 and Virtex-2 Pro FPGA Data Sheets, 2002.

[157] Xilinx Inc. Processor Local Bus (PLB) v3.4, 2003.

[158] Xilinx Inc. Virtex-4 Data Sheets, 2005.

[159] Xilinx Inc. Virtex-4 User Guide, 2005.

[160] Xilinx Inc. Virtex-5 User Guide, 2007.

[161] Xilinx Inc. Virtex-5 Data Sheets, 2008.

[162] Xilinx Inc. Virtex-6 Data Sheets, 2009.

[163] Xilinx Inc. Virtex-7 Data Sheets, 2012.

[164] S. Young, P. Alfke, C. Fewer, S. McMillan, B. Blodget, and D. Levi. A HIGH I/O Reconfigurable Crossbar Switch. In *Proc. Int'l Conference on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–10, Apr. 2003.

A
Glossary

This chapter provides a guide to the language used in this thesis. Section A.1 contains the list of abbreviations, Section A.2 contains the list of terminology, and Section A.3 provides the list of legends that are used in the thesis.

A.1 List of Abbreviations

The list of abbreviations explains the most commonly used abbreviations in this thesis.

AM Application Manager

AGU Address Generation Unit

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

BIST built in self test

BSI boundary scan infrastructure

CDC Clock Domain Crossing

CLB Configurable logic block

DMA Direct Memory Access

DSM Deep Sub-Micron

DTL Device Transaction Level

FIFO First In, First Out

Fnode FPGA node

FPGA Field Programmable Gate Array

FSM Finite State Machine

GALS Globally Asynchronous Locally Synchronous

GT Guaranteed throughput

HDL Hardware Description Language

HPU header parsing unit

HWNoC Hardwired Network on Chip

ICAP Internal Configuration Port

(I)DCT (Inverse) Discrete Cosine Transform

IOB Input Output Block

IP Intellectual Property

ITRS International Technology Roadmap for Semiconductors

LUT Lookup Table

MB Macro Block

MMIO Memory Mapped Input Output

MPSoC Multi-processor System-on-Chip

MTCR Minimum Test Configuration Region

NI Network Interface

NoC Network on Chip

NRE Non-Recurring Engineering

ORA Output Response Analyser

OS Operating System

PE Processing Element

PPSD Point-to-Point Streaming Data

PUMA Placement Unification with Mapping and Allocation

QoS Quality of Service

RNUT Region Not Under Test

RTR Run Time Reconfiguration

RUT Region Under Test

SDR Software-Defined Radio

SM System Manager

SoC System-on-Chip

TAM Test Access Mechanism

TCFR Test Configuration Functional Region

TDM Time-division Multiplexing

TPG Test Pattern Generator

TTM Time To Market

UCF User Constraint File

In the following discussion we explain the terminology (along with the respective page number) that is used in our thesis.

A.2 List of Terminology

A *hardwired* or *hard* IP is directly implemented in silicon, page 1.

A *soft* IP is mapped on the reconfigurable resources (e.g. CLBs) of FPGA, page 1.

An *application* can be defined as a program that is designed to perform a specific function, page 2.

Functional data (or simply data) stands for the data that is computed or stored by the IPs, page 2.

Control data is used to program the IPs by writing to their memory-mapped input output (MMIO) registers, page 2.

A *use-case* is defined as the set of applications that execute in parallel at a given time, page 3.

Run time is defined as the time during which an application executes, page 51.

Throughput is the average data transfer rate that is required over a communication connection, page 3.

Latency stands for the amount of time data takes to traverse the communication connection, page 4.

(Re)configuration is the installation of new functionality in the FPGA by sending a bitstream to a reconfiguration region, page 7.

Dynamic partial reconfiguration allows the reconfiguration of selected area of FPGA without shutting down the applications that run on rest of FPGA, page 12.

Configuring an IP means loading its bitstream in the configuration plane, page 14.

Programming an IP means changing the state of its registers, page 14.

An application is said to be: (i) *placed* when its IPs are placed on FPGA logic plane, (ii) *mapped* when its IP ports are connected to the functional interconnect, and (iii) *allocated* when its IPs can communicate (after programming the NoC) with each other as per QoS constraints. We term the whole process of placing, mapping, and allocation as *binding*, page 124.

Compile time is defined as the time during which the user specifications are being translated into the executable code (for hardware and software), page 51.

Online testing verifies the FPGA chip while the system is operational, page 22.

Stuck-at fault defines the state of a logic block or wire when it always stays at `1` or `0` and can not be reversed, page 78.

Minimum test configuration region, i.e., an MTCR is the minimum region that can be tested or configured. In Virtex-4 it consists of a column of 16 CLBs and the associated programmable interconnect [154], page 182.

In the following discussion we will explain the legends that are used in our thesis.

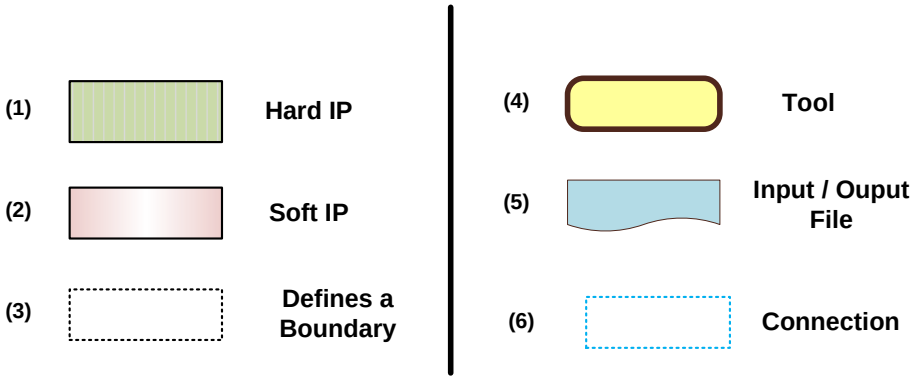


Figure A.1: Showing Different Figures that are Used in the Thesis.

A.3 List of Legends

In this section we provide the list of legends that are used in the thesis as shown in Figure A.1. For example a hard IP is represented with a rectangular box with multiple line patterns, as shown in Figure A.1. However, the box with multiple line patterns can have different colors depending upon the type of the hard IP, e.g., a BRAM, DSP block, and PowerPc, etc., all can be categorised as hard but with different types. Similarly, a soft IP is represented with a rectangular box with shaded color pattern, as shown in Figure A.1. However, the box with shaded color pattern can have different colors depending upon the type of the soft IP, e.g., an NI shell, BUS Macro units, CLB units , etc., all can be categorised as soft but with different types.

B

System XML specification

This chapter shows the XML specifications that are used as input to the design flow, presented in Chapter 3. First, we look at the architecture specification in Section B.1, followed by the application specification in Section B.2.

B.1 Architecture specification

The architecture specification lists a number of Intellectual Property (IP) components, each a number of ports. In addition the area (MTCRs) and bitstream address of each IP is also specified. For each port, type, protocol and other relevant architecture parameters are specified.

```
<architecture  id="thesis">
  <!---HWNoC Specifications--->
  <parameter id="clk" type="int" value="500" />
  <parameter id="slotsize" type="int" value="3" />
  <parameter id="slots" type="int" value="166" />

  <ip id="AM" type="IP" size="31" bitaddr="0">
      <port id="A1" type="Initiator" protocol="MMIO_DTL">
         <parameter id="width" type="int" value="32"/>
         <parameter id="blocksize" type="int" value="32"/>
      </port>
      <port id="A2" type="Target" protocol="MMIO_DTL">
         <parameter id="width" type="int" value="32" />
      </port>
   </ip>
   <ip id="Residue" type="IP" size="13" bitaddr="3781">
      <port id="B1" type="Initiator" protocol="MMIO_DTL">
         <parameter id="width" type="int" value="32"/>
```

```
        <parameter id="blocksize" type="int" value="32"/>
    </port>
    <port id="B2" type="Target" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32" />
    </port>
</ip>
<ip id="DCT" type="IP" size="18" bitaddr="49044">
    <port id="C1" type="Initiator" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32"/>
        <parameter id="blocksize" type="int" value="32"/>
    </port>
    <port id="C2" type="Target" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32" />
    </port>
</ip>
<ip id="QNT" type="IP" size="19" bitaddr="116948">
    <port id="D1" type="Initiator" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32"/>
        <parameter id="blocksize" type="int" value="32"/>
    </port>
    <port id="D2" type="Target" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32" />
  </port>
</ip>
<ip id="AM2" type="IP" size="31" bitaddr="400000">
    <port id="E1" type="Initiator" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32"/>
        <parameter id="blocksize" type="int" value="32"/>
    </port>
    <port id="E2" type="Target" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32" />
    </port>
</ip>
<ip id="Residue2" type="IP" size="13" bitaddr="3781">
    <port id="F1" type="Initiator" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32"/>
        <parameter id="blocksize" type="int" value="32"/>
    </port>
    <port id="F2" type="Target" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32" />
    </port>
</ip>
<ip id="DCT2" type="IP" size="18" bitaddr="49044">
    <port id="G1" type="Initiator" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32"/>
```

```
            <parameter id="blocksize" type="int" value="32"/>
        </port>
        <port id="G2" type="Target" protocol="MMIO_DTL">
            <parameter id="width" type="int" value="32" />
        </port>
    </ip>
    <ip id="IQNT" type="IP" size="19" bitaddr="188624">
        <port id="B1" type="Initiator" protocol="MMIO_DTL">
            <parameter id="width" type="int" value="32"/>
            <parameter id="blocksize" type="int" value="32"/>
        </port>
        <port id="B2" type="Target" protocol="MMIO_DTL">
            <parameter id="width" type="int" value="32" />
        </port>
    </ip>
    <ip id="IDCT" type="IP" size="18" bitaddr="256528">
        <port id="C1" type="Initiator" protocol="MMIO_DTL">
            <parameter id="width" type="int" value="32"/>
            <parameter id="blocksize" type="int" value="32"/>
        </port>
        <port id="C2" type="Target" protocol="MMIO_DTL">
            <parameter id="width" type="int" value="32" />
        </port>
    </ip>
    <ip id="Reconstruct" type="IP" size="13" bitaddr="305572">
        <port id="D1" type="Initiator" protocol="MMIO_DTL">
            <parameter id="width" type="int" value="32"/>
            <parameter id="blocksize" type="int" value="32"/>
        </port>
        <port id="D2" type="Target" protocol="MMIO_DTL">
            <parameter id="width" type="int" value="32" />
        </port>
    </ip>
</architecture>
```

B.2 Application Specification

The specifications below correspond to the running example in Chapter 6. It consists of 3 applications and two use-cases that allow app0 and app1, and app1 and app2 to execute in parallel. Each application specification consists of a number of connections, where each connection corresponds to a requestor. For each requestor, the write type is specified, along with burst sizes, required

bandwidth and latency requirements.

```
<communication>
 <application id="app0">
   <connection id="0" qos="GT">
        <initiator ip="AM" port="A1" />
        <target ip="Residue" port="B2" />
       <write bw="92" burstsize="32" latency="1000" />
   </connection>
   <connection id="1" qos="GT">
        <initiator ip="Residue" port="B1" />
        <target ip="DCT" port="C2" />
       <write bw="62" burstsize="32" latency="1000" />
      </connection>
   <connection id="2" qos="GT">
        <initiator ip="DCT" port="C1" />
         <target ip="QNT" port="D2" />
       <write bw="94" burstsize="32" latency="1000" />
      </connection>
   <connection id="3" qos="GT">
        <initiator ip="QNT" port="D1" />
         <target ip="AM" port="A2" />
       <write bw="94" burstsize="32" latency="1000" />
   </connection>
 </application>
 <application id="app1">
   <connection id="0" qos="GT">
        <initiator ip="AM2" port="E1" />
         <target ip="Residue2" port="F2" />
       <write bw="46" burstsize="32" latency="1000" />
   </connection>
   <connection id="1" qos="GT">
        <initiator ip="Residue2" port="F1" />
         <target ip="DCT2" port="G2" />
       <write bw="31" burstsize="32" latency="1000" />
   </connection>
   <connection id="2" qos="GT">
        <initiator ip="DCT2" port="G1" />
         <target ip="AM2" port="E2" />
       <write bw="47" burstsize="32" latency="1000" />
   </connection>
 </application>
 <application id="app2">
   <connection id="0" qos="GT">
        <initiator ip="AM" port="A1" />
```

```
        <target ip="IQNT" port="B2" />
       <write bw="94" burstsize="32" latency="1000" />
   </connection>
   <connection id="1" qos="GT">
        <initiator ip="IQNT" port="B1" />
        <target ip="IDCT" port="C2" />
       <write bw="62" burstsize="32" latency="1000" />
   </connection>
   <connection id="2" qos="GT">
        <initiator ip="IDCT" port="C1" />
         <target ip="Reconstruct" port="D2" />
       <write bw="94" burstsize="32" latency="1000" />
   </connection>
   <connection id="3" qos="GT">
        <initiator ip="Reconstruct" port="D1" />
         <target ip="AM2" port="A2" />
       <write bw="92" burstsize="32" latency="1000" />
   </connection>
 </application>
<constraint type="allow" appl="app0" with="app1" />
<constraint type="allow" appl="app1" with="app2" />
</communication>
```

List of Publications

International Journals

1. M. A. Wahlah and K. Goossens, **TeMNOT: A Test Methodology for the Non-Intrusive Online Testing of FPGA with Hardwired Network on Chip**, *In Microprocessors and Microsystems (MICPRO), Elsevier*, http://dx.doi.org/10.1016/j.micpro.2012.05.011, 2012.

2. J. Y. Hur, K. Goossens, L. Mhamdi, M. A. Wahlah, **Comparative Analysis of Soft and Hard On-Chip Interconnects for FPGAs**, *IET Computers and Digital Techniques (IET CDT)*, (to appear), 2012.

International Conferences

1. M. A. Wahlah and K. Goossens, **A Non-Intrusive Online FPGA Test Scheme using a hardwired network on chip**, *In Proc. Euromicro Symposium on Digital System Design* (DSD), 2011

2. M. A. Wahlah and K. Goossens, **PUMA: Placement Unification with Mapping and guaranteed throughput Allocation on an FPGA using a hardwired network on chip**, *In Proc. Euromicro Symposium on Digital System Design* (DSD), 2011

3. M. A. Wahlah and K. Goossens, **Modeling reconfiguration in a FPGA with a hardwired network on chip**, *Reconfigurable Architecture Workshop* (RAW), 2009

4. M. A. Wahlah and K. Goossens, **3-Tier Reconfiguration Model For FPGAs Using Hardwired Network on Chip**, *Proceedings of the International Conference on Field-Programmable Technology* (FPT), 2009

5. M. A. Wahlah and K. Goossens, **Composable And Persistent-State Application Swapping On FPGAs Using Hardwired Network on Chip**, *Proceedings of the International Conference on ReConFigurable Computing and FPGAs* (ReConFig), 2009

6. K. Goossens, M. Bennebroek, J. Y. Hur, and M. A. Wahlah, **Hardwired Networks on Chip in FPGAs to unify Data and Configuration Interconnects**, *Network on Chip Symposium* (NOCS), 2008

National Conferences

1. M. A. Wahlah and K. Goossens, **Run-Time FPGA Testing Using Hardwired Network on Chip**, *In Proc. Annual Workshop on Circuits, Systems and Signal Processing* (ProRisc), 2009

2. M. A. Wahlah and K. Goossens, **Hardwired NOC Infrastructure with Integrated Configuration and Functional Architecture**, *In Proc. Annual Workshop on Circuits, Systems and Signal Processing* (ProRisc), 2008

Samenvatting

De bovengenoemde applicatie en architectuur trends hebben tot een aantal problemen geleid. (1) Een toenemend aantal toepassingen op een FPGA vereist vaak dynamische herconfiguratie van een toepassing die interferentie kan veroorzaken met andere actieve toepassingen. (2) De toenemende complexiteit van een applicatie kan mogelijk niet toegekend worden aan de FPGA, wat verlies van data gedurende dynamische partile herconfiguratie tot mogelijke gevolg kan hebben. (3) De van natuur diverse toepassingen maken het moeilijk om aan Quality-of-Service eisen van een applicatie te voldoen. (4) Ook is het moeilijk om (fysieke) timing haalbaarheid in een SoC te bereiken, als gevolg van het toenemende aantal en verscheidenheid van de IP cores. (5) Het neerwaarts schalen van de technologie leidt tot FPGA architecturen die meer vatbaar zijn voor fouten, bv. geconfigureerde geheugens en logische elementen in een FPGA kunnen op een bepaalde waarde vast zitten. (6) Omdat de communicatie architectuur en IPs beide toegekend worden als soft IPs in dezelfde logische vlak van de FPGA legt hun plaatsing vele beperkingen op, om dynamische gedeeltelijke herconfiguratie mogelijk te maken.

In dit proefschrift willen we de bovengenoemde problemen aanpakken door de architectuur en design flow van een nieuw FPGA voorstellen.

De belangrijkste bijdrage van dit proefschrift is het voorstellen van de FPGA architectuur met een hardwired network on chip (HWNoC), en meerdere testen, configuratie en functionele regio's (TCFRs). Wij noemen het hardwired, omdat de NoC in een FPGA uit silicium is gebouwd en niet door herconfigureerbare delen te gebruiken. Met een HWNoC kunnen we een globaal asynchrone lokaal synchroon (GALS) omgeving hebben, die op zijn beurt ervoor zorgt dat data niet verloren gaat tijdens inter-IP-communicatie. De HWNoC scheidt de communicatie en de berekening in twee disjuncte vlakken dat beperkingen op de plaatsing van IP cores verlicht. De tweede bijdrage van dit proefschrift is het laten zien hoe we de HWNoC kunnen gebruiken om uniform testen, configuratie, en functionele gegevens te transporteren naar TCFRs, voor testen, sneller configureren, en storingsvrije communicatie tijdens de uitvoering van applicaties. De derde bijdrage van het proefschrift is het laten zien dat de voorgestelde design flow voorspelbaar applicatie gedrag garandeert door te voldoen aan de QoS eisen. We presenteren ook een 3-tier herconfiguratie model dat gebruik maakt van de HWNoC die contention-free communicatie garandeert op architectuur niveau, om de prob-

lemen van interferentie en toestand verlies te overwinnen, respectievelijk tijdens inter-applicatie en intra-applicatie herconfiguratie. Nog een bijdrage van dit proefschrift is het voorstellen van een niet-intrusieve testmethodologie dat de HWNoC gebruikt als een test toegang mechanisme om de aanwezigheid van fouten van FPGA architectuur te testen. Met andere woorden, de voorgestelde methodologie zorgt ervoor dat applicaties altijd herconfigureerd en uitgevoerd worden in een betrouwbaar gebied van een FPGA en zonder het benvloeden van de andere applicaties.

Curriculum Vitae

Muhammad Aqeel Wahlah was born on December 21, 1978 in Lahore, Pakistan. From 1996 to 2000 he studied in the University Of Engineering and Technology (UET) Lahore. He received Bachelor of Science degree in Electrical engineering with specialization in communications. He completed his Master of Science in Information Technology from Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad.

In November 2006, he joined the Computer Engineering laboratory of Delft University of Technology in the Netherlands, and, under the advisory of Professor Kees Goossens, he started his PhD study, working on hardwired networks on chip for Field Programmable Gate Arrays. The research work was funded by the Higher Education Commission (HEC) of Pakistan. The results of this work are presented in the current dissertation. Muhammad Aqeel Wahlah's research interests include Embedded Systems, Reconfigurable Computing, Networks on Chip, and Field Programmable Gate Arrays.