# A FRAMEWORK AND METHOD FOR THE RUN-TIME ON-CHIP SYNTHESIS OF MULTI-MODE SELF-ORGANIZED RECONFIGURABLE STREAM PROCESSORS

by

Victor Dumitriu
Bachelor of Engineering (B.Eng.), Ryerson University, Toronto 2006
Master of Applied Science (MASc), Ryerson University, Toronto 2008

A dissertation
presented to Ryerson University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2015

# A Framework and Method for the Run-Time On-Chip Synthesis of Multi-Mode Self-Organized Reconfigurable Stream Processors

Victor Dumitriu
Doctor of Philosophy, 2015,
Electrical and Computer Engineering,
Ryerson University

## ABSTRACT

A number of modern digital processing systems implement complex multi-mode applications with high performance requirements and strict operating constraints; examples include video processing and telecommunication applications. A number of these systems use increasingly large FPGAs as the implementation medium, due to reduced development costs. The combination of increases in FPGA capacity and system complexity has lead to a non-linear increase in system implementation effort. If left unchecked, implementation effort for such systems will reach the point where it becomes a design and development bottleneck. At the same time, the reduction in transistor size used to manufacture these devices can lead to increased device fault rates. To address these two problems, the Multi-mode Adaptive Collaborative Reconfigurable self-Organized System (MACROS) Framework and design methodology is proposed and described in this work. The MACROS Framework offer the ability for run-time architecture adaptation by integrating FPGA configuration into regular operation. The MACROS Framework allows for run-time generation of Application-Specific Processors (ASPs) through the deployment, assembly and integration of pre-built functional units; the framework further allows the relocation of functional units without affecting system functionality. The use of functional units as building blocks allows the system to be implemented on a piece-by-piece basis, which reduces the complexity of mapping, placement and routing tasks; the ability to relocate functional units allows fault mitigation by avoiding faulty regions in a device. The proposed framework has been used to implement multiple video processing systems which were used as verification and testing instruments. The MACROS framework was found to successfully support run-time architecture adaptation in the form of functional unit deployment and relocation in high performance systems. For large systems (more than 100 functional units), the MACROS Framework implementation effort, measured as time cost, was found to be one third that of a traditional (monolithic) system; more importantly, in MACRO Systems this time cost was found to increase linearly with system complexity (the number of functional units). When considering fault mitigation capabilities, the resource overhead associated with the MACROS Framework was found to be up to 85 % smaller than a traditional Triple Module Redundancy (TMR) solution.

# Acknowledgments

The author wishes to thank Dr. Lev Kirischain for sharing his knowledge and experiences, as well as providing support in his role as doctoral studies supervisor.

Additionally, the author wishes to acknowledge the support of the Department of Electrical and Computer Engineering at Ryerson University, the CMC Microsystems Corporation for providing all required CAD tools and systems, the Ontario Centers of Excellence (OCE), Federal Development (FedDev) Ontario, Connect Canada and the Natural Sciences and Engineering Research Council (NSERC) of Canada for financial support during the period of study.

The author also wish to acknowledge the support of industrial partners: MDA Space Missions, Soft-R US Inc. and UBS Ltd. for professional consulting and financial support. The author would like to acknowledge Dr. Karen Safaryan for his valuable consultations and assistance during the R&D projects and internship.

Finally, the author wishes to thank his family for the unwavering support they have provided across the years of study; the presented research would not have been possible without them.

# Contents

**List of Tables** **xi**

**List of Figures** **xii**

**1 Introduction** **1**

1.1 Motivation . . . . . . . . . . 1

1.1.1 Trends in Modern Digital System Design . . . . . . . . . . 1

1.1.2 Manufacturing Trends and Device Resilience . . . . . . . . . . 6

1.2 Modular And Dynamic Systems . . . . . . . . . . 9

1.3 Research Objective . . . . . . . . . . 11

1.4 Contribution . . . . . . . . . . 13

1.5 Thesis Document Structure . . . . . . . . . . 14

**2 Background and Related Work** **16**

2.1 Introduction . . . . . . . . . . 16

2.2 Mechanisms for Functional Unit Deployment and Relocation in Dynamically Reconfigurable Field-Programmable Gate Arrays (FPGAs) . . . . . . . . . . 17

2.2.1 Dynamic Partial Reconfiguration in Modern FPGAs . . . . . . . . . . 17

2.2.2 Free Functional Unit Placement and Relocation . . . . . . . . . . 20

2.3 On-Chip Communication Architectures for Dynamic Partially Reconfigurable Systems 23

2.3.1 Shared Buses . . . . . . . . . . 23

2.3.2 Network-on-Chip . . . . . . . . . . 24

2.3.3 Fully-Connected Crossbar . . . . . . . . . . 26

2.3.4 Additional Communication Architectures . . . 27
2.4 Board and System-Level Run-Time Assembly and Integration Methods . . . 28
2.5 Mechanisms for On-Chip Processor Assembly . . . 30
2.5.1 Configuration Control . . . 31
2.5.2 Configuration Scheduling . . . 31
2.5.3 Link Establishment . . . 32
2.6 Mechanisms for Functional Unit Integration . . . 33
2.7 Fault Mitigation Using Dynamic Partial Reconfiguration of FPGAs . . . 34
2.7.1 Scrubbing-Based Fault Mitigation . . . 35
2.7.2 Fault Mitigation Using Both Scrubbing and Relocation . . . 36
2.8 Summary . . . 38

**3 MACROS Framework Operating Principles 39**
3.1 Introduction . . . 39
3.2 Run-time Architecture-to-Mode Adaptation . . . 40
3.3 Digital Data Streams and Stream Workload . . . 43
3.3.1 Digital Data Stream Definition . . . 43
3.3.2 Stream Workload Structure . . . 44
3.4 Mode-Based Architecture Changes and Stream Workloads . . . 45
3.5 Seamless Run-Time Architecture Adaptations . . . 46
3.6 Summary . . . 49

**4 Multi-mode Adaptive Collaborative self-Organized System (MACROS) Framework Mechanisms 51**
4.1 Introduction . . . 51
4.2 Functional Unit Deployment and Relocation . . . 52
4.2.1 Partially Reconfigurable Region Structure and Functional Unit Placement . . 53
4.2.2 Free Functional Unit Deployment and Relocation . . . 55
4.2.3 Functional Unit Deployment and Relocation in MACROS Framework Systems 56
4.3 On-Chip Communication Architectures . . . 57
4.3.1 Communication Requirements . . . 58

4.3.2 Communication Infrastructure Analysis . . . . . 59
4.3.3 MACROS Framework Communication Architecture . . . . . 62
4.3.4 Delay Variation Mitigation . . . . . 63
4.4 On-Chip Assembly Mechanism . . . . . 64
4.4.1 Functional Unit Allocation and Configuration Scheduling Analysis . . . . . 65
4.4.2 Connectivity Change Analysis . . . . . 67
4.4.3 Non-Preemptive Connectivity Change Algorithms . . . . . 69
4.5 Functional Unit Integration Mechanisms . . . . . 74
4.5.1 Scheduling and Data Synchronization in Multi-Mode and Dynamically Reconfigurable Pipe-Lined and Data-Flow Environments . . . . . 75
4.5.2 Scheduler Implementation Solutions . . . . . 75
4.6 Central Versus Distributed Control Structures in the MACROS Framework . . . . . 79
4.7 Summary . . . . . 82

**5 MACROS Framework Architecture 83**

5.1 Introduction . . . . . 83
5.2 Top-Level MACROS Framework Architecture . . . . . 83
5.2.1 Top-Level Architecture . . . . . 84
5.2.2 Link Structures in the MACROS Framework . . . . . 86
5.2.3 Data and Control Flow . . . . . 87
5.2.4 System-Level Customization . . . . . 89
5.3 Collaborative Macro-Function Unit . . . . . 89
5.3.1 Collaborative Macro-Function Unit (CMFU) Responsibilities and Behavior . 90
5.3.2 CMFU Structure . . . . . 90
5.3.3 Co-Op Unit Structure and Behavior . . . . . 93
5.3.4 CMFU Customization . . . . . 97
5.4 Distributed Communication and Control Infrastructure . . . . . 98
5.4.1 Distributed Communication and Control Infrastructure (DCCI) Responsibilities and Behavior . . . . . 98
5.4.2 DCCI Structure . . . . . 99

5.4.3 Local Connection Control Unit . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

5.4.4 Control Data Broadcast Network . . . . . . . . . . . . . . . . . . . . . . . . . . 108

5.4.5 DCCI Customization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

5.5 Mode-Mapper . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

5.6 Bit-stream and Configuration Manager . . . . . . . . . . . . . . . . . . . . . . . . 114

5.6.1 Bit-stream and Configuration Manager (BCM) Responsibilities and Behavior 114

5.6.2 BCM Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

5.6.3 Architecture Look-Up Engine . . . . . . . . . . . . . . . . . . . . . . . . . . . 117

5.6.4 Allocation Engine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119

5.6.5 Configuration Engine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122

5.6.6 BCM Customization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124

5.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125

**6 MACROS Framework Run-Time Behavior** **127**

6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

6.2 Multi-Mode Application Behavior in MACROS Systems . . . . . . . . . . . . . . . 127

6.2.1 System Start-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

6.2.2 Normal Processing Operations . . . . . . . . . . . . . . . . . . . . . . . . . . 129

6.2.3 Run-Time Mode-Based ASP Generation . . . . . . . . . . . . . . . . . . . . . 130

6.3 Fault Mitigation Behavior in the MACROS Framework . . . . . . . . . . . . . . . 131

6.3.1 Targeted Fault Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132

6.3.2 MACROS Framework Fault Mitigation Principle . . . . . . . . . . . . . . . . 133

6.3.3 BCM Behavior for Fault Mitigation . . . . . . . . . . . . . . . . . . . . . . . 134

6.3.4 Relocation-Based Fault Mitigation Process . . . . . . . . . . . . . . . . . . . 136

6.3.5 Extensions for Fault Tolerance in MACROS Framework Systems . . . . . . . 137

6.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139

**7 MACROS Framework Experiments** **141**

7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

7.2 General Experimental Set-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

7.3 Experiment: Framework Verification and Performance Measurements . . . . . . . . . 142

7.3.1 Experimental Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
7.3.2 Experimental Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
7.4 Experiment: Monolithic System Implementation Effort Measurement . . . . . . . . . 149
7.5 Experiment: Frameworks Resource Cost Measurements . . . . . . . . . . . . . . . . 151
7.6 Experiment: Central Versus Distributed Control System Analysis . . . . . . . . . . . 152
7.7 Experiment: Communication Architecture Performance Measurements . . . . . . . . 155
7.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157

**8 Results and Analysis 158**

8.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
8.2 MACROS Framework Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
8.3 Multi-Mode Run-Time Architecture Adaptation Timing Analysis . . . . . . . . . . . 160
8.4 Control System Performance and Resource Cost Analysis . . . . . . . . . . . . . . . 163
8.5 Communication Architecture Performance and Resource Cost Analysis . . . . . . . . 171
8.6 Modular Implementation Effort Analysis . . . . . . . . . . . . . . . . . . . . . . . . 174
8.7 Fault Mitigation Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179
8.8 Complete Architectural Comparisons . . . . . . . . . . . . . . . . . . . . . . . . . 182
8.9 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184

**9 Conclusion 186**

9.1 Problems in Modern Digital System Design and the Concept of Modular Dynamic Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
9.2 The MACROS Framework: Run-time Architecture Adaptation in Stream Processors 187
9.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190

**Bibliography 192**

**Glossary 206**

# List of Tables

8.1 Bit-Stream Sizes Configuration Times . . . . . . . . . . . . . . . . . . . . . . . . . . 160
8.2 Application-Specific Processor (ASP) Generation Cost Values for the Canonical and Minimal System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
8.3 Stream Interrupt Factors for the Canonical and Minimal System . . . . . . . . . . . 162
8.4 MACROS Test System Resource Cost . . . . . . . . . . . . . . . . . . . . . . . . . . 164
8.5 Control Overhead in the MACROS Framework . . . . . . . . . . . . . . . . . . . . . 165
8.6 Crossbar Size in Relation to Number of Ports . . . . . . . . . . . . . . . . . . . . . . 172
8.7 Network-on-Chip (NoC) and Crossbar Comparison . . . . . . . . . . . . . . . . . . . 173
8.8 Data-Rate per MHz for Various Communication Architectures . . . . . . . . . . . . 174
8.9 Implementation Times for Canonical System . . . . . . . . . . . . . . . . . . . . . . 175
8.10 Implementation Times for Minimal System . . . . . . . . . . . . . . . . . . . . . . . 176
8.11 Static 1X Implementation Time on Various FPGA Devices . . . . . . . . . . . . . . 176
8.12 Static 1X Implementation Time Using Constraints (XC7VX980T) . . . . . . . . . . 176
8.13 Static 1X, 2X, 3X and 4X Implementation Times on XC7VX980T FPGA . . . . . . 177
8.14 Static 1X, 2X, 3X and 4X Implementation Times on XC7V980T FPGA . . . . . . . 177
8.15 Implementation Time Cost Using Hierarchical Design . . . . . . . . . . . . . . . . . 178

# List of Figures

1.1 Multi-Mode Stream Processor Example . . . . . . . . . . . . . . . . . . . . . . . . 4

1.2 Resource Capacity in Modern FPGA Devices . . . . . . . . . . . . . . . . . . . . . 5

1.3 Manufacturing Process for Virtex Device Families . . . . . . . . . . . . . . . . . . . . 7

2.1 Configuration Memory Architectures . . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.2 Slot-Based and Region-Based Partially Reconfigurable Regions . . . . . . . . . . . . 21

2.3 Multiplexer-Based Shared Bus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.4 Mesh and Fat Tree NoC Topologies . . . . . . . . . . . . . . . . . . . . . . . . . . 25

2.5 Basic Crossbar Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

2.6 Multiplexer-Based Fully Connected Crossbar . . . . . . . . . . . . . . . . . . . . . 27

3.1 Traditional and Dynamically Reconfigurable System Implementations of Multi-Modal Processing Circuits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

3.2 Fault Mitigation in a Dynamic System . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.3 Workload Example in a Video Processor . . . . . . . . . . . . . . . . . . . . . . . . 45

3.4 Examples of Workload Interference Due to ASP Generation . . . . . . . . . . . . . 47

3.5 Overlap of Processing and Configuration . . . . . . . . . . . . . . . . . . . . . . . . 48

4.1 Module Communication Methods in Region-Based Systems . . . . . . . . . . . . . 54

4.2 Effects on Static Nets Due to Bit-stream Manipulation . . . . . . . . . . . . . . . . 55

4.3 Static Net Routing For PRR Avoidance . . . . . . . . . . . . . . . . . . . . . . . . 56

4.4 Comparison of Deployment and Relocation Methods and PRR Structures . . . . . . 57

4.5 Concurrent Links in Stream Processing Systems . . . . . . . . . . . . . . . . . . . . 58

4.6 Link Contention in Mesh NoC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.7 Link Contention in Dynamic Systems . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.8 Wire Length Variation in Dynamic Systems . . . . . . . . . . . . . . . . . . . . . . . 64
4.9 Delay Variation Conversion Into Latency Variation . . . . . . . . . . . . . . . . . . . 65
4.10 Allocation Algorithms for Different Slot Configurations . . . . . . . . . . . . . . . . . 66
4.11 Preemptive Mode Change . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.12 Non-Preemptive Mode Change . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.13 Graph Representation of a Stream Processor . . . . . . . . . . . . . . . . . . . . . . 70
4.14 Example of the Canonical Connectivity Change Process . . . . . . . . . . . . . . . . 71
4.15 Canonical Connectivity Change Process . . . . . . . . . . . . . . . . . . . . . . . . 72
4.16 Seamless Mode Change Using the Canonical Process . . . . . . . . . . . . . . . . . . 73
4.17 Minimal Connectivity Change Process . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.18 Feasibility of Minimal Connectivity Change Process . . . . . . . . . . . . . . . . . . 73
4.19 Minimal Connectivity Change Process Example . . . . . . . . . . . . . . . . . . . . 74
4.20 Data-flow Synchronization in MACRO Systems . . . . . . . . . . . . . . . . . . . . 76
4.21 System-Wide Scheduling Circuit Structure and Behavior . . . . . . . . . . . . . . . . 76
4.22 Central Scheduling Circuit Structure and Behavior in Multi-Mode Scenario . . . . . 77
4.23 Distributed Scheduling Circuit Structure and Behavior . . . . . . . . . . . . . . . . . 78
4.24 Distributed Connectivity Control Concept . . . . . . . . . . . . . . . . . . . . . . . 81

5.1 MACRO System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.2 Link Types Encountered in Pipe-lined and Data-Flow Systems . . . . . . . . . . . . 87
5.3 Link Support in the MACROS Framework . . . . . . . . . . . . . . . . . . . . . . . 88
5.4 CMFU General Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.5 CMFU Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.6 CMFU Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.7 CMFU Identification Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.8 Co-Op Unit Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.9 Co-Op Unit Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
5.10 DCCI Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100

5.11 4-Port Crossbar Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
5.12 Canonical Local Connection Control Unit (LCCU) Structure . . . . . . . . . . . . . 103
5.13 Canonical LCCU Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
5.14 Minimal LCCU Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.15 Minimal LCCU Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.16 Point-to-point Link Broadcast Network . . . . . . . . . . . . . . . . . . . . . . . . 109
5.17 Aggregation Element-Based Broadcast Network: A) Multiple Aggregation Elements; B) Single Aggregation Element . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
5.18 Mode-Mapper General Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.19 Example Mode-Mapper Architectures . . . . . . . . . . . . . . . . . . . . . . . . 113
5.20 BCM General Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
5.21 Interface Between The Architecture Look-Up Engine and Allocation Engine . . . . . 116
5.22 Interface Between The Allocation Engine and Configuration Engine . . . . . . . . . 117
5.23 Mode-Specific Architecture Look-Up Table Structure . . . . . . . . . . . . . . . . . 117
5.24 Architecture Look-Up Engine General Architecture . . . . . . . . . . . . . . . . . . 118
5.25 Architecture Look-Up Engine General Behavior . . . . . . . . . . . . . . . . . . . . 119
5.26 Memory Map Structure for the Bit-stream Storage Memory . . . . . . . . . . . . . 120
5.27 Allocation Engine General Architecture . . . . . . . . . . . . . . . . . . . . . . . . 121
5.28 Allocation Engine General Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . 122
5.29 Configuration Engine General Architecture . . . . . . . . . . . . . . . . . . . . . . 123
5.30 Configuration Engine General behavior . . . . . . . . . . . . . . . . . . . . . . . . 124

6.1 MACRO System Start-Up Process . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.2 Distributed Scheduling Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
6.3 Run-Time ASP Generation Procedure - CMFU Configuration . . . . . . . . . . . . 132
6.4 Comparison of Mitigation Procedures . . . . . . . . . . . . . . . . . . . . . . . . 134
6.5 Distributed Nature of DCCI System . . . . . . . . . . . . . . . . . . . . . . . . . 135
6.6 Allocation Engine Behavior for Fault Mitigation . . . . . . . . . . . . . . . . . . . 136
6.7 Relocation-Based Mitigation Procedure . . . . . . . . . . . . . . . . . . . . . . . . 138

7.1 Experimental Setup Using an Integrated Logic Analyzer . . . . . . . . . . . . . . . 142

7.2 Spatial Masking Procedure . . . . . 144
7.3 Video Test Application . . . . . 144
7.4 Application Spatial Masks . . . . . 145
7.5 Modes for Run-time Architecture Adaptation Tests . . . . . 146
7.6 Modes for Run-time Fault Mitigation Tests . . . . . 146
7.7 MACROS Framework Test Architecture . . . . . 147
7.8 System On-Chip Layout for Test System Using Canonical LCCUs . . . . . 148
7.9 System On-Chip Layout for Test System Using Minimal LCCUs . . . . . 149
7.10 Static Test System . . . . . 150
7.11 Test Circuit Behavior for Central and Distributed Schedulers . . . . . 153
7.12 Single Instruction-Based Processor Test System . . . . . 154
7.13 Duplex Instruction-Based Processor Test System . . . . . 155
7.14 Crossbar Performance Test System . . . . . 156

8.1 Canonical Connectivity Change Procedure Timing Results . . . . . 161
8.2 Minimal Connectivity Change Procedure Timing Results . . . . . 161
8.3 Connectivity Change Time Cost for Various System Sizes . . . . . 164
8.4 Per-slot Overhead in Different Sized Systems . . . . . 165
8.5 Comparison Between Central and Distributed Control . . . . . 166
8.6 Comparison Between Central and Distributed Control in Multi-Mode System . . . . 167
8.7 Finite-State Machine (FSM) State Transition Comparison Between Central and Distributed Control in Multi-Mode Systems . . . . . 168
8.8 Control System Overhead Comparison for Various Sized Systems . . . . . 169
8.9 Fault Tolerant Control System Overhead Comparison for Various Sized Systems . . 170
8.10 Connectivity Change Time Cost Comparison Between Central and Distributed Approaches . . . . . 171
8.11 Full Versus Partial Bit-stream Component Mitigation Speedup . . . . . 180
8.12 Resource Overhead Comparison for Fault Mitigation Methods . . . . . 181

# Chapter 1

# Introduction

## 1.1 Motivation

### 1.1.1 Trends in Modern Digital System Design

#### Application Complexity

Integrated circuit manufacturing methods have advanced at a steady pace over the last 70 years, allowing for the inclusion of an ever increasing number of on-chip elements, through manufacturing method improvements and the miniaturization of component feature sizes. At the same time, applications have advanced in complexity at an even more rapid pace, such that the increased capabilities of integrated circuits (and the systems they create) have constantly been taxed to the limit.

Modern-day applications continue to exhibit the same complexity increases, in particular in the multimedia and telecommunications fields, which rely on complex real-time processing tasks, applied to high volumes of information in limited time periods (some modern applications can exhibit data rates in the hundreds of Gigabits per second). This complexity is increased further, as system requirements often impose hard constraints on size, cost, and power consumption, both in consumer-oriented systems as well as infrastructure components.

One of the ways in which modern applications have significantly increased complexity is through the incorporation of multiple modes of operation. The specifications for these applications include multiple variations in the application algorithm, designed to offer various features or trade-offs in various situations. Depending on the application, these variations in algorithm can number in the tens or hundreds, and when taken into account, they lead to extremely large, complex system implementations. Two examples are described briefly in the following paragraphs: the Digital Video Broadcasting - Satellite to Hand-held (DVB-SH) transmission standard and the H.264 Video Coding standard.

The DVB-SH standard (ETSI EN 302 583) describes a method of providing satellite communication services to hand held devices [1]. The standard specifies the framing structure of the data stream, the coding method and the modulation method. The system receives as input an Moving Pictures Experts Group (MPEG) Transport Stream which it frames, codes and then modulates for RF transmission. The standard specifies both satellite and terrestrial transmission; the terrestrial transmission path uses Orthogonal Frequency Division Multiplexing (OFDM) transmission, while the satellite path can use either OFDM or Time-Division Multiplexing (TDM) transmission [2]. Both transmission paths share a common section, however, where framing and coding takes place. This common path itself supports one of two modes of operation, based on whether a regular or low latency stream is received. The main operations performed in both cases are framing, Forward Error Correction (FEC) coding using the Third Generation Partnership Project 2 (3GPP2) turbo code [3], convolutional time interleaving, stream adaption for low latency streams and multiplexing between stream types.

In the modulation phase, each modulation scheme supports different data rates through different modulation variations. TDM transmission supports Quadrature Phase Shift Keying (QPSK), 8 Phase Shift Keying (PSK) and 16 Amplitude and Phase Shift Keying (APSK) modulation [2]. OFDM likewise supports QPSK modulation, as well as Quadrature Amplitude Modulation (QAM) 16 modulation [2]. Furthermore, the OFDM transmission mode supports 8k, 4k, 2k and 1k modes, each of which specifies a different number of carriers. By taking into account both transmission modes, the modulation variations, the different number of carriers for OFDM transmission, the emerging system exhibits a large number of variations in operating mode. The FEC coding step can be treated as a stand-alone application in its own right, and also includes multiple operating modes (see [3] for further details).

The ITU-T H.264 advanced video coding standard specifies a compression and coding technique which allows video information to be treated as general computer data, and be stored or transmitted efficiently over various mediums [4]. The H.264 compression method relies on the extraction of differences between frames to achieve high compression rates; for any sequence of frames, only a subset of frames need to be processed completely, while remaining frames are represented by their difference from one (or more) of the complete frames. Complete frames are referred to as Intra-frames (I-frames), while difference-based frames are referred to as predictive-frames (P-frames) and bi-directional frames (B-frames, so named because they use both future and past frames to compute differences).

The H.264 standard incorporates the concept of profiles and levels [4]. Different profiles specify different compression variations as well as different coding methods. These profiles vary in complexity as well as the obtainable compression rate, and are aimed at different applications. Variations between profiles include the representation of color information (for example using various YCbCr variations), the bit-width used, whether B-frames are supported or the type of coding being used. In addition, each profile can specify different levels, which refer to the maximum bit-rate a decoder must support, in terms of frame rate and frame size. Currently, 11 profiles exist; some profiles including multiple variations.

## Implementation Methods

These types of applications (high data-rate stream processing) rely on macro-function specific pipeline circuits to achieve the necessary processing rate while meeting energy consumption requirements; sequential, instruction-based processors are not generally used in these instances, as their achievable performance is limited, and entails increased power consumption (due to the use of increased processor operating frequencies). Given that a specific functionality is implemented as a specific circuit, variations in functionality are implemented as multiple circuits. A multi-mode application-specific system is usually implemented as a collection of functional units, only some of which are active at any point in time (see Figure 1.1). These systems can be very large (in terms of system gates), and very complex to implement; part of this complexity stems from the fact that all functional units needed by the system must be present in the system *at all times*, regardless of whether they are active or not.

Traditionally, these types of systems were implemented as Application-Specific Integrated Circuits (ASICs), and incurred the costs associated with this approach (manufacturing, testing). How-

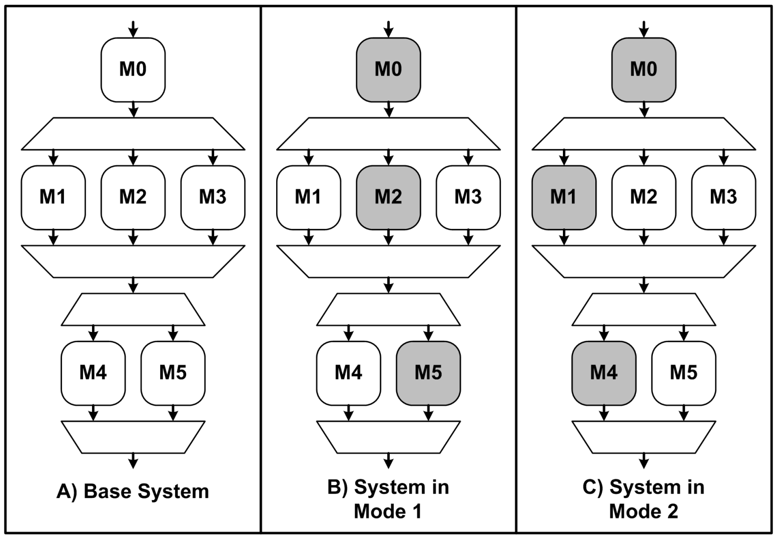
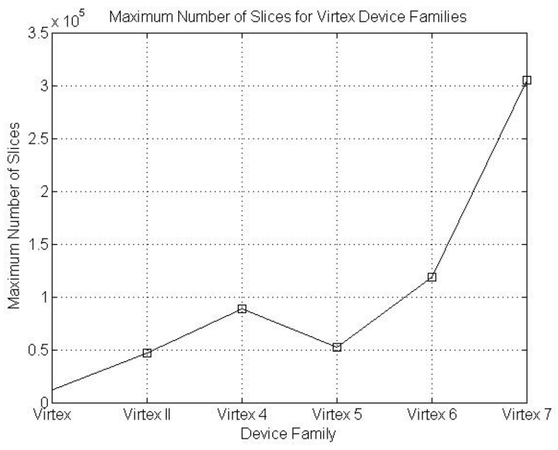


Figure 1.1: Multi-Mode Stream Processor Example

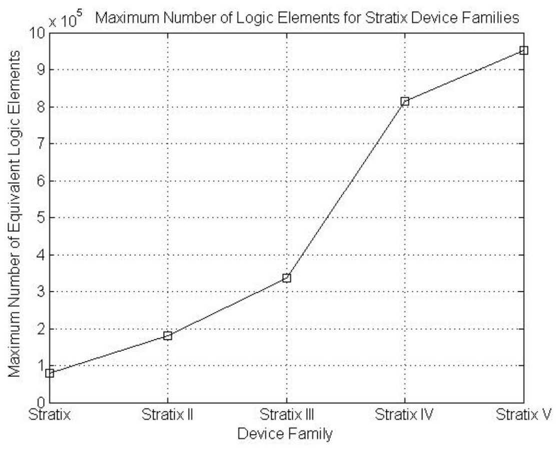
ever, the aforementioned advances in integrated circuit manufacturing technology have resulted in the emergence of a second implementation medium for digital systems. The capabilities of Programmable Logic Devices (PLDs), and in particular FPGAs have expanded in terms of the number and type of resources being integrated and the achievable system performance (logic delays and achievable operating frequencies).

The pace at which FPGAs have grown is illustrated in Figures 1.2(a) and 1.2(b), which show the progression of logic capacity in the Virtex [5, 6, 7, 8, 9, 10] and Stratix [11, 12, 13, 14, 15] device families, offered by Xilinx and Altera Corporation, respectively; Slices are listed for Virtex devices and equivalent Logic Elements are listed for Stratix devices. Both figures show an almost continuous increase when going from one family to the next; in the case of the transition from Virtex 4 to Virtex 5, while the total number of slices drops slightly, the capacity of each slice is doubled. The actual system capacity is, in fact, larger, as micro-architectures have shifted from 4-input Look-Up Table (LUT) to 6-input ones [16, 17].

The above trends look set to continue: as outlined in the International Technology Road-Map for Semiconductors, feature sizes for on-chip logic are expected to continue dropping in the next 15 years, and device integration is expected to continue increasing [18]. There is no reason to assume FPGA devices will deviate from this trend; on the contrary, the two largest FPGA manufacturers are pursuing new manufacturing methods aimed at very large scale integration. Xilinx has introduced the use of the Stacked Silicon Interconnect (SSI) in the largest of the Virtex 7 FPGAs [10], while



(a) Virtex Device Families



(b) Stratix Device Families

Figure 1.2: Resource Capacity in Modern FPGA Devices

Altera is developing FPGA architectures targeting the Intel 14 nm Tri-Gate Process [19]; both these technologies will help increase the attainable device size.

As a result, due to both general manufacturing improvements as well as improvements in micro-architecture design and implementation, modern FPGAs have gotten closer to ASIC devices in terms of accommodated performance and system size. In fact, FPGAs are now large enough to accommodate complete Systems on Chip (SoCs), and incorporate heterogeneous components such as memory blocks and dedicated multipliers to further facilitate this task. At the same time, the design and implementation of systems on FPGAs is less expensive, complex and time-consuming than ASIC implementations, due to the exclusion of manufacturing considerations; thanks to integrated Computer-Aided Design (CAD) tool-flows, a system designer can obtain a working prototype quickly using FPGA implementation. When short and medium-length production runs are considered, the reduced design and implementation cost associated with FPGAs makes up for the base cost of the FPGA itself, and has led to system designers increasingly adopting FPGAs as implementation platforms instead of ASICs.

### The Problem of Increasing Implementation Effort

The current situation can be summarized as follows: currently, FPGA devices can accommodate large systems-on-chip (exceeding 50 million system gates), and all indicators point towards a continued increase in their resource capacity. At the same time, there is no shortage of complex, high data rate applications, which are being implemented as very large, monolithic systems; because of their reduced design and development cost, a number of such systems are implemented using FPGA devices. In essence, designers are driven to implement ever larger and more complex systems, which is currently possible thanks to ever larger FPGA devices which can accommodate said systems.

However, the continued increase of FPGA capacities and the systems they accommodate has exposed a limitation of the traditional design process being used: the effort associated with implementing large designs is increasing in a non-linear fashion. The normal design flow consists of synthesis, translation, placement and routing of a given design; of these four tasks, placement and routing belong to a class of problems know as NP-hard [20], meaning that the computation time for these tasks will increase exponentially in relation to the size of the circuits being implemented. This is due in large part to the fact that design implementation is reduced to the placement and routing of mostly fine-grain elements (look-up tables and D Flip-Flops). Implementation times for modern devices range from minutes to more than a day, for systems that can reach 50 million equivalent system gates; this increase in implementation time is not linear in relation to system size, despite various attempts to optimize and improve the process [21, 22, 23].

If application complexity and FPGAs size continue to increase as they have (and all evidence indicates that they will), then the effort required to implement the resulting systems onto the available devices (which look set to reach and then exceed 100 million system gates in the near future) will become a major bottleneck in FPGA-based on-chip system design.

### 1.1.2 Manufacturing Trends and Device Resilience

#### Reduced Feature Size and Radiation-Induced Faults

As stated previously, circuit integration has continued at a relatively steady pace, resulting in the continued on-chip resource capacity increase seen in the previous section. This increase in integration has been driven in large part by a continued decrease in the minimum feature size used in manufacturing processes [24]. Figure 1.3 shows this trend represented in FPGA manufacturing, showing the feature size of the manufacturing process used for the various Virtex families of devices [5, 6, 7, 8, 9, 10]; as can be seen, the feature size has dropped rapidly, meaning that transistors in a Virtex 7 device are much smaller than those found in the original Virtex family.

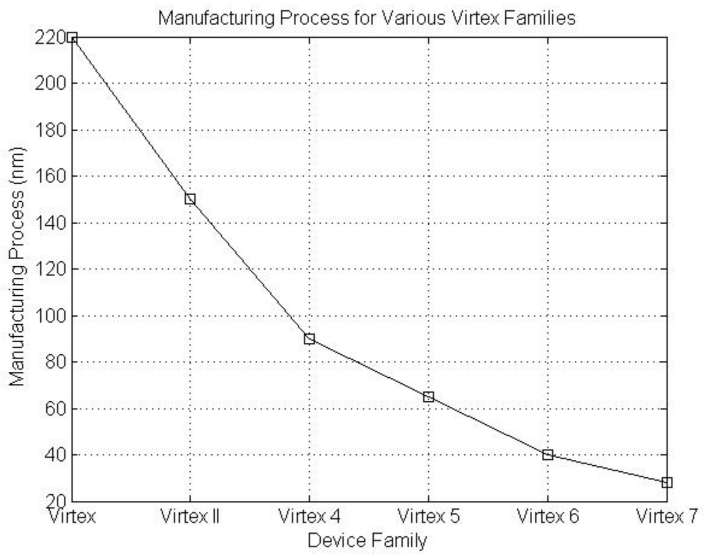


Figure 1.3: Manufacturing Process for Virtex Device Families

This reduction in transistor size has led to the creation of ever-larger devices (in terms of capacity); however, this reduction has also had less beneficial consequences. One of the fields which has seen a significant increase in FPGA adoption rates is that of space-borne computing platforms [25]. These types of systems benefit from the reduced development costs associated with FPGA system deployment, as well as the power efficiency of FPGA-based processing systems. However, these systems differ from their terrestrial counter-parts in that their operating environment is much more in-hospitable; specifically, radiation levels are much higher.

The increased radiation level increases the incidence of faults in FPGA-based systems; these faults take on both transient and permanent forms. Transient faults are caused by high-energy particle strikes to the FPGA fabric; they are referred to as Single Event Effects (SEEs) and come in the form of Single Event Upsets (SEUs) and Single Event Functional Interrupts (SEFIs) [26]. These effects can affect all resource types inside an FPGA, including configuration memory; when affecting logic and memory, their effect appears as temporary corruption of system data, and can be addressed through the use of Error-Correction Codes (ECCs). When configuration memory is affected, transient faults must be mitigated by re-writing the configuration memory of the device with the correct values, a process known as *scrubbing* [27].

Permanent faults can have multiple causes: radiation, manufacturing faults, thermal cycling. Radiation-induced faults are caused by large Total Ionizing Dose (TID) exposure [28], or single event latch-up. The thermal variation in space-borne environments can cause break-down in on-chip structures. Finally, hidden manufacturing defects can affect some portion of the device; this problem can be further exacerbated by the thermal cycling process. Permanent faults cannot be eliminated; mitigation techniques rely on avoiding faulty regions of the device, most often through module redundancy.

The continued reduction in transistor size has increased the severity of radiation-induced permanent and transient faults. High-energy particle strikes can cause Multiple-Event Upsets (MEUs), and similar total ionizing doses can result in higher fault rates in modern devices. The reduction in transistor size has placed a much heavier emphasis on fault mitigation mechanisms in these types of systems, particularly for permanent faults.

### The Problem of Increasing Fault Rates with Decreasing Transistor Size

Given the continued planned decrease in transistor sizes, the problem of radiation-induced transient and permanent faults looks set to migrate to terrestrial system. While the terrestrial environment does not exhibit the same radiation levels as orbital and inter-planetary systems, background levels of radiation nonetheless exist. This background radiation can affect on-chip structures if they are small enough; in fact, modern Dynamic Random-Access Memory (DRAM) devices already exhibit this problem. Such devices have taken advantage of improved manufacturing methods to provide increased memory capacity through reduction of the capacitive memory elements used; however, such systems have to rely on error-correction codes, as bit values are routinely affected by radiation effects [29, 30].

The continued reduction in transistor size looks set to increase the incidence of radiation-induced faults, both transient and permanent, in both space-borne and terrestrial applications. While fault-tolerant design methods do exist, they currently rely on module and system-level redundancy; taking this approach will not be sustainable as system size continues to increase, and the implementation complexity problem described above fully takes hold.

## 1.2 Modular And Dynamic Systems

As described above, current design methods prescribe the implementation of multi-mode systems (in particular processing systems using application-specific architectures) as large, monolithic structures. When implemented using FPGA devices (an approach which is becoming common) these systems are translated into large, monolithic net-lists, which must then be mapped, placed and routed [31]. These net-lists are translated into device-specific primitives such as look-up tables, D flip-flops, memory blocks, multiplier blocks [32, 33, 34]; of these, look-up tables and D flip-flops have fine granularity in relation to the total system size, and they, very often, make up the majority of a design. Because of this, mapping, placement and routing tasks must operate primarily on large numbers of fine-grain elements while meeting various constraints (area or combinational delay), which makes these tasks NP-hard in nature. Modern CAD systems attempt to avoid this problem by using various heuristic approaches (for example, Simulated Annealing [21, 22]) to find near-optimal solutions for each task without an exhaustive search. However, the heuristic methods used must still search large solution spaces due to the fine-grained nature of the building blocks used.

As established in Section 1.1.1, multi-mode systems can be logically divided into a collection of on-chip functional units. These functional units can be considered to be coarse-grained building blocks of the system; however, as described above, during the implementation process they are reduced to their component building blocks (LUTs and flip-flops) and treated as large flat collections of small elements. If the modular nature of the design could be extended into the implementation phase, then the NP-hard nature of the implementation task could be controlled such that it would result in a reduced implementation effort. Specifically, the implementation task would be applied to smaller modules of the system (individual functional units), where the problem complexity would be limited by the reduced size of the design being implemented. In essence, rather than having to address one large NP-hard optimization problem, a designer would have to deal with a linear combination of small NP-hard problems.

A second observation can be made about multi-mode systems: at any one time, only a subset of all system functional units are active, as dictated by the current operating mode, while all others are idle (see Figure 1.1 above). An ideal implementation of these systems would be one where the system always consists of only active functional units. This implies that such a system would be able to change its architecture to one of multiple variants at run-time; each architecture would correspond to a different system mode, and would consist of different combinations of Functional Units. Each mode-specific architecture would then be smaller in complexity and size than a traditional implementation containing all needed functional units.

The second potential problem afflicting modern FPGA-based digital systems is a potential increased incidence of faults (both transient and permanent) as transistor sizes continue to shrink. Transient fault can be mitigated through the use device reconfiguration (scrubbing); however, permanent faults cannot be mitigated as easily. The current defense against permanent faults is the use of various levels of redundancy; however, this approach relies on the addition of a large resource overhead to a system, and further exacerbates the problem of implementation complexity and effort.

A potential alternative mitigation solution to permanent system faults can be arrived at by extending the idea of run-time changing system architecture described above. If architectural differences could be extended to include the physical location of functional units inside a target device, then permanent faults could be mitigated through architecture changes, such that the faulty region of the device could be avoided.

These three theoretical ideas can be unified into the concept of a Modular and Dynamic System. Such a system would be implemented not as a single, monolithic design but as a collection of functional units; mapping, placement and routing activities would be applied to functional units as opposed to complete systems. These functional units would then be assembled and integrated into an ASP at run-time. Changes in mode would be reflected by changes in the combination of functional units present in the system, and the ASP being implemented. Permanent faults would be mitigated through *re-location* of functional units to different regions of the same device. Such a system would lead to reduced implementation effort due to the fact that the implementation is itself modular and limits the mapping, placement and routing tasks to smaller portions of the design (functional units). Furthermore, the system would exhibit increased cost-efficiency by only housing active functional units. Finally, the system would offer increased fault tolerance, in particular to permanent faults which are more difficult to address.

The current discussion is aimed at FPGA-based systems; conventional implementations using

FPGA devices employ device configuration as a simple start-up step, to be performed only once. However, the field-programmable nature of these devices means that configuration activities can be integrated into run-time system behavior; this, in turn, could lead to a dynamic system architecture, as described above. This approach would be limited in large part by the configuration time needed to program the FPGA being used (which may be prohibitively large). However, a number of modern FPGA devices support a feature called dynamic partial reconfiguration [35], which allows portions of the configuration memory to be written, while the rest of the device continues to operate normally; this capability can reduce the impact of configuration activities by limiting the amount of data being configured at any time. Thus, Modular and Dynamic Systems implemented using Dynamic Partially Reconfigurable FPGAs could offer a solution to the two problems introduced above.

## 1.3 Research Objective

Based on the above preamble, the objective of the presented research was the development of a framework for the creation of Modular and Dynamic FPGA-based digital processing systems; the modular and dynamic nature of these systems would be achieved through run-time integration of dynamic partial reconfiguration activities. Such systems would permit modular implementation of macro-function oriented functional units, run-time generation of mode-specific architectures through assembly and integration of said functional units with a dedicated on-chip infrastructure and fault mitigation capabilities via on-chip functional unit relocation.

The main emphasis of the framework being introduced was the development of high data-rate stream processing systems; as described in previous sections, stream processing applications are becoming increasingly prevalent. More importantly, streaming applications (in particular high data rate ones) impose strict requirements on the system implementation; these requirements must be taken into consideration when implementing architecture adaptation procedures, lest these procedures interfere with and degrade the performance of such systems. Thus, the method being proposed was built from the ground up to address stream processing tasks. Having said this, the same design method is applicable to non-streaming applications, where the operating constraints may be different or more lax.

To accomplish the above objective, a number of uncertainties were identified which led to topics of research. These topics were investigated, and the results of these investigations were then combined into the framework being presented. Both the uncertainties and their associated topics of research are listed below.

**Uncertainty 1:** What mechanism is used for the on-chip allocation and relocation of functional units, and what are the characteristics of this mechanism?

**Topic 1:** Research into functional unit allocation and relocation mechanisms in dynamic partially reconfigurable FPGAs. Emphasis is placed on analysis of Dynamic Partial Reconfiguration (DPR) support in modern FPGAs, and mechanisms based on DPR to support functional unit allocation and relocation.

**Uncertainty 2:** How can on-chip functional units communicate, and what impact will the dynamic nature of the system have on said communications?

**Topic 2:** Research into available on-chip communication architectures. Emphasis is placed on support for high data-rate communications and the impact of dynamic architecture changes on communication activities.

**Uncertainty 3:** How will a complete ASP be assembled from individual functional units?

**Topic 3:** Research into processor on-chip assembly procedures and mechanisms using dynamic functional units. Emphasis is placed on communication link establishment in response to changing system modes.

**Uncertainty 4:** Once the application-specific processor is assembled (i.e. is structurally complete) how will individual functional units (with individual behavior) be integrated into a *working* processor.

**Topic 4:** Research into functional unit integration procedures and mechanisms. Emphasis placed on scheduling and synchronization of processing activities.

The topic of application analysis and segmentation into functional units was not one of the objectives of the presented research; this topic has received attention in recent years and a number of research works and solutions have been proposed (e.g. [36], [37]). The presented research assumes the use of a separate mechanism for application segmentation and functional unit generation. Likewise, fault detection was *not* part of the proposed research objectives, as a significant body of work already exists addressing this topic [38].

## 1.4 Contribution

The main contributions of the presented work are the identification of the two main problems described in Section 1.1 (those being increasing implementation complexity for large systems and a potential increase in the incidence of faults in these systems), the proposal of a potential approach to solving these problems for a class of applications (stream processing), the identification of mechanisms which can support this approach, and their investigation. This investigation consisted of the implementation of tests systems, and the collection of results; these results verified that the proposed approach A) can address the problems initially identified and B) can do so more effectively, for the given application class, than existing approaches.

The result of this research work is the proposal of the MACROS Framework, which specifies the architecture and behavior for a class of run-time reconfigurable stream processors. The systems are considered Multi-mode Adaptive due to their ability to adapt their on-chip architecture in response to changes in algorithm, data structure or constraints associated with a given application [39, 40]. They are Reconfigurable because they make use of the configuration capabilities of modern FPGA devices to implement on-chip architecture adaptation [41, 42]. Finally, the systems are referred to as Collaborative and self-Organized because they implement a distributed assembly and integration mechanism [43]; the mechanism functionality is divided between system functional units and a distributed control infrastructure. In this way, the number of central points of failure in the system is reduced [44, 45].

Systems built using the MACROS Framework have the ability to run-time self-assemble and self-integrated into mode-dependent application-specific processors using a library of macro-operation functional units (derived from pre-existing Intellectual Property (IP) Cores) and a dedicated on-chip infrastructure which supports these assembly and integration operations. Run-time mode changes are reflected in the system architecture through changes in the distribution of functional units present and their interconnection. A second type of architecture change is also supported, whereby the physical on-chip location of functional units can change without affecting the functionality of the ASP being implemented.

The presented framework offers the following four primary benefits:

1. By supporting run-time assembly and integration of complete processors from parts (functional units and an infrastructure), the proposed framework allows for the *modular implementation* of a given system. In this way, the problem of increasing implementation effort can be mitigated by imposing limits on the complexity of mapping, placement and routing tasks; these tasks are applied to individual functional units and the on-chip infrastructure, as opposed to complete systems.

2. By supporting run-time architecture changes through functional unit allocation in response to mode changes, the proposed framework supports run-time architecture-to-task adaptation. This feature allows for improved power and cost efficiency when implementing complex stream processing systems.

3. By supporting functional unit relocation without affecting processor functionality, the proposed framework offers a fault mitigation mechanism which can address the increased incidence of both transient and *permanent* faults in modern FPGA-based systems. Secondly, by making functional unit relocation the primary repair mechanism in all fault cases, the repair time can be controlled and kept constant, regardless of the fault type encountered.

4. By explicitly targeting streaming applications, the proposed framework offers a method for the integration of run-time architecture adaptation operations into stream processing workloads while minimizing or eliminating the impact of these operations, thus allowing for *seamless* changes in system architecture with no interruption in processing tasks.

## 1.5 Thesis Document Structure

The remainder of the document is organized as described below:

**Chapter 2** will present background material and related works addressing the research topics identified in Section 1.3. This chapter constitutes the first step in investigation process which will result in the selection of mechanisms for the MACROS Framework.

**Chapter 3** will present the basic principle of operation of MACROS Framework systems. The basic behavior identified in this chapter will then act as the specification for the class of systems defined by the proposed framework. Based on analysis of the presented behavior, fundamental characteristics and performance metrics for this system class are also established.

**Chapter 4** will present a comparative analysis of mechanisms in the four topics of research identified in Chapter 1. Using the material presented in Chapters 2 and 3, the analysis of Chapter 4 will result in a selection of specific mechanisms which will be integrated into the framework.

**Chapter 5** will present the full MACROS framework architectural template. The presented architecture will reflect the analysis and selection made in Chapter 4.

**Chapter 6** will describe the run-time behavior specified by the MACROS framework, consisting of run-time system assembly, mode-specific ASP generation and run-time fault repair procedures via functional unit relocation.

**Chapter 7** will describe the test procedures used to verify and analyzed the proposed design method, including the structure of constructed test systems and the measurement procedures used.

**Chapter 8** will present the results obtained, including performance and resource characteristics, as well as associated implementation effort. Where possible, these results yielded by the proposed design approach will be compared with existing approaches in the field. Chapter 8 constitutes the final step of the investigation process into the topics of research identified in Chapter 1, by validating the mechanism selection made in Chapter 4 through experimental results.

**Chapter 9** will conclude the thesis.

# Chapter 2

# Background and Related Work

## 2.1 Introduction

Having established a number of research topics in need of investigation as prerequisites to the chosen goal - a design method for digital processors with run-time adaptive architectures based on dynamic partially reconfigurable FPGAs - the first step in the investigation process is survey and analysis of existing work in the given field. This chapter will tackle the first of these two items, by presenting background information and surveys of existing works for the four topics identified in Chapter 1.

The chapter will begin with an analysis of mechanisms for functional unit deployment and relocation in modern FPGA devices. Following this, various architectures for on-chip communications will be presented; in each case, the basic principle of operation will be identified. Third, mechanisms for on-chip system assembly will be analyzed; the topics being addressed are configuration control and system link establishment. Once the topic of system assembly is covered, the related topic of system integration is addressed, paying particular attention to scheduling of processing activities and data synchronization.

Finally, the topic of Fault Tolerance methods using dynamic partial reconfiguration will be discussed. This form of fault tolerance actively relies on some or all of the other mechanisms identified above. Thus, a system that can support functional unit deployment and relocation, run-time assembly and integration can automatically support some forms of fault mitigation using these capabilities. However, existing research works have traditionally addressed Fault Tolerance capabilities in such systems as a separate topic; thus, as a reflexion of this trend, they are presented in a separate, stand-alone section.

## 2.2 Mechanisms for Functional Unit Deployment and Relocation in Dynamically Reconfigurable FPGAs

Dynamic Partially Reconfigurable FPGAs allow run-time deployment of functional units via write procedures to portions of their configuration memory. The characteristics and limitations of this process must be established, given that this feature is the corner-stone of the proposed design method. For the systems being proposed, free functional unit deployment is desired, as it increases the flexibility of the system and allows more efficient utilization of system resources. Furthermore, functional unit relocation is a form of free deployment, in that the same functional unit is re-deployed to a different region of the device. As such, the topics of free functional unit placement (the ability to deploy a functional unit to one of multiple on-chip locations) and relocation (the ability to move a deployed functional unit from one on-chip location to another) must also be investigated.

### 2.2.1 Dynamic Partial Reconfiguration in Modern FPGAs

Modern FPGA devices incorporate a large number of configurable logic resources (most often implemented using look-up tables which act as programmable logic functions), augmented by additional types of resources: memory blocks, dedicated multiplication and adder blocks, input/output blocks, clock generation blocks and others [32, 33, 34, 46, 47]. In addition, an FPGA contains routing resources and configuration memory used to store the configuration information for all programmable resources. To accomplish a given circuit, FPGA basic blocks are configured and connected together to achieve the desired functionality. A given circuit will result in a global configuration for a given FPGA; the needed configuration information is stored in a file, commonly referred to as a configuration bit-stream.

FPGAs can be categorized according to the type of configuration memory they use; the two categories are based on the use of volatile (e.g. [10, 15]) and non-volatile (e.g. [48]) configuration memory. Given that the inclusion of non-volatile memory requires specialized manufacturing processes, the majority of medium and large FPGAs currently available make use of volatile memory. These devices loose their configuration information when not powered, and require an initial configuration step prior to regular operation when powered on. To accommodate this step, such devices incorporate a number of configuration interfaces, including serial (e.g. Joint Test Action Group (JTAG) port, serial port) and parallel ones (e.g. [49]), all connected to a configuration controller. Corrupted bit-streams can lead to incorrect circuit structure, meaning that the implemented circuits either do not meet the original specification, or can outright damage the device. To avoid this, modern devices make use of Cyclic Redundancy Check (CRC) codes to ensure that the bit-stream structure remains intact as it is transferred to the configuration circuitry [49].

The principle of DPR is an extension of regular configuration support. The principle states that configuration memory can be made addressable (with a certain minimum addressable word), and portions of the configuration memory (and, therefore, the implemented functionality) *can be changed while the rest of the device continues to operate.* In this way, portions of the circuit structure can be changed at run-time, and yield run-time adaptive architecture changes. This feature can be extended further through the inclusion of an internal configuration access port on-chip (e.g. ICAP [49]); in this way, an FPGA device now has the ability to change its own architecture at run-time by accessing its own configuration memory.

The partial reconfiguration capabilities of a given device are defined in large part by the minimum addressable configuration memory element of the device architecture. At the fine-grain end of the scale, certain devices allow bit-streams to address the configuration memory for single logic blocks; however, this approach leads to high addressing overhead for the configuration memory, and can become prohibitively expensive as the device size increases. At the coarse-grain end of the scale are column based devices (Virtex II [6]); in these devices, the minimum configuration memory addressable element controls a column of resources spanning the full height of the device (and some fixed width, see Figure 2.1 - A). This approach reduces addressing overhead, but at the cost of functional unit allocation flexibility.

A third approach lies in the middle, and has been adopted in a large number of modern FPGA families (Virtex 4, 5, 6 and 7 Series [7, 8, 9, 10]): tile-based configuration memories. In these devices, the minimum addressable configuration memory element is a tile, which controls a column of resources; however, this column does not span the full length of the FPGA (see Figure 2.1 -

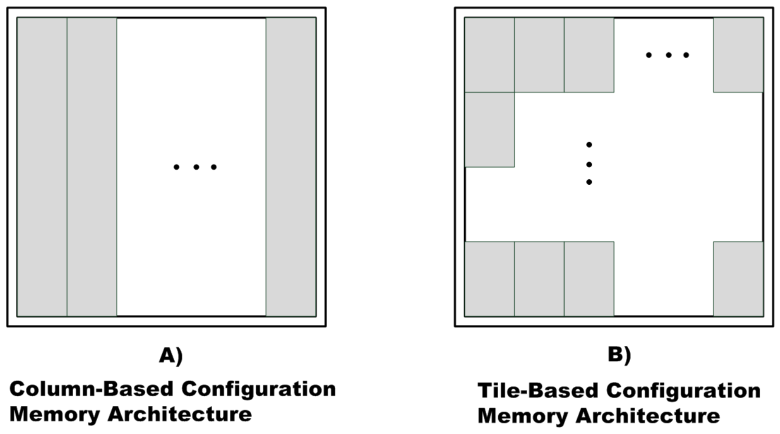

A)
Column-Based Configuration Memory Architecture
B)
Tile-Based Configuration Memory Architecture

Figure 2.1: Configuration Memory Architectures

B). In such devices, individual tiles contain a specific type of configurable resource (logic blocks, memories, multiply-accumulate units). Given the prevalence of tile-based architectures in modern devices, the remainder of this work will address exclusively tile-based architectures.

Beyond the basic considerations discussed above, the capabilities of dynamic partially reconfigurable FPGAs are manufacturer- and family-dependent. For the remainder of this document, all discussions will use as example Xilinx modern FPGA families (Virtex 4, 5, 6 and 7-series families [7, 8, 9, 10]). The reason for this is the fact that, for a long period of time, including the time when the presented research was first began, Xilinx devices were the only ones to support dynamic partial reconfiguration. However, the presented design method, and the class of systems it leads to can be implemented using any FPGA device which supports free functional unit placement and relocation (for example, the Stratix V FPGA [15]).

A dynamic partially reconfigurable system is logically and structurally divided into two parts: a static portion of the system, and dynamic portions (i.e. those that change via the loading of partial bit-streams at run-time). Certain infrastructure elements must always be in the static portion of the system, such as clock generation elements [35]; clock distribution to dynamic partially reconfigurable regions is controlled by the DPR CAD tool-chain, as it is dependent on device micro-architecture. Likewise, Input/Output (I/O) blocks are often considered static in nature, given that they exhibit fixed connections at the system level. Finally, if on-chip functional units are used to perform configuration activities (using an on-chip configuration access port), then these units must be static (they cannot change their own architecture). The remainder of a given device can be segmented into one or more *Partially Reconfigurable Regions (PRRs)* [35]; these region are reserved for the housing of dynamic elements of the system, and are targeted by partial bit-streams. Dynamic functional units housed in these PRRs are referred to as *Partially Reconfigurable Modules (PRMs)* [35].

## 2.2.2 Free Functional Unit Placement and Relocation

The structure of PRRs being deployed, as well as the method through which functional units are placed and relocated in these regions define the capabilities of a system, and impose requirements on its structure. The PRRs structure of a system can follow one of two approaches, as described below:

**Slot-based approach:** The slot-based system is the simplest conceptually: a set of partially reconfigurable regions inside the target FPGA are reserved as *system slots* at design time. Each system slot accommodates *only one* functional unit, regardless of the actual size of the unit; thus, the number of slots in the system must be equal to the maximum number of concurrent functional units. Each slot contains a set amount of various resources, and slots must be allocated such that all functional units fit in all slots (meaning that the available type and amount of resources per slot are enough for each module type). This system arrangement is shown in Figure 2.2 - A.

**Region-based approach:** The region-based approach reserves one or more large (usually rectangular) region inside the target FPGA. These regions will house all system functional units, with *more than one unit* housed in each region. In this approach, functional unit placement is dictated by the minimum addressable element of the target FPGA micro-architecture, the distribution of resources in the region and its boundaries. This type of functional unit deployment is shown in Figure 2.2 - B.

Likewise, the allocation and relocation of functional units to PRR can be implemented in one of two ways:

**Separate Bit-streams:** Each functional unit net-list is mapped to each slot region in the system, and bit-streams are generated for each mapping. This leads to $numberofslots \times$

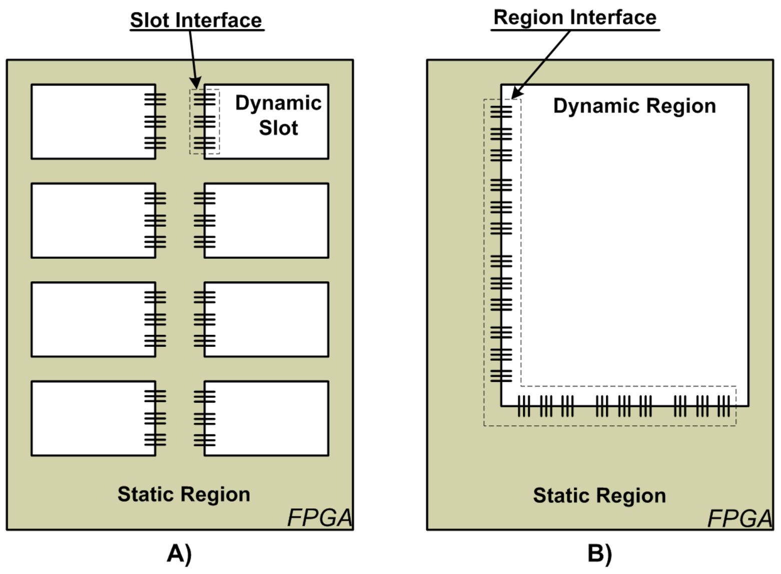


Figure 2.2: Slot-Based and Region-Based Partially Reconfigurable Regions

$numberoffunctionalunits$ bit-streams.

**Bit-Stream Manipulation:** Only one bit-stream is created per functional unit, mapped to a target region which can accommodate the given unit (assuming the right combination of resources is available). This base bit-stream is then manipulated at run-time, prior to being downloaded to the target FPGA, so that it targets a different region of the device.

Of the above combinations of PRR structures and allocation and relocation mechanisms, the simplest to implement is a slot-based approach using multiple separate bit-streams. This approach is currently directly supported by DPR CAD tool-chains [35]. However, this approach has two primary limitations: a large number of partial bit-streams must be stored, and the fixed nature of slots may lead to internal fragmentation and wasted resources. Multiple slot-based reconfigurable computing architectures have been proposed, including [50, 51, 52, 53, 54, 55, 56]

Because of the above limitations associated with slot-based PRRs and the use of separate bit-streams, mechanisms oriented towards region-based PRRs have been pursued; these mechanisms rely on bit-stream manipulation for both allocation and relocation. A method for functional unit allocation and relocation in column-oriented devices is presented in [57], targeting Virtex E devices; the proposed approach allows communication between modules through shared buses embedded in each unit, using tri-state buffers. An updated communication method is presented in [58], based on shift register embedded in deployed functional units. A column-oriented relocation method is presented in [59]; the method relies on bit-stream manipulation and can relocate full column bit-streams, as well as portions of a column via merging of target and destination bit-streams.

The authors of [60, 61, 62] present a method for both one and two dimensional placement and relocation in Virtex II devices which relies on bit-stream read-back and modification. To permit communication between modules, routers must be embedded inside functional unit logic, and specialized routing components must be used inside each column. A second placement and relocation method aimed at columnar (Virtex II) devices is presented in [63]; the method relies on the use of reserved routing for static communication signals in dynamic functional units. The proposed method relies on read-back of a partial bit-stream (targeting a specific region of the FPGA), merging with a new destination bit-stream (targeting a new region of the FPGA, which may already contain circuitry) and finally write-back; while it is aimed at columnar (Virtex II) devices, it can also be applied to tile-based (Virtex 4) devices.

A functional unit relocation method is presented in [64]; the process used relies on altering partial bit-stream structure (primarily the Frame Address Register (FAR)) to target different regions of the device; the proposed method is aimed at columnar devices such as the Virtex II. A second relocation method based on bit-stream manipulation is presented in [65]; this method allows functional unit relocation in tile-based FPGAs such as the Virtex 4 family of devices.

When undertaking relocation via bit-stream manipulation (in particular changes to the FAR register), the source and destination regions must be identical in structure (both in terms of logic blocks as well as routing resources); this implies a homogeneous structure for the device. However, modern devices are not homogeneous in structure, containing logic, memory and dedicated circuitry (multipliers, I/O blocks, clock generators and others, as seen in, e.g. [7, 8]). In [66], a relocation method aimed at tiled devices is presented; the method permits relocation of functional units of a pre-determined size, which incorporate communication buses. To allow for the heterogeneous nature of modern FPGA, the authors define reconfigurable regions only over pure logic sections (thus excluding elements like memory and multipliers). An alternative solution to this problem is presented in [67]; here, the authors propose to classify functional units by the types of resources they incorporate (memory, logic, etc.) and only allow relocation of a unit to a region which contains the same types of tiles, in the same arrangement. The authors also describe a communication method based on embedded macros, which are incorporated into the structure of each unit (reminiscent of previous works).

In the above works, communication between freely-placed and relocated functional units relied on the inclusion of dedicated infrastructure elements into individual functional units. An alternative approach is presented in [68, 69]; here, functional units incorporate memory blocks which are used to store input and output data. To extract this information, configuration memory read-back is used to read the data stored in Random-Access Memory (RAM) modules; this same approach can then be used to implement communication between modules by injecting data into component bit-streams prior to loading them to the device. These tasks are performed by a dedicated communication controller and must be performed sequentially, as only one configuration interface is available.

## 2.3 On-Chip Communication Architectures for Dynamic Partially Reconfigurable Systems

Once allocated or relocated, functional units are expected to communicate in order to implement the functionality of a given application. As such, an investigation of existing on-chip communication infrastructures must be undertaken. The main aspects being considered for each architecture are the achievable performance and associated cost.

### 2.3.1 Shared Buses

The shared bus is the most basic method of allowing multiple functional units to communicate. The underlying principle of operation is to use a shared communication medium (a bus in this case), and to arbitrate access to this medium. Figure 2.3 shows the basic structure of a bus-based system using multiplexers. All communication links in shared buses are multiplexed in time; a priority scheme can be used when arbitrating access, such that certain functional units are given higher priority.

Shared bus systems have found popularity in architectures aimed at dynamic partial reconfiguration. One of the earliest examples of a self-reconfigurable system [70, 71] relies on a shared bus architecture. The dynamically reconfigurable architecture presented in [72, 52] likewise relies on a shared bus [73] to implement communication between functional units. A third example

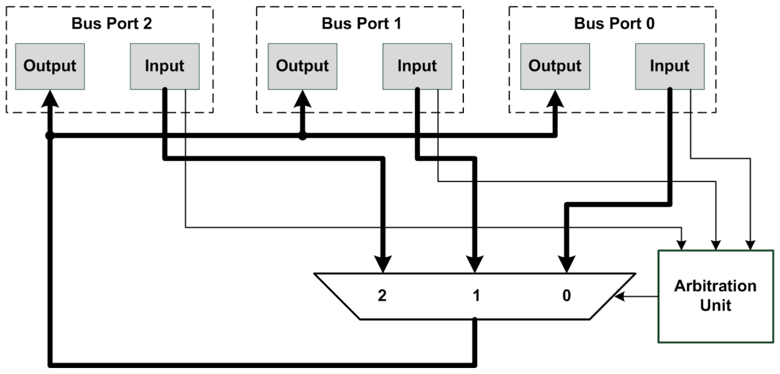


Figure 2.3: Multiplexer-Based Shared Bus

of an architecture using shared bus communications is presented in [74, 75]. Additional examples of dynamically reconfigurable on-chip architectures using shared bus systems can be found in [76, 77, 78, 79, 80, 56].

Shared bus architectures have also been applied to multi-device (cluster) architectures; here, the shared bus serves as both chip level and system level communication architecture. Examples of this approach include [81, 82, 83].

### 2.3.2 Network-on-Chip

NoC are a relatively recent option when considering on-chip communication architectures. The NoC concept emerged as a solution for on-chip communications in Multi-Processor Systems on Chip (MPSoCs) [84]; the aim was to provide an alternative to the shared bus approach, which was found to be a performance bottleneck in systems which relied on inter-processor communications. At their core, NoC are composed of switches and links; links can connect two switches or a switch and a system functional unit and, along with switch ports, usually support full duplex communication. Figure 2.4 shows two topology types often found in NoC systems: the mesh and the fat tree; each topology dictates a certain connection pattern between switches and functional units.

The main defining aspect of a NoC is the method by which data is transferred through the network. The two main approaches on offer are: circuit-switched and packet-switched systems [84]. In a circuit-switched network, a *circuit* is established between a source and a destination, consisting of reserved links and switch ports. Once data transfer completes, this circuit can be released, meaning that all resources previously used become available for other circuits. The alternative

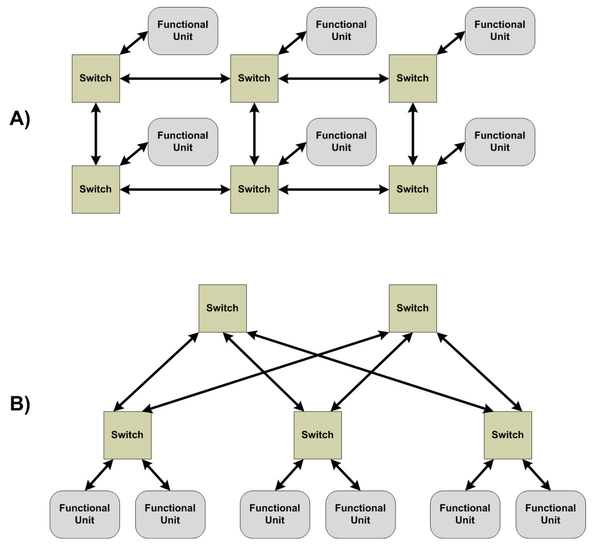


Figure 2.4: Mesh and Fat Tree NoC Topologies

approach is to transfer data as individual *packets*, each of which is routed through the network from the source to the destination. In packet-switched systems, data must be converted to packet form, which usually involves the addition of headers (containing the source and destination address, as well as routing tables in some instances); this task is accomplished by *network interfaces* connected directly with processing elements.

A number of research efforts have been dedicated towards taking the NoC concept and adapting it to DPR design. A packet switched NoC architecture is presented in [85, 86, 87]; the proposed architecture implements linear or two-layer mesh architectures (four links per router, one to the processing element). The structure of the network itself can be changed (through the addition of routers) via partial reconfiguration. Similarly, a packet switched NoC architecture is presented in [88, 89], based on the SpaceWire protocol. Finally, a packet-switched NoC based on fat tree architectures is presented in [90].

Packet switched architectures are, by definition, best-effort in nature; communication data-rates are usually not guaranteed due to arbitration and traffic contention. To by-pass this problem, circuit switched and hybrid NoC combinations have also been proposed. A circuit-switched architecture is presented in [91]; here, circuits are established via configuration of the routers themselves. An NoC with hybrid routing system is presented in [92]; both packed switched and time-slot based circuit-switched communications are supported. Likewise, [93] presents a combination NoC using

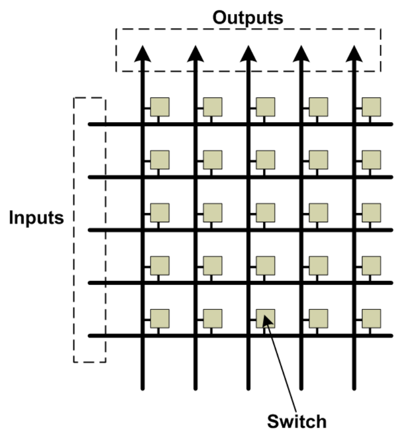


Figure 2.5: Basic Crossbar Structure

packet switched communication for control purposes and circuit-switched communication for data transfer; circuits are established via reconfiguration of dedicated routers. Finally, the Star-Wheels NoC architecture makes use of packet switched control communication and circuit-switched data transfer [94]; this architecture was further updated to support packet-switched communication in cases where circuits cannot be established due to deadlock [95].

### 2.3.3 Fully-Connected Crossbar

Unlike the NoC concept, the fully connected crossbar has existed in some form for many years; this types of communication architecture has been used in telecommunication systems [96] as well as computing architectures. The basic structure of a 1-bit crossbar is shown in Figure 2.5: each input can be connected to any of the outputs. The most important feature of the fully connected crossbar is the fact that it is *non-blocking*, meaning that links between separate input-output pairs will in no way affect each other. A diagram of a multi-bit fully-connected crossbar, based on multiplexers, is shown in Figure 2.6. The crossbar consists of inputs and outputs, with each output driven by a multiplexer; via this multiplexer, any input can be connected to any output.

A number of crossbar and crossbar-like architectures have been deployed in dynamic partially reconfigurable systems. Not all architectures presented here are strict crossbars, but all offer *non-blocking communications* the most fundamental feature of a crossbar. The Erlangen Slot Machine [50, 51] is an example of an architecture using crossbar or crossbar-like communication architectures; the system includes an external programmable crossbar, and can also use an on-chip crossbar-like

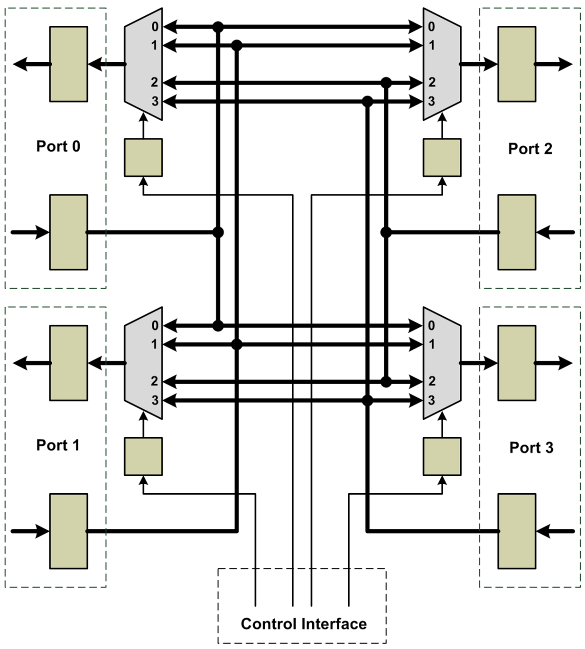


Figure 2.6: Multiplexer-Based Fully Connected Crossbar

architecture [97] which implements multiple parallel buses connected through switches, each of which can connect two functional units directly. A fully connected network architecture is presented in [98]; the architecture allows processing elements to connect directly to all other processing elements via multiple communication channels in a non-blocking fashion, achieving the same link pattern as a fully-connected crossbar.

A crossbar architecture referred to as an *IO Bar* is presented in [99]; this communication infrastructure is used for high data rate communications in video processing systems. A multi-stage non-blocking communication architecture is presented in [100]; the system supports both uni-cast and multi-cast communications via a multi-stage switching architecture. A Morphing Crossbar architecture is presented in [101], which allows reconfigurable functional units to be connected to different I/O pins as the system-level connectivity changes.

### 2.3.4 Additional Communication Architectures

The three communication architectures presented above represent the most common approaches to functional unit communication in modern SoC. However other approaches to on-chip communications do exist. A potential approach is the use of dedicated point-to-point links: the COMMA method presented in [102, 103] implements point-to-point links between functional units mapped to reserved system slots; similarly, [104, 105] present examples of systems where dedicated processing elements use point-to-point links for communication purposes.

Another potential approach to on-chip communication is through the use of shared memories. An example of this approach can be found in the Erlangen Slot Machine [50, 51] architecture, where adjacent modules utilize external Static Random-Access Memory (SRAM) interfaces for communication purposes. Likewise, [106] presents an architecture where functional units communicate via a multi-port memory controller, as well as a dedicated mail-box system for short messages.

## 2.4 Board and System-Level Run-Time Assembly and Integration Methods

A number of functional unit run-time assembly and integration mechanisms exist at the board and system levels; at this level, functional units consists of either complete single devices (FPGAs, ASICs, instruction-based processors) or collections of such devices, assembled into boards. These mechanisms make possible *hot plug* and *hot swap* procedures, whereby functional units are added to the system or replaced while the system is running. In these instances, the deployment mechanisms used is a human operator, who adds or removes the functional unit in question from the system. Given that the topic of on-chip functional unit assembly and integration is being considered, an examination of existing system-level solutions may be beneficial.

Two examples of communication interfaces which support such hot-plug and hot-swap procedures are the Peripheral Component Interconnect (PCI) Express computer bus [107] and the Universal Serial Bus (USB) [108]. Both interfaces have been designed for deployment in a wide variety of systems, and have seen wide-spread adoption. These interfaces are primarily used to connect I/O and storage functional units to central processing systems, such as various network interfaces, non-volatile storage, and human input and output interfaces (mice, keyboards, video and sound interfaces). Thus, in most instances, the functional units connected through these interfaces consist of one or more boards containing multiple devices.

The PCI Express interface is considered a third generation peripheral interface [107], and differs from previous (first and second generation) peripheral interfaces in that it eschews the shared bus topology for a network structure. A PCI Express system consists of a Root Complex, a number of

Switches and multiple endpoints [107]. The root complex acts as the interface between the network and system processors which issue various transactions; transactions are converted into packets by the root complex and sent to various endpoints on behalf of the Central Processing Unit (CPU). System switches forward packets between their various ports; finally, endpoint devices respond to various packet types by undertaking a specific operation.

The Universal Serial Bus (USB) architecture follows a similar approach, in that it can be seen as a networked structure. A USB system consists of a USB Host, a number of USB Hubs and finally a number of USB Peripheral Devices [108]. Communication is accomplished via the transfer of packets between the host and various peripheral devices, with the hubs acting as intermediate connection points for peripheral devices. Thus, a USB system has a tiered star topology, with the host at the center of the first tier, and hubs at the center of secondary tiers.

The behavior of both interfaces is divided into multiple layers, including a physical layer, a link layer, and a transport layer [107, 108]. Thus, the hot-plug and hot-swap processes consist of activities at each of these layers. At the physical layer, power distribution and link training [108] tasks must be addressed. At the link and transport layers, functional units must be identified and, possibly, configured prior to their utilization [107, 108]. The identification process often consists of the construction of data structures needed by the instruction-based processors which use these interfaces to communicate with various peripherals.

A number of observations can be made which apply to both the USB and PCI Express interfaces, which must be taken into consideration if either of these interfaces are to be used to inform on-chip procedures. The first is that both these interfaces and their associated protocols assume a certain minimum level of complexity from the functional units they interact with. Both interfaces rely on complex protocols, hence the separation of said behavior into layers, and all functional units which form the system, be they endpoints (PCI Express) or peripheral devices (USB) must support these protocols. This means that functional units must incorporate complex behavioral components (often smaller processors or dedicated ASICs) as well as storage (for incoming packets) to interact with these interfaces, in addition to any other components needed for their functionality.

Secondly, both architectures rely on central elements and control points, in the form of the Root Complex and the USB Host Controller. The USB specification explicitly addresses only transactions between peripheral devices and the host controller [108]. PCI Express, due to its network-oriented nature, allows communications between any two endpoints, as well as endpoints and the root complex [107]; however, the majority of existing PCI Express systems exhibit only communication between endpoints and the root complex (on behalf of a controlling central processing unit). Thus, both communication architectures are aimed towards certain communication patterns, including the flow of control information (used for system configuration during the hot plug or hot swap processes); this may impose limitations on the type of applications being supported.

The third item to note is that many system-level assembly and integration mechanisms are inherently slow in their operation; as mentioned above, functional unit deployment is accomplished by a human operator. Thus, the timing parameters are much more relaxed, and complex protocols can be used for link establishment; this timing overhead, along with the existing resource overhead (discussed above) must be taken into consideration. Finally, both communication architectures considered here use packet-based communication methods. Using such an approach, control information (such as packet headers) has a similar temporal distribution to data, in that control information is sent with every packet. In a high data-rate environment, control information is expected to propagate through the system at a much slower rate than the data being processed; in effect, control information is needed only when some aspect of the system structure (functional unit distribution, communication patterns) changes.

## 2.5 Mechanisms for On-Chip Processor Assembly

In a dynamically reconfigurable system, the assembly process consists of two main activities: allocation of functional units through configuration activities and establishment of links between the functional units present in the system. Each of these two activities will be analyzed in turn in this section.

Functional Unit allocation via configuration is a known quantity, in that the procedure itself follows a well defined algorithm, as defined in documentation for the chosen FPGA (for example, in a device configuration guide [49]). Thus, the open questions for the allocation process are: A) which element of the system is responsible for configuration activities? B) what process is used to determine when configuration activities are undertaken?

### 2.5.1 Configuration Control

The most popular method of undertaking configuration activities, particularly in self-reconfigurable systems, is through the use of a sequential, instruction-based processor acting as the control mechanism; this processor will either interact directly with the on-chip configuration interface, or it will rely on a secondary dedicated circuit, aimed at accelerating the process. This approach has been applied in architectures and systems aimed at general computing [70, 71, 72, 52, 74, 75], MPSoCs [109, 110, 111, 112, 113], High-Performance Computing (HPC) [106, 80] and video and multimedia-oriented systems [76, 114, 53, 56, 77, 78, 104, 105, 99, 115]. In these architectures, the configuration controller is integrated on-chip with the rest of the system; thus, the FPGA device used can self-reconfigure.

An alternative approach to dynamic partially reconfigurable system design is to rely on external configuration controllers; in this way, the system is no longer self-configurable (secondary external devices are needed). However, the popularity of sequential, instruction-based processors persists in these architectures as well. The Erlangen Slot Machine architecture [50, 51] uses a PowerPC processor in conjunction with a dedicated configuration controller (based on a Spartan FPGA) to accomplish configuration tasks. Likewise, the architectures presented in [81, 82, 83] also rely on external configuration control elements using instruction-based processors.

A somewhat different approach to configuration control is adopted in [55]; here, control over configuration activities is shared amongst multiple entities. Multiple hardware functional units are present in the system, each containing a hardware controller as well as a partially reconfigurable region which houses the functional logic of each hardware unit. These hardware controllers are responsible for monitoring various system conditions and requesting changes in the locally deployed circuit; these changes are authorized by a central coordinator, and implemented by a sequential instruction-based processor.

### 2.5.2 Configuration Scheduling

In addition to performing the configuration activity, a configuration controller must also make the decision as to *when* said configuration should take place. The majority of configuration scheduling efforts which have been undertaken have taken the approach that functional unit allocation scheduling can be considered an extension of traditional software scheduling procedures (with some added considerations). In these approaches, partially reconfigurable regions (organized either as slots or regions) are considered to be shared resources, and tasks must be allocated and scheduled on top of these shared resources.

In a number of works, functional unit allocation is performed following schedules derived from application task graphs; in these instances, nodes of the graph may correspond to functional units, and will be configured based on a derived allocation and execution schedule. Examples include [116, 117, 118, 119, 120, 121]. These works take into consideration the configuration overhead when deriving configuration and execution schedules; some attempt to reduce this overhead by "caching" re-used tasks (in essence, allowing the task to remain inside the FPGA with the understanding that it will be used again in the future).

Methods have also been proposed for the allocation and scheduling of independent tasks; the tasks in question may either be known either at design time or at run-time. Examples of such scheduling methods can be found in [122, 123, 124, 125, 126]. Both on-line and off-line scheduling methods are proposed; in cases where no a prior information regarding incoming tasks is known, tasks may be rejected, depending on available resources and relative deadlines.

### 2.5.3 Link Establishment

Once a functional unit has been deployed to a device via configuration, a connection link must be established between the functional unit and the rest of the system. This process can take multiple forms, depending on the communication infrastructure being used. When connectivity is provided by a shared bus architecture, the assembly process stops at the configuration stage; once the functional unit is physically connected to the shared bus (often through a bus interface, such as the one described in [127]) the connection process is complete, and bus-based data transfers can be initiated. The reason for this is that a shared bus architecture does not require link reservation. However, if free functional unit placement and relocation are supported, then added resources are needed to track the location of functional units.

A similar situation occurs in packet switched NoCs; once again, link reservation does not take place in such systems, meaning that once a functional unit is connected to the network (often through a Network Interface [84]) the assembly process is complete. However, as before, dedicated resources are needed to track the location of functional units in the system. This can be more problematic in a packet-switched network, as different deployment locations will lead to different routes.

Circuit switched NoCs do require link establishment prior to the initiation of communication tasks. In [91], links are established in the system via configuration of individual routers. A similar approach is adopted in [93], where, once again, switch links are established through configuration. Finally, [94, 95] presents an NoC architecture which implements both packet and circuit switched communications. In these systems, links are established in the circuit-switched portion of the NoC via control packets sent through the packet-switched portion. Once again, additional resources are needed to track the location of functional units in the system. The links are established in response to requests originating in the functional units themselves.

As with circuit-switched NoCs, crossbars and crossbar-like architectures also require the establishment of links for communication purposes. In [97], links are established through switches present in the communication architecture; these links are established in response to requests from the functional units themselves. In [101], crossbar links are established and changed by a central system controller, implemented using an instruction-based processor.

## 2.6 Mechanisms for Functional Unit Integration

Once a complete processor has been assembled from functional units, these units must still be integrated into a functioning processor. Here the term integration refers to the combination of functional unit behaviors such that a given functionality is accomplished. Two main integration tasks are identified: schedule enforcement and synchronization. Schedule enforcement refers to the act of ensuring that processing activities in individual functional units occur at the correct time (according to some schedule). Synchronization refers to the act of ensuring that data and control signals coming to the functional unit align and interact correctly with local circuit behavior. Data synchronization is of particular importance, in particular in complex system architectures which exhibit one-to-many and many-to-one communication patterns.

In most systems, the synchronization task is enforced through interface circuits between the functional units and the communication infrastructure being used, such as bus or network interfaces;

[127] presents an example of such a bus interface. These interface circuits often perform translation operations, rearranging data from the format used in the local data-path to that used in the system communication architecture.

The schedule enforcement task can be dealt with in multiple ways and depends in large part on the control elements present in the system. A number of systems which rely on sequential, instruction-based processors as central control elements have been presented in Section 2.5.1 above. In the majority of such systems, the control element (instruction-based processor) is responsible for both configuration activities as well as the schedule enforcement once the functional units have been deployed; in essence, these processors implement the scheduling algorithms described in Section 2.5.2.

Alternatively, the functional units may be implemented to operate autonomously; in such cases, the functional unit itself will be capable of undertaking processing activities provided the right circumstances are in effect. This approach relies on the inclusion of control circuitry into the functional unit which is responsible for monitoring various external and/or internal parameters (such as the presence of data and control signal at input interfaces), and making decisions as to when processing tasks should be undertaken.

A third approach can combine aspects of external control and local autonomous behavior. An example of this approach can be found in the Run-time Adaptive Multi-Processor System-on-Chip (RAMPSoC) system architecture [109, 110, 128, 111], which targets on-chip multi-processor systems. Here, functional units can take the form of instruction-based processors or macro-function processors augmented with smaller instruction-based controllers. In such a system, functional units receive commands from a central server (including executable files); however, each functional unit also exhibits some degree of autonomy both in terms of local processing as well as communications.

## 2.7 Fault Mitigation Using Dynamic Partial Reconfiguration of FPGAs

As mentioned in Chapter 1, FPGAs are emerging as viable alternative platforms for space-based applications, due to their flexibility, faster development time and reduced cost relative to ASICs [25]. Applications targeting this environment suffer from being exposed to high levels of radiation (compared to the Earth's surface); in the case of FPGAs, such faults can target both the logic resources present in the system (logic blocks, memory blocks, I/O blocks, etc.) as well as infrastructure circuitry (routing resources, configuration memory, configuration control circuits). To address this increased fault rate, designers have traditionally relied on spatial redundancy coupled with error correction and detection codes (where applicable). FPGAs offer added benefits, in the form of full or partial reconfiguration, which can be used to mitigate both transient and permanent errors in the device [25, 129, 130].

In theory, any system architecture which allows free module deployment and relocation can offer fault mitigation capabilities, meaning that the majority of works already discussed above can be applied, to a lesser or greater extent, to the problem of fault mitigation. However, a number of research efforts have been directed towards the use of full and partial configuration activities purely for fault mitigation tasks. Towards this end, this section will present a survey of fault mitigation methods based on and oriented towards FPGA-based systems and partial reconfiguration.

The survey will be organized based on the type of faults being mitigated by the presented method: transient faults or both transient and permanent faults. The first of the following two sections presents fault detection and mitigation solutions and frameworks based solely on scrubbing procedures; these solutions can only address transient faults. The second section presents methods and frameworks which rely on both scrubbing as well as relocation methods for fault mitigation.

### 2.7.1 Scrubbing-Based Fault Mitigation

Transient faults affecting the configuration memory of an FPGA will can result in (incorrect) changes to the implemented circuitry, and they will affect the functionality of the device until they are mitigated; the mitigation process consists of re-writing, either fully or in part the configuration memory of the target FPGA. By making use of dynamic partial reconfiguration, the time cost of scrubbing operations can be reduced by writing only a portion of the device memory. Secondly, DPR allows a system to self-scrub portions of its architecture (those implemented as PRMs).

A number of research works have been proposed which implement systems consisting of a static control portion and a number of PRRs which can house functional units [131, 132, 133, 134]. Transient faults targeting functional units in PRRs can be mitigated through partial scrubbing; furthermore, fault detection can be accomplished by loading multiple copies of the same functional unit and comparing generated results. In this way, the system can implement module redundancy selectively, based on the expected fault rate.

The authors of [135] present a fault mitigation mechanism based on Triple-Module Redundancy (TMR) methods; the proposed approach makes use of additional voting circuits in order to permit better fault detection capabilities. In instances where faults are detected, individual modules are scrubbed using partial bit-streams. Finally, a self-healing processor architecture is presented in [136]; the proposed processor architecture once again relies on TMR as the mechanisms for fault detection, and proposes module scrubbing as the mitigation mechanism for SEUs.

### 2.7.2 Fault Mitigation Using Both Scrubbing and Relocation

In addition to transient faults in the configuration memory, FPGA devices are susceptible to permanent faults (either in logic or infrastructure resources); these fault types can only be mitigated through avoidance. The mitigation process used to avoid such faults is to alter the on-chip architecture such that the faulty region is not used by any system circuit. These alterations to the architecture can be accomplished either through full or partial configuration.

Given their popularity in embedded system design, a number of fault tolerance techniques have been aimed at mitigating faults in sequential instruction-based processors. A potential approach to this problem is to separate a processor into multiple elements (often pipe-line stages in pipe-lined processors), and perform scrubbing or relocation on these elements. Examples of this approach to processor protection are presented in [137, 138]. An alternative approach is presented in [139], where the complete processor is treated as a functional unit and implemented as a PRM; the method supports scrubbing, as well as a form of relocation referred to as *tiling*, whereby multiple versions of the same PRM targeting the same PRR but with different spatial distributions are implemented and used. An approach to fault tolerance in multi-processor systems is presented in [140, 141]. In the case of transient faults, individual processors can scrub each other; permanent faults in a processor are mitigated by migrating software tasks to the other processors.

A general fault detection and mitigation method is presented in [142]; the proposed method makes use of *roving* test circuits which are migrated across the FPGA and used to detect faults. The authors also propose a method of system design which ensures that, for a given design circuit, each active programmable logic block is near a spare logic block, which allows functionality to be migrated in case of a detected fault. The proposed method is aimed explicitly towards fully homogeneous FPGA micro-architectures, and was implemented and tested using the ORCA 2C series of FPGA [143].

A fine granularity approach to fault mitigation is presented in [144, 145, 146]. The presented method proposes the use of logic redundancy at the logic block level; in the case of a fault, small configuration activities can be undertaken to migrate logic from faulty regions (in essence a very fine grained form of relocation). In a similar vein, the authors propose the implementation of fault mitigation on a per-frame basis [147]; faults covered by a configuration frame can be avoided by changing configuration data in the frame, such that faulty components are avoided. These types of approaches rely on very detailed knowledge of the micro-architecture of the FPGA being used, and can result in very large combinations of bit-streams for a given design.

An on-chip framework based on evolvable hardware is presented in [148, 149]. The core of the method is the use of two-dimensional arrays of Processing Element (PE), connected in a mesh topology. These processing elements are implemented as PRMs, and can be loaded into the system to accomplish various operations. A processor with a specific functionality can be assembled from individual PE, using a genetic algorithm; a control CPU is used to control the evolutionary process and perform all reconfiguration tasks. Faults can be avoided through this same evolutionary process; in essence, the connection pattern and functionality of PEs reaches a pattern where the faulty region is avoided or has no impact on the generated output.

A method aimed primarily at permanent faults due to aging effects is presented in [150]. The the method relies on implementing a design using multiple bit-stream variations, where each bit-stream avoids a certain region of the targeted device. This is similar to the tiling technique described in [139] above, but reliant on the use of full as opposed to partial reconfiguration. In contrast, methods using module relocation using partial bit-streams are presented in [151, 152]; in these works, multiple module regions are reserved, and modules can be loaded into different regions, thus avoiding permanent faults in the FPGA fabric.

Finally, A number of fault detection and mitigation methods for dynamic partially reconfigurable systems are combined in [153, 154, 155, 156] in the form of a design flow for fault tolerance. Potential mitigation methods include module redundancy (either triple or double), scrubbing and relocation; the relocation process is accomplished via full reconfiguration, and is based on the tiling method described above.

## 2.8 Summary

The preceding chapter has presented background information and a literature survey for the four primary mechanisms required by the MACROS Framework. The background information and literature survey were presented together as the information contained in the former lends structure to the material presented in the latter. This chapter forms the starting point for further investigations into the topics of on-chip functional unit deployment, relocation, assembly, integration and communication.

However, before said investigations are undertaken, the basic principle of operation of the class of systems defined by the MACROS Framework must be described more formally, as it informs all future work. Thus, Chapter 3 will formally present the principle of operation of the MACROS Framework and conduct an initial analysis of this type of system, with the aim of determining how run-time architecture adaptation can impact a running system. In essence, Chapter 3 will act as the specification for the class of systems implemented by the MACROS Framework. Chapter 4 will then continue the investigations began in Chapter 2, guided by the analysis and conclusions of Chapter 3.

# Chapter 3

# MACROS Framework Operating Principles

## 3.1 Introduction

The Multi-mode Adaptive Collaborative Reconfigurable self-Organized System (MACROS) Framework defines a class of systems which permit run-time architecture adaptations through deployment and relocation of macro-function processing elements (referred to herein as functional units) in dynamic partially configurable FPGAs. The MACROS Framework is aimed primarily at the creation of mode-adaptive and fault tolerant stream processing systems; however, the framework can be applied to non-streaming multi-mode systems as well.

This chapter represents the first step in describing this class of systems through a description of their principle of operation; at this stage, the systems are treated primarily as black boxes, in that the mechanisms which accomplish the described behavior are not yet fully elaborated. These mechanisms have been introduced in Chapters 1 and 2 and will be investigated in more detail in Chapter 4. In this chapter, the emphasis is placed on the integration of run-time architecture adaptation procedures into stream processing workloads. Thus, a definition for digital data streams will be introduce, and the structure of stream workloads will be derived.

Based on the presented principle of operation and workload structure an analysis will be conducted to determine how run-time architecture adaptation procedures can affect stream workloads.

This analysis will introduce the concept of a Stream Interrupt Factor $F_{si}$, an indicator of how stream workloads are affected by these adaptation procedures; based on this factor, the concept of *seamless mode-based architecture adaptation* will be derived, and its requirements will be determined. In essence, this chapter acts as a specification of requirements for the class of systems defined by the MACROS Framework. These requirements are behavioral in nature, and address how run-time architecture adaptation procedures must be implemented and incorporated into the stream work-load.

## 3.2 Run-time Architecture-to-Mode Adaptation

The basic operating principle of a system using the MACROS Framework is the generation of full ASP via run-time assembly and integration of functional units, thus resulting in run-time architecture adaptation. ASPs are generated in response to received system modes or detected system faults. This concept is illustrated in Figure 3.1; Figure 3.1 - A shows a traditional system consisting of multiple functional units, activated in different combinations depending on the operating mode. Figure 3.1 - B shows a version of the system which incorporates dynamic architecture adaptation: for each mode, an ASP is generated using a library of functional units.

Architecture adaptation in MACROS Framework systems can consists not only of functional unit deployment, but also of functional unit relocation. The relocation process requires that the system remains functionally identical, while the physical location of one or more functional units changes. The relocation procedure is used to mitigate permanent system faults; an example of this process is shown in Figure 3.2. Fault detection for the system is assumed to be accomplished via Built-In Self-Test (BIST) circuitry embedded in the functional units themselves [38].

By allowing this functionality, the MACROS Framework can address the two problems introduced in Chapter 1: the increasing implementation complexity of modern systems, and the potential increase in fault rates associated with reduced transistor sizes. Implementation complexity is reduced by assembling processors at run-time from pre-built modules (functional units), while faults can be mitigated through relocation of these same functional units. It must be noted that the two problems being addressed are not independent from one another and neither are their respective solutions. Traditional mitigation methods (using module redundancy) exacerbate the problem of

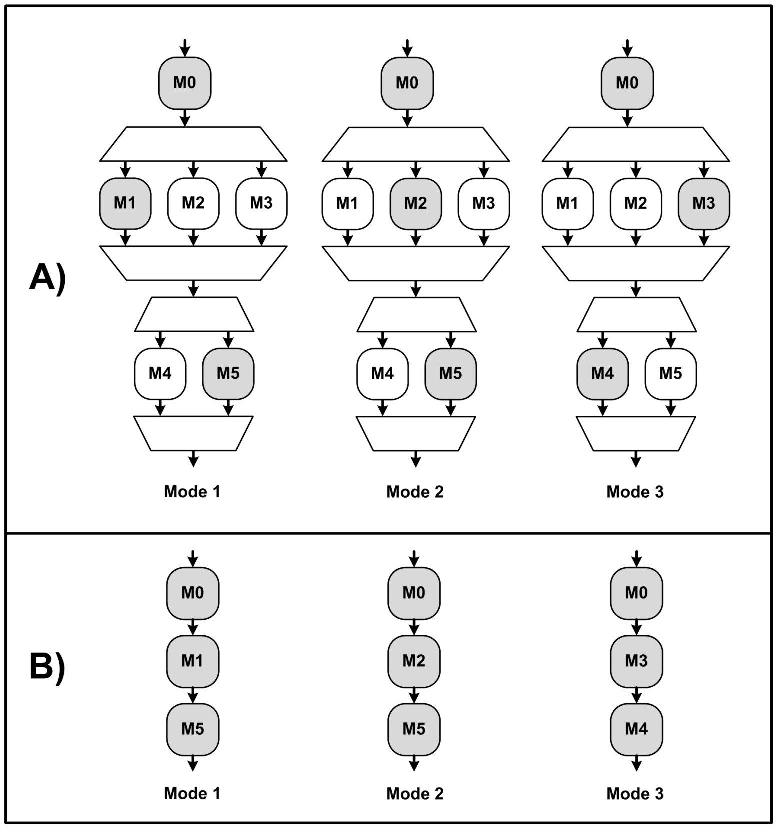


Figure 3.1: Traditional and Dynamically Reconfigurable System Implementations of Multi-Modal Processing Circuits

increasing system complexity. By assembling ASPs from pre-built functional units and using only those functional units necessary, the system size is reduced, which leads to a smaller number of potential fault sites. Likewise, by offering relocation-based fault mitigation (as opposed to relying on resource duplication), the complexity of the implemented system can be reduced.

With the above in mind, the run-time ASP generation process can be more formally defined as follows:

1. Sample mode or detect fault and determine the needed ASP architecture.

2. Assemble the new ASP by performing configuration activities and establishing system links.

3. Integrate functional units into an operational ASP.

The ASP generation process is driven by the system mode and/or system faults. Thus, the process always starts with a decision as to what ASP architecture is needed based on the mode/fault

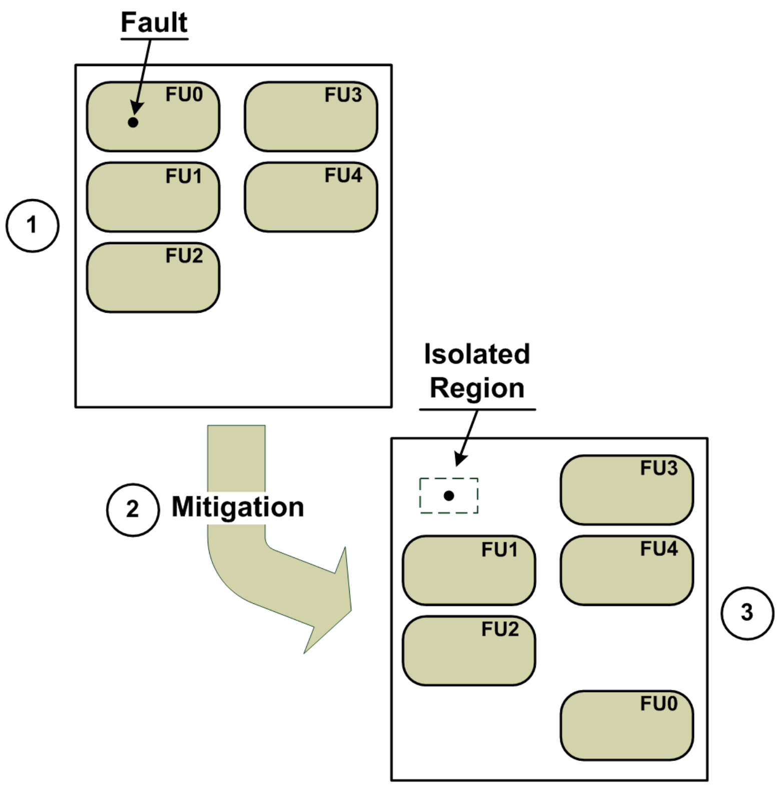


Figure 3.2: Fault Mitigation in a Dynamic System

detected, and an analysis of how the needed architecture differs from the current one. This analysis will result in a sequence of steps needed to assemble the new ASP processor; these steps may include configuration activities (given that FPGAs are the deployment medium) and system link establishment. Finally, the assembled functional units will be integrated into a working ASP architecture which meets the mode requirement and, if necessary, mitigates a given fault.

As previously indicated, the MACROS Framework is aimed primarily at high data rate stream processing applications, where large volumes of information must be processed under strict temporal constraints. For this reason, the concept of time division multiplexing of functional units onto the same underlying resources is omitted, as it limits the achievable processing performance; a design approach based on time-division multiplexing of FPGA resources for custom architecture systems can be found in [36]. In the MACROS Framework, sharing of underlying resources does occur when the system architecture changes from one mode to the next; however, once a new mode is in effect, all functional units are expected to be present in the system continuously, which allows the fastest achievable data through-put.

## 3.3 Digital Data Streams and Stream Workload

Having defined the basic principle of operation of MACROS Framework systems, the next step is to analyze how this proposed operating principle - run-time ASP generation - can affect a stream processing system. As a preamble to this analysis, this section presents a definition of digital data streams which will be used in the remainder of the presented work. As well, the section describes the workload structure of any system built to process stream information, as derived from the stream structure.

### 3.3.1 Digital Data Stream Definition

A digital data stream is defined as an *infinite sequence* of digital data packets; these packets form a hierarchy, with larger packets formed from combinations of smaller ones. The *Stream Packet Hierarchy (SPH)* for a digital data stream is formally defined as a set of pairs of values:

$$SPH = \{(lvl_i, numP_i) | lvl_i \in N, numP_i \in N^*, 1 \leq i \leq n\} \tag{3.1}$$

The stream packet hierarchy defines, as the name suggests, a hierarchy of packets. Each pair in the set defines a packet at a certain level; above, $lvl$ specifies the packet level and $numP$ specifies how many packets of the next lower level ($lvl_{i-1}$) form one packet of the current level. In the case of all streams, $lvl_1 = 0, numP_1 = 1$; the level 0 packet is the smallest atomic element of the stream, and each such element constitutes one level 0 packet. To clarify this definition, the following example is provided of the SPH of a 480p60 video stream uncompressed video stream using 24-bit color [157].

$$SPH_{VGA} = \{(0,1), (1,640), (2,480)\} \tag{3.2}$$

Where:

1. $(0,1)$: level 0 packet, consisting of one atomic element (one 24-bit pixel value).
2. $(1,640)$: level 1 packet, consisting of 640 level 0 packets (one line).
3. $(2,480)$: level 2 packet, consisting of 480 level 1 packets (one frame).

In addition to this hierarchy of data packets, a digital data stream also incorporates control information which describes the structure of the stream. This control information can take one of two forms: explicit control information, or implicit control information. Explicit information is transmitted along with stream data; examples include flags (such as line-valid and frame-valid or other synchronization signals) or packet headers. Implicit control information takes the form of *standardized, known information*; the packet hierarchy described above is the most obvious type of implicit control information. Other examples include timing information, such as the frequency of the synchronizing clock being used or the number of clock cycles needed for transmission of one packet, or packet structure, such as the length of a packet header. For the purposes of this discussion, the most important aspect of control information is that it can be used to specifies the beginning and end of the various packets in a data stream. In the remainder of this document the above-mentioned control information will be referred to as *control indicators.*

### 3.3.2 Stream Workload Structure

A stream processing application will be defined as performing a number of operations on one or more packets of a given data stream. This leads to a specific workload, which must follow the temporal distribution of the stream packets themselves. The data stream is assumed to be either infinite in length or long enough to eliminate the possibility of complete buffering; thus, the throughput capability of the system must match the stream data-rate.

For any given combination of data stream and application, two important parameters can be determined: $T_w$, the length of time needed to perform one iteration of the implemented application algorithm and $T_p$, the period of the packet or collection of packets over which the application is defined to operate; note that $T_w$ includes any latency value ($T_l$). The term work period is used to describe the execution and completion of all operations needed to process one packet (or collection of packets, depending on the application) given a specific ASP architecture. A graphical illustration of such a workload is shown in Figure 3.3; the figure shows the workload of an application-specific video processor performing operations on frames of an uncompressed video stream. It should be noted that $T_w$ can be larger than $T_p$, depending on the structure of the operation performed; this can happen if the latency of the processing task is great enough. The same temporal workload structure, exhibited by a complete system, extends to individual functional units in the system as

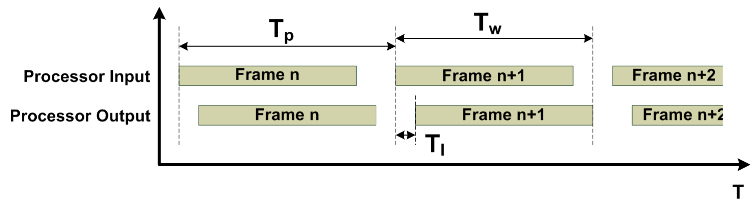


Figure 3.3: Workload Example in a Video Processor

well.

## 3.4 Mode-Based Architecture Changes and Stream Workloads

Despite being very general in nature and omitting all technical details, the above ASP generation procedure can be analyzed to determine how it will interact with a stream workload. In any system which supports run-time architecture adaptation, the adaptation process constitutes a second layer of system behavior, superimposed on top of the regular functionality of the system. This second layer is not encountered in traditional system designs, and must be analyzed separately. The most important parameter which can be identified for any system which supports run-time architecture adaptation is the *mode-based ASP generation cost* $Tagen_{i-j}$; this parameter defines the time cost needed to change the system architecture from that of mode i to that of mode j. $Tagen_{i-j}$ has to be defined in terms of current and desired modes, because this dictates what functional units must be added to the system. The set of functional units which must be added to the system when changing modes from $i$ to $j$ is defined as:

$$Sconf_{i-j} = \{Sm_j - (Sm_i \cap Sm_j)\} \tag{3.3}$$

Where:

- $Sconf_{i-j}$ represents the set of functional units which must be added to the system to transition from mode i to mode j; because FPGAs are targeted, this represented the set of functional units that must be deployed via configuration.
- $Sm_i$ represents the set of modules needed for mode i.
- $Sm_j$ represents the set of modules needed for mode j.

The mode-based ASP generation cost for a given mode transition will be composed of the following elements:

- $Tsel_{i-j}$: the time needed to select the required ASP architecture for mode j, and the set of functional units that must be added to the system to permit the architecture transition.
- $Tconf_{i-j}$: the time needed to configure the set of functional units $Sconf_{i-j}$ defined above.
- $Tai_{i-j}$: the time needed for the completion of the assembly process (link establishment) and integration activities.

Thus, the complete mode-based ASP generation cost $Tagen_{i-j}$ is defined as:

$$Tagen_{i-j} = Tsel_{i-j} + Tconf_{i-j} + Tai_{i-j} \tag{3.4}$$

This assembly time cost must be incurred whenever a mode change must take place; its placement in time in relation to the system workload distribution dictates what interference the assembly process will impose on regular processing activities. Depending on the value of $Tagen_{i-j}$, the ASP generation process may interfere (overlap temporally) with 0, 1 or more work periods; these possibilities are shown in Figure 3.4. The *stream interrupt factor* $Fsi_{i-j}$ can be defined as an indicator of the total interference that can be expected from a given assembly activity.

$$Fsi_{i-j} = \begin{cases} 0 & \text{if } Tagen_{i-j} \leq (T_p - T_w), \\ \lceil \frac{Tagen_{i-j}}{T_p} \rceil & \text{if } Tagen_{i-j} > (T_p - T_w). \end{cases} \tag{3.5}$$

$Fsi_{i-j}$ will have a value of 0 or some positive integer, which will indicate the number of work periods missed (and the loss of the associated stream packets) due to ASP generation activities.

## 3.5 Seamless Run-Time Architecture Adaptations

The definition for $Fsi_{i-j}$ suggests that run-time architecture adaptation can be implemented in such a way that no work periods are affected; this is accomplished if $Tagen_{i-j} \leq (T_p - T_w)$. This condition also implies that $(T_p - T_w) > 0$, which states that a seamless change can only occur in the time period when no functional units are working, and this time period must be greater than

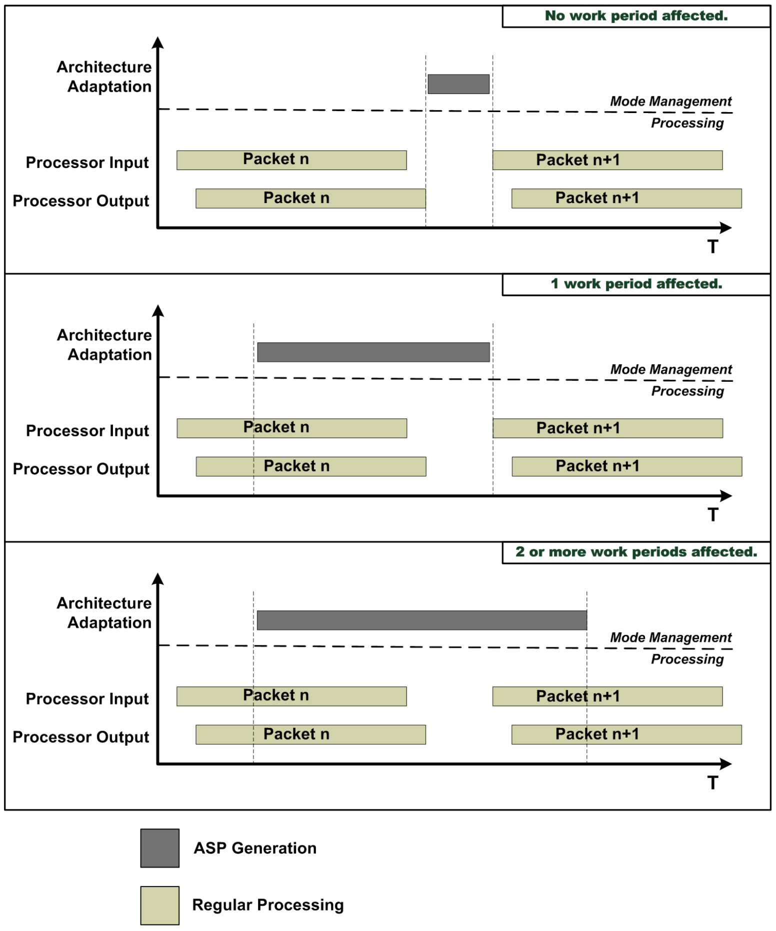


Figure 3.4: Examples of Workload Interference Due to ASP Generation

0. Given that $T_p - T_w$) is the more difficult parameter to change ($T_p$ is fixed for a given stream, and $T_w$ is often imposed by implementation constraints), a potential method of ensuring seamless architecture adaptation is to minimize $Tagen_{i-j}$.

A method to accomplish this can be derived by taking advantage of the defining aspect of DPR operations: while reconfiguration activities take place, any regions not affected by said operations can operate normally.

A method to accomplish this minimization is to overlap potential configuration tasks with processing tasks; in this way, configuration activities can be omitted from $Tagen_{i-j}$. This principle can be applied to single or multi-FPGA systems; the remainder of the presented discussion will focus on single-FPGA systems using dynamic partial reconfiguration. Thus, if the target FPGA device being used has spare resources, the set of functional units $Sconf_{i-j}$ can be configured in parallel with regular processing operations; in essence, $Tsel_{i-j}$ and $Tconf_{i-j}$ are overlapped with

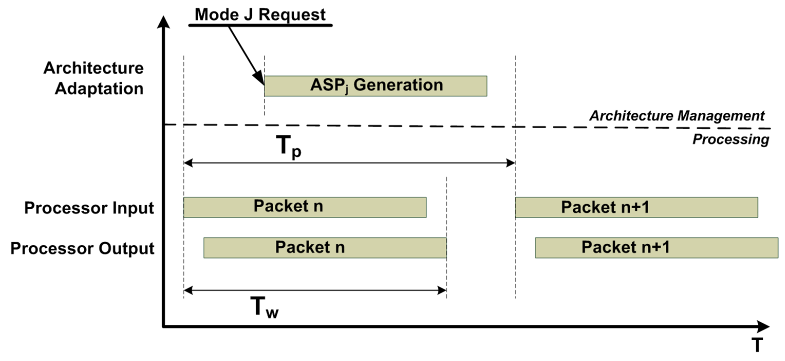


Figure 3.5: Overlap of Processing and Configuration

processing activities. Once the work period completes, the final part of ASP generation can take place. If this approach is adopted, the stream impact factor $Fsi_{i-j}$ becomes:

$$Fsi_{i-j} = \begin{cases} 0 & \text{if } Tai_{i-j} \leq (T_p - T_w), \\ \lceil \frac{Tai_{i-j}}{T_p} \rceil & \text{if } Tai_{i-j} > (T_p - T_w). \end{cases} \tag{3.6}$$

$Tai_{i-j}$ is expected to be much smaller than the complete ASP generation time cost $Tagen_{i-j}$, due to the fact that all configuration activities are omitted.

To be able to fully take advantage of this feature, the primary requirement which must be met is that the configuration time for all missing functional units must be smaller than the work period time, i.e. $Tconf_{i-j} \leq T_w$. An indirect secondary requirement emerges, in that the configuration activities must start early enough that they finish before the end of the current work period; this is illustrated in Figure 3.5. Assuming the mode change request arrives sometime during the current work period, this can be accomplished in one of two ways: A) the mode change request arrives at least $Tconf_{i-j}$ seconds before the start of the next work period; B) the mode change can be predicted by at least $Tconf_{i-j}$ seconds.

Based on the previous discussion, a set of requirements can be defined for seamless mode changes in a system implementing run-time architecture adaptation using dynamic partially reconfigurable FPGAs:

- $Tagen_{i-j} \leq (T_p - T_w)$.

or

- The target system has spare resources such that it can accommodate the set of missing functional units $Sconf_{i-j}$ for any combination of $i$ and $j$ modes.
- $Tconf_{i-j} \leq T_w$ for any combination of $i$ and $j$ modes.
- The mode change request can be predicted by at least $Tconf_{i-j}$ seconds, or is guaranteed to arrive at least $Tconf_{i-j}$ seconds before the start of the next work period.
- $(T_p - T_w) > 0$ or $T_p > T_w$.

$(T_p - T_w) > 0$ is the foundational idea behind the concept of a seamless architecture change; in essence, some non-zero time is needed between consecutive work periods, at least for link establishment and integration (assuming that configuration activities are omitted). However, this requirement can impose its own costs and limitations. First of all, depending on the application algorithm, it may not be possible to enforce the above constraint. Secondly, if the application allows it, $T_w$ can be reduced to less than $T_p$; however, this can lead to increased hardware or power consumption due to the use of larger parallel circuits or faster clock periods. For any given application, seamless run-time architecture adaptation may come at a cost; whether this cost is acceptable will depend on the application requirements.

## 3.6 Summary

The preceding chapter has taken the first step in describing the MACROS Framework by introducing the principle of operation underpinning this class of systems: the integration of full or partial configuration of FPGA devices into the regular operation of the system, thus allowing for run-time architecture adaptation. In an attempt to determine how architecture adaptation activities would affect a system, the principle of operation was analyzed, and the stream interrupt factor $Fsi_{i-j}$ was introduced as a measure of this interference. Finally, based on this analysis, the concept of *seamless architecture adaptation* was formally defined for situations where the architecture adaptation process does not affect the system workload in any way.

Having presented the this basic aspect of the MACROS Framework, the specific mechanisms which support this behavior can be determined. These mechanisms will address the problems of functional unit deployment and relocation, on-chip communication, ASP assembly and integration.

Existing work in this field has already been presented in Chapter 2; with the benefit of the added information derived in this chapter, Chapter 4 will continue the investigation process into the selection and derivation of mechanisms for the MACROS Framework.

# Chapter 4

# MACROS Framework Mechanisms

## 4.1 Introduction

In Chapter 1, the basic concept of a MACROS Framework system was introduced as a system which can implement run-time architecture adaptation by generating application-specific processors from functional units. To accomplish this basic concept, a number of uncertainties and related topics of research were identified which would have to be investigated. Chapter 2 presented background information and existing mechanisms for each of the research topics identified. Chapter 3 presented the basic principle of operation for MACROS Framework systems, thus establishing a base-line functionality which must be supported. Chapter 4 will conduct analyses on existing mechanisms for each topic being addressed, based on the principle of operation introduced in Chapter 3; as a result of this analysis, mechanisms will either be selected or derived to address the topics identified in Chapter 1. The final step of the selection process is a technical analysis which confirms or repudiates the chosen mechanisms, or the chosen implementation method. However, the mechanisms used in MACROS Framework systems interact heavily with one another and require full system implementations for complete verification and analysis; for this reason, the technical analysis of these mechanisms will be presented after the complete Framework has been described.

Topics of research must be presented in a specific order, as dependencies between topics exist; the various mechanisms being considered depend on one another in various ways, as will be discussed below. The first topic to be addressed will be the selection of a mechanism for free functional unit deployment and relocation. Following this, an architecture for on-chip communication between functional units must be selected; this communication architecture must reflect the capabilities and limitations of the deployment process outlined above.

The third topic to be addressed is the on-chip assembly process; here, methods for the scheduling of configuration activities and link establishment will be analyzed, and a mechanism will be derived. The fourth topic being addressed is that of mechanisms for functional unit integration, in the form of synchronization and configuration scheduling. Finally, the section concludes with an analysis of central and distributed control structures for on-chip assembly and integration mechanisms; the aim of this section is to determine whether MACRO Systems, given their emphasis on high data-rate stream processing, will benefit more from distributed control structures.

## 4.2 Functional Unit Deployment and Relocation

The MACROS concept, as introduced, proposes the creation of stream processing systems which can run-time self-assemble into working ASPs, and can further re-assemble into different ASP architectures (including the relocation of functional units). This functionality allows such systems to be implemented in a modular fashion and then be assembled at run-time, and allows for efficient permanent fault mitigation in such systems. To permit the aforementioned behavior, a mechanism is needed which allows functional units to be deployed and relocated at run-time; the mechanism targeted in this case is full and partial reconfiguration of FPGAs.

As established in Chapter 2, functional units are deployed into PRR, using partial bit-streams. Chapter 2 also showed that two types of PRR structures exist, as well as two methods for functional unit deployment and relocation. This section will analyze the benefits and shortcomings associated with PRR structures and functional unit deployment and relocation methods; based on this analysis, a mechanism will be derived for functional unit deployment and relocation in MACROS Framework systems. As already stated in Chapter 2, FPGAs with tile-based architectures are considered in this discussion.

### 4.2.1 Partially Reconfigurable Region Structure and Functional Unit Placement

The concept of dynamic partial reconfiguration relies on the use of PRRs; these regions will hold all PRM, i.e. some or all of the system functional units. As already mentioned, the the structure of PRRs and the mapping of PRMs to PRRs will lead to two types of systems: slot-based systems (one PRM per PRR, many PRRs per device) or region-based systems (multiple PRMs per PRRs, few or one PRRs per device). Slot-based functional unit placement is the simpler of the two placement methods and is considered first. It should also be noted that slot-based placement is currently directly supported by existing DPR CAD tool-chains [35].

The disadvantage associated with the slot-based approach is that FPGA resources may not be optimally utilized. Slot sizes and locations have to be selected such that all functional units will fit; depending on the variation in resource use in various functional units, one or more slots may be under-utilized. This problem can be partially addressed by using multiple slot types, but this solution reduces system flexibility. On the other hand, there are a number of benefits associated with adopting this approach. As already noted, a slot-based solution can be implemented directly using current DPR CAD tool-chains. Since all slots are allocated at design time to fixed locations, slot communications can be more easily implemented, since slot interface points can be fixed at design time. An extension of this design-time allocation is the fact that delays in the system can be carefully controlled and accounted for, which makes it easier to enforce desired performance parameters. Thirdly, the use of slots reduces the scope of the allocation task; the problem of fragmentation (beyond what happens inside the slot due to functional unit size variations) no longer has to be considered.

Region-based module placement offers the primary benefit that it can, potentially, maximize the resource usage of a given FPGA for a given set of functional units. However, region-based module placement also suffers from a number of limitations. First of all, this approach cannot be implemented using multiple bit-streams, as the number of bit-streams per functional unit for a region can be large; as such, some form of bit-stream manipulation automatically becomes a necessity. Secondly, adopting this approach complicates inter-functional unit communication significantly; unit interfaces are no longer fixed, and the communication infrastructure must be more complex to deal with location variation and gaps between modules and the edges of the partially reconfigurable region (see Figure 4.1).

A number of solutions have been proposed to address this problem, the majority of which rely on the integration of communication modules inside the functional unit structure [57, 58, 63, 60, 61, 62]; one of the large potential problems associated with this approach is the need to load dummy modules, as shown in Figure 4.1, to support communication between functional units and interfaces on the PRR region boundary. An alternative is the use of configuration read-back for the purposes of inter-functional unit communication, as described in [68, 69]. However, this approach limits achievable communication rates to the read-back rate of the configuration interface; this reduction is further exacerbated by the fact that a shared communication medium (the configuration interface) is used.

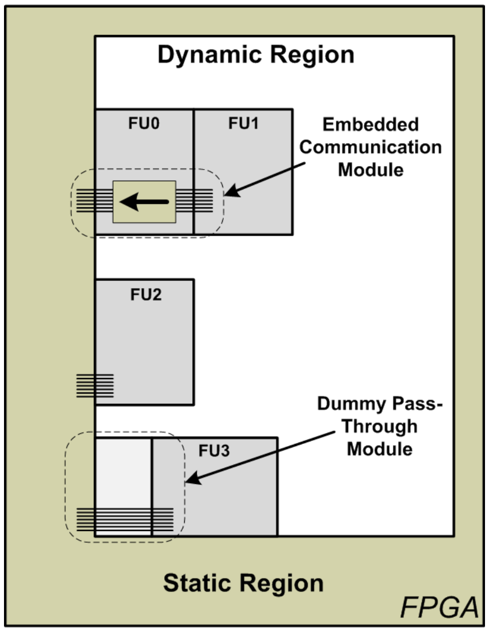


Figure 4.1: Module Communication Methods in Region-Based Systems

These solutions require increased design and implementation effort, and may impose limits on the achievable communication performance inside the system. The third and final limitation associated with this approach is that it imposes added requirements on functional unit implementation. Each functional unit requires certain resources for its implementation (such as look-up tables, D flip-flops memory blocks, etc.), and a functional unit can only be mapped to a region which contains these resources *in the right combination and with the right spatial arrangement*; accommodating

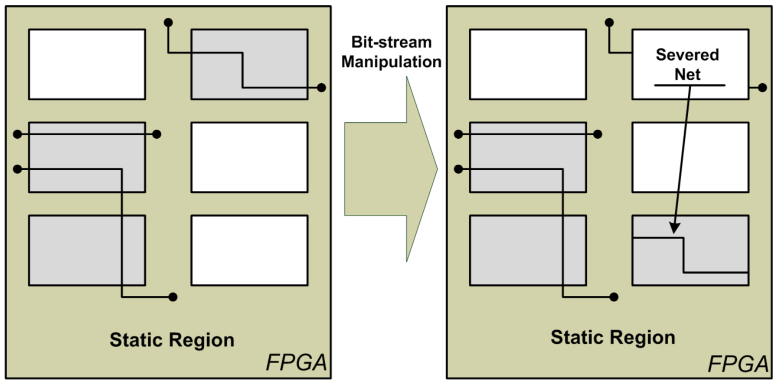


Figure 4.2: Effects on Static Nets Due to Bit-stream Manipulation

these requirements generally requires complex implementation procedures [66, 67].

### 4.2.2 Free Functional Unit Deployment and Relocation

As with placement, relocation can be accomplished in one of two ways: multiple bit-streams can be generated for each functional unit, one for each potential unit location, or a single base bit-stream can be generated and then manipulated. Once again, the analysis will begin with the simpler of the two approaches, the use of multiple bit-streams. It should be noted that this approach can only realistically be used in conjunction with slot-based placement; region-based placement would result in extremely large numbers of bit-stream combinations. In the case of slot-based placement, the number of needed bit-streams equals $numberofslots \times numberofmodules$.

The main disadvantage of this approach is that it leads to increased storage requirements for partial bit-streams, as well as additional implementation steps needed to generate said partial bit-streams. The main benefit of this approach is that it is currently supported directly by DPR CAD tool-chains, and more importantly it supports the routing of static nets through partially reconfigurable regions. Figure 4.2 shows potential routing techniques where static nets are routed though system slots; this approach is currently used by the Xilinx DPR place-and-route process [35]. As the right half of the figure shows, if a functional unit which houses a static net is relocated simply through bit-stream manipulation, one or more static nets may be severed. This problem could be addressed by excluding static nets from dynamically reconfigurable regions; however, using the current CAD tool-chain, this is difficult and time consuming to accomplish.

The alternative, as mentioned previously, is to use a single bit-stream, and perform manipulation

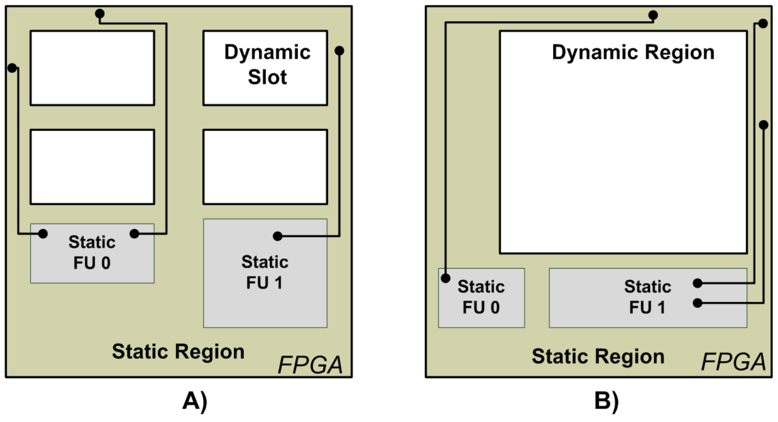
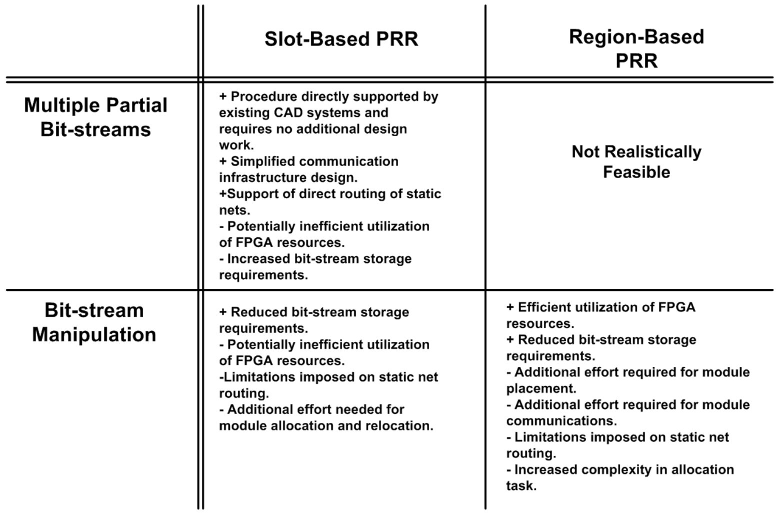


Figure 4.3: Static Net Routing For PRR Avoidance

steps (such as changing the FAR and CRC values in the bit-stream [158, 159, 160, 49]) to target a different region. Using this method of functional unit deployment and relocation requires that all slots are identical and have the same configuration memory structure; if this requirement is not met, re-mapped bit-streams will affect different slot regions incorrectly. For this technique to work, the problem described in the previous paragraph must be avoided, by forcing static nets to not cross partially reconfigurable regions. A potential way to enforce this approach is to create hard macros (hand-made circuit structures with controlled layout and routing); however, this approach can be very time-consuming. More importantly, in systems with high I/O utilization, nets have to travel to multiple regions on the device; ensuring that all these nets avoid all slots can lead to congestion of nets on certain routing resources, and can lead to difficulties in meeting timing requirements for these same nets. Figure 4.3 shows this routing approach in a slot-based system and in a region-based system; in both cases, net lengths are increased when attempting to avoid partially reconfigurable regions.

### 4.2.3 Functional Unit Deployment and Relocation in MACROS Framework Systems

A summary of the potential solutions for placement and relocation, and the benefits and limitations of each is shown in Figure 4.4. Based on the above discussions, the MACROS Framework use slot-based functional unit placement, and allocation and relocation via separate partial bit-streams. Slot based functional unit placement was selected because the offered benefits far outweigh the one disadvantage (the potential for inefficient slot utilization). The use of multiple partial bit-



| | Slot-Based PRR | Region-Based PRR |
|---|---|---|
| Multiple Partial Bit-streams | + Procedure directly supported by existing CAD systems and requires no additional design work.<br>+ Simplified communication infrastructure design.<br>+Support of direct routing of static nets.<br>- Potentially inefficient utilization of FPGA resources.<br>- Increased bit-stream storage requirements. | Not Realistically Feasible |
| Bit-stream Manipulation | + Reduced bit-stream storage requirements.<br>- Potentially inefficient utilization of FPGA resources.<br>-Limitations imposed on static net routing.<br>- Additional effort needed for module allocation and relocation. | + Efficient utilization of FPGA resources.<br>+ Reduced bit-stream storage requirements.<br>- Additional effort required for module placement.<br>- Additional effort required for module communications.<br>- Limitations imposed on static net routing.<br>- Increased complexity in allocation task. |

Figure 4.4: Comparison of Deployment and Relocation Methods and PRR Structures

streams for free allocation and relocation leads to higher numbers of bit-stream, which means added bit-stream storage requirements and a longer bit-stream generation step in the implementation process. This disadvantage becomes much less pronounced when one considers the added design steps needed to separate static nets from partially reconfigurable regions, and the fact that bit-streams do not require additional manipulation. One of the guiding principles when selecting both the on-chip PRR structure and the placement and relocation method was easy integration into existing CAD tools; the theoretic potential benefits associated with using region-based PRRs and bit-stream manipulation methods are outweighed by the added design steps and associated limitations associated with these approaches.

## 4.3 On-Chip Communication Architectures

Once functional units have been deployed (or relocated), they create an ASP by combining their functionality; this functionality combination relies on the creation of communication links between individual functional units. Thus, some form of on-chip communication mechanism is needed to provide system connectivity between functional units. This section will present an analysis of existing communication architecture, which will lead to the selection of an on-chip communication architecture for the MACROS framework.

The section begins by analyzing the requirements imposed on the communication architecture in

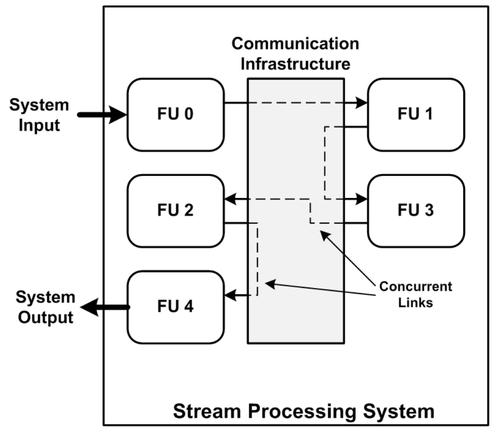


Figure 4.5: Concurrent Links in Stream Processing Systems

a MACROS Framework system. Following this analysis, the communication methods introduced in Chapter 2 (shared bus, crossbar, NoC and hybrid architectures) are analyzed and compared; based on their respective characteristics and the imposed requirements, an architecture will be selected. The section concludes with a discussion of run-time wire delay variation, a unique feature of run-time architecture adaptation, and proposes a method of addressing this effect in MACROS Framework systems.

### 4.3.1 Communication Requirements

The MACROS Framework and design method is aimed specifically at digital stream processors; more specifically, high data-rate streams are targeted. With that in mind, the first requirement is that the communication infrastructure must be able to provide a high aggregate data throughput to each functional unit in the system. A number of concurrent links are expected to be present in the system (see Figure 4.5), and each link will require a guaranteed data rate to ensure correct operation of the system.

In addition, functional unit distribution in a MACROS Framework system is run-time variable; the number and location of units is expected to change *as part of the regular operation of the system*. Thus, communication links in the system will likewise have to change to reflect this. In other words, the communication infrastructure being used must offer programmability, so that links can be established or changed at any time during system operation. A secondary desired characteristic would be that the programming process used to update connectivity is not overly complex in nature.

### 4.3.2 Communication Infrastructure Analysis

Chapter 2 identified four main types of communication architectures: the shared bus, the Network-on-Chip, the crossbar and variation on point-to-point links and shared memory systems. This section will analyze these architecture types, in each case identifying their benefits and limitations.

#### Shared Buses

The shared bus is the most basic method of allowing multiple elements to communicate. Due to its reliance on a single shared medium (the shared bus), it can only offer one communication link at any time; the data throughput of this one link may be large (clock rate $\times$ bit width), but only one link is ever active. For systems where X simultaneous links are needed, each with Y bit-rate, a shared bus would have to offer a peak bit-rate rate of $X \times Y$ to be able to accommodate all links through a time-division multiplexing scheme. Achieving such a bit-rate implies the use of a clock frequency X times larger than that used for the rest of the system, which will lead to: A) increased power consumption and B) the need for additional buffering and clock domain cross-over circuitry. More importantly, as the number of links in the system increases, the required operating frequency will rapidly out-pace the capabilities of the underlying implementation circuitry.

#### Fully-Connected Crossbar

The fully-connected crossbar finds itself at the opposite end of the spectrum in terms of communication capabilities. The main advantage of a crossbar is its *non-blocking* nature: linked input-output pairs will be completely isolated and will in no way interfere with one another. In addition, the communication links themselves consist of nothing but wires and one multiplexer (when using multiplexer-based implementations), meaning that high clock rates can be achieved. The combination of high clock rates and full connectivity leads to very large aggregate data-rates, and, more importantly, the ability to guarantee the data rate on each link.

The primary limitation of this communication architecture lies in its resource cost. Multiplexer-based crossbar implementations require one such multiplexer per output port. Depending on the bit-width of the communication channels used, and the number of input ports in the system, these

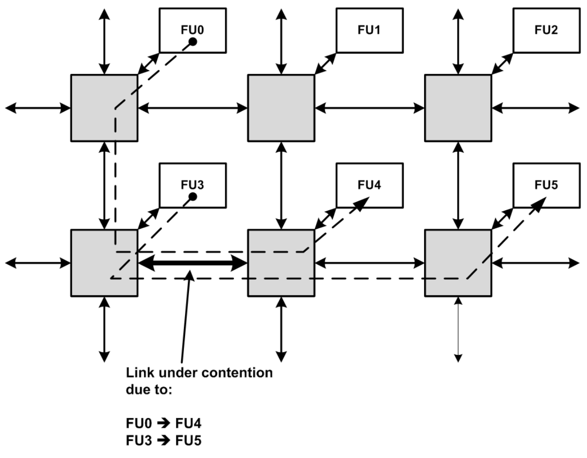


Figure 4.6: Link Contention in Mesh NoC

multiplexers can grow quickly. Thus, the resource cost of fully-connected crossbars grows at a faster-than-linear rate in relation to the size of the system (in terms of functional units connected).

**Network-on-Chip**

NoCs offer higher aggregate data-rates than shared buses (this is one of the main reasons for their existence). However, they are not non-blocking communication methods by default, meaning that links between pairs of elements can interfere with each other (see Figure 4.6). It is possible to choose a topology and module distribution such that links are kept separate, such that a non-blocking system arrangement is obtained; however, this approach relies on careful system allocation and arrangement at design time. More problematic is the fact that, in a dynamic partially reconfigurable system, modules can and often will change position at run-time, which will invalidate the topology selection made at design time. An example of this type of problem in a mesh topology is shown in Figure 4.7; as a result of functional unit relocation, link contention can appear.

If non-blocking communications cannot be guaranteed, the question then arises: can NoCs still offer data rates high enough for the stream applications being targeted? As mentioned in the requirements discussion, multiple concurrent communication links must be supported; in the case of non-blocking communications these links may interfere with each other. This interference manifests itself in the form of contention over various network resources; in packet-switched network, this contention will result in degraded performance [161], and is proportional to the volume of traffic congregating in one place. In a circuit-switched system, circuits can be established for some of the

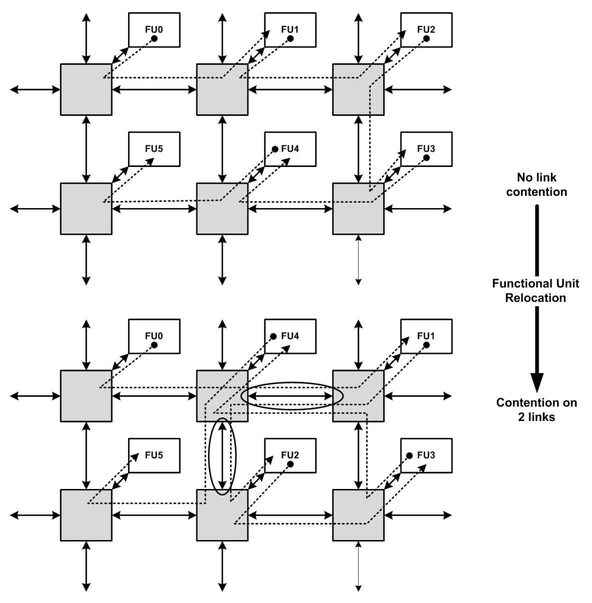


Figure 4.7: Link Contention in Dynamic Systems

system links; however, due to link contention, the establishment of some circuits will automatically block others, which is not an acceptable solution in a high data rate pipeline. This type of problem can be partially addressed by using a combination of circuit and packet-switched methods (see [95]); however, this means that some links (the packet-switched ones) will suffer from degraded performance to a varying extent, depending on topology and traffic patterns.

### Point-to-Point Links and Shared Memory Solutions

Given the desirability for non-blocking communication in stream processing architectures being considered here, point-to-point links may be a desirable solution. However, by default, point-to-point links are fixed, and thus will not reflect changes in the system architecture. This can be addressed by allowing the links themselves to be changed via partial reconfiguration; however, this leads to increased complexity when designing the system (the addition of one or more PRRs to house said links), and a much slower connectivity programming process (due to the need for configuration).

Shared memory systems can, like-wise, offer non-blocking communications, depending on their implementation. Multi-ported memories can allow a number of separate links to be established, and isolate them by assigning each a memory range. However, multi-port memory implementation is limited; most RAM blocks included in modern FPGAs are dual-ported (e.g. [33]). Implementing a larger system is possible through the use of multiple such memory blocks and dedicated control circuitry; however, at that point, such a solution resembles a more expensive fully connected crossbar, making it irrelevant. Alternatively, a single memory block can be used as a shared medium, thanks to a time-multiplexing scheme; however, this approach shares the same problems as the shared bus architecture.

### 4.3.3 MACROS Framework Communication Architecture

Of the five types of communication infrastructures presented above, one or more may be selected for the MACROS Framework architecture. The selection process is informed by the communication requirements associated with stream processors, as outlined at the beginning of this section; the core requirement is support for large aggregate sustained data rate in multiple simultaneous links. Additionally, the dynamic nature of the targeted systems must also be considered when an architecture is selected.

The shared bus architecture allows only single links to be established at any time. It is aimed at master-slave situations, and works best in situations where one master element initiates all communication tasks; it is uniquely unsuited to high data-rate applications and is not considered further here. Dedicated links of various types (point-to-point, point-to-multi-point) may offer excellent communication performance, but are not built to support dynamic changes in their topology natively; their nature does not lend itself easily to programmability.

The third potential communication architecture, which has proven popular in recent years, is the Network-on-Chip. As discussed above, both packet and circuit-switched architectures suffer from potential contention on individual links, which will adversely affect the performance of individual circuits. In streaming systems where the processing task is divided between multiple functional units, all links must offer the same data rate, to ensure that all components can maintain their requisite throughput. While contention can be eliminated through judicious selection of network topology, this selection cannot be maintained easily once the distribution of functional units changes, as is expected in a system with dynamic architecture.

The final potential architecture, and the one that is selected for the MACROS Framework, is the fully-connected crossbar; this architecture is selected because its benefits outweigh its shortcomings. The most important guiding consideration is the fact that the chosen architecture must support high data-rate stream applications, by allowing high sustained data rates in all system links. This is accomplished by the fully-connected crossbar thanks to its non-blocking nature. This feature also ensures that link data-rates can be maintained regardless of the functional unit distribution in the system. Secondly, the fully connected crossbar supports the dynamic nature of the system. Link establishment is easy to implement via programming of output multiplexers. The fact that a fully connected architecture is used facilitates the allocation task, by allowing functional units to be deployed anywhere in the system.

The main shortcoming of this architecture is the large resource overhead it imposes in large systems; however, this same overhead makes the architecture non-blocking, and is, therefore, considered an unavoidable cost in such a system. This overhead can be reduced to a certain extent through Clos network decomposition [162] in situations where this overhead is considered to taxing.

### 4.3.4 Delay Variation Mitigation

The mobility of functional units in dynamic partially reconfigurable systems leads to a secondary problem: run-time delay variation due to the relocation of components. Figure 4.8 illustrates this effect, showing the relocation of a functional unit, and the increased wire length for the new position, which leads to an increase in signal delay. Variation in wire length due to on-chip position can also be found in static systems; however, in static systems the distribution of wire lengths is known at all times, and does not change at run time.

The standard solution in static systems is to select a clock frequency for each link such that the delay does not violate setup times; if a single system clock is used, the selected frequency must be chosen so that the worst delays are supported. This approach can be used in dynamic systems as well; however, in a dynamic system, the increase in delay may only occur in a fraction of modes.

An alternative solution which has been proposed for the mitigation of line length variations is the principle of latency insensitive design [163]. This approach take all wires and divides them into stages of fixed length; longer wires lead to more stages. Each stage is separated by a register, so that changes in delay are converted to changes in latency (see Figure 4.9). The length of each stage wire is selected to meet the target clock frequency requirement. In this way, system links will vary in terms of their latency, with longer wires resulting in a longer latency; however, in all cases, data throughput remains the same.

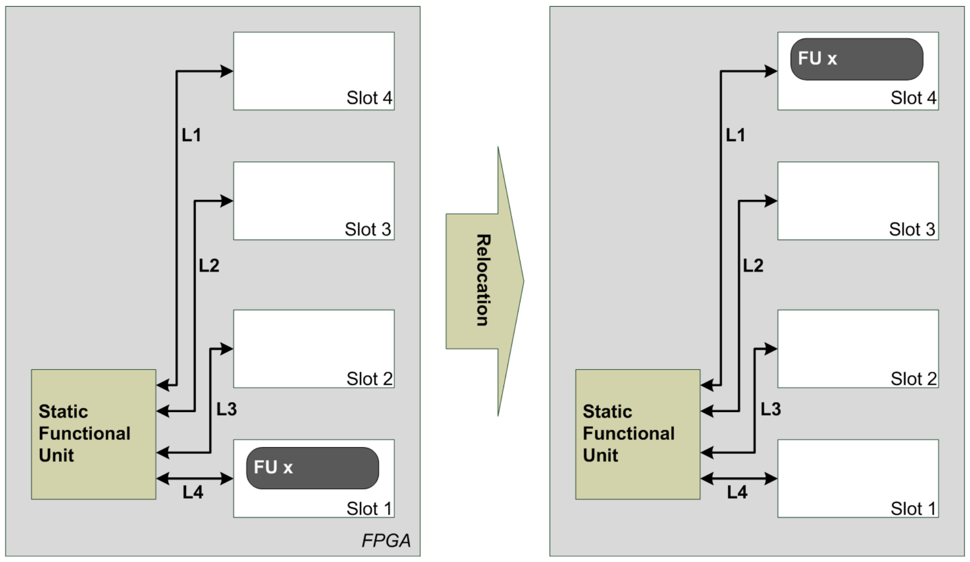


Figure 4.8: Wire Length Variation in Dynamic Systems

This approach lends itself well to pipe-lined systems, as its cost is added pipeline latency. In the case of data-flow architectures, this approach may complicate data synchronization tasks; to address this problem, delay stages will have to be included on select links, to ensure all latencies are identical. Nonetheless, because it decouples clock frequency from on-chip physical relations, MACRO Systems incorporate the latency insensitive design method to ensure that target clock rates are met regardless of physical module distributions.

## 4.4 On-Chip Assembly Mechanism

MACROS Framework Systems aim to address the problems of large implementation complexity and increasing fault rates by allowing run-time architecture adaptation; through architecture adaptation, processors are generated from smaller component parts (functional units), and can be re-generated with different physical circuit distributions via relocation. The architecture adaptation process is composed of two procedures: system assembly and system integration. The assembly process likewise consists of two steps: the downloading of functional unit bit-streams to the target FPGA (configuration) and the establishment of communication links.

This section begins by analyzing the configuration process, with the aim of determining a mechanism for functional unit allocation and configuration scheduling. Following this, the process of link establishment and connectivity change will be analyzed, keeping in mind the communication

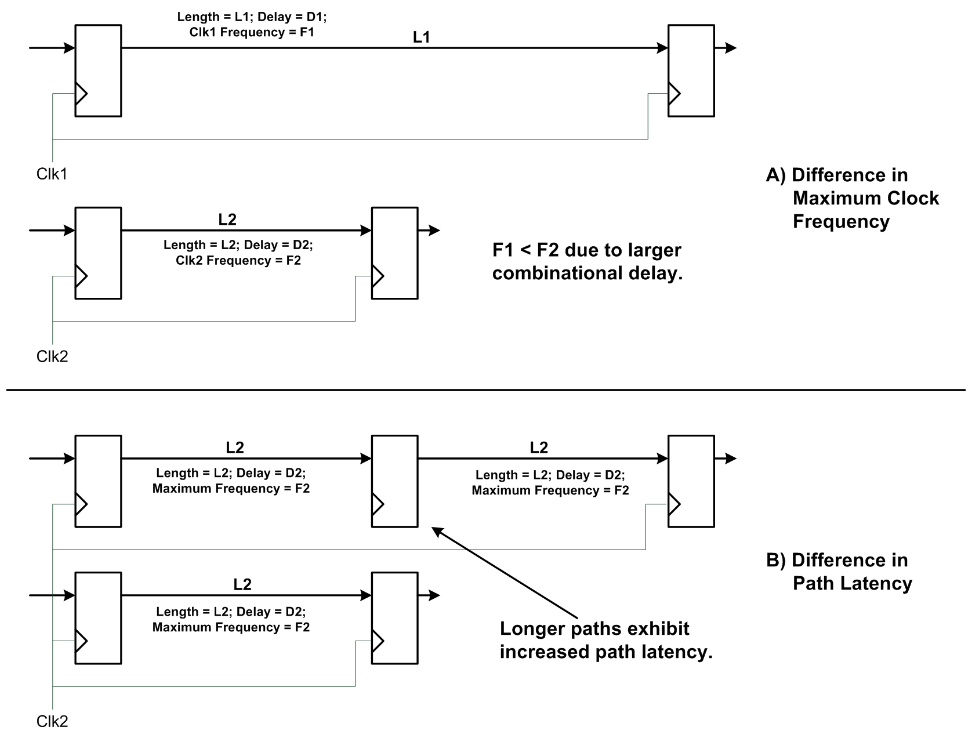


Figure 4.9: Delay Variation Conversion Into Latency Variation

architecture selected above. Based on this analysis, connectivity change algorithms will be derived which dictate how connectivity change takes place in the system.

### 4.4.1 Functional Unit Allocation and Configuration Scheduling Analysis

The mechanisms by which functional units are deployed and relocated have been elaborated in Section 4.2: each functional unit is mapped to a system slot, and one bit-stream is created per slot and functional unit combination. To deploy a functional unit into the system, a partial bit-stream is written to configuration memory via the chosen configuration interface. This configuration process is directly dictated by the device used; no further investigation is required. This leaves two aspects for consideration: functional unit slot allocation, and configuration activity scheduling.

As established in Chapter 3, due to the nature of the targeted applications, resource sharing via time-division multiplexing of functional units is not currently considered for the MACROS Framework. This exclusion will affect both the allocation and scheduling processes described below; it will also exclude the use of existing scheduling methods which treat the configurable system as a shared resource to be multiplexed in time [122, 116, 123, 124, 125, 117, 118, 126, 119, 120, 121].

**Uniform System Slots**

```
for(set of missing functional units){
   for(all system slots){
     If(system slot is free){
          Allocate functional unit to current system slot;
     }
   }
}
```

**Multiple Slot Types**

```
for(set of missing functional units){
   for(system slots supporting current functional unit){
     If(system slot is free){
          Allocate functional unit to current system slot;
     }
   }
}
```

Figure 4.10: Allocation Algorithms for Different Slot Configurations

As described in Chapter 3, changes in mode or detected faults will lead to the generation of a new ASP; as part of this ASP generation process, a collection of missing functional units will be found which must be allocated into the system. This allocation task is simplified by the use of slots versus regions; rather than having to select a two-dimensional coordinate (and deal with the problem of fragmentation), an index must be selected instead (corresponding to a system slot).

The allocation process is dictated by the type of system slots used; the system slots may be uniform, or may be separated into types. If uniform slots are used, then all slots have the same resource distribution and capacity, and all slots can house any functional unit. Alternatively, system slots may be separated into types, based, once again, on their resource content; this approach may be needed in instances where functional units exhibit a wide variation in size or in the type of resource they use. Figure 4.10 shows the allocation processes for these two situations; in each case, the allocation process consists of searching through a slot record for an entry which matches a specific criteria (i.e. finding the first free slot that can house a given functional unit) and assigning said functional unit there; thus, the complexity of both algorithms is $O(n)$, where n represents the number of system slots.

Once allocation is complete functional unit deployment can be considered. Configuration activities are scheduled based on the nature of the system, and follow one of three paths:

- Configuration for systems where $Tagen_{i-j} \leq (T_p - T_w)$.

- Configuration for systems where $Tagen_{i-j} > (T_p - T_w)$ and spare slots are available.
- Configuration for systems where $Tagen_{i-j} > (T_p - T_w)$ and no spare slots are available.

If $Tagen_{i-j} \leq (T_p - T_w)$, configuration activities are deferred until the point in time when all system functional units find themselves between work periods. At this time, the missing allocated functional units are configured sequentially. If $Tagen_{i-j} > (T_p - T_w)$ and spare system slots are available, configuration activities are undertaken as soon as missing functional units have been allocated. Finally, if $Tagen_{i-j} > (T_p - T_w)$ and no spare slots are available, system processing will have to be interrupted prior to undertaking configuration activities; this is done to avoid the inclusion of erroneous data due to uncontrolled switching during the configuration process. Therefore, configuration of all allocated functional units will take place once all processing has been interrupted; this interruption process is described in the following section.

It is important to re-iterate at this point that the above discussion has addressed *only* functional unit allocation and deployment via configuration. Thus, the scheduling task described above explicitly addresses only deployment (configuration activities). A second layer of scheduling activities takes place once functional units are deployed in the system and processing tasks must be scheduled; such scheduling activities are considered in Section 4.5.

### 4.4.2 Connectivity Change Analysis

Connectivity changes consist of three specific operations: link establishment, link changes and functional unit disconnection. The way in which these operations are implemented is based on the type of communication infrastructure being used. The MACROS Framework makes use of fully connected crossbars, meaning connectivity change operations are implemented by setting appropriate control values for each output multiplexer in the system crossbar.

The third connectivity change operation is a disconnection: through this action, a functional unit is isolated from the rest of the system (i.e. no stream data can reach it). This is accomplished by driving some constant value to the component data inputs; the specific value selected depends on the nature of the system (usually a constant 0 value can be used). Towards this end, every output multiplexer in the crossbars used will have one setting which connects the output to a constant value, to be determined based on the application specifications. This disconnection operation will

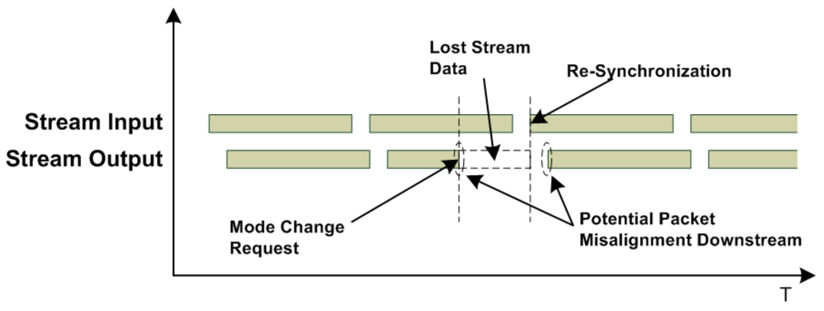


Figure 4.11: Preemptive Mode Change

be used if a functional unit is no longer needed in a new mode, or when the system must be interrupted to allow configuration activities (as described above).

Connectivity changes in response to mode changes can be undertaken in one of two ways: in a preemptive fashion, or a non-preemptive one. A preemptive connectivity change takes place as soon as the mode change request is received and configuration activities have been completed (or the complete system was disconnected). Alternatively, a non-preemptive approach would see connectivity changes deferred to some later point in time, with the aim of minimizing the impact of the mode change on system behavior (see Chapter 3).

The main advantage of preemptive connectivity changes is that system response to mode change requests is very fast. The main disadvantage of preemptive mode changes is that they may interfere with the structure of the data stream being generated. If a mode change occurs during a system work period, the changes in connectivity needed for the new mode will interrupt data transfer in the system; various pipelines will then have to be flushed, and associated control circuitry will have to be reset. The structure of the data stream being generated will also be interrupted, as shown in Figure 4.11; this interruption in stream structure will cause problems for processors further downstream, as they will have to re-synchronize with the new data stream, and flush previous data from their pipelines.

The non-preemptive approach to connectivity changes delays said changes until the system completes the current work period; this basic principle was discussed in Chapter 3. Taking this approach allows for the possibility of seamless mode changes, such that resets and pipeline flushing are not required. If seamless connectivity changes (as defined in Chapter 3) cannot be implemented then the temporal arrangement of stream packets will be temporarily altered, as shown in Figure

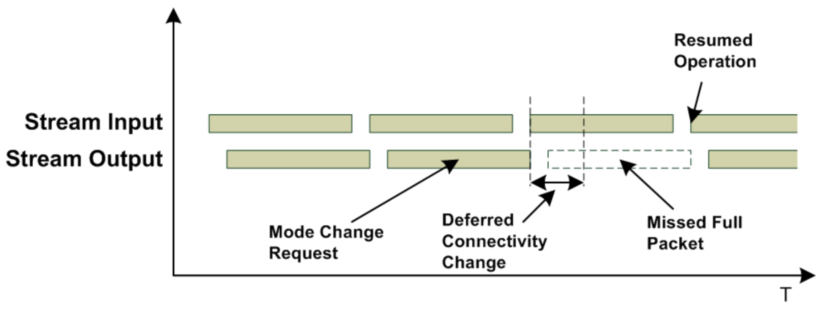


Figure 4.12: Non-Preemptive Mode Change

4.12. This means that lost data will conform to packet boundaries (data is lost in packet increments), which facilitates re-synchronization down-stream. Because of this feature, as well as the possibility for seamless mode changes, the MACROS Framework use non-preemptive connectivity change procedures.

### 4.4.3 Non-Preemptive Connectivity Change Algorithms

Above it was established that connectivity changes associated with a change in mode will be performed between work periods, so that work periods are not affected. When treated at the functional unit level, connectivity changes applied to a given unit must: A) not interrupt local processing activities inside the functional unit and B) not interrupt the processing activities of functional units down-stream. An algorithm must therefore be constructed which ensures that connectivity changes for specific functional units occur when they *as well as all down-stream modules* find themselves between work periods.

To facilitate the process of data dependence analysis, a MACRO System can be represented as a directed acyclic graph, where nodes represent functional units and directed edges represent unidirectional data links between units (Figure 4.13 shows an example system, including data dependencies during a connectivity change). To ensure that processing activities are not interrupted, a change in connectivity in a given node requires that all descendant nodes find themselves between work periods. This can be accomplished by disconnecting all descendants, starting with the one furthest down-stream, as they finish their current work period. This process continues until all descendants have been disconnected, at which time it is safe to change the connectivity of the target node. After this change, each disconnected node is re-connected to its parent, once said parent

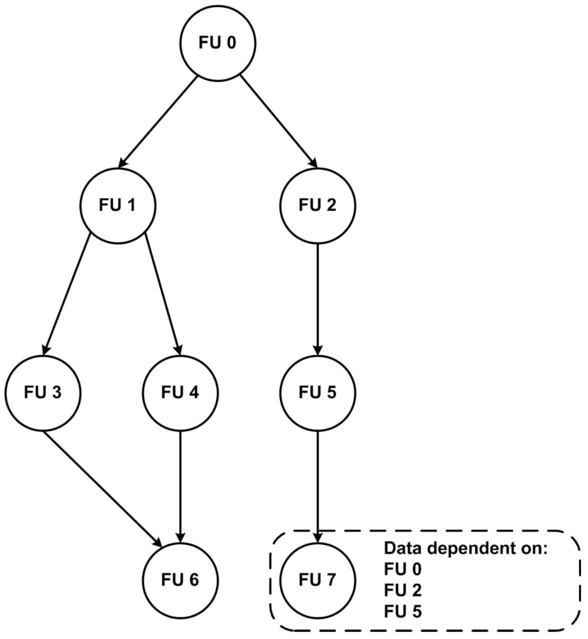


Figure 4.13: Graph Representation of a Stream Processor

has been connected. This connectivity change process is, hence-forth, referred to as the Canonical Connectivity Change process, as it stems directly from the concept of non-preemptive connectivity changes as described above, and makes no further assumptions about the workload structure.

To better illustrate the Canonical Connectivity Change process, the example in Figure 4.14 is presented. Here, a simple pipe-lined system consisting of four active functional units is shown. As a result of mode change, functional unit FU2 must be replaced with FU2b; thus, functional unit FU2 must be disconnected from the system, and functional unit FU2b must be connected into the system, after having been deployed into a system slot via configuration. Following the process prescribed above, functional unit FU4 is disconnected from the system (step 1) once it finishes its current work period; this is considered a safe operation, as there are no other functional units further down-stream. Following this, functional unit FU3 is disconnected once it enters it finishes its work period (step 2). At this point, functional unit FU2 can be disconnected from the system, once it finishes its current work period (step 3). Functional unit FU2b can now be connected into the system, once its up-stream parent (FU1) has been connected; given that FU1 has not changed connectivity during this process, FU2b can be connected into the system immediately (step 4). Finally, FU3 can be re-connected to the system once its parent (FU2b) has been connected (step 5), followed by FU4, once its parent (FU3) has been connected (step 6).

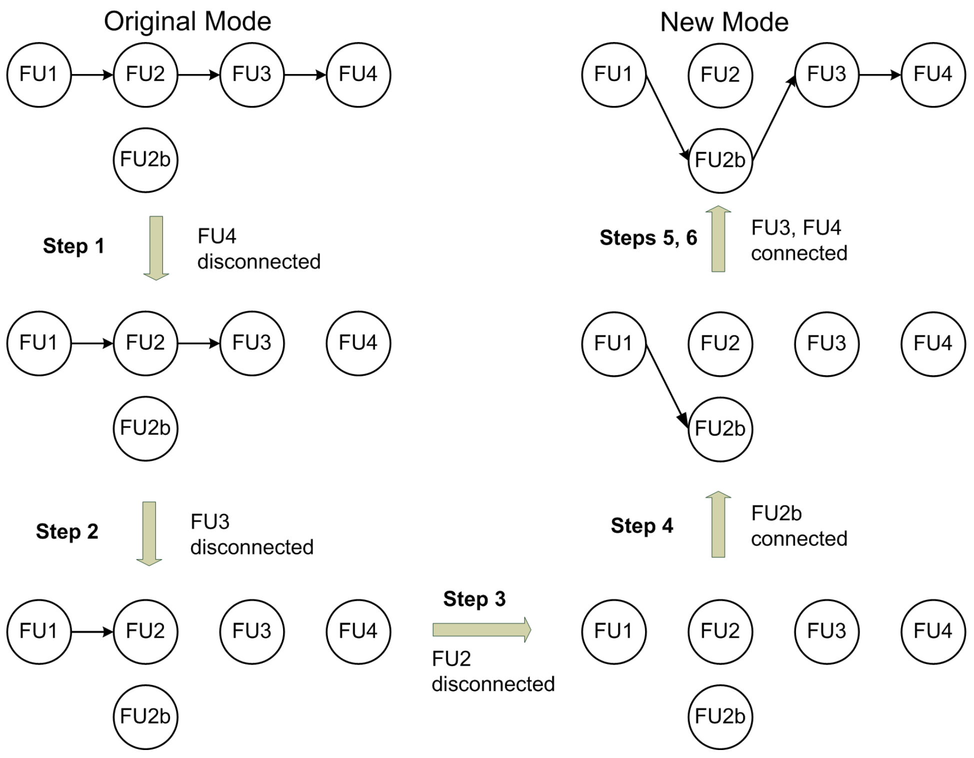


Figure 4.14: Example of the Canonical Connectivity Change Process

The pseudo-code for the Canonical Connectivity Change algorithm is shown in Figure 4.15. In the worst case, the algorithm dictates that, for each functional unit in the system, two searches must be made: one to determine if all down-stream functional units have been disconnected, and one to determine if all up-stream parents have been connected. Both these searches must be conducted across all functional units in the system. Thus, the complexity of this algorithm is $O(n^2)$, where n represents the number of functional units in the system.

This algorithm relies on isolating down-stream modules from the system when they find themselves between work periods; returning to the Chapter 3 analysis, if $T_p - T_w$ is long enough, the modules will be disconnected, the connectivity change will be made and all components will be reconnected within this time frame. If this requirement is met, the mode change is seamless; an example of this process is shown in Figure 4.16. Alternatively, the mode change will not be seamless and will affect one work period; however, the fact that modules are disconnected between work periods means that when they are reconnected, they will be able to resume processing without the need for global or local reset or other forms of interference.

An alternative approach to ensuring that all child nodes are between work periods before mak-

```
while(local or up-stream connectivity request){
   if(FU has descendants)then{
      if(all FU descendants are disconnected)then{
         if(FU is not in a work period)then{
            if(FU parent is connected)then{
               change local connectivity;
            }
            else{
               wait for parent to be connected;
            }
         else{
            wait until FU finishes current work period;
         }
      }
      else{
         request disconnection of all descendants;
      }
   }
   else{
      if(FU is not in a work period)then{
         if(FU parent is connected)then{
            change local connectivity;
         }
         else{
            wait for parent to be connected;
         }
      }
      else{
         wait until FU finishes current work period;
      }
   }
}
```

Figure 4.15: Canonical Connectivity Change Process

ing connectivity changes is to implement said changes *only when all system functional units find themselves between work periods.* Taking this approach requires less steps as the algorithm in Figure 4.17 shows. Given that all connectivity changes happen at the same time, (once all units are between work periods), this algorithm can be performed faster, and its requisite time is not dependent on the number of child nodes. The algorithm relies on a determination of when all functional units are in their safe state; thus, the algorithm requires one search across the system, and has $O(n)$ complexity, where n represents the number of functional units. This means that this approach to connectivity change is more likely to offer seamless mode changes. This algorithm is referred to as the Minimal Connectivity Change process; it *requires* that the time periods between

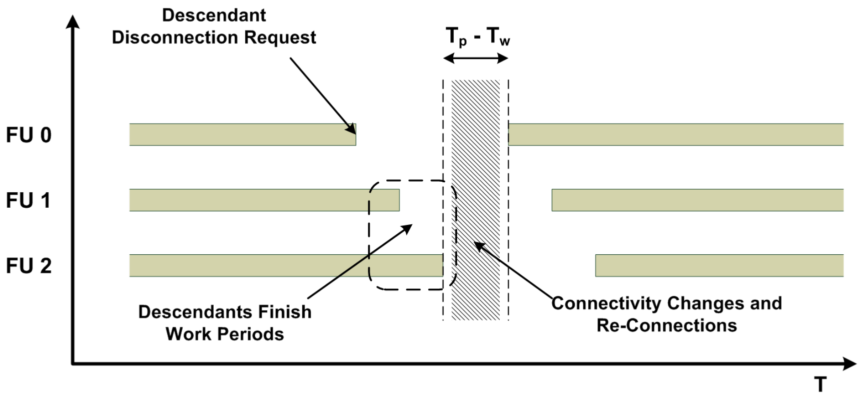
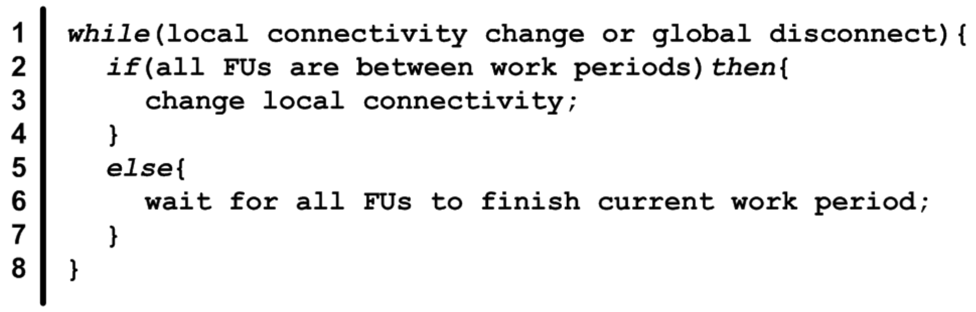

Descendant Disconnection Request
$T_p$ - $T_w$
FU 0
FU 1
FU 2
Descendants Finish Work Periods
Connectivity Changes and Re-Connections
T

Figure 4.16: Seamless Mode Change Using the Canonical Process



```
while(local connectivity change or global disconnect){
  if(all FUs are between work periods)then{
    change local connectivity;
  }
  else{
    wait for all FUs to finish current work period;
  }
}
```

Figure 4.17: Minimal Connectivity Change Process

consecutive work periods for all modules overlap at some point (see Figure 4.18), and *will* offer faster connectivity changes than the Canonical Connectivity Change process if this requirement is met.

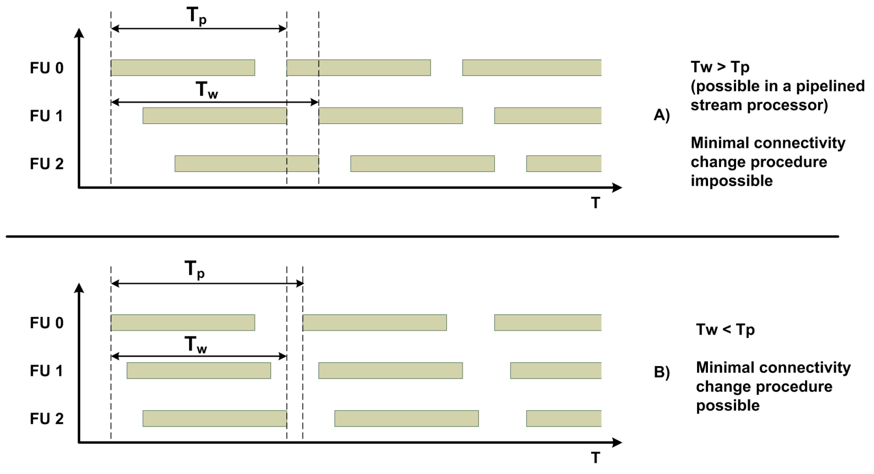

$T_p$
$T_w$
FU 0
FU 1
FU 2
T
A)
Tw > Tp
(possible in a pipelined stream processor)
Minimal connectivity change procedure impossible
$T_p$
$T_w$
FU 0
FU 1
FU 2
T
B)
Tw < Tp
Minimal connectivity change procedure possible

Figure 4.18: Feasibility of Minimal Connectivity Change Process

Adopting once again the example system used above, a Minimal Connectivity Change would occur as shown in Figure 4.19. Once a mode change is detected, the system must wait until *all* functional units have finished a work period; once this occurs, all components are either re-connected or disconnected (step 1). Finally, the system resumes operation (step 2).

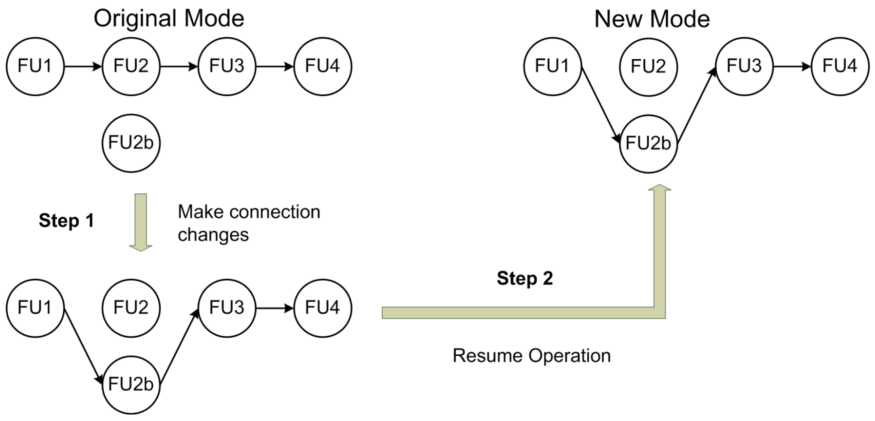


Figure 4.19: Minimal Connectivity Change Process Example

When comparing the two connectivity change processes, the Canonical process is more complex, whereas the minimal process is simpler to implement. However, the Canonical process will still work when $T_w > T_p$, in which case seamless connectivity changes and the Minimal connectivity change process cannot be implemented. In such instances, the Canonical algorithm controls the nature in which stream data is lost; here, data is lost along packet boundaries, which will facilitate the re-synchronization process and may eliminate the need to flush pipelines.

## 4.5 Functional Unit Integration Mechanisms

As stated above, the MACROS Framework aims to implement run-time ASP generation in a class of systems; this generation process relies on architecture adaptation which consists of system assembly and integration into a mode-specific ASP. The assembly process has already been addressed in previous sections of this Chapter. Once a full ASP has been assembled, the functional units which make up its functionality must be integrated into a running system. This integration step consists of scheduling of processing activities and synchronization of data transfers. This section will analyze the problem of processing activity scheduling in systems which do not rely on shared resource models, as well as the problem of data synchronization in pipe-lined and data-flow systems; it is, once again, noted that the processing activity scheduling task is separate from the configuration scheduling task discussed in Section 4.4.

### 4.5.1 Scheduling and Data Synchronization in Multi-Mode and Dynamically Reconfigurable Pipe-Lined and Data-Flow Environments

In a shared resource environment, the scheduling problem addresses the question of when various tasks obtain access to one of the available computing resources (which could be either an instruction-based processor or a reconfigurable resource). All tasks have an associated deadline, and the aim is to arrive at a schedule which permits (ideally) all tasks to complete before their respective deadlines. The scheduling task can occur either on-line (scheduling decisions are made as the system is running) or off-line (where the schedule is derived ahead of time, and is enforced at run-time). Chapter 2 lists a number of examples of scheduling methods for shared-resource environments.

If a shared resource model is not used, then all functional units needed for a given ASP are present in the system as long as the ASP is active. However, scheduling of processing activities must still take place; individual functional units must be activated such that they commence processing at the correct time in relation to the data being processed. Thus, functional unit activation schedules must still be derived and enforced. These schedules will directly rely on the data synchronization mechanisms being used.

In a pipe-lined system, data flow follows a single path; thus, data synchronization is reduced to matching cycle time between functional units. In MACROS Framework systems, this synchronization is accomplished by ensuring that all functional unit architectures share the same cycle time. In a data-flow architecture, the data synchronization task requires that different potential latencies and cycle times are accounted for. In the MACROS Framework, this synchronization is accomplished by enforcing that A) all functional unit have the same cycle time and B) *all* parallel functional units have the same latency; these requirements are illustrated in Figure 4.20.

### 4.5.2 Scheduler Implementation Solutions

#### System-Wide Scheduling

For a given application and system architecture the scheduling process can be implemented in multiple ways. The most basic approach is to determine a schedule off-line, based on the latencies of

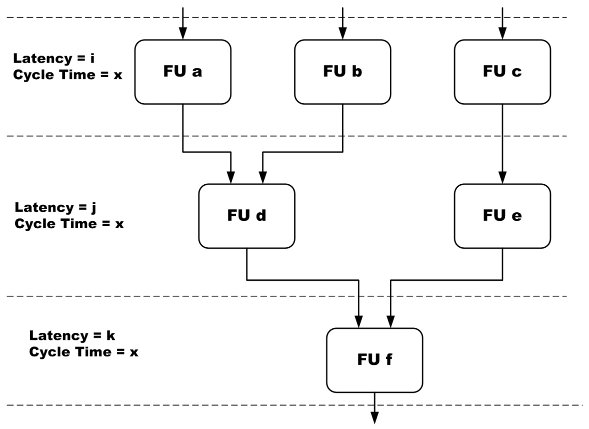


Figure 4.20: Data-flow Synchronization in MACRO Systems

each module; this schedule can then be enforced using a simple counter-based time-keeper which activates each functional unit in sequence. The behavior and structure of such a system scheduler is shown in Figure 4.21; the scheduler monitors incoming stream data, and once the correct control information is detected, all functional units are activated in sequence. The structure of this scheduler is based on a simple control FSM, a counter and a number of comparators, indicating when one or multiple modules must be activated.

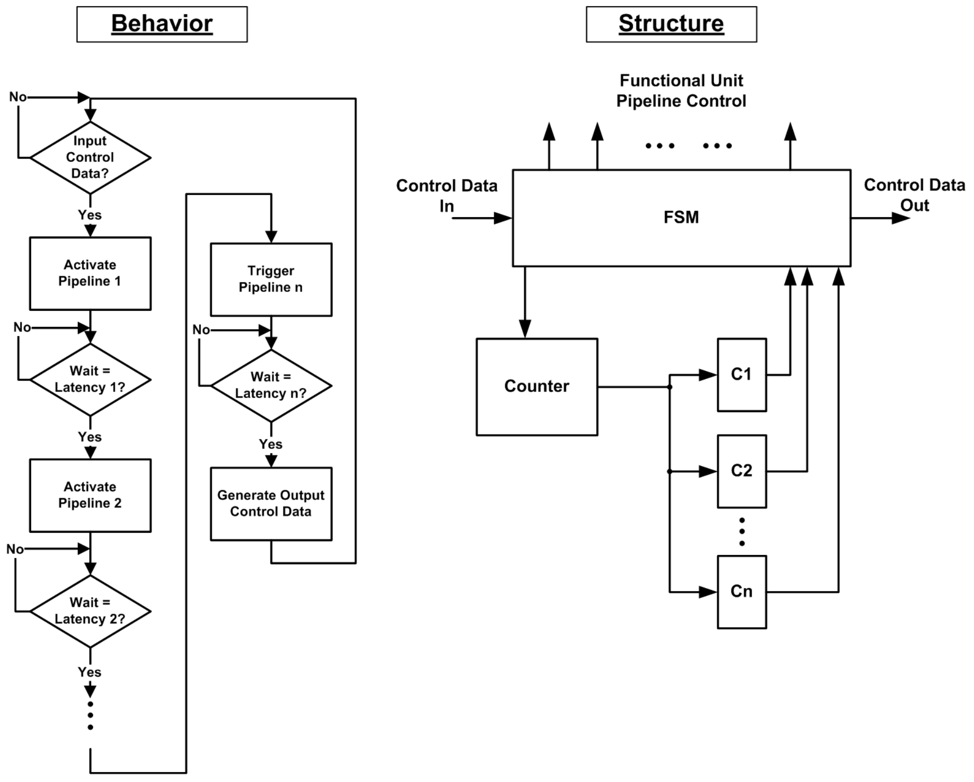


Figure 4.21: System-Wide Scheduling Circuit Structure and Behavior

In a multi-mode system, the above scheduler behavior must change, as shown in Figure 4.22; for each mode, the scheduler may have a different schedule, with different activation times. In terms of implementation, this complicates matters, since now multiple comparators are needed, and the FSM must include additional states for the various modes. The circuit complexity for such a scheduler immediately increases, and the increase is proportional to the number of functional units and modes.

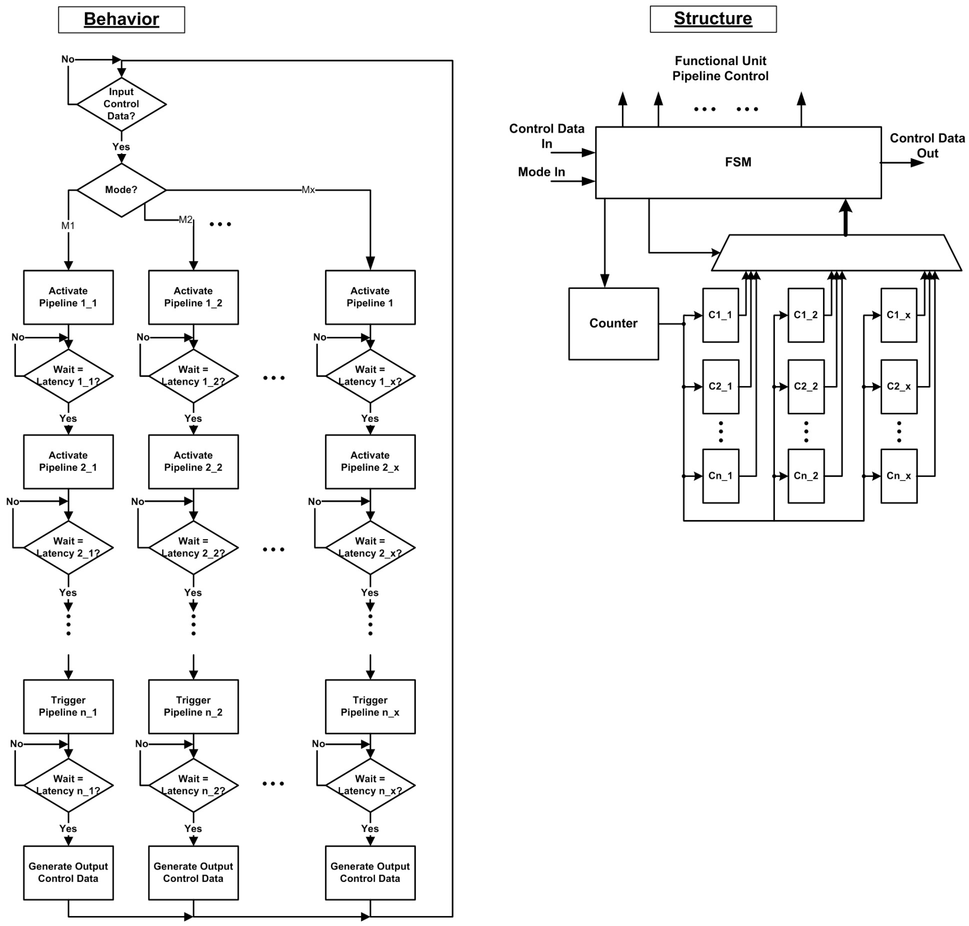


Figure 4.22: Central Scheduling Circuit Structure and Behavior in Multi-Mode Scenario

An alternative implementation method for such a scheduling element is to use a programmable instruction-based processor, which can (in theory) support a large number of schedule variations. However, such processors impose both a resource and performance overhead. A soft-core processor such as MicroBlaze [164] requires significant resources to implement, and may be limited in the performance it offers. Before selecting such an approach, a quantitative analysis of such a solution would have to be undertaken to determine its capabilities.

### Per Functional Unit Scheduling

When analyzed from the point of view individual functional units, the scheduling task is simplified to one activity: activating the local functional unit data-path at the correct moment, so that incoming data is registered and processed correctly. This task can be accomplished by monitoring control data as it arrives at the functional unit input port. Based on this observation, the scheduling task is divided into multiple smaller sub-tasks, each addressing the schedule of only one functional unit. Taking this approach, each scheduling circuit will have minimal complexity; the general structure and behavior of such a scheduler is shown in Figure 4.23. However, this circuit becomes a fixed overhead in each functional unit.

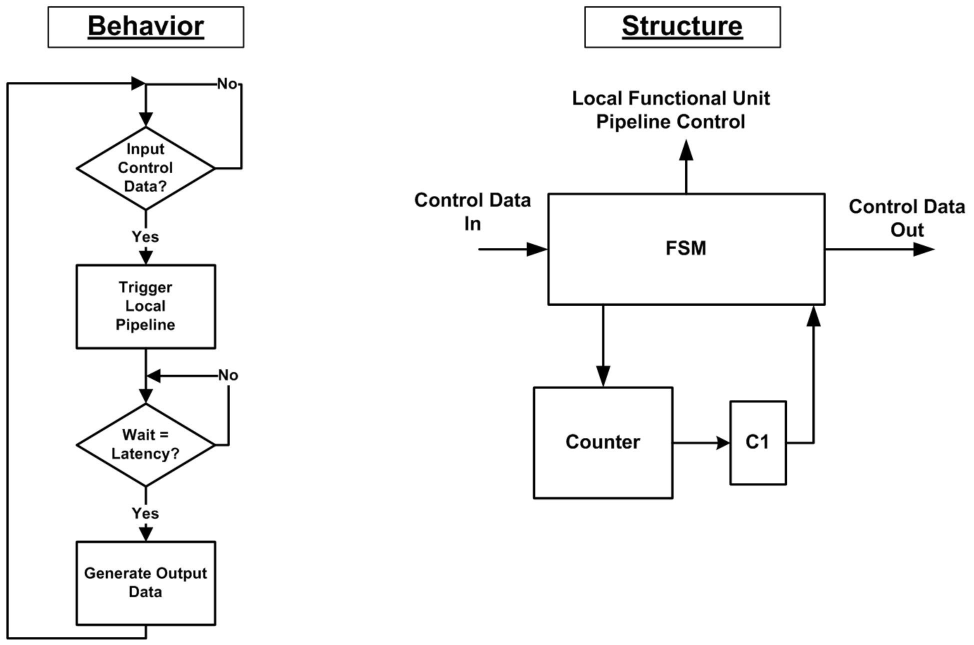


Figure 4.23: Distributed Scheduling Circuit Structure and Behavior

Using this distributed scheduling approach leads to the simplification of the scheduling task when multiple modes are considered. Each scheduling circuit is responsible for determining if a specific functional unit is present or not; if the scheduling circuits are embedded *inside* the functional units, then this step can be omitted. The local scheduler will work if it is present in the system (i.e. has been configured, is being clocked and receiving input data) as a result of the fact that the functional unit is present; when the functional unit is absent, so is the scheduling circuit. For such a solution to be used, both stream data and control information must be propagated through the system, which can lead to increased resource use in communication architecture.

## 4.6 Central Versus Distributed Control Structures in the MACROS Framework

The above discussion on scheduling approaches suggested (in a qualitative fashion) that central control circuits may exhibit larger complexity and resource requirements than distributed implementations, particularly when variations in architecture based on multi-mode behavior are considered. Furthermore, the applications being considered exhibit strict timing constraints; control operations in the resulting systems should add minimal timing overhead, which suggests the exclusion of sequential, instruction-based processors from this task. With these basic considerations in mind, this final section will outline the use of a distributed control architecture in the MACROS Framework. The complete verification and validation of this control architecture, via quantitative analysis and comparison, will be presented in Chapter 8.

When looking at the MACROS Framework, three main control aspects emerge: functional unit bit-stream management and configuration, link establishment and connectivity change procedures, and scheduling and synchronization. The functional unit configuration process is implicitly reliant on a singular element: the configuration ports available in a given device, and the bit-stream storage system used; as a result, the configuration process is not distributed in nature. In a distributed implementation, access to these two elements would have to be managed via some form of central interface element (a memory controller, for example); the inclusion of such an element would add to the complexity and the overhead of the system, and would not alleviate the sequential nature of the configuration process in single chip solutions. Thus, in the case of the configuration and bit-stream management subsystem, a distributed architecture is not expected to reduce the structural complexity or the needed resources; it may be beneficial in multi-chip systems, however, by allowing multiple configurations to be undertaken in parallel.

The implementation of on-chip scheduling has already been discussed in the previous section, and consists of attaching small scheduling circuits to functional units which activate functional units individually. By adopting this approach, no further intervention from other system elements is required for the scheduling task; as long as stream data (including all explicit control information) is made available, the functional units will operate automatically.

The third area being considered is the on-chip link establishment process. Two algorithms for non-preemptive connectivity change procedures have already been derived in Section 4.4. These algorithms can be treated in a distributed fashion, by analyzing connectivity changes from the point of view of individual functional units. The Minimal Connectivity Change process requires only a global flag which indicates when local connection changes can be made. The Canonical Connectivity change process requires that, locally, three search are conducted to determine if up-stream connectivity changes exist, if down-stream functional units have been disconnected and if the up-stream parent has been connected. When analyzed from the point of view of individual functional units, the complexity of both algorithms decreases. The Minimal Connectivity Change process now has constant complexity, regardless of of the number of functional units in the system. Meanwhile, the Canonical Connectivity Change process now has complexity $O(n)$, as three searches are conducted across all functional units. These algorithms can be implemented through the use of control circuits allocated to each system slots, with the sole responsibility of implementing connectivity changes, via programming of the system crossbar; this concept is illustrated in Figure 4.24. The reduction in complexity for each algorithm means that high performance can be achieved in these control circuits without undue expenditure of resources.

Each connectivity control circuit will implement either the Canonical or Minimal connectivity change process; thus, each control circuit will track connectivity information for its associated functional unit, and must also receive connectivity information for all other functional units in the system. To allow the implementation of the connectivity change algorithms, all connectivity control circuits must be interconnected and communicate, thus allowing each circuit to determine up and down-stream connectivity in the system.

The distributed approach described so far does not address how system link information is stored and distributed; before establishing the needed communication links, the connectivity control circuits must know what these desired links are. For each mode, this information takes the form of a sequence of entries describing the connections of each functional unit source and sink (input and output). This information can be stored centrally (as is the case with the the configuration and bit-stream management subsystem); however, since this information is specific to each mode and functional unit, an alternative is to embed storage of this information inside each functional unit

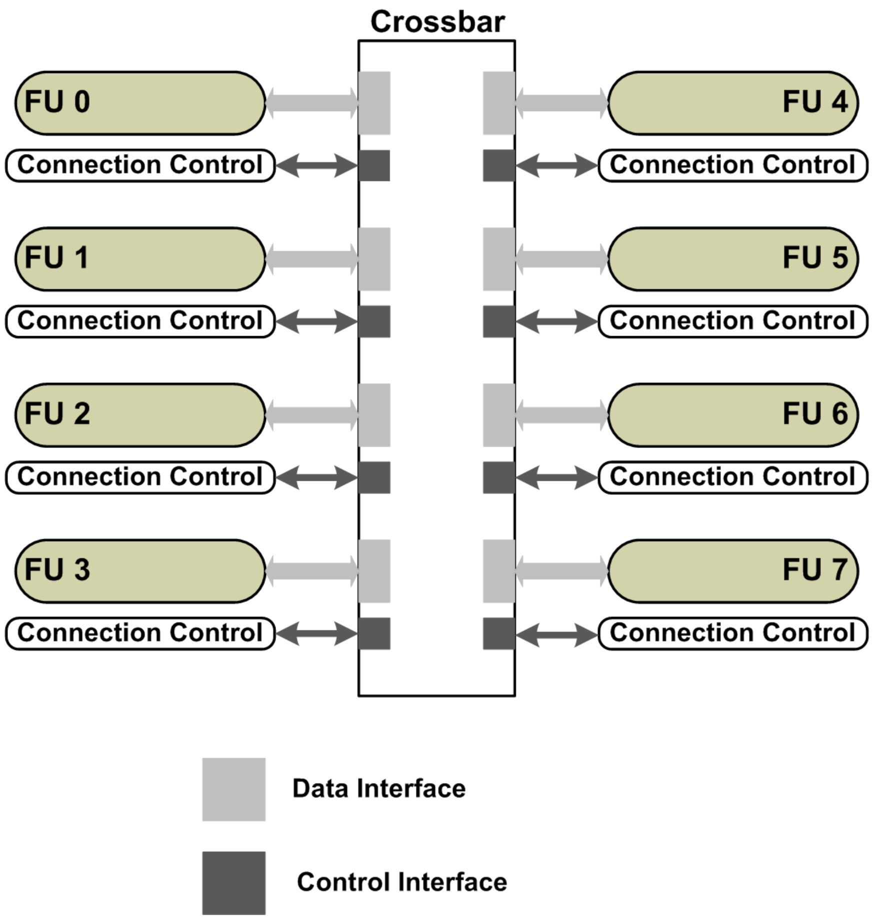


Figure 4.24: Distributed Connectivity Control Concept

proper. In essence, each functional unit will receive a mode identifier, and based on this information make the appropriate connectivity request of its connectivity control circuit.

The above control architecture was selected with the aim of permitting the execution of multiple control operations in parallel, which can accelerate portions of the assembly and integration process. Link establishment, data synchronization and processing schedule enforcement tasks are all performed in parallel by individual control circuits in the distributed control system; these operations can be overlapped in time, which can reduce the time cost of the assembly and integration tasks. In contrast, central control architectures (either based on instruction-based processors or dedicated control circuits) do not have the same capability for parallel operation; such solutions can add either resource overhead, timing overhead, or both, to a system. A more detailed analysis of such potential overheads, as well as a comparison of central and distributed control architectures is provided in Chapter 8. In addition, when considered from a fault resilience point of view, a central control architecture introduces a single point of failure into the system architecture, and would require additional hardening steps; in contrast, a distributed architecture avoids the presence of such a single point of failure, and can by-pass the need for additional hardening steps.

System-level communication and control architectures can also be adapted for on-chip utilization; this would negate the need for developing a new control architecture for the MACROS Framework. Chapter 2 examined two such architectures, the PCI Express bus and the Universal Serial Bus (USB), both of which support run-time system assembly and integration. However, such system-level architectures and their associated protocols exhibit a number of characteristics which make them unsuitable to on-chip, high data-rate systems. First of all, system-level assembly and integration mechanisms operate over long time frames; functional unit deployment is often undertaken by human operators, and the assembly and integration procedures are considered over time periods of seconds or longer. The mechanisms used in the assembly and integration task are complex, require long periods of time to complete, and rely on the interaction of complex system elements; as a result, a minimum level of complexity is assumed for functional units. In an on-chip environment resources are limited, and on-chip functional units are less complex than system level ones; the control infrastructure must offer fast assembly operations with limited resource overhead, which favors simple assembly protocols. Secondly, in a high data-rate on-chip environment control information exhibits a much lower bandwidth requirement compared with data being processed; this aspect is not reflected in system and board-level architectures, which transmit control information with all transactions. A dedicated on-chip control architecture can take advantage of this difference to provide fast assembly and integration operations with limited overhead.

## 4.7 Summary

The previous Chapter has presented an initial analysis as part of the investigation into the mechanisms used in the MACROS Framework. This initial analysis is of a conceptual nature; thus, the analysis at this stage is primarily qualitative in nature. A more detailed analysis and validation relies on the technical details of the implementation of the presented mechanisms, which is presented in in Chapters 5 and 6. Thus, a quantitative analysis will be deferred until Chapter 8.

# Chapter 5

# MACROS Framework Architecture

## 5.1 Introduction

The preceding chapter has presented the basic principle of operation of the MACROS Framework, and the mechanisms which make this operating principle possible. However, these mechanisms were presented at a conceptual level, and could only be analyzed in a qualitative fashion at that stage. Chapter 5 takes the conceptual aspects introduced thus far and transforms them into a concrete on-chip architecture template; the term template is used because it reflects the customizable aspect of the architecture in response to application specifics.

The chapter will begin by introducing the top-level on-chip architecture of MACROS Framework systems; the building blocks of such systems are introduced, and their general interaction is described. Each individual building block is then addressed in detail through descriptions of their architecture and behavior. Since a framework is being discussed, each section will conclude with a discussions of how the presented architecture can be modified depending on various design requirements.

## 5.2 Top-Level MACROS Framework Architecture

A top-down approach will be adopted when describing the MACROS architecture. The top-level system architecture will be described first; at this level, system sub-modules are treated primarily as black boxes. The emphasis here is to establish the connectivity and behavioral interdependency between the major elements of the system. The internal aspects of these elements will then be described in further sections. Likewise, the initial customization discussion addressed top-level parameters of the system.

### 5.2.1 Top-Level Architecture

When considered at the top level, all MACROS Framework systems consist of four main elements. The first is a collection of CMFU; these are augments functional units which support semi-autonomous behavior (their collaborative aspect). The second is a DCCI; this system component provides all communication links, as well as all control functionality needed for on-chip processor assembly. Third, the system contains a BCM which is responsible for bit-stream storage, CMFU allocation and configuration activities. The fourth major system element is the Mode-Mapper which is responsible for identifying current modes. Finally, as with any digital system, a MACROS Framework systems will also include power and clock distribution, as well as I/O interfaces. The complete top-level system structure is shown in Figure 5.1.

The final building blocks of the MACROS Framework on-chip architecture are *system slots* used to house CMFUs; the rationale for their use was provided in the previous chapter. Unlike all other elements of the architecture, system slots are defined not through schematics or HDL, but rather through constraints provided to the CAD tool-chain used [165]. Each system slot has a fixed data and control interfaces to the DCCI; the details of this interface are based on the type of stream being processed, and will be described in detail in the next Section. System slots can be either uniform, or of different types, differentiated by their resource content, size and connectivity capabilities.

The collection of system CMFUs form various ASP architectures based on their combination in the system. CMFUs can be separated into two classes: static and dynamic. Static CMFUs are expected to be present in the system continuously, due to the fact that they are needed for every mode of operation. Input and Output oriented CMFUs connected to device pins will often fall in this category, due to the static nature (in most systems) of board-level connections. Dynamic CMFUs, on the other hand, are not needed at all times; they are loaded into system slots depending on the current mode. Static CMFUs can also be allocated to system slots and implemented as partial bit-streams, despite their static nature; this will improve system fault resilience, as it allows static

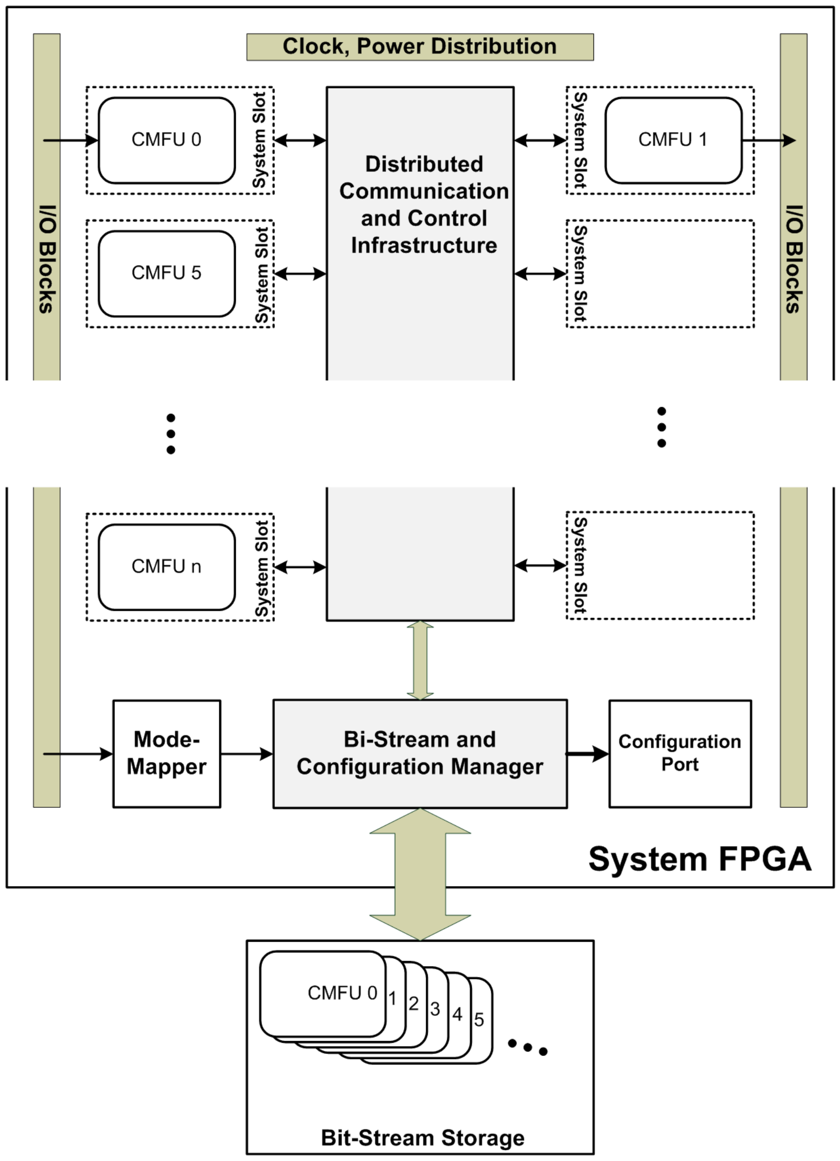


Figure 5.1: MACRO System Architecture

CMFUs to be scrubbed.

The DCCI is responsible for providing all communication links between system slots (and, by extension, the CMFUs residing there) via a fully-connected crossbar. The DCCI also contains all circuitry needed to implement one of the non-preemptive connectivity change procedures described in Chapter 4. Although it is presented as a single element in Figure 5.1, the DCCI has an distributed internal architecture (hence its name); this distributed architecture will be described in detail later in this chapter.

In Figure 5.1, the BCM is shown as an on-chip component. However, the BCM can be implemented either on-chip or external to the target FPGA. If implemented on-chip the BCM requires an I/O interface to external non-volatile storage where bit-streams are stored. In addition, for on-chip implementations, the BCM requires some form of external configuration support which performs the initial configuration step when the device is powered (for example, the Xilinx Platform Flash XL configuration and storage device [166]). If implemented as an external device, the BCM can perform all configuration steps (including the initial one); however, this approach requires the use of a secondary external device, such as a smaller FPGA.

Finally, as previously mentioned, mode data is received from a separate system component, referred to as the Mode-Mapper (so called because it maps stream and environmental conditions into a mode ID); this component can be either external or on-chip. The Mode-Mapper is considered to be a static system element; it must always be present in the system (on-chip or off) to provide mode information which the rest of the system can use. In terms of connectivity, the Mode-Mapper must receive stream data for monitoring purposes; as well, depending on the modes supported, it may require I/O interfaces to external sensors (for example, in a power limited system).

### 5.2.2 Link Structures in the MACROS Framework

The MACROS Framework architecture supports both pipe-lined and data-flow system structures. This subsection will describe how each system type is implemented in the presented architecture. Due to the high data-rate nature of the targeted applications, full duplex communication is required; thus, it is assumed that each CMFU has separate input and output ports. Input ports are connected to data sources referred to as *up-stream* sources and output ports are connected to data sinks referred to as *down-stream sinks*.

In a pipe-lined system, all functional units are assumed to have *one* input and *one* output interface; this leads to the linear link structure associated with such systems (Figure 5.2 - A). In data-flow systems, each CMFU may have one or more input interfaces, and one output interfaces; in this way, all potential link types can be established, as shown in Figure 5.2 - B.

In a pipe-lined system, each CMFU will use one DCCI port, which offers one duplex link (input and output); this implementation is shown in Figure 5.3 - A. In a data-flow architecture, two types of CMFUs can occur: A) CMFUs with one input and one output interface and B) CMFUs with multiple input interfaces and one output interface. CMFUs with one input and one

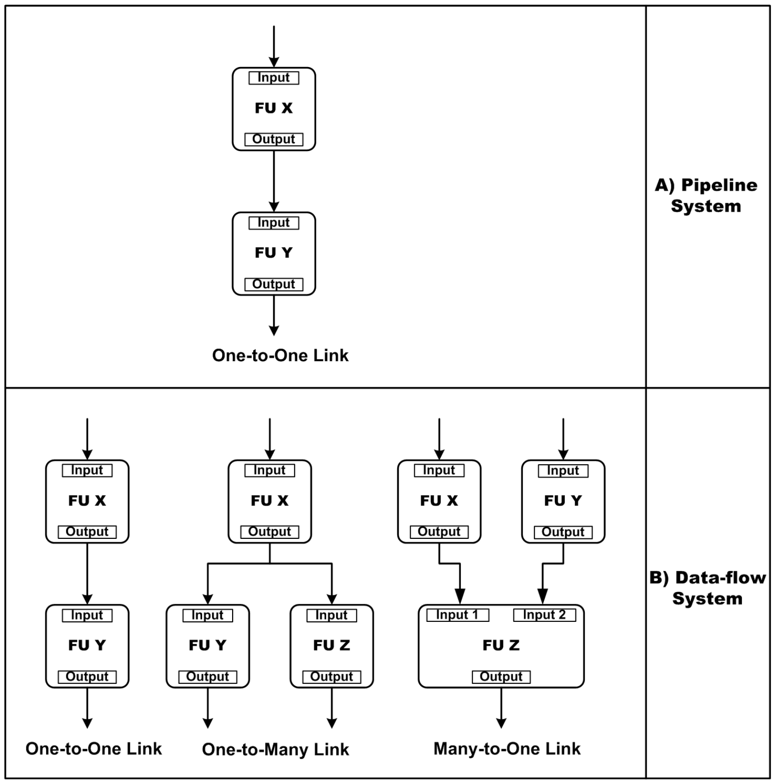


Figure 5.2: Link Types Encountered in Pipe-lined and Data-Flow Systems

output interface will, once again, use one DCCI port (once again, see Figure 5.3 - A). CMFUs with multiple input interfaces will require the use of multiple DCCI ports; this is shown in Figure 5.3 - B. To accommodate this requirement, system slots will have to be created which include multiple interfaces connected to multiple DCCI ports; thus, in data-flow architectures, multiple types of system slot will have to be used.

## 5.2.3 Data and Control Flow

In any MACROS Framework system, data flow will originate in CMFUs acting as input interfaces (connected to to device input pins). This stream data will then be transferred from CMFU to CMFU via links in the DCCI crossbar; finally, this data flow ends at CMFUs which act as output interfaces (and, once again, are connected to output pins). Each CMFU is capable of self-scheduling and will operate correctly if it is provided stream data via an established link (assuming the CMFU in question is used in the current mode).

Control data is inherently based on the mode of the system; as such, the first path to consider

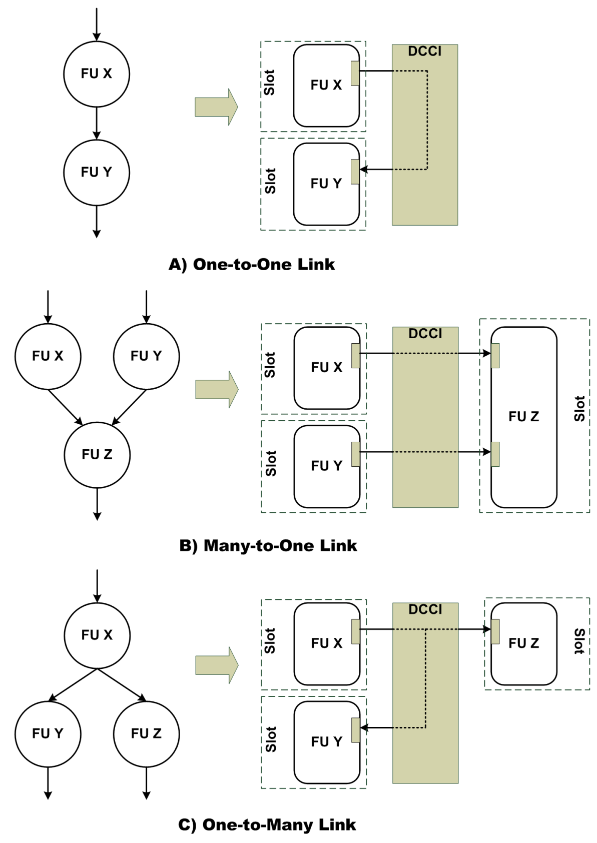


Figure 5.3: Link Support in the MACROS Framework

is that of mode information generation and distribution. Mode information, in the form of a mode ID, originates in the Mode-Mapper, and is transferred to the BCM. The BCM will use this information to perform configuration activities, and will then propagate the mode data to system CMFUs via the DCCI. Once this mode information is received, each system CMFU can determine its connectivity setting for the current mode.

Once the desired connectivity has been determined, each CMFU may make a connectivity request of the DCCI, if a change in connectivity is needed. The DCCI will then implement the requested connectivity change at some point in the future, as defined by the connectivity change algorithm being used (Minimal or Canonical); given that non-preemptive connectivity change methods are used, the connectivity change will usually not occur immediately.

### 5.2.4 System-Level Customization

The main customization aspect at the top architecture level is the size and number of system slots used. Given a set of functional units and mode information, the number and type of slots can be determined, as well as the slot size. For a uniform slot system, the slot size will be dictated by the largest functional unit in the system. If functional units vary significantly in size, multiple tiers of slots can be selected; the number of slot types and their size will be dictated by an analysis of functional unit size distributions. If data-flow systems are being deployed, then slot types will also differ by the number of DCCI interfaces they connect to.

Once the types of slots have been selected, the maximum number of concurrent functional units needed in the system can be derived based on existing mode information; this leads to a minimum number of system slots that the system must contain. Based on the selected slot size, number and type, the worst-case ASP Generation time cost ($Tagen_{i-j}$) can be found; if this worst case time cost is smaller than $T_p - T_w$, then the system will support seamless mode-based ASP generation.

If $Tagen_{i-j} > (T_p - T_w)$ and $(T_p - T_w) > 0$, then the system may support seamless ASP generation via the inclusion of spare system slots. The worst-case difference in functional units between system modes can be found; this will be the number of added system slots the system must incorporate. If fault tolerance is considered, this will also lead to the inclusion of additional spare slots; spare slots should be included for all slot types being used.

## 5.3 Collaborative Macro-Function Unit

The Collaborative Macro-Function Unit (CMFU) is the primary building block of a MACROS Framework system; this is where all of the system functionality resides. As discussed in Chapter 1, the MACROS design method makes use of higher granularity building blocks; as such, the CMFU, as the name implies, is a macro-function processor, constructed by augmenting various IP Cores. The distribution of CMFUs in the system, including their connectivity, dictates the ASP architecture being implemented. This section will describe the behavior and general structure of a CMFU, and will then describe the parametrization steps available when constructing CMFUs for a given application.

### 5.3.1 CMFU Responsibilities and Behavior

The CMFU is semi-autonomous in behavior; this semi-autonomy of behavior allows multiple CMFUs to accomplish more complex functionality than what is contained in any single CMFU (hence the claim that CMFUs are collaborative). The two main responsibilities of any CMFU are A) autonomous operation and B) autonomous connectivity tracking. Autonomous operation refers to a CMFU's ability to begin processing operations without external intervention, so long as stream data is present at its inputs; in essence, *plug-and-play* behavior. Autonomous connectivity tracking refers to a CMFU's ability to make the correct connectivity request of the communication infrastructure, so long as mode information is received. In this way, connectivity information tracking is distributed amongst system CMFUs, and is simplified.

The behavior of a CMFU, once it is loaded into a system slot, is shown graphically in Figure 5.4. A CMFU is disconnected from the system by default, and the first step for the CMFU is to sample mode information and determine what connectivity is required. Once the connectivity is selected, the CMFU makes a connectivity request from the DCCI and waits for the request to be implemented. Once connected to the system, the CMFU can begin local processing (once the correct data is received); local processing will henceforth take the form of work periods, as discussed in Chapter 3. During each work period, the CMFU will generate both data as well as stream control information, which is used by down-stream CMFUs for scheduling purposes. At the end of each work period, the CMFU will re-sample the incoming mode information and, potentially, make a new connectivity request. If the CMFU is not active in a given mode, it will disconnect from the system, and periodically sample data until such time as the mode indicates it should be reconnected.

### 5.3.2 CMFU Structure

A CMFU is created by augmenting an existing functional unit; often such functional units exist in the form of preexisting IP Cores. To this core element, additional circuitry is added to support autonomous behavior. The complete CMFU structure is shown in Figure 5.5, and consists of the following: IP Core, system-specific interface, and Co-Op Unit. The IP Core will not be discussed in further detail here, as it is well covered in existing literature. The Co-Op unit is responsible for

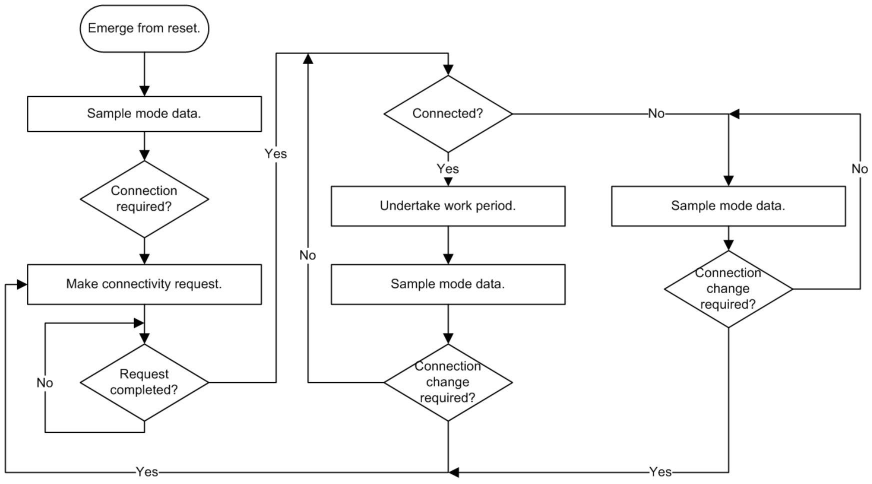


Figure 5.4: CMFU General Behavior

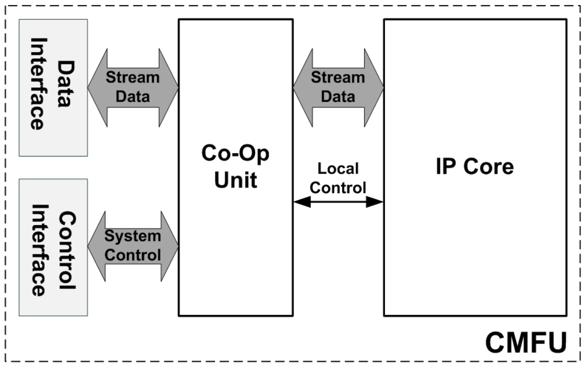


Figure 5.5: CMFU Structure

local scheduling and synchronization (if necessary), as well as tracking mode data and negotiating all connectivity changes with the DCCI. Finally, the system-specific interface combines data and control interfaces for the CMFU.

The CMFU interface consists of two primary components: stream data and system control. The generalized CMFU interface is shown in Figure 5.6. The stream data portion of the interface, itself, may be further separated into data and control portions, depending on the stream structure. Scheduling activities are conducted based on control information received over the stream data interface.

The system control interface is responsible for all on-chip assembly activities; it is used to make connectivity requests and negotiate connectivity activities. It consists of an ID interface, a status port, a mode port, and two hand-shake control signals. The ID interface is used by the CMFU to identify itself through the use of a numerical ID code; it is also used to transmit a second ID

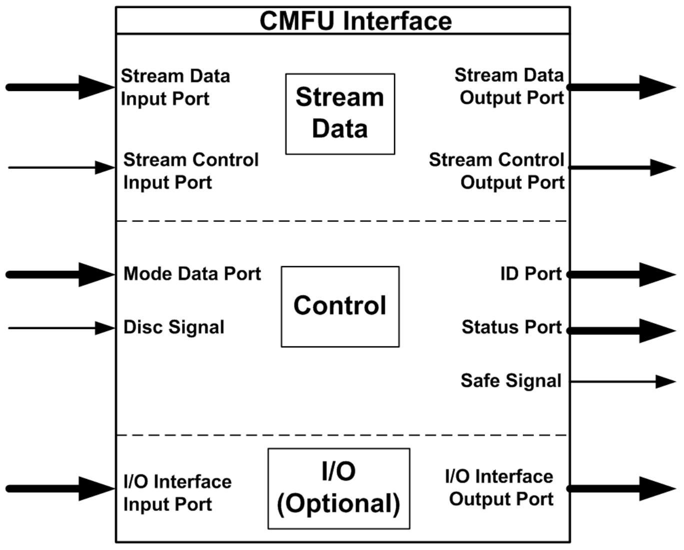


Figure 5.6: CMFU Interface

code, which identifies the up-stream CMFU for the current operating mode. The status port is used to indicate to the CMFU whether a change in connectivity is required. The mode port is used to receive mode data from the BCM. Finally, the *safe* and *disc* hand-shake signals are used for negotiation of connection procedures. If the CMFU makes use of two DCCI ports (in data-flow systems) then a separate data and control interface will be used for each DCCI port used.

CMFU identification is used to allow a MACROS Framework system to assemble processors correctly despite the fact that CMFU locations can change at run time. Each CMFU has a numerical ID which is uses to identify itself to the DCCI, regardless of its location in the system (using the port described above). This ID takes the form of a numerical value; the structure of this numerical value depends on the structure of the potential processor architectures. To accommodate the connection change mechanisms describe in Chapter 4, descendants and ancestors of a node must be found; to facilitate this search process, the complete CMFU ID consists of a node ID as well as multiple branch IDs. This identification scheme is shown in Figure 5.7; the node IDs are assigned through a breadth-first search of the graph. Branches are, likewise, identified in a breadth-first fashion.

Figure 5.6 shows a second, optional System I/O interface; this interface is present only static CMFUs which act as system-wide I/O sources and sinks. Such CMFUs are locked to specific system slots in response to the fact that A) they are expected to be present for all operating modes and B) the system level connectivity is expected to remain static (as dictated by the system board),

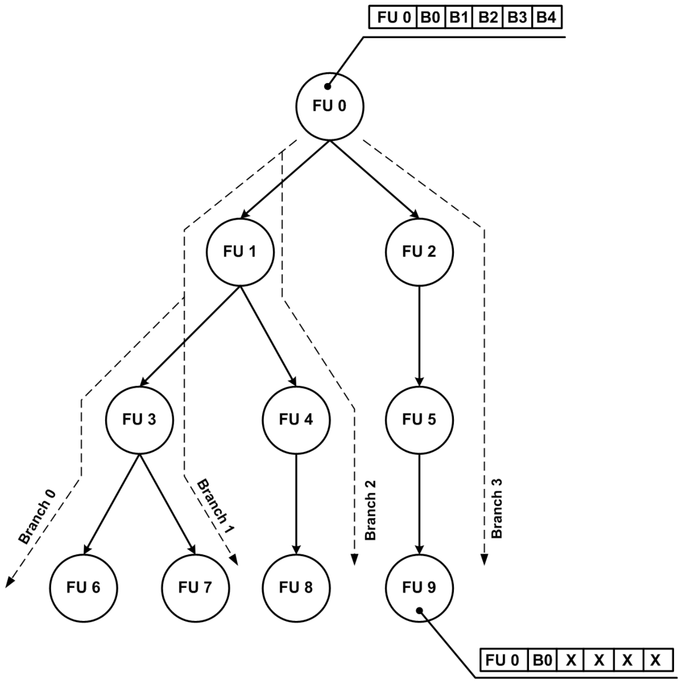


Figure 5.7: CMFU Identification Process

meaning that the I/O pins used will remain equally static. This I/O interface is application specific, and dictated by the external device on the other end of the interface.

### 5.3.3 Co-Op Unit Structure and Behavior

The Co-Op unit is a control-oriented circuit added to an IP Core which performs two functions: local scheduling of processing activities and connectivity control for the local CMFU; the scheduling task may not be necessary, depending on the IP Core implementation. Certain types of IP Cores are built explicitly for data streams, i.e. they are aware of the periodic nature of stream data and automatically trigger their pipeline when the correct stream indicators are detected (for example, the RGB to YCbCr Color Space Converter IP Core offered by Xilinx [167]). Alternatively, IP cores may be built to operate on single data structures, such as a fixed-length array of data elements; in such instances, the IP Core must be activated repeatedly, as stream data is received. Likewise, the IP Core may or may not generate stream control data; if no control data is generated, the Co-Op unit will have to generate output control data for the CMFU.

If data-flow architectures, the Co-Op unit can be used for data synchronization purposes by adding delay stages to an existing data-path to achieve a target latency. However, as discussed in

Chapter 4, all functional units being used must exhibit the same cycle time; if existing IP cores are used, then this requirement will have to be enforced.

The connectivity control task is mandatory, regardless of the IP Core type. It consists of three sub-activities: monitoring of mode data, making connectivity requests, and negotiating connectivity procedures. To accommodate the first two tasks, the Co-Op unit incorporates a local record which defines the CMFU connectivity for every supported mode. For each mode, the ID of the CMFU and the ID of its source are defined. For each sampled mode value, the Co-Op unit will have a connectivity requirement; if the mode was found to have changed, a new connectivity request will be made, reflecting the new connectivity requirement. In situations where multiple DCCI ports are used, the Co-Op unit will generate separate mode-based connectivity information for each port.

Connectivity requests are made the Co-Op unit via the status port introduced previously. From the point of view of connectivity requirements, a CMFU can be in one of 4 states, each of which has an associated status code (a 3-bit binary value); these states and the associated codes are listed below.

- Normal operation; code = 101.
- Request connectivity change; code = 010;
- Request disconnection from system  disconnected from system; code = 110;
- Blank CMFU indicator; code = 111;

When either code 010 or 110 is received by the DCCI, a connectivity change is initiated. This connectivity change is performed following one of the connectivity change procedures described in Chapter 4. These procedures rely on the DCCI knowing when each CMFU finds itself between work procedures; thus, the Co-Op unit drives a *safe* handshake signal, which is asserted between work periods. At the same time, the DCCI may need to force the CMFU to remain between work periods (ignore incoming stream control data); the *disc* signal is used by the DCCI to force this behavior in the CMFU via the Co-Op unit.

The structure of the Co-Op unit is shown in Figure 5.8. The core of the Co-Op unit is a central FSM, used to control all other circuits and generate the required behavior. Secondly, the Co-Op unit contains mode processing circuitry, consisting of a mode look-up table (used to store

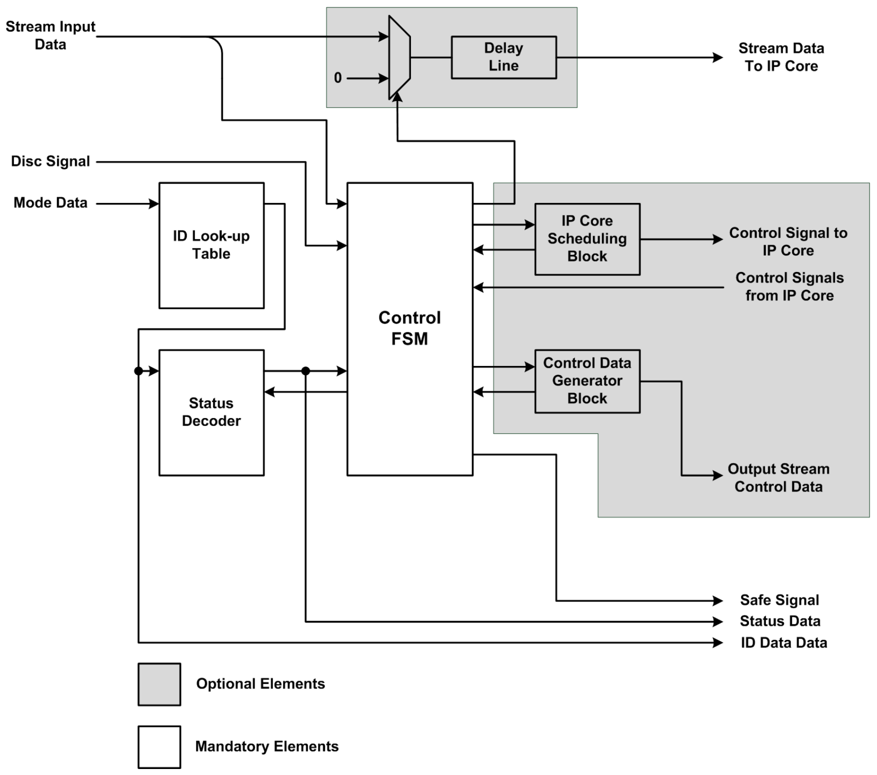


Figure 5.8: Co-Op Unit Structure

the records described above) and a status decoder which determines the status code for the CMFU; the outputs of these two circuits are stored locally in registers. The default stored status, when the CMFU is first loaded into the system or coming out of reset is 110 (i.e. disconnected from the system); in this way, all CMFUs start disconnected. The CMFU will then change this status as it samples incoming mode data, and will be connected to the system by the DCCI.

For the purposes of IP core scheduling and synchronization, the Co-Op unit *may* incorporate a scheduling block containing dedicated counters and comparators used to track time periods and trigger the IP core at the appropriate time, as well as potential feed-back lines from the IP Core. A data-flow multiplexer can also be included to allow the Co-Op unit to isolate the IP Core from incoming data. Finally, delay lines may be used for the purpose of synchronizing stream data with control signals, as well as latency equalization. As well, depending on the IP Core behavior, the Co-Op unit may have to generate output stream control data for down-stream scheduling. To accommodate this behavior, a control data generator block may be included, consisting of comparators, decoders or storage elements; control data can either be stored and reproduced from input data, or generated by decoder. The inclusion of these circuits depends on the nature of the

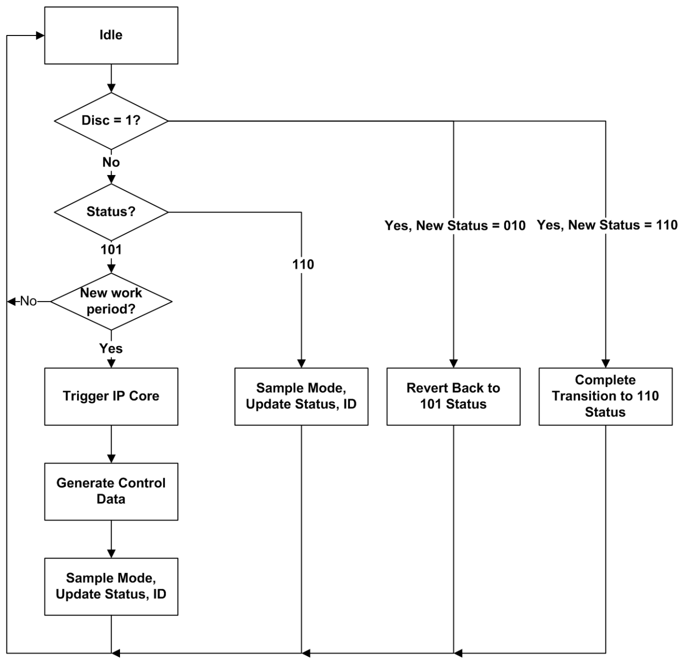


Figure 5.9: Co-Op Unit Behavior

IP Core and its capabilities.

The Co-Op unit provides the majority of the CMFU behavior described above, which is codified in its control FSM. The FSM behavior is shown in Figure 5.9 in the form of a flow-chart. The FSM upon coming out of reset, enters the Idle state, where it spends most of the time. During regular operation, the FSM will react to decoders monitoring incoming stream control data; once the correct control pattern is detected, the FSM will enter a processing loop of triggering the IP core and generating output stream control data. This loop may be undertaken once or multiple times, depending on the behavior of the IP Core (specifically, if the IP Core needs to be activated multiple times per work period); if the IP core does not require scheduling, this loop may be omitted. Finally, once the work period is over, mode data is sampled (and then propagates to the mode look-up table and status decoder) and the CMFU ID and status are updated.

If a change in status occurs and the DCCI responds by asserting the *disc* signal (indicating that a connectivity change is ready to take place), the FSM will transition to one of two state sequences: one will return the CMFU to regular operation (after a connectivity change); the other will complete the transition process to the disconnected state. Finally, if the CMFU is in a disconnected state but still present in the system, it continuously samples incoming mode data to determine if a connectivity change (re-connection to the system) is needed.

### 5.3.4 CMFU Customization

CMFU customization can be divided into three main areas: interface parametrization, mode record storage selection and the IP Core behavior matching. The used control and data interfaces must be matched to the stream data format used and the control characteristics of the system. The stream data interface should have a bit-width large enough to accommodate stream packets in the selected format, as well as any additional explicit control information being transmitted (for example flags); this will apply equally if multiple DCCI ports are used.

The control interface consists of the 3-bit status port, the two hand-shake signals, the mode port and the ID port(s). Each of these ports must be large enough to accommodate the data type transmitted through them. If multiple DCCI ports are used, each will have its own separate control interface. In pipe-lined systems, only the local ID needs to be transmitted; the up-stream ID for connection purposes can be derived by subtracting 1 from the local ID. In a data-flow system two ID values must be looked up and sent to the DCCI. The most straight-forward way to accomplish this is using two separate ports. However, if the port bit-width is found to be too large, a single port can be used, and a time multiplexing scheme can be used for both IDs.

Each CMFU must store ID information locally, describing the connectivity of all input interfaces; this information is codified into a look-up table, where each mode ID can be mapped to a CMFU ID (or multiple IDs for data-flow architectures); if the IP core is absent for a given mode, then a special “not in system” code should be used, which is interpreted by the status decoder (see the previous Subsection).

The ID look-up tables can be implemented using either a simple decoder or a Read-Only Memory (ROM). If a memory element is used, then the inclusion of this type of resource should be noted when selecting the slot size. Alternatively, if a decoder implementation is chosen, the added logic resources used by this circuit must be taken into account when allocating slot size. The use of a decoder may be more efficient, as a number of modes may map to the not-in-system code; this, of course, depends on the CMFU mode distribution.

The final parametrization task consists of determining if scheduling and stream control data generation circuitry must be included into the co-op unit. If the local IP core was designed for stream operations, then no scheduling circuits need be included. The only element needed will be a data-flow multiplexer which is used to isolate the IP Core from data lines during connectivity changes. On the other hand, counters and comparators will be needed if the IP Core must be triggered; as well, a delay line may be included to ensure that control signals from the Co-Op unit to the IP Core control unit are aligned properly, or when latency must be equalized.

Given that distributed scheduling is being used, stream control data must be transferred between CMFUs. If the local IP Core can generate stream control data (this is very likely to be true for IP Cores built to operate on stream data) then no dedicated circuitry is needed for output stream control data generation. Otherwise, either decoders or storage elements need to be included, as described above. If scheduling circuits are included as well, included counters can be shared for both tasks; otherwise dedicated counters must also be added to the Co-Op unit.

## 5.4 Distributed Communication and Control Infrastructure

The Distributed Communication and Control Infrastructure (DCCI) is the second primary building block in the MACROS Framework; it facilitates the implementation of full system functionality by providing communication links between the partial functionality of CMFU. Given the dynamic nature of such systems, the DCCI also implements the on-chip assembly process via link establishment (through connectivity change procedures). To perform this task, the DCCI incorporates a distributed control network which negotiates the inclusion of connection procedures into the processor workload.

### 5.4.1 DCCI Responsibilities and Behavior

The DCCI has three primary responsibilities in a MACROS Framework systems, the most fundamental of which is to provide communication links via an integrated, fully connected crossbar. Links are expected to change as modes change, and different CMFU combinations are used to compose new processor architectures. Thus, the second responsibility of the DCCI is to implement these changes in processor architecture by responding to connectivity change requests from CMFUs. Finally, the DCCI is the main transmission medium for mode information from the BCM to system CMFUs.

During regular operation, the DCCI control system sits idle and monitors incoming control information from system CMFUs; the crossbar is actively transferring data between CMFUs via the programmed system links. As mode information is transmitted to CMFUs, requests for connectivity changes may occur; these changes can come either from pre-existing CMFUs, or from newly configured CMFUs which make new connectivity requests. At this time, portions of the DCCI control system begin active operations in an attempt to implement one of the two connectivity change procedures described in Chapter 4. Since the control system is distributed both in behavior and in structure, not all elements of this system will be involved in all connectivity change operations.

### 5.4.2 DCCI Structure

The DCCI internal structure is shown in Figure 5.10, and consists of the system crossbar, a collection of LCCU, a Control Data broadcast network and a Mode Data broadcast bus. The system crossbar provide the communication links used by the system. It includes data registration in input and output ports, and a dedicated control interface associated with each port, where control data can be loaded to control connectivity for that port. A four-port MACROS Framework crossbar example is shown in Figure 5.11. As shown, the crossbar connectivity is provided by multiplexers on the output ports; this architecture reflects the underlying architecture of the targeted FPGAs, which incorporate multiplexers as part of their logic block structure [32], and can implement this architecture efficiently. If pipe-lined system are implemented, the number of port-pairs in the crossbar will be equal to the number of system slots, thus allowing all slots to communicate in a uniform fashion. If data-flow system are implemented, the crossbar must contain one port-pair for every CMFU input interface in the system; one of these port-pairs will be used as a duplex link (input and output), while the others will be used as input-only links.

The collection of LCCUs forms the distributed control system used to establish connections in a MACROS Framework systems. Each LCCU is assigned to a crossbar port and the associated system slot or slots; its main responsibility is to program the local crossbar in response to the connectivity requests received from the CMFU. This programming comprises a connectivity change, and is done in accordance with the connectivity change procedures described in Chapter 4; thus, LCCUs can be either of Canonical type or of Minimal type. The internal structure and behavior of these two types of LCCUs will be described in detail in the following section.

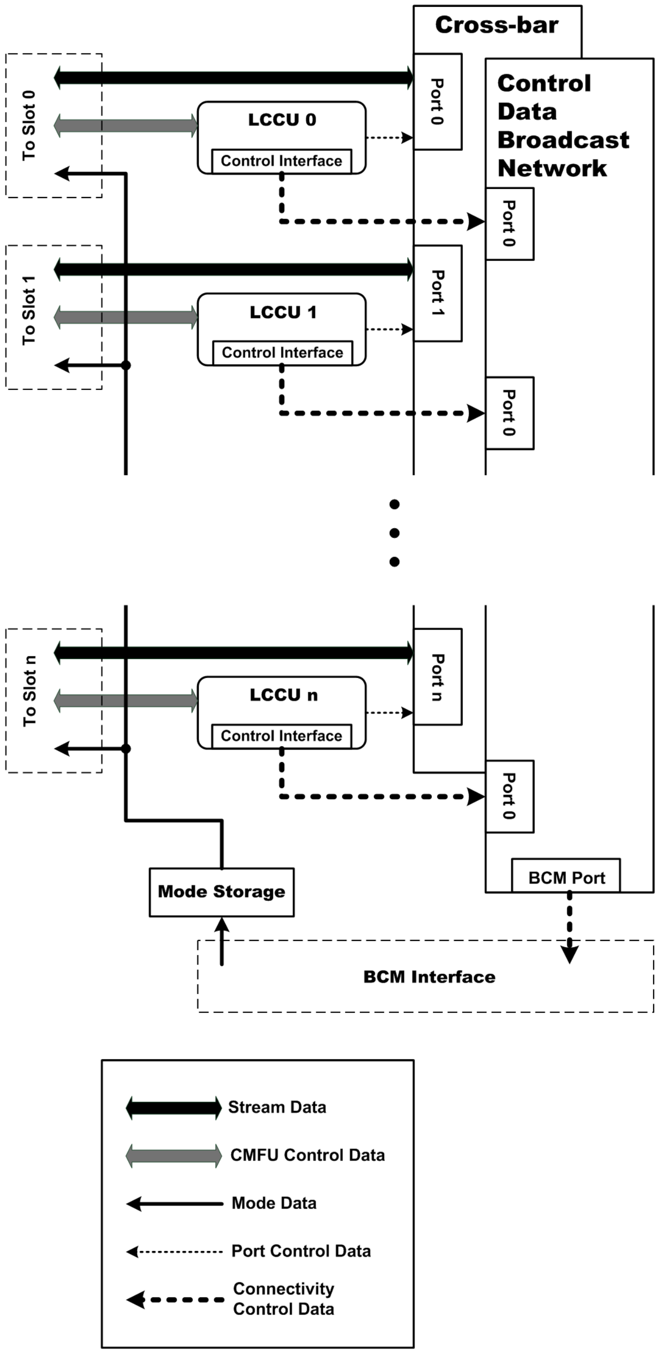


Figure 5.10: DCCI Structure

Both the Canonical and Minimal connectivity change procedures rely on information about connections in the system, as well as the distribution of CMFUs inside the system. The latter item is of particular importance, as CMFU location can change as the system operates. Because of this, all LCCUs are connected to one another using a Control Data Broadcast Network; this network manages the distribution of non-local control information in the system. The network can

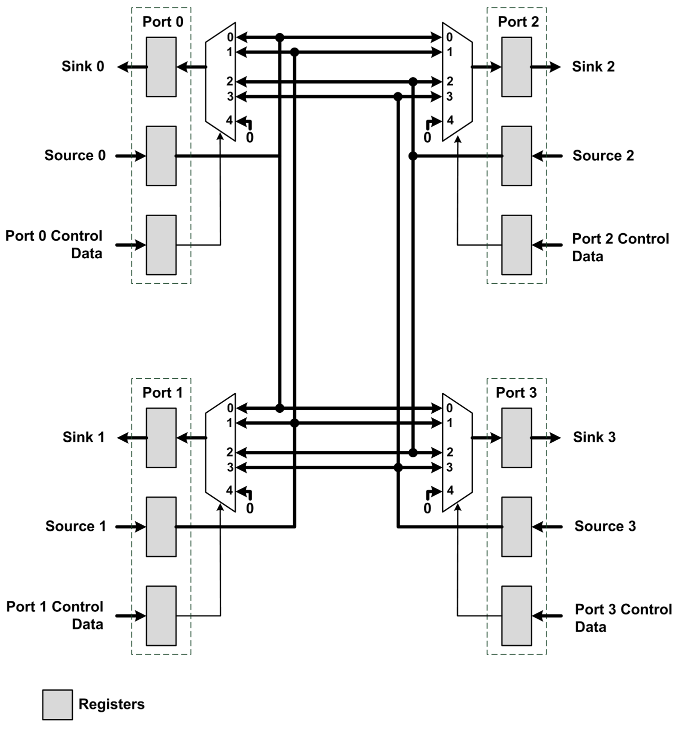


Figure 5.11: 4-Port Crossbar Example

have various structures, and the type of control information it carries will change depending on the connectivity change procedure used; however, in all cases, this network will carry the IDs of CMFUs currently in the system. The Control Data Broadcast Network also includes an interface to the BCM, allowing it to monitor the connectivity status of CMFUs.

Finally, the DCCI contains the Mode Broadcast Bus, which is used to transmit mode information to system CMFUs. This bus has only one source, the BCM, and is connected to all CMFUs via their control interface. As shown in Figure 5.10, mode information is kept entirely separate from all other DCCI elements; it is used only by CMFUs to make individual connectivity changes. In this way, the complexity of the Local Connectivity Control Units and the Control Data Broadcast Network can be minimized.

In Figure 5.10 above, the DCCI is shown as a collection of separate elements. This feature of the DCCI design allows it to be implemented as separate modules. In this way, during system implementation, these DCCI modules can, themselves, be assigned to separate PRRs; in this way, scrubbing operations can be applied to these elements if necessary.

### 5.4.3 Local Connection Control Unit

The Local Connection Control Unit (LCCU) is the primary agent responsible for the implementation of connectivity changes in MACROS Framework systems. These connectivity changes can follow one of two procedures introduced in the previous chapter; because of this, two architectures are presented for the LCCU. The two architectures differ in a number of respects; however, they also share certain similarities.

Both presented architectures use the same control interface to system CMFUs, and access crossbar control interfaces in the same way. Secondly, both architectures are modular in nature; each architecture consists of a central control FSM, a number of local storage registers (used to store local parameters regarding the local CMFU) and an interface to the Control Data Broadcast Network.

#### Canonical Local Connection Control Unit Structure and Behavior

As established in Chapter 4, the Canonical Connectivity Change procedure is the more complex procedure from a behavioral point of view; this complexity is imparted to the architecture of the Canonical LCCU. This complexity begins with an analysis of the information elements needed to implement this connectivity change procedure. Each LCCU must be aware of the distribution of CMFUs in the system, which means that CMFU IDs must be stored and distributed throughout the system. Secondly, flags are needed for each system slot, indicating if a CMFU is present, and if it is connected to the system. Finally, a flag is needed for each LCCU to indicate a request for all down-stream CMFUs in a given branch to be disconnected. This leads to the control information listed below:

- CMFU ID: the ID of the local CMFU.
- POP flag: flag indicating if the local slot is populated.
- CON flag: flag indicating if the local slot is populated.
- DR flag: flag indicating if a disconnection request was made.

The structure of the Canonical LCCU is shown in Figure 5.12; its core, as stated, is a control FSM. The LCCU also contains local storage registers for the control information described above, as

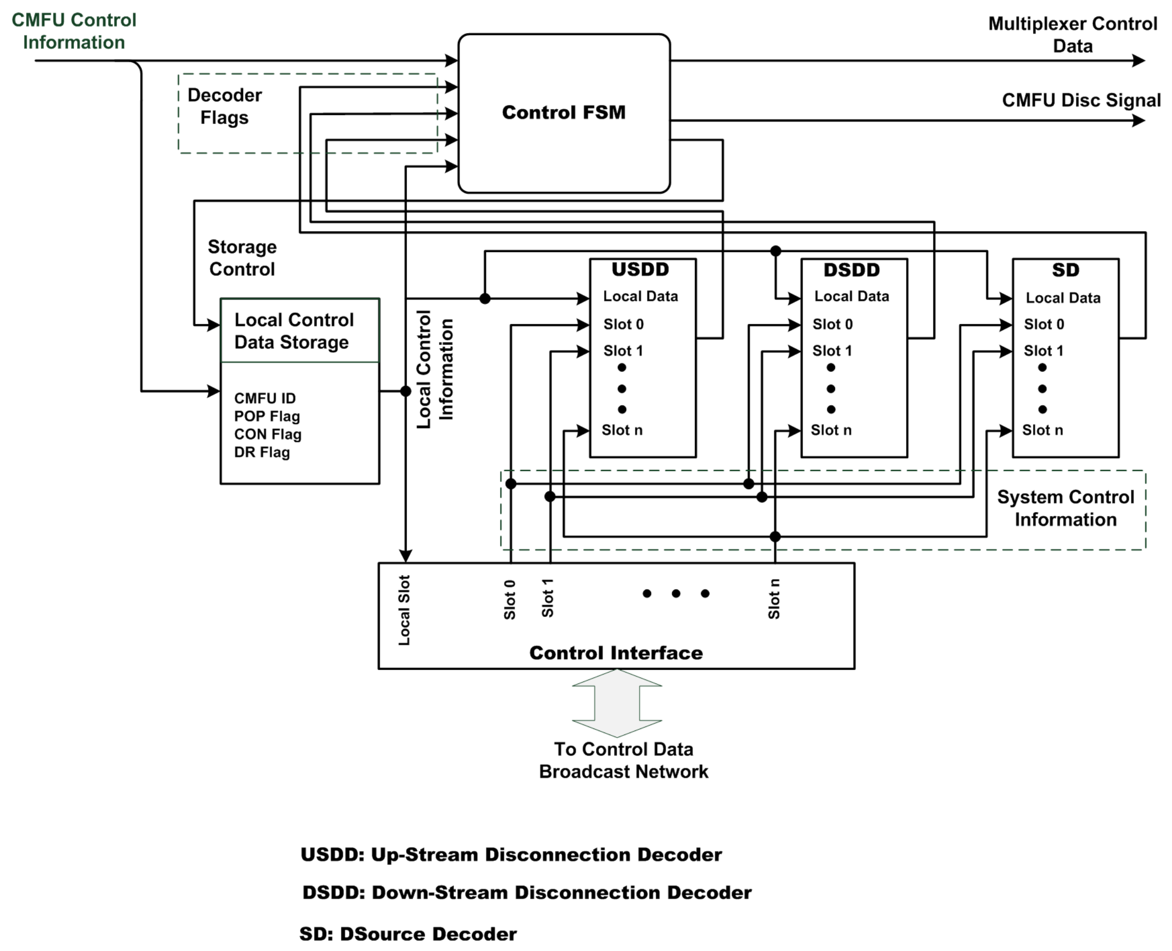


Figure 5.12: Canonical LCCU Structure

well as a register storing the CMFU status. This control information is received through a dedicated control interface connected to the Control Data Broadcast Network; the structure of this interface, as well as the network itself are discussed in the next Section. Finally, the Canonical LCCU contains three decoders: the up-stream disconnection decoder, the down-stream disconnection decoder and the source decoder.

The up-stream disconnection decoder (USDD in Figure 5.12) is responsible for determining if *any* LCCU associated with an up-stream CMFU has made a disconnection request (this implies that the local CMFU will need to be disconnected temporarily). It determines if any DR flags are asserted, and determines if the associated CMFUs are up-stream by comparing ID values. The down-stream disconnection decoder (DSDD in Figure 5.12) is responsible for determining if *all* down-stream CMFUs have been disconnected. The decoder identifies all down-stream CMFUs through ID comparison, and determines if their associated CON flag is asserted or not. Finally, the source decoder (SD in Figure 5.12) is responsible for determining the correct crossbar setting for a given CMFU source ID, based on the IDs received from other LCCUs in the system; it also determines if the source CMFU is connected.

The control FSM implements the actual connectivity change procedures according to the Canonical connectivity change procedure, using the information generated by the three decoders described above. A flow-chart describing the behavior of the control FSM is shown in Figure 5.13. The FSM emerges from reset into its idle/monitoring state; in this state, the FSM waits for either changes in the local status of the CMFU or up stream disconnection requests (as indicated by the appropriate decoder). If the local CMFU is replaced with a blank bit-stream, or vice-versa (as indicated by transitions in status between values 111 and 110), the FSM simply updates the local POP flag. If the local CMFU makes a connectivity change request or if an up-stream disconnection request was received, the FSM will inter its main control sequence. This control sequence consists of: 1) asserting the local DR flag, thus indicating to all down-stream components to disconnect temporarily; 2) waiting for confirmation that all down-stream CMFUs are disconnected, 3) wait for the source CMFU to re-connect; 4) load the new connectivity control value for the local crossbar port.

A final aspect of the behavior of the Canonical LCCU which must be considered is the potential for dead-lock in the system. As described above, the LCCU FSM will actively wait for certain conditions to be met, as reported by transmitted system information. If system information (CMFU IDs and connectivity flags) fail to be transmitted, or are transmitted incorrectly, then the LCCU will stall. Thus, dead-lock can occur in situations where faults induce incorrect system behavior; for a further discussion of how this can be avoided, see Chapter 6. Secondly, incorrect behavior and dead-lock can be induced if CMFUs output incorrect connectivity information; therefor, individual CMFUs must be verified prior to deployment to ensure that their Co-Op Units contain correctly populated look-up tables for system connectivity.

### Minimal Local Connection Control Unit Structure and Behavior

The Minimal connectivity change procedure is less complex than the Canonical one, and, once again, this reduction in complexity is reflected in the structure of the associated LCCU, as well as the needed control data. For the Minimal process, the following control information is needed:

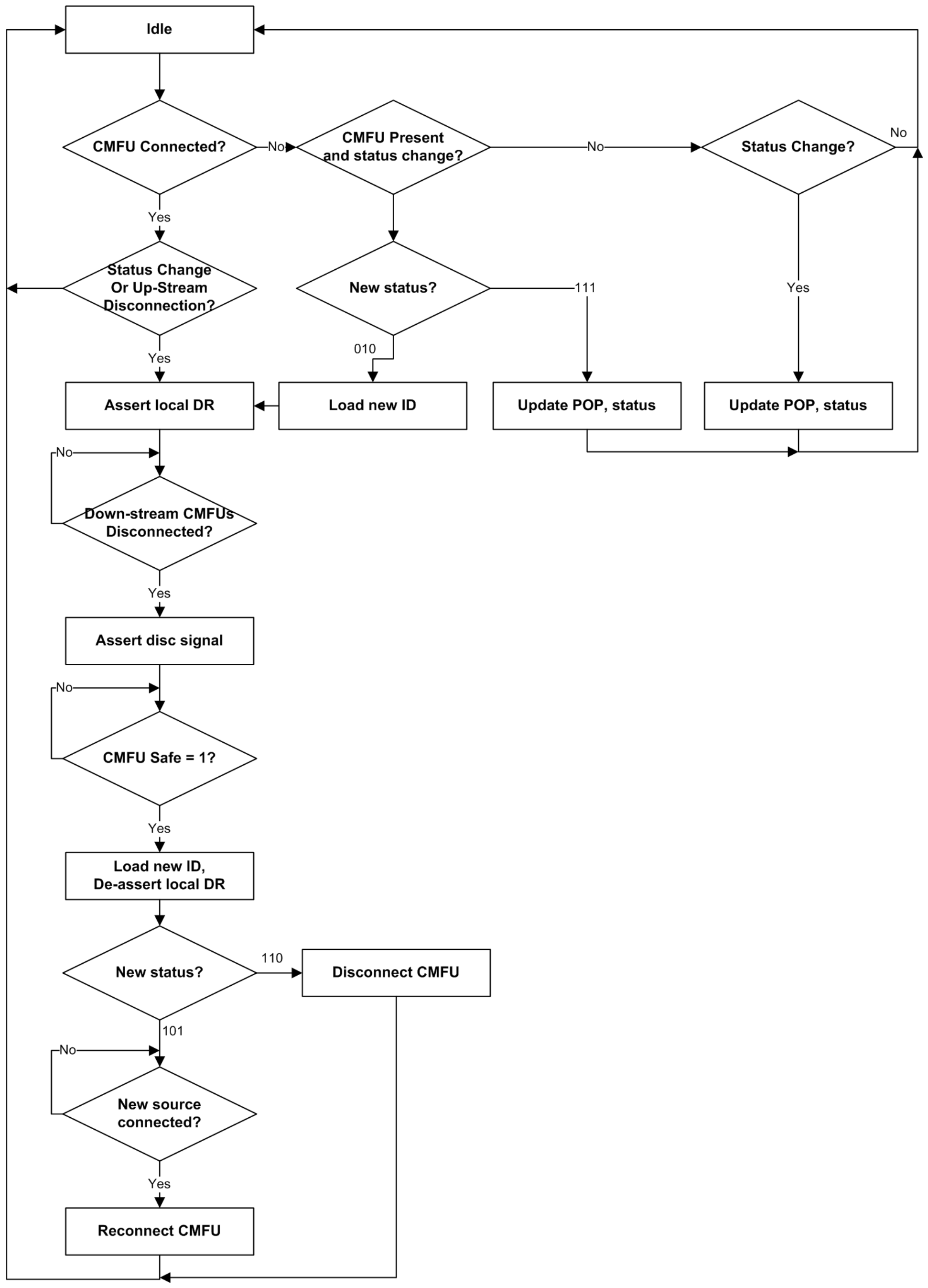


Figure 5.13: Canonical LCCU Behavior

- CMFU ID: the ID of the local CMFU.
- IGNORE flag : flag used to indicate that the local CMFU is not connected in the system.
- DR flag: flag indicating if a disconnection request was made.

The structure of the Minimal LCCU is shown in Figure 5.14; as usual, the core of the LCCU consists of a control FSM. In addition, the LCCU contains storage registers for local control information. Unlike the Canonical LCCU, the Minimal LCCU does not require specialized local decoders; rather, two global flags are computed, one indicating if *any* LCCUs have made disconnection requests, the other indicating if all LCCUs have made disconnection requests; in either case, the CMFUs that have made requests are assumed to be between work periods (in keeping with the CMFU behavior described above). The LCCU also include crossbar control generator, which generates a control value for the local crossbar multiplexer based on the distribution of IDs in the system. Finally, the LCCU contains an interface to the Control Data Broadcast Network through which it receives control information from other system elements.

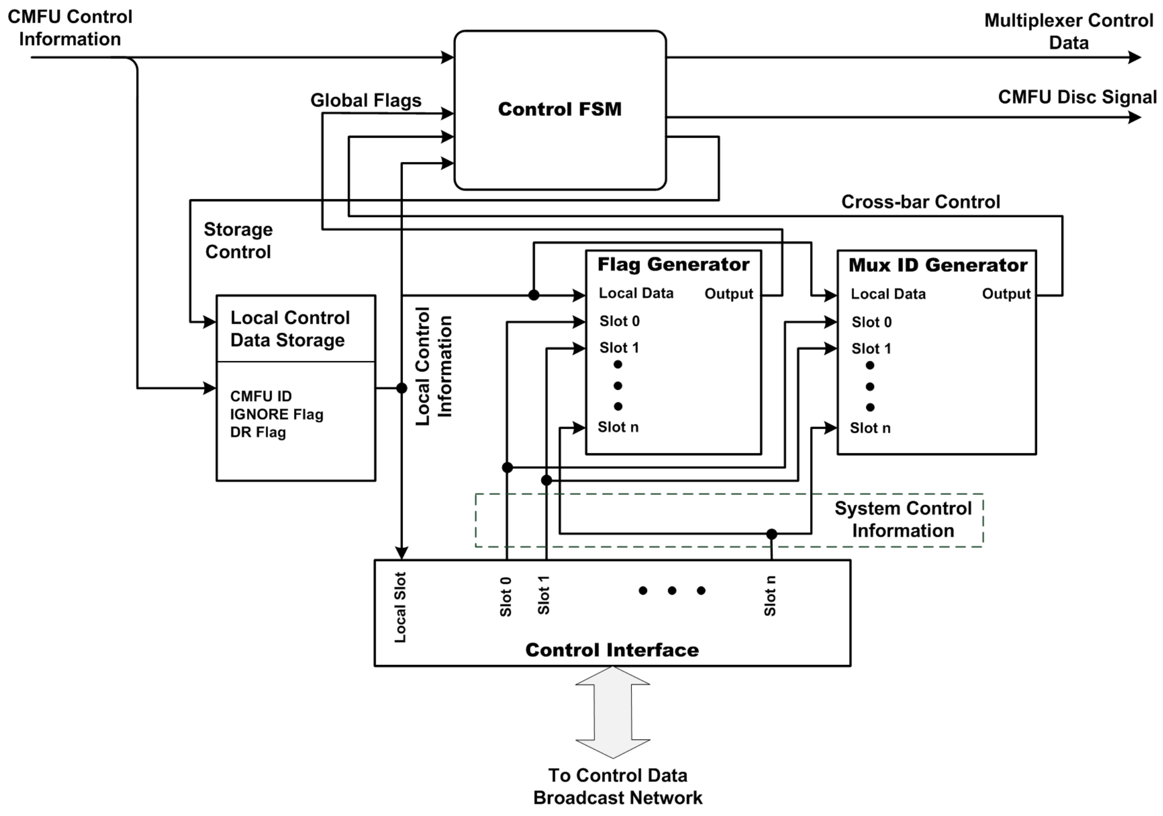


Figure 5.14: Minimal LCCU Structure

The state transition diagram for the Minimal LCCU control FSM is shown in Figure 5.15. As before, the FSM emerges from reset into an idle state, where it monitors local connectivity changes as well as disconnection requests coming from other LCCUs. If a local connectivity change occurs, or if a disconnection request is detected from *any* other LCCU in the system (via the flag introduced above), then the control FSM enters its main control process. In this main process, the FSM does the following: 1) wait for the local CMFU to finish the current work period (i.e. assert its safe signal); 2) assert the local DR flag (thus indicating that the local CMFU is between work periods); 3) wait for all DR flags to become asserted (indicated, once again, via the second flag introduced above); 4) make the local connectivity change requested, and update the local control information.

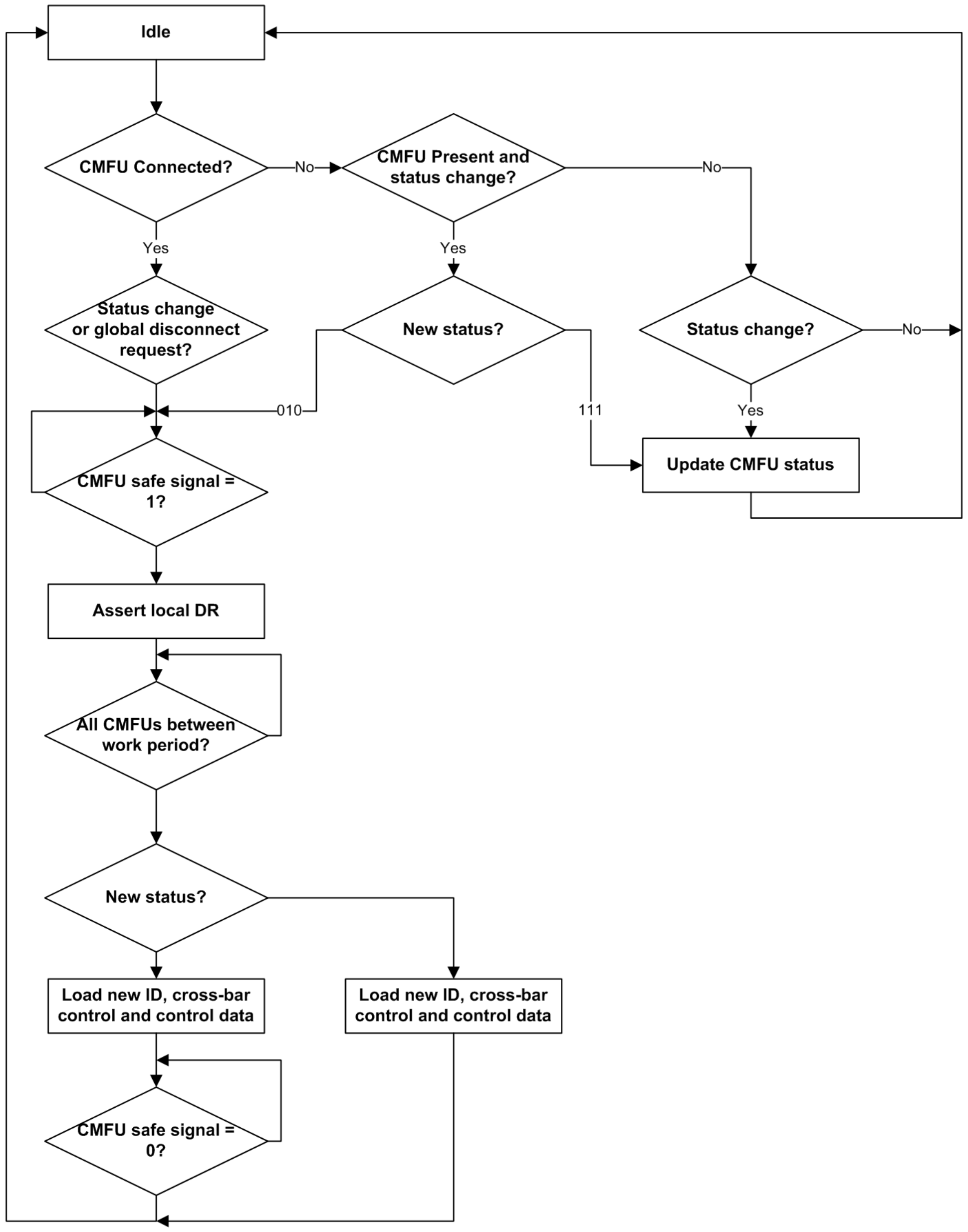

Idle
CMFU Connected?
No
CMFU Present and status change?
No
Yes
Yes
Status change or global disconnect request?
New status?
Status change?
No
010
111
Yes
CMFU safe signal = 1?
Update CMFU status
Assert local DR
All CMFUs between work period?
New status?
Load new ID, cross-bar control and control data
Load new ID, cross-bar control and control data
CMFU safe signal = 0?

Figure 5.15: Minimal LCCU Behavior

As with the Canonical LCCU, Minimal LCCUs can enter into dead-lock conditions if incorrect connectivity information is transmitted through the system, either due to faults or incorrect data connectivity data in Co-Op units. However, in addition to these problems, the Minimal Connectivity Change process relies on the implicit assumption that *all* functional units find themselves between work periods *at the same time* (for more on this, see section 4.4.3). This condition must be verified externally, as the Minimal LCCUs themselves do not have the ability to do this verification; furthermore, the Minimal LCCU behavior does not, currently, make provisions for such situations.

### 5.4.4 Control Data Broadcast Network

The Control Data Broadcast Network is responsible for transmitting all requisite control information between system LCCUs; this network, and the associated interfaces embedded in system LCCUs can have multiple architectures. These variations in architecture will lead to different temporal costs for data transmission, and can thus dictate the temporal cost of the two connectivity change procedures.

The simplest architecture for the broadcast network is the use of multiple point-to-point links, as shown in Figure 5.16. This approach implies the use of one duplex link per LCCU pair, meaning that each LCCU will have $N_{slot} - 1$ such links as part of their control interface. Furthermore, data through these links can be transmitted in parallel, in a bit-serial fashion, or a combination of these two extremes (such as serialization to 4 or 8-bit values). If fully parallel transition of data is used, then all control information will require one clock cycle to be transmitted; this offers the fastest transfer method, and requires no additional control circuitry, but leads to the use of more wires in the system, and potential routing congestion. Alternatively, bit-serial transmission can be used; this leads to a much slower transmission process, and requires transmission control circuits (albeit small ones) for each link. In essence, when using individual links, the primary trade-off is between the number of wires used versus the logic overhead in individual LCCUs.

A second approach to the broadcast network structure uses the concept of aggregation elements. The structure of such a broadcast network is shown in Figure 5.17. In essence, this network structure reduces the number of links connected to each LCCU to one duplex link; this approach may be desirable if the LCCU is implemented as a PRM, and the number of wires crossing PRR boundaries needs to be minimized [35]. In this approach, aggregation elements are used to collect control data from LCCUs and transmit it. Figure 5.17 - A shows an example of this approach where one aggregation element is assigned to each LCCU, while Figure 5.17 - B shows this approach using

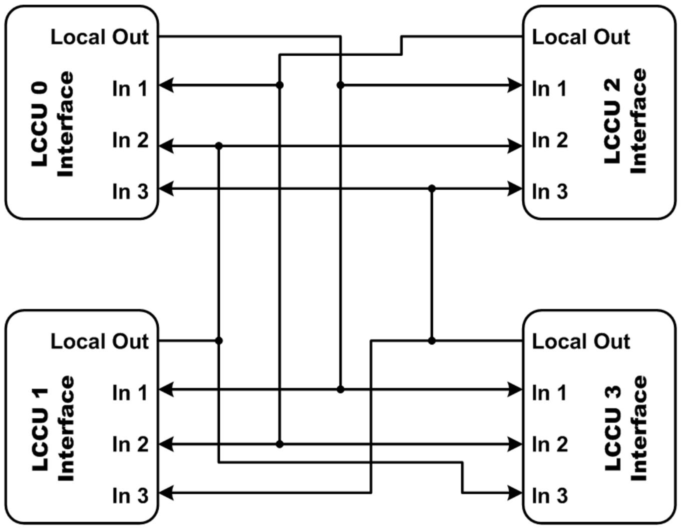


Figure 5.16: Point-to-point Link Broadcast Network

a single aggregation element. Using multiple aggregation elements improves the fault resilience of the system through redundancy, while the use of a single element reduces the resource overhead.

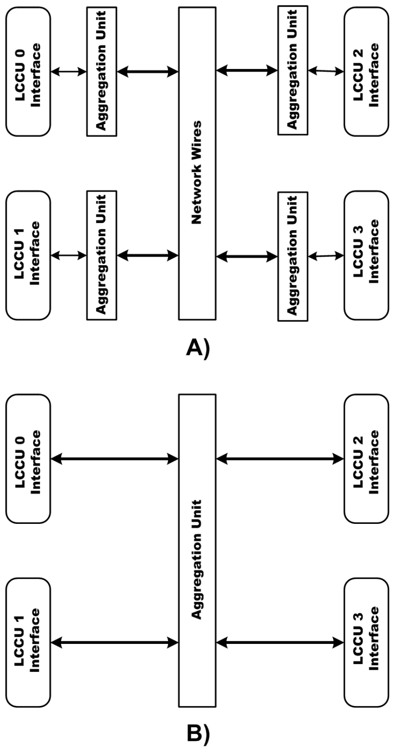


Figure 5.17: Aggregation Element-Based Broadcast Network: A) Multiple Aggregation Elements; B) Single Aggregation Element

When the minimal connectivity change process is used in conjunction with aggregation elements, the global flags used by system LCCUs can be implemented either locally in the LCCU, or in the aggregation elements. Using this approach, these flags are computed faster, as the final transmission stage to the LCCU is omitted. This approach can be take due to the reduced logic cost needed for flag computation; the same would not be possible if the canonical connectivity change process were used.

### 5.4.5 DCCI Customization

The DCCI customization is primarily accomplished in five ways:

1. Selection of a crossbar architecture parameters.
2. Delay to latency conversion in the crossbar.
3. Selection of the connectivity change process used.
4. Decoder implementation for Canonical LCCUs.
5. Control Data Broadcast Network architecture selection.
6. On-chip layout of DCCI elements.

The crossbar used must have port pairs equal to the number of system slots; in situations where slot types require the use of multiple DCCI ports, the crossbar will have to be increased accordingly; the port bit-width must be such that it accommodates stream data and control information. If large devices or high clock rates are used, long lines in the crossbar must be segmented using registers. This process can be done automatically using the CAD tool-chain, or manually; the manual approach will be more time consuming, but can offer the highest performance in a given situation by minimizing wire delays between registers.

Since the minimal connectivity change procedure involves reduced resource cost and procedure complexity it is the preferred connectivity change procedure. However, it can only be used under the following two conditions:

- $(T_p - T_w) > 0$.

- Individual work periods are aligned such that all functional units find themselves between work periods *concurrently* for a non-zero period of time.

If these two conditions are met, the Minimal LCCU architecture should be used. Alternatively, the Canonical LCCU architecture will have to be used; if the $(T_p - T_w) > 0$ condition cannot be met, seamless mode changes will not be feasible.

If the Canonical LCCU architecture is used, the structure of the three decoders used by each LCCU must also be selected. These decoders can be implemented in one of two ways: fully parallel architectures which can generate results in a single clock cycle but require large amounts of resources, or sequential implementation which require multiple clock cycles but use reduced logic. The decision as to which decoder implementation to use should be dictated by the size of the system (the size of the parallel decoders will increase in proportion to the number of LCCUs sending control information), the desired logic resource overhead and the required performance in the connectivity change process.

The architecture of the Control Data Broadcast Network must also be selected. The primary considerations in selecting this architecture are the required connectivity change performance (as described above, various implementations will lead to different time costs for control data distribution), the resource cost associated with this network, and the fault resilience capabilities desired. Architectures based on aggregation elements should be considered if the number of wires leaving the LCCU must be minimized (in instances where the LCCUs are implemented as PRM and the system size is large). If this limitation is not imposed, architectures using individual links can offer better performance and resilience, as all control data distribution tasks are distributed in the system.

The final customization aspect of the DCCI is optional in nature, and consists of physical placement of various elements on-chip. This task requires the system designer to create constraints for some or all DCCI components prior to implementation of the system infrastructure. Such placement may help or hinder the implementation process, in terms of implementation effort as well as timing closure [168]. This customization aspect will be heavily informed by the nature of the application as well as external constraints such as I/O location limitations.

## 5.5 Mode-Mapper

All MACROS Framework elements introduced thus far assume that mode information is provided in the form of a mode ID. The Mode-Mapper is the system module responsible for generating this mode ID for the rest of the system. Depending on the application characteristics, the system mode can be derived based on one or more of the following pieces of information: A) information embedded in or derived from the data stream, B) data obtained from various system sensors or C) information provided by other system functional units (in cases where complex analysis tasks are needed to determine system mode). Using these source of information, the Mode-Mapper essentially acts as a look-up table, translating disparate pieces of information into a single mode ID for the system.

The general Mode-Mapper architecture is shown in Figure 5.18, and consists of stream data, sensor and /or functional unit inputs, a specialized decoder block, a look-up table and a control FSM. The decoder block is responsible for analyzing all incoming information, be it stream data, sensor inputs or data from functional units, and generating an address for the look-up table; the address will point to a mode ID which corresponds to the detected conditions. Both the structure of the decoder block, as well as the entries in the look-up table are based on the characteristics of the application being implemented. The control FSM is responsible for sending control signals to the decoder (if necessary), and for storing the mode ID value, once generated.

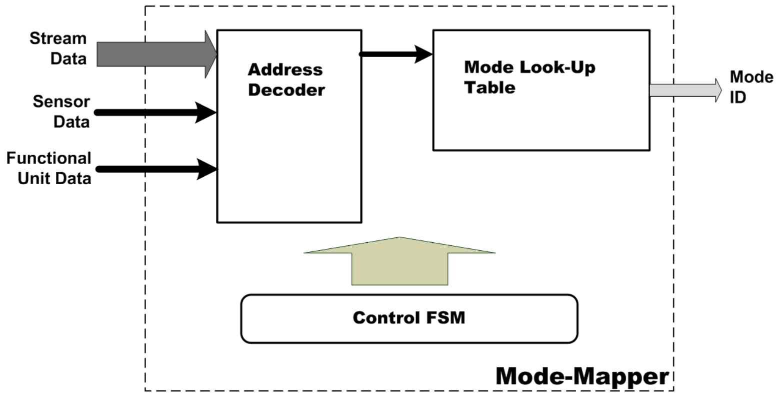


Figure 5.18: Mode-Mapper General Architecture

To better illustrate the structure of the Mode-Mapper, two examples are provided here. The first, shown in Figure 5.19 - A, is the Mode-Mapper of a system which changes mode based on sensor data; the system always stores the previous sensor values into registers, and reacts to any changes in the sensor inputs. The decoder block is, in this case, a pure combinational circuit; the look-up table simply stores a mode ID for every potential combination of sensor inputs. A more complex example of a Mode-Mapper is shown in Figure 5.19 - B; here stream data is organized into packets, and mode information is derived from packet headers. The Mode-Mapper is activated whenever a new packet is received, and the decoder block consists of a cascade of registers, used to store a complete packet header, as well as comparators. In essence, the decoding operation for this Mode-Mapper is spread across multiple operating clock cycles.

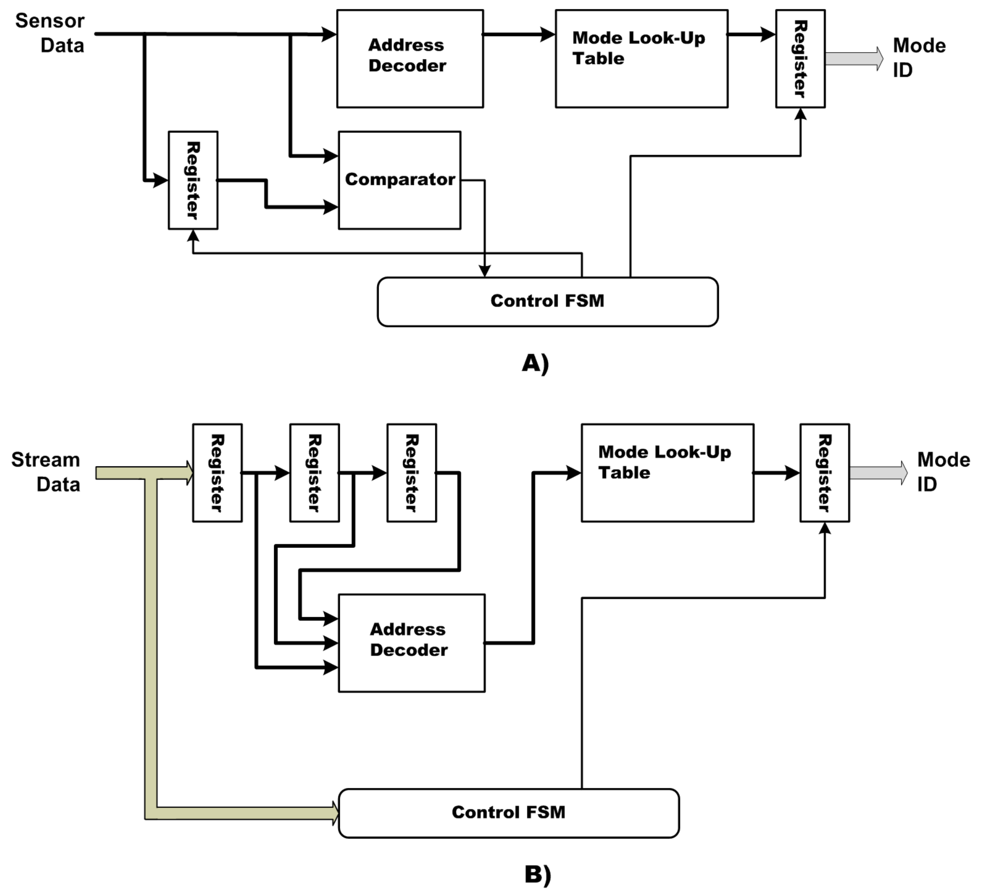


Figure 5.19: Example Mode-Mapper Architectures

The Mode-Mapper component can be implemented as either an on-chip element, or an off-chip component. If implemented on-chip, the Mode Mapper will be a system element, similar to the BCM and DCCI, meaning that it is always required for correct system operation. However, as with elements of the DCCI, the Mode-Mapper can be implemented as a PRM, such that fault mitigation operations can be undertaken (this possibility is elaborated further in Chapter 6). If the system size of the main processing FPGA is limited (thus leading in a reduction in the system cost), the Mode-Mapper can be implemented as an external circuit; in this case, a smaller (and cheaper) implementation medium can be used.

## 5.6 Bit-stream and Configuration Manager

The Bit-stream and Configuration Manager (BCM) is the final element of a MACROS Framework system; such an element, in one form or another, is present in any system which relies on run-time reconfiguration, be it full or partial. The main roles of the BCM are bit-stream storage and processor assembly operations via full and partial system configuration. The remainder of this section will describe the BCM structure and behavior, and will discuss the parametrization steps associated with this component. The presented architecture is device-independent, and is not based on or aimed at any one FPGA architecture; some elements of the architecture must be tailored to the FPGA architecture being used (specifically the configuration interface supported), and these elements will be identified as such.

### 5.6.1 BCM Responsibilities and Behavior

As stated above, the BCM is responsible for storing all bit-streams associated with the system, as well as performing all configuration tasks. As part of the configuration task, the BCM must also track the distribution of CMFUs in the system, allocate new CMFUs to system slots, and perform ancillary "house keeping operation". The latter item can consist of loading blank bit-streams into the unused system slots, or specific fault mitigation steps (this will be discussed further in Chapter 6).

The general behavior of the BCM consists primarily of waiting for mode changes and performing configuration tasks in response to said changes. If the system finds itself in the start-up phase (either due to emerging from reset or due to receiving power), the BCM will first perform and initial "default" configuration using a full bit-stream; this first bit-stream will consist of a default configuration aimed at a default mode. Once this operation is complete, the BCM monitors incoming mode data. once a mode change is detected, the BCM will begin run-time mode-change procedures. The first step in these procedures is to determine what the system architecture is for the current mode (i.e. which CMFUs are needed for this mode). The second step is to determine which CMFUs are present in the system, and which must be loaded. Each missing CMFU must then be allocated to a slot and configured into the system. At this time, the mode information is allowed to propagate to system CMFUs via the Mode Broadcast Bus. If spare system slots are not available, CMFUs are stalled prior to undertaking these run-time mode change procedures.

### 5.6.2 BCM Structure

The general behavior described above translates into significantly more complex behavior when actual circuit implementations are considered. Because of this, the BCM is modular in its internal architecture, as shown in Figure 5.20. The BCM is composed of three sub-blocks: an Architecture Look-Up Engine, an Allocation Engine and a Configuration Engine. The reason for the structural modularity is two-fold: fist, through a modular approach to design, an upper bound can be placed on system complexity; second, multiple variations of one block can be implemented and used in various circumstances without affecting the design of the others.

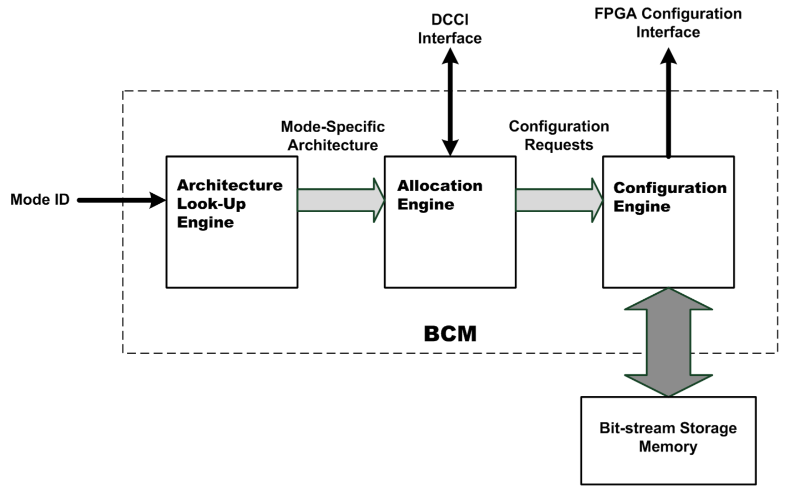

DCCI Interface
FPGA Configuration Interface
Mode-Specific Architecture
Configuration Requests
Mode ID
Architecture Look-Up Engine
Allocation Engine
Configuration Engine
BCM
Bit-stream Storage Memory

Figure 5.20: BCM General Architecture

The Architecture Look-Up Engine receives mode information from the Mode-Mapper, and uses this information to determine what the associated processor architecture should be. This architecture information takes the form of a listing of CMFU IDs; because each CMFU is responsible for its own connectivity setting, no other information is required to describe the system architecture. This architecture listing is transferred to Allocation Engine through a dedicated interface built on the basis of a First-In First-Out (FIFO) buffer (which is used to transfer said listing). This interface is shown in Figure 5.21; it consists of a FIFO control portion (write control for the Architecture Look-Up Engine and read control for the Allocation Engine) as well as two hand shake signals.

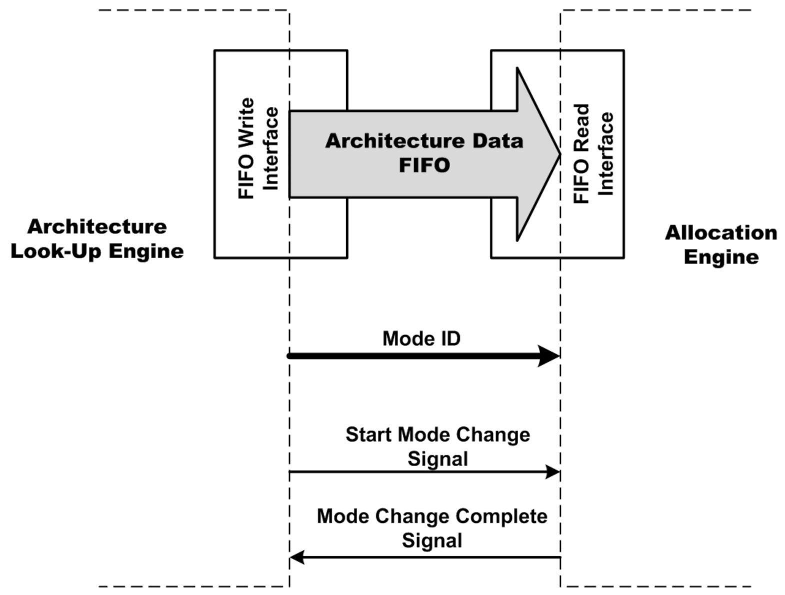


Figure 5.21: Interface Between The Architecture Look-Up Engine and Allocation Engine

The Allocation Engine receives architecture listings and must determine which CMFUs are missing from the system and allocate them. For each missing CMFU, the Allocation Engine will allocate it to a system slot (either a free one, if available, or to one housing a CMFU that is no longer used); based on this allocation, a partial bit-stream will be selected, and the associated information will be sent to the Configuration Engine via a second dedicated interface. This interface takes the form shown in Figure 5.22, and consists of a start address and length identifying the bit-stream location in the storage memory, as well as additional hand shake signals.

Finally, the Configuration Engine is responsible for reading bit-stream information from the non-volatile storage memory in the system, and writing it to the FPGA configuration interface. The structure and behavior of the Configuration Engine can vary extensively, depending on the type of storage memory being used, and the type of configuration memory being targeted. Thanks to the modular approach being adopted and the adherence to set internal interfaces, the rest of the BCM need not be aware of this variation.

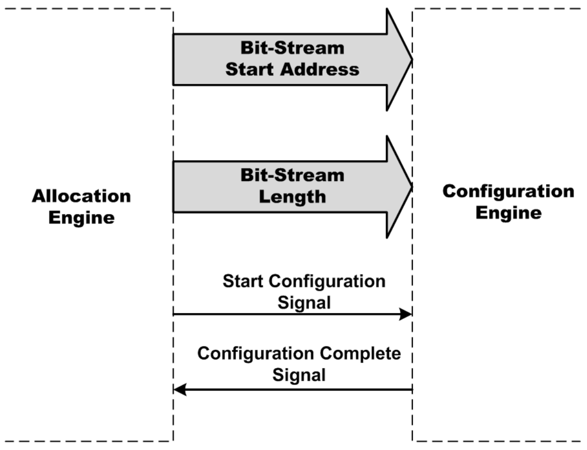


Figure 5.22: Interface Between The Allocation Engine and Configuration Engine

### 5.6.3 Architecture Look-Up Engine

The Architecture Look-Up Engine is responsible for determining, for any given mode, what the processor architecture must be. It receives mode information from the Mode-Mapper, and determines the architecture for said mode through a look-up process. Mode-specific architectures are described as shown in Figure 5.23; for each mode, the number of CMFUs used and their IDs are listed. The mode ID can be used as the basis of the look-up process, and each entry is stored as a sequence of $Nc_{CMFU} + 1$ memory words describing the architecture; $Nc_{CMFU}$ refers to the maximum number of concurrent CMFUs needed for any mode, as defined in Section 5.2 above. If multiple types of slots are used, then each entry will consist of two words, one being the CMFU ID and the other being the slot type the CMFU requires.

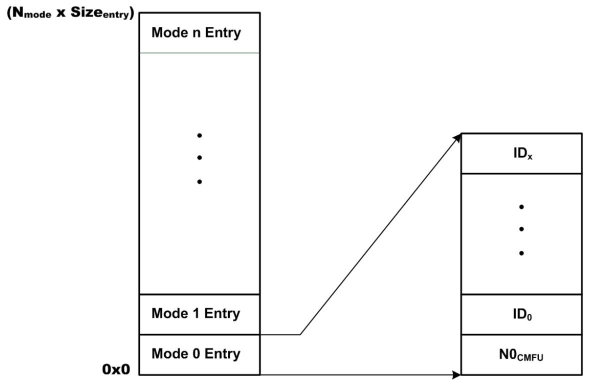


Figure 5.23: Mode-Specific Architecture Look-Up Table Structure

The general architecture of the Architecture Look-Up Engine is shown in Figure 5.24. The core elements of the Engine are a look-up memory and a control FSM. The look-up memory stores architecture entries, while the control FSM is responsible for providing all control procedures. In addition, the Engine also includes a mode comparator block, a start address decoder and an address counter. The mode comparison block consists of a register and comparator, and is used to determine if the mode input has changed. The start address decoder determines the start address in the look-up table for a given mode ID; this start address is derived using the mode ID and the size of architecture entries. Finally, the address counter is used during to read a complete entry from the look-up memory (each entry consists of multiple words).

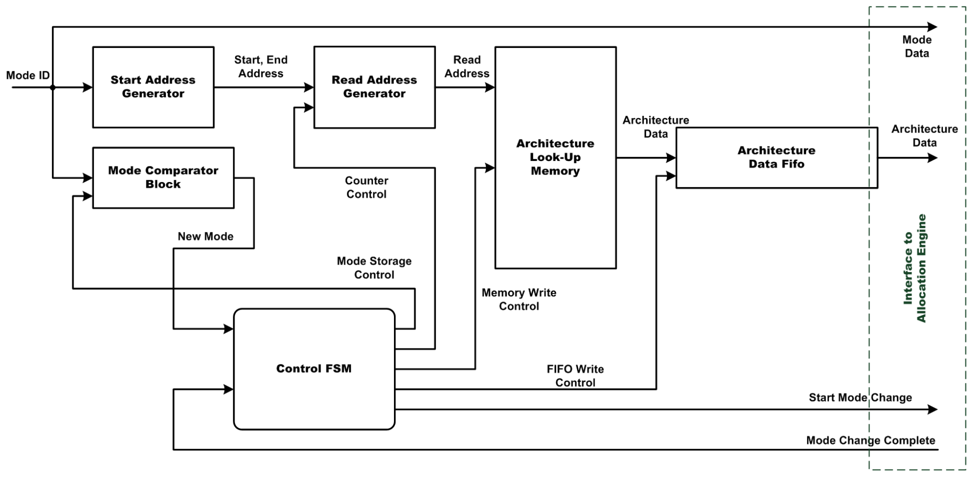


Figure 5.24: Architecture Look-Up Engine General Architecture

The behavior of the control FSM is shown in Figure 5.25, in the form of a state transition diagram. The control unit will remain idle until a mode change is detected. At this point, the input mode information will be stored locally, and start and end addresses for the appropriate architecture entry will be computed and loaded into the address counter. Following this, words will be read from the look-up memory and written to the interface FIFO; once the final word is read (determined by comparing the counter address with a computed end address), the Allocation Engine will be activated via an interface flag. The Architecture Look-Up Engine will ignore all new mode requests until the Allocation Engine finishes the allocation and configuration procedures for the current mode.

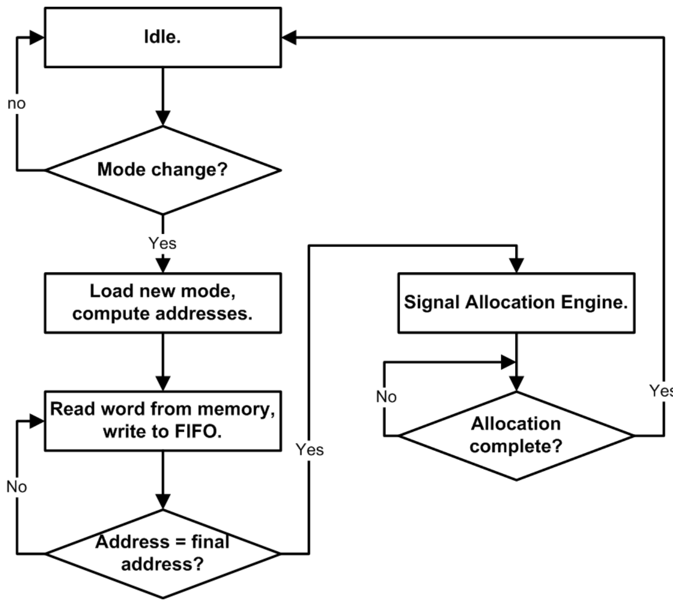

Idle.
no
Mode change?
Yes
Load new mode, compute addresses.
Read word from memory, write to FIFO.
No
Address = final address?
Yes
Signal Allocation Engine.
No
Allocation complete?
Yes

Figure 5.25: Architecture Look-Up Engine General Behavior

### 5.6.4 Allocation Engine

The Allocation engine receives a mode-specific architecture entry from the Architecture Look-Up Engine and must ensure that all needed CMFUs are either present in the system or loaded to the system if missing. To accomplish this task, the Allocation Engine uses two types of records: a slot allocation map and a bit-stream memory map. The slot allocation map describes the distribution of CMFUs in system slots; it has one entry for each slot in the system, consisting of the CMFU ID assigned there, and a flag indicating if the slot is empty or not (if a slot is marked as empty, this means that it is not used for the current mode). If multiple slot types are used, a second set of entries is stored per slot, identifying its type. These entries are implemented as register files; this approach is feasible due to the fact that the number of entries is limited.

The second type of record used is a bit-stream memory map, which is used to identify the location of a given bit-stream (full or partial), and is implemented using a look-up memory. This memory map takes as input an address derived from a combination of CMFU ID and slot number, and will yield two data words: a start address and a length for the given bit-stream; the structure of this record implementation is shown in Figure 5.26. The total number of entries is equal to $N_{slot} \times N_{CMFU}$ (system slots $\times$ total number of CMFUs) plus a number of full bit-streams $N_{sbits}$ (at least 1).

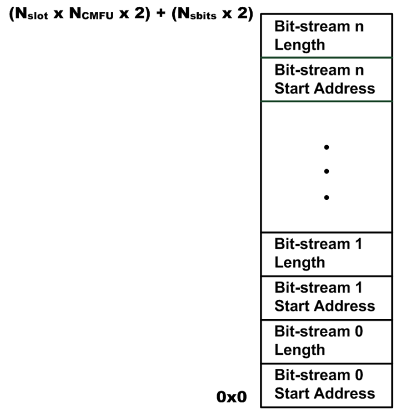

(Nslot x NCMFU x 2) + (Nsbits x 2)
Bit-stream n Length
Bit-stream n Start Address
Bit-stream 1 Length
Bit-stream 1 Start Address
Bit-stream 0 Length
0x0
Bit-stream 0 Start Address

Figure 5.26: Memory Map Structure for the Bit-stream Storage Memory

The structure of the Allocation Engine is shown in Figure 5.27. The engine consists of a slot allocation map (composed of two register files), a bit-stream memory map look-up table (implemented as a read-only memory), an address decoder, an allocation counter, a search counter, a mode storage block and local storage registers for architecture data received from the Architecture Look-Up Engine. The address decoder is responsible for generating an address for the bit-stream memory map look-up table based on a specified CMFU ID, slot number and the size of memory-map entries. The allocation counter is used to track CMFU allocation. The search counter is used to during search procedures in the system, as described below. Finally, the mode storage block consists of two register and a multiplexer; this block can be used to either disconnect all CMFUs or to delay the propagation of the new mode until after all allocation and configuration activities are finished.

Finally, the control FSM is responsible for controlling all Engine circuits and implementing the desired procedures. The behavior of the FSM is shown as a flow-chart in Figure 5.28. Upon exiting from reset, the FSM makes an automatic configuration request to the Configuration Engine; a full bit-stream is specified for this initial configuration, using hard-coded parameters (start of the bit-stream and length). This behavior assumes that the BCM is external; if the BCM is internal, this step is omitted.

Once the initial configuration is complete, the FSM enters an idle state where it waits to be triggered by the Architecture Look-Up Engine. Once triggered, the FSM enters its main processing loop. If the system contains no spare system slots, a special “disconnect” mode is broadcast to all CMFUs; once all CMFUs are disconnected, the allocation procedure can proceed. Alternatively,

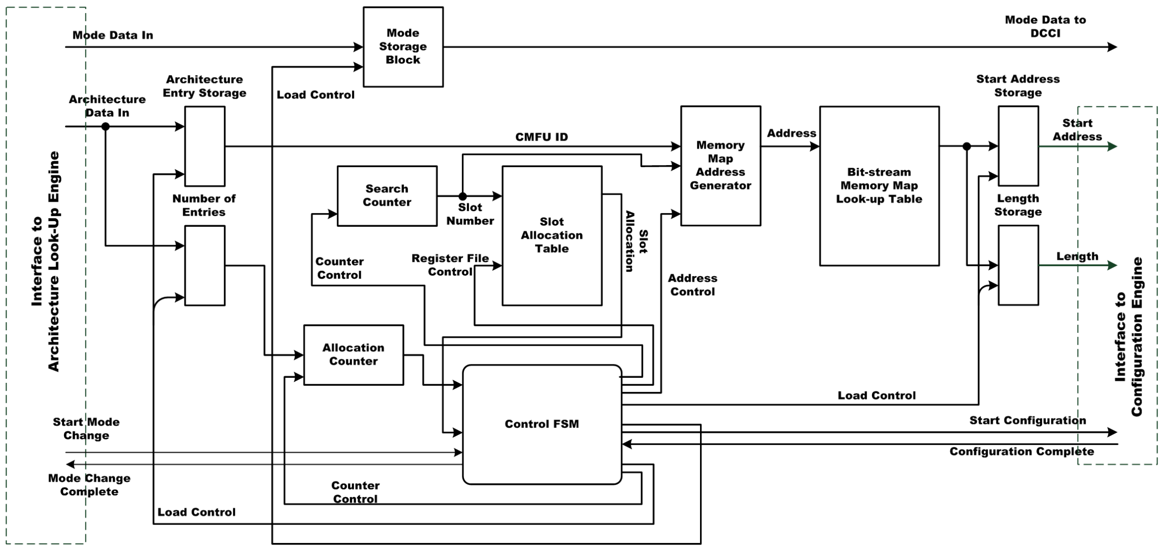
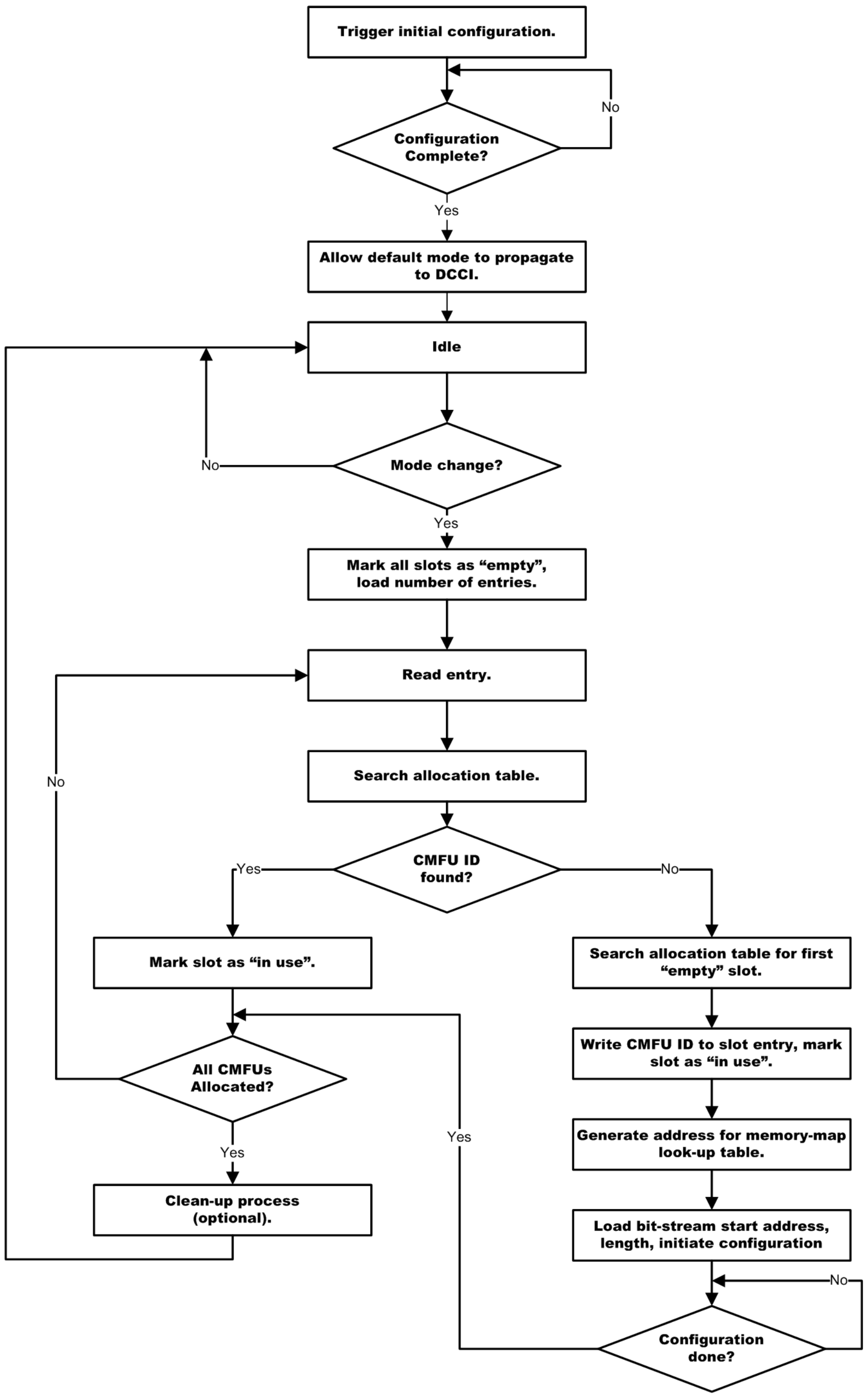


Figure 5.27: Allocation Engine General Architecture

if spare slots are available, the allocation procedure begins directly. In either case, the first step is to reset part of the allocation table, such that all slots are marked as empty. The rest of the procedure is iterative, and is repeated for each CMFU used by the current mode. For each CMFU ID, the allocation table is searched to determine if the CMFU has been previously allocated; if yes, then the associated slot is marked as "not empty" and the next iteration begins. If the CMFU is not allocated, a second search is initiated to find the first empty slot in the system; once this slot is found, the CMFU is allocated to said slot, and a configuration request is made to the Configuration Engine. Once configuration of the CMFU to the selected slot is complete, the slot is marked "not empty" and the next iteration begins. Once all CMFUs are allocated, the Allocation Engine allows the mode information to propagate to the rest of the system. Optionally, at the end of the allocation procedure, "clean-up" operations can be performed; these consist of finding each slot marked as empty and loading it with a blank bit-stream. Taking this approach can reduce system power consumption to some extent (by eliminating the switching associated with disconnected CMFUs); alternatively, the CMFUs can be left in the system, in case they are needed in the future (this can reduce the number of configuration actions performed in future mode changes).



Trigger initial configuration.
Configuration Complete?
No
Yes
Allow default mode to propagate to DCCI.
Idle
Mode change?
No
Yes
Mark all slots as "empty", load number of entries.
Read entry.
Search allocation table.
CMFU ID found?
Yes
No
Mark slot as "in use".
Search allocation table for first "empty" slot.
Write CMFU ID to slot entry, mark slot as "in use".
All CMFUs Allocated?
No
Yes
Generate address for memory-map look-up table.
Clean-up process (optional).
Load bit-stream start address, length, initiate configuration
No
Configuration done?
Yes

Figure 5.28: Allocation Engine General Behavior

### 5.6.5 Configuration Engine

The Configuration Engine receives commands from the Allocation Engine, which essentially consist of readying a set number of words from the bit-stream storage memory and writing them to the

FPGA configuration interface. The Configuration Engine does not deal with any records as part of its operation; rather, its structure reflects the nature of the interfaces it interacts with. As such, the architecture and behavior of the Configuration Engine can only partially be defined here.

The architecture of the Configuration Engine is shown in Figure 5.29, and consists of an address generator block, a data storage and conversion block, a control FSM and a number of storage registers. As usual, the control FSM implements the behavior of the Configuration Engine; the FSM drives control signals for the Engine circuitry, as well as for the storage memory and configuration interfaces. The storage registers are used to store the start address and length values of the bit-stream being configured. The address generator block is used to generate read addresses for the bit-stream storage memory, and is responsible for determining when the full bit-stream has been read (based on its length). Finally, the data storage and conversion block stores incoming data from the bit-stream memory, and if necessary converts its form for writing to the configuration interface (for example serialization).

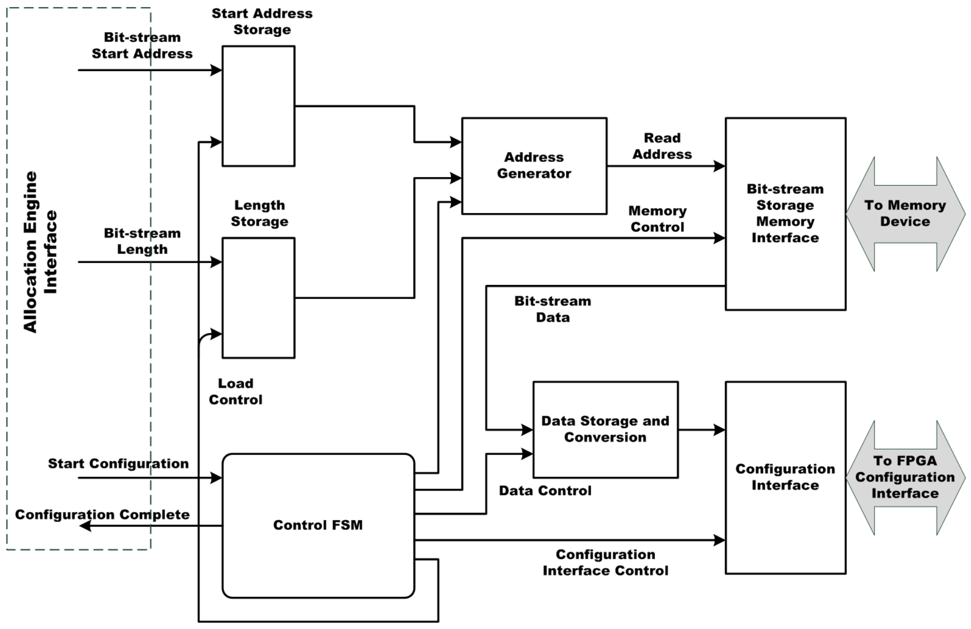


Figure 5.29: Configuration Engine General Architecture

The behavior of the control FSM is shown in flow-chart form in Figure 5.30. The control FSM emerges from rest into an idle state where it monitors a trigger signal coming from the Allocation Engine. When this trigger is received, the start address and length of the bit-stream are stored locally and the final read address is computed (via the address generator block). The FSM then enters a main process loop, where it remains until the complete bit-stream is read. Each loop iteration consists of updating the read address for the configuration memory, reading a new word and storing it into the data storage and conversion block, and writing it to the configuration interface (via manipulation of the appropriate control signals in the configuration interface).

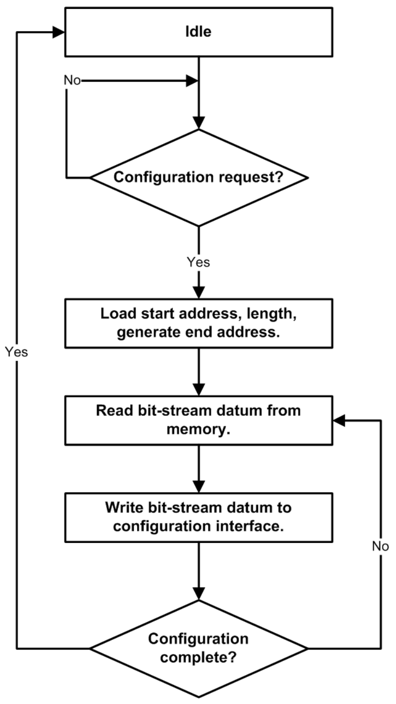

Idle
No
Configuration request?
Yes
Load start address, length, generate end address.
Yes
Read bit-stream datum from memory.
Write bit-stream datum to configuration interface.
No
Configuration complete?

Figure 5.30: Configuration Engine General behavior

### 5.6.6 BCM Customization

The first customization aspect of the BCM is its implementation location: internal to the target FPGA, or external, using a secondary device. If the BCM is internal to the FPGA, then it will require additional external support; the initial configuration of the FPGA will have to be undertaken by an external device (such as the Platform Flash XL devices introduced above [166]). As well, if included internally, the BCM will consume some device resources, both in terms of programmable blocks (logic and memory) as well as I/O blocks. More specifically, memory interfaces to the bit-stream storage memory will have to be included. Finally, to support an internal BCM, the target FPGA must offer some form of internal configuration port.

External implementation implies the addition of a secondary device to the system, which houses the BCM; this device will probably take the form of a secondary, smaller FPGA which incorporates non-volatile configuration capabilities (e.g [48]). For both internal and external implementation, a secondary component must be added to the system; the trade-off then becomes which of these approaches leads the smallest overhead (power, area, cost). Additional considerations emerge if the primary system aim is fault resilience (see Chapter 6).

The second important customization step for the BCM is the preparation of all system records for implementation. Both read-only memories used in the Architecture Look-Up Engine and the Allocation Engine must be initialized with the correct data structures (this is usually encoded into the configuration bit-stream used to configure the device). Likewise, the slot allocation register files must be initialized with the correct settings for the initial configuration. The interfaces between the three engines must tailored to reflect the parameters of the system; this includes the FIFO between the Architecture Look-Up Engine and the Allocation Engine, and the bit-stream information ports (bit-stream start address and length) between the Allocation Engine and the Configuration Engine.

Finally, the internal structure of the Configuration Engine must be tailored to match the structure and behavior of the bit-stream storage memory interface and the FPGA configuration interface. Control signals must be added for each interface according to its specification, and the requisite behavior must be codified in the control FSM. The structure of the data storage and conversion block must also be customized, based on the relation between the storage memory interface and the configuration interface. This block can be as simple as a single register, may consist of multiple registers, or may include serialization circuits of various types.

## 5.7 Summary

The above chapter has formally introduced the MACROS Framework architecture template, which is composed of Collaborative Macro-Function Units (CMFUs), a Distributed Communication and Control Infrastructure (DCCI), a Bit-Stream and Configuration Manager (BCM) and a Mode-Mapper. Each element was described in detail from both an architectural and behavioral perspective; in each case, potential customization options were also discussed, which would allow the architecture to adapt to various application requirements.

While the general operating principle of MACROS Framework Systems has already been introduced at a conceptual level, it has not been presented in detail. In this chapter, detailed behavioral descriptions were provided for individual system building blocks; however, their interaction was described only in general terms. Chapter 6 will integrate these disparate behaviors into a complete description of system behavior. This system behavior can be divided into two types, depending on the aspect being addressed: support of multi-mode operation or fault mitigation. However, both behavioral types rely on the same building blocks, which were presented above.

# Chapter 6

# MACROS Framework Run-Time Behavior

## 6.1 Introduction

The MACROS Framework operating principle was introduced in Chapter 3, and the associated on-chip architecture was formally defined in Chapter 5; this architectural definition included descriptions of the behavior of individual components. This chapter will combine these individual behaviors into a unified system wide operating behavior description; in essence, it will show how the principle of Chapter 3 is implemented using the architecture of Chapter 5.

System behavior will be considered from two points of view: multi-mode application support, and fault mitigation support. System behavior in multi-mode operation will be divided into start-up behavior, regular operation and mode-change behavior. In each case, the elements involved in the process will be identified, and their interaction will be described. The description of the fault mitigation process will follow the same general approach, but will be preceded by a discussion of the supported fault model, and the fault mitigation principle being employed.

## 6.2 Multi-Mode Application Behavior in MACROS Systems

To support multi-mode applications, MACROS Framework systems generate mode-specific ASP architectures at run-time using a library of CMFUs; this general behavior can be sub-divided into three types of behavior: system start-up, regular operation and run-time ASP generation. The system start-up process describes the assembly and integration of an initial ASP when the system first emerges from global reset (for example, when power is first applied). Regular system operations consists of scheduling and synchronization activities in the individual CMFUs which implement mode-specific functionality. Finally, run-time ASP generation is accomplished through an interaction between BCM, DCCI and CMFUs in response to mode changes. Each type of behavior will be described in the following subsections.

### 6.2.1 System Start-Up

System start-up will always consists, first and fore-most, of the initial configuration of the system FPGA; this will be the case for any reconfigurable computing system. This start-up configuration procedure will take one of two forms, depending on whether the BCM is implemented as an external (off-chip) element, or an internal one. As specified before, on-chip BCM implementations require a secondary external support system; for example, a Platform Flash XL configuration device can be used to load the initial system bit-stream [166]. This initial bit-stream will contain all system infrastructure (including the on-chip BCM), as well as CMFUs for a default system mode. If the BCM is external to the FPGA, when the Allocation Engine emerges out of reset it will immediately make a request to the configuration engine for an initial configuration. Special parameters will be specified for this configuration, pointing to a static bit-stream.

In both cases, once the main FPGA is configured, the BCM will allow the default mode ID to propagate to the DCCI, and then, through the DCCI, to all system CMFUs. All CMFUs emerge from reset with status 110 (disconnected), and will continuously sample the mode ID until a mode indicates their addition to the system; as mentioned in Chapter 5 the initial, default mode broadcast to all CMFUs is a “disconnect all” mode, in which all CMFUs are disconnected. As soon as the new mode reaches the CMFUs, they will all make connection requests. At this point, each LCCU will attempt to implement a connectivity change (either through the Canonical or Minimal process); as part of this process all LCCUs will broadcast control information through the Control Data Broadcast Network. Each LCCU will then connect each CMFU to a specified source, until the complete system is assembled. During this whole process, only LCCU connected to populated slots will be active; LCCUs associated with currently empty slots will remain idle. A graphical

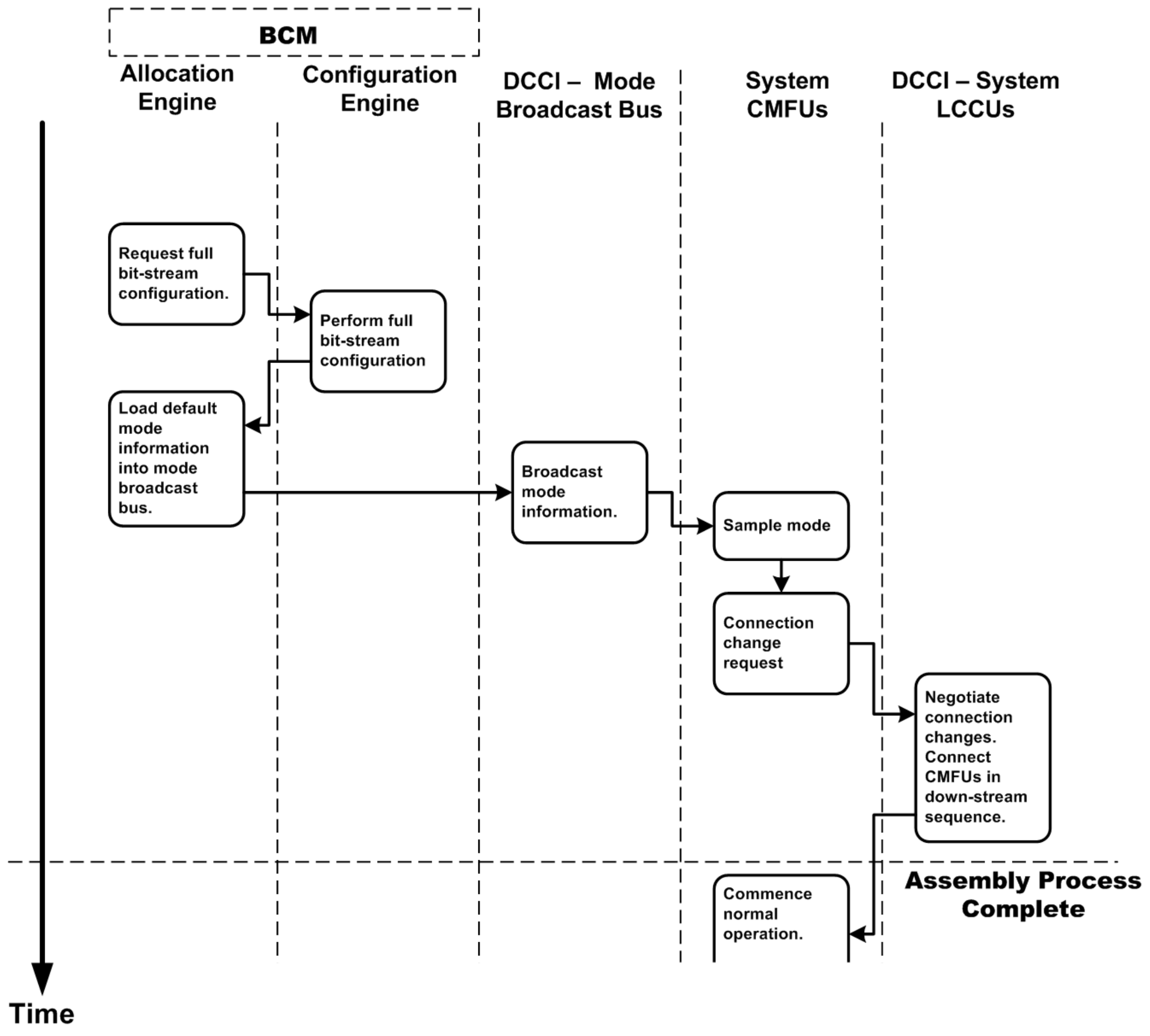

BCM
Allocation Engine
Configuration Engine
DCCI – Mode Broadcast Bus
System CMFUs
DCCI – System LCCUs
Request full bit-stream configuration.
Perform full bit-stream configuration
Load default mode information into mode broadcast bus.
Broadcast mode information.
Sample mode
Connection change request
Negotiate connection changes. Connect CMFUs in down-stream sequence.
Assembly Process Complete
Commence normal operation.
Time

Figure 6.1: MACRO System Start-Up Process

representation of this process is shown in Figure 6.1.

## 6.2.2 Normal Processing Operations

During normal processing, system CMFUs are the primary active element in a MACROS Framework system; the control units of most infrastructure elements find themselves in their “Idle” states (as described in Chapter 5). When in these states, system elements are monitoring various environment conditions. The Architecture Look-Up Engine of the BCM is monitoring the mode data coming from the Mode-Mapper, waiting for a change in mode. The Allocation engine is monitoring the control inputs from the Architecture Look-Up Engine, waiting for a mode change to be indicated. Finally, the Configuration Engine monitors control signals from the Allocation Engine, waiting for configuration requests. Outside the BCM, all system LCCUs are monitoring control signals from their local CMFU as well as control data from all other LCCUs; each LCCU waits for either a local connectivity change or a disconnection request up-stream.

The more active elements of the system are the CMFUs, the system crossbar, and the Mode-Mapper. During regular operation, disconnected CMFUs (not used in the current mode) continuously sample the system mode via their Co-Op unit. Connected CMFUs (those used for the current mode) are periodically undertaking work periods as dictated by the stream structure. For each work period, the Co-Op Unit activates the local IP Core (if necessary), generates stream control data (once again, if necessary), samples the system mode and updates local status and connectivity information; this process is shown graphically in Figure 6.2. The crossbar directly supports processing activities by transferring stream data over all active links in the system. Finally, the Mode-Mapper is responsible for generating mode IDs based on detected conditions, either in the data stream processed or in environmental and operating conditions (via sensor inputs). Depending on the nature of the application and the monitored conditions, multiple processing steps, each varying in complexity, may be needed; the period of these processing steps will be, once again, dictated by the nature of the application.

### 6.2.3 Run-Time Mode-Based ASP Generation

Any mode-based architecture change originates in the Mode-Mapper, and then propagates into the rest of the system. The architecture change procedure associated with a mode change can be divided into configuration activities and connectivity changes (i.e. downloading of bit-streams and on-chip assembly of the system). The change in mode will activate the Architecture Look-Up Engine, which will determine the new system architecture for the new mode. Once all architecture entries are read from the look-up memory and loaded into the interface FIFO, the change of mode is indicated to the allocation engine. The Allocation Engine will then allocate each missing CMFU and request its configuration from the Configuration Engine; once all configurations are complete, the mode is allowed to propagate into the system (this process assumes the existence of spare system slots). The complete mode-change activities performed by the BCM are illustrated in Figure 6.3.

Once the mode data has propagated into the DCCI and reached all system CMFUs, the same process described in Section 6.2.1 will take place. System CMFUs will make connectivity requests

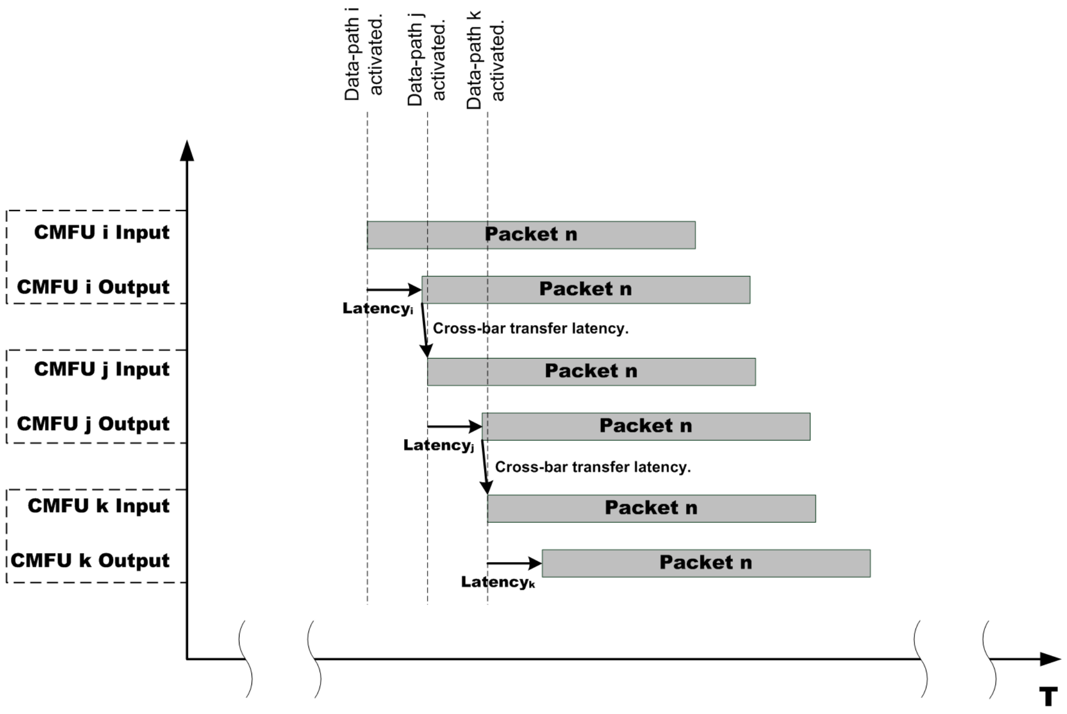


Figure 6.2: Distributed Scheduling Process

as they finish their current work periods and sample the new mode. System LCCUs will respond to these requests by indicating globally (via the Control Data Broadcast Network) that connectivity changes are required. As CMFUs finish work periods, connectivity changes can be implemented according to either the canonical or minimal connectivity change process.

## 6.3 Fault Mitigation Behavior in the MACROS Framework

Fault mitigation in MACROS Framework systems is accomplished via the same basic mechanism used to support multi-mode operation: run-time architecture adaptation (i.e. ASP generation). This section describes in detail how the relocation process is implemented using the elements introduced in the previous chapter. The section will first describe the fault model being targeted, followed by a description of the principle behind the use of component relocation as a primary means of fault mitigation. Following this, an expanded BCM behavior will be introduced, aimed at supporting complete mitigation operations, and the mitigation process proper will be described. Finally, the section will conclude with a brief discussion of fault detection and hardening steps that can be applied to the system.

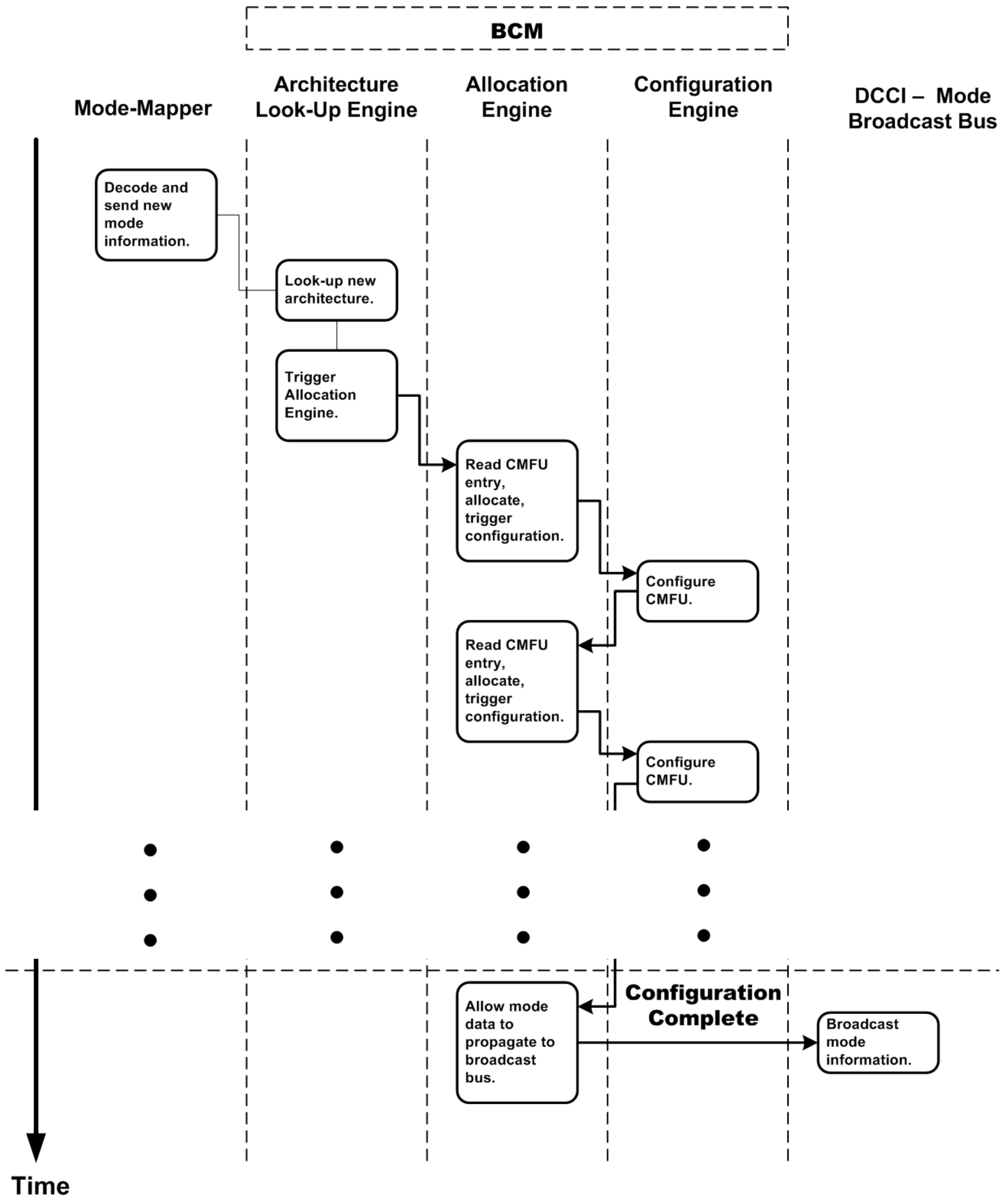

BCM
Mode-Mapper
Architecture Look-Up Engine
Allocation Engine
Configuration Engine
DCCI – Mode Broadcast Bus
Decode and send new mode information.
Look-up new architecture.
Trigger Allocation Engine.
Read CMFU entry, allocate, trigger configuration.
Configure CMFU.
Read CMFU entry, allocate, trigger configuration.
Configure CMFU.
Configuration Complete
Allow mode data to propagate to broadcast bus.
Broadcast mode information.
Time

Figure 6.3: Run-Time ASP Generation Procedure - CMFU Configuration

### 6.3.1 Targeted Fault Model

As previously discussed, faults affecting FPGAs are expected to be of two types: transient and permanent. Transient faults are primarily caused by high-energy particles impacting the FPGA die, and can lead to SEUs and SEFIs [26]. Permanent faults can have multiple causes: exposure to high Total Ionizing Doses [28], aging of the die or thermal cycling of the device. SEUs can affect single transistors, or multiple transistors in a region; however, this effect is localized to the region where the impact took place. Over a long enough time-frame, permanent faults can affect a large portion of a device, depending on the cause of the fault. However, over short time-frames, individual faults manifest themselves in limited regions of the device.

Based on these observations, a simple fault model can be derived. First of all, only one fault is expected to occur over short time-frames; specifically, the time-frame in question is the mitigation time in a MACROS Framework system. In other words, the system is expected to have to mitigate one fault at a time, although multiple faults may occur over the life-time of the system. Secondly, faults are expected to be localized to specific regions, which will correspond to one functional unit or control element. In MACROS Framework systems, this fault localization leads to faults occurring in single CMFUs, or affecting only one port of the DCCI.

### 6.3.2 MACROS Framework Fault Mitigation Principle

As previously stated, MACROS Framework systems undertake fault mitigation via relocation of functional units, coupled with additional test operations. To understand why this approach is taken, Figure 6.4 lists the MACROS Framework mitigation process which automatically performs the relocation activity followed by potential scrubbing and testing, as well as a mitigation process which begins with scrubbing, followed by relocation. As the figure shows, in the case of the MACROS framework mitigation process, the system can resume operation after one configuration step. If the alternative approach is used, scrubbing is performed first, and relocation may or may not be undertaken if the scrubbing operation fails; however, in this case, at least one and possible more configuration activities may be needed before the system can resume correct operation.

By undertaking relocation immediately, the MACROS framework accomplishes two things: 1) the mitigation time is kept constant regardless of the fault type; 2) the mitigation time is always minimal, regardless of the fault type. In essence, the MACROS Framework takes advantage of the fact that relocation and scrubbing require the same amount of configuration time; given that relocation *may* happen, the system performs this task immediately and forgoes scrubbing/testing operations until after the system has resumed operation, thus taking full advantage of the DPR concept.

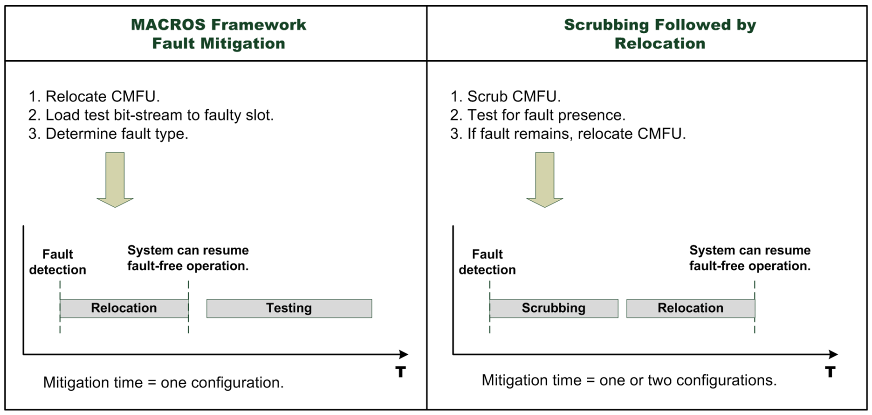

MACROS Framework Fault Mitigation
Scrubbing Followed by Relocation
1. Relocate CMFU.
2. Load test bit-stream to faulty slot.
3. Determine fault type.
1. Scrub CMFU.
2. Test for fault presence.
3. If fault remains, relocate CMFU.
Fault detection
System can resume fault-free operation.
Relocation
Testing
T
Mitigation time = one configuration.
Fault detection
System can resume fault-free operation.
Scrubbing
Relocation
T
Mitigation time = one or two configurations.

Figure 6.4: Comparison of Mitigation Procedures

In addition to scrubbing and relocation, mitigation can also be accomplished via the use of ECCs as well as module redundancy (TMR). Both of these approaches, however, have limitations that the relocation process does not have. Both methods can only support a limited number of faults: in the case of a TMR system, faults in more than one module will cause incorrect system behavior; likewise, ECCs can only correct a set number of faults. On the other hand, relocation can mitigate any number of faults provided that A) spare slots are available and B) the faults follow the fault model described above.

The proposed mitigation process relies on the relocation of CMFUs to various system slots; this mitigation procedure does not explicitly address faults in the DCCI. However, the DCCI structure is distributed in nature, with logic circuitry being uniformly distributed among system slots. This aspect is illustrated in Figure 6.5, which shows all elements of the DCCI which are associated with a system slot; as can be seen, each system slot has an associated crossbar port, LCCU, and portion of the Control Data Broadcast Network (for fault tolerance purposes, only separate links or separate aggregation elements should be used in the Control Data Broadcast Network). By relocating the CMFU, faults in any of these elements can be by-passed; once again, the MACROS Framework attempts to take full advantage of the DPR concept.

### 6.3.3 BCM Behavior for Fault Mitigation

As with any fault-tolerant system, fault mitigation is preceded by fault detection; the topic of fault detection via built-in self-test (BIST) has been analyzed in past works [38] and will not be addressed in detail here. Once a fault is detected, the MACROS framework specifies that the

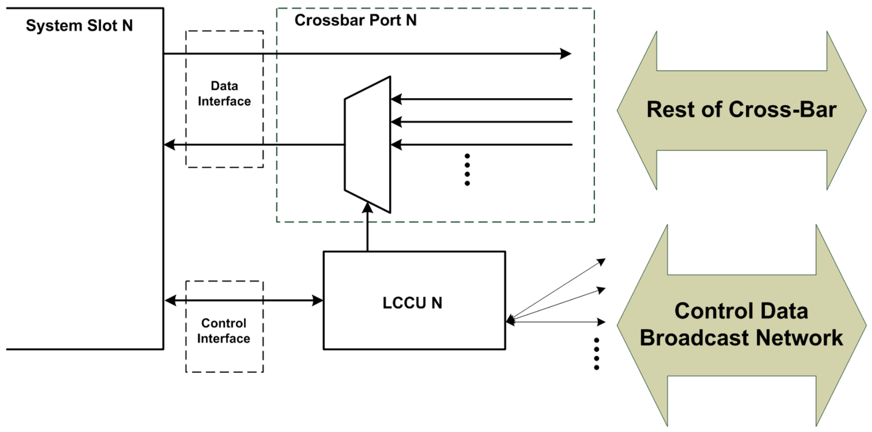


Figure 6.5: Distributed Nature of DCCI System

CMFU associated with the detected fault is relocated; this process will automatically avoid faults in elements of the DCCI (as described above). Thus, the main mitigation mechanism is relocation, potentially followed by testing to determine if the fault is permanent, and it is implemented by the BCM.

To implement this mitigation process, the BCM behavior as previously presented must be altered; more specifically, this leads to behavioral alterations to the Allocation Engine. The Allocation Engine must expand its slot allocation map, to include a flag for each slot, indicating if it is faulty of not. Using this flag, the Allocation Engine behavior is altered as shown in Figure 6.6. If a fault is detected, the slot in question is flagged as faulty; the Allocation Engine then takes the CMFU assigned to the faulty slot and re-allocates it to the first free slot of the correct type. Based on this allocation, a bit-stream is selected, and the configuration engine is activated, thus implementing a relocation procedure.

Once the CMFU has been relocated, the faulty slot has to be tested to determine if the fault is transient or permanent; this is accomplished by loading a test component into the faulty slot. If the slot continues to be reported faulty after the test component has been loaded into the system, the fault is assumed to be permanent. Otherwise, the fault is transient, and the slot is no longer marked as faulty. Once all these operations are complete, the Allocation Engine returns to its idle state.

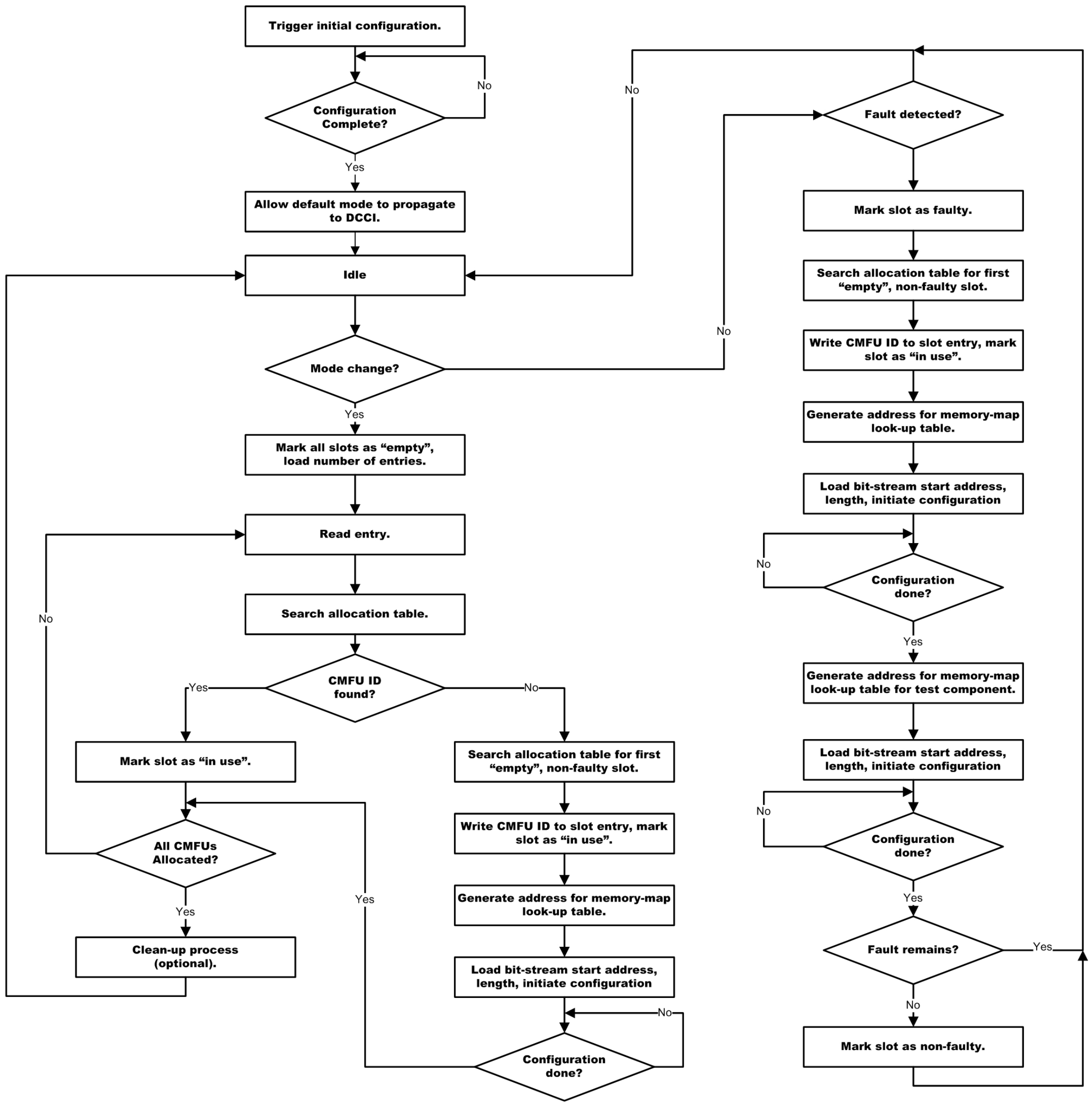

Trigger initial configuration.
Configuration Complete?
No
Yes
Allow default mode to propagate to DCCI.
Idle
Mode change?
Yes
No
Mark all slots as "empty", load number of entries.
Read entry.
Search allocation table.
CMFU ID found?
Yes
No
Mark slot as "in use".
All CMFUs Allocated?
Yes
Clean-up process (optional).
Search allocation table for first "empty", non-faulty slot.
Write CMFU ID to slot entry, mark slot as "in use".
Generate address for memory-map look-up table.
Load bit-stream start address, length, initiate configuration
Configuration done?
Fault detected?
Mark slot as faulty.
Generate address for memory-map look-up table for test component.
Fault remains?
Mark slot as non-faulty.

Figure 6.6: Allocation Engine Behavior for Fault Mitigation

### 6.3.4 Relocation-Based Fault Mitigation Process

Because of the distributed nature of the MACROS Framework, the fault mitigation process is accomplished via the interaction of multiple system elements, all working in tandem. All fault detection originates in local BIST circuits spread throughout the system; once a fault is detected, it is reported to the BCM. The fault will be associated with a specific slot, and will originate in a specific LCCU; at that point, all other connectivity data associated with this LCCU (CMFU ID in particular) will be ignored by other LCCUs.

The BCM will react to this reported fault by immediately relocating the CMFU associated with the reported fault; the individual steps for this process were described in detail in Section 6.3.3. Once the CMFU has been relocated, it will be present in the system at a different location; since all CMFUs start with a disconnected status, the first action taken by the CMFU will be to request connection into the system according to the current mode. This connection request will lead to all descendants of the CMFU being disconnected from the system and then reconnected, following either the Canonical or Minimal connectivity change procedure. Through this reconnection process, all CMFUs immediately down-stream will adjust their connectivity so that the relocated (as opposed to the faulty) CMFU is now specified as their data source. At this point, the relocation process is complete; this part of the mitigation procedure is shown graphically in Figure 6.7.

Once the CMFU has been relocated, the faulty region can be scrubbed and tested further to determine if the fault is permanent or not. If the fault persists after scrubbing, then it is treated as permanent in nature and the slot will remain marked as faulty. Alternatively, if the fault ceases to be reported, this indicates that a transient fault has been mitigated; thus, the slot is marked as non-faulty, and can now act as a spare system slot.

### 6.3.5 Extensions for Fault Tolerance in MACROS Framework Systems

As presented above,MACROS Framework systems offer the possibility of fault mitigation via relocation of CMFUs. However, a fault tolerant system must incorporate both fault detection as well as mitigation operations. While the topic of fault detection in reconfigurable systems is not a research objective of the presented work, it is discussed briefly here for the sake of completeness. In addition, although the MACROS Framework architecture is distributed in nature, two elements - the BCM and Mode-Mapper - are singular in nature; thus, the section also includes a discussion of how these systems may be hardened.

#### Fault Detection

As currently presented, the MACROS Framework architecture offers the possibility for fault mitigation via relocation. However, any MACROS system should also incorporate fault detection capabilities to ensure correct operation. Such fault detection can be provided through the inclusion of BIST circuitry in system CMFUs as well as the crossbar, LCCUs, and the Control Data Broad-

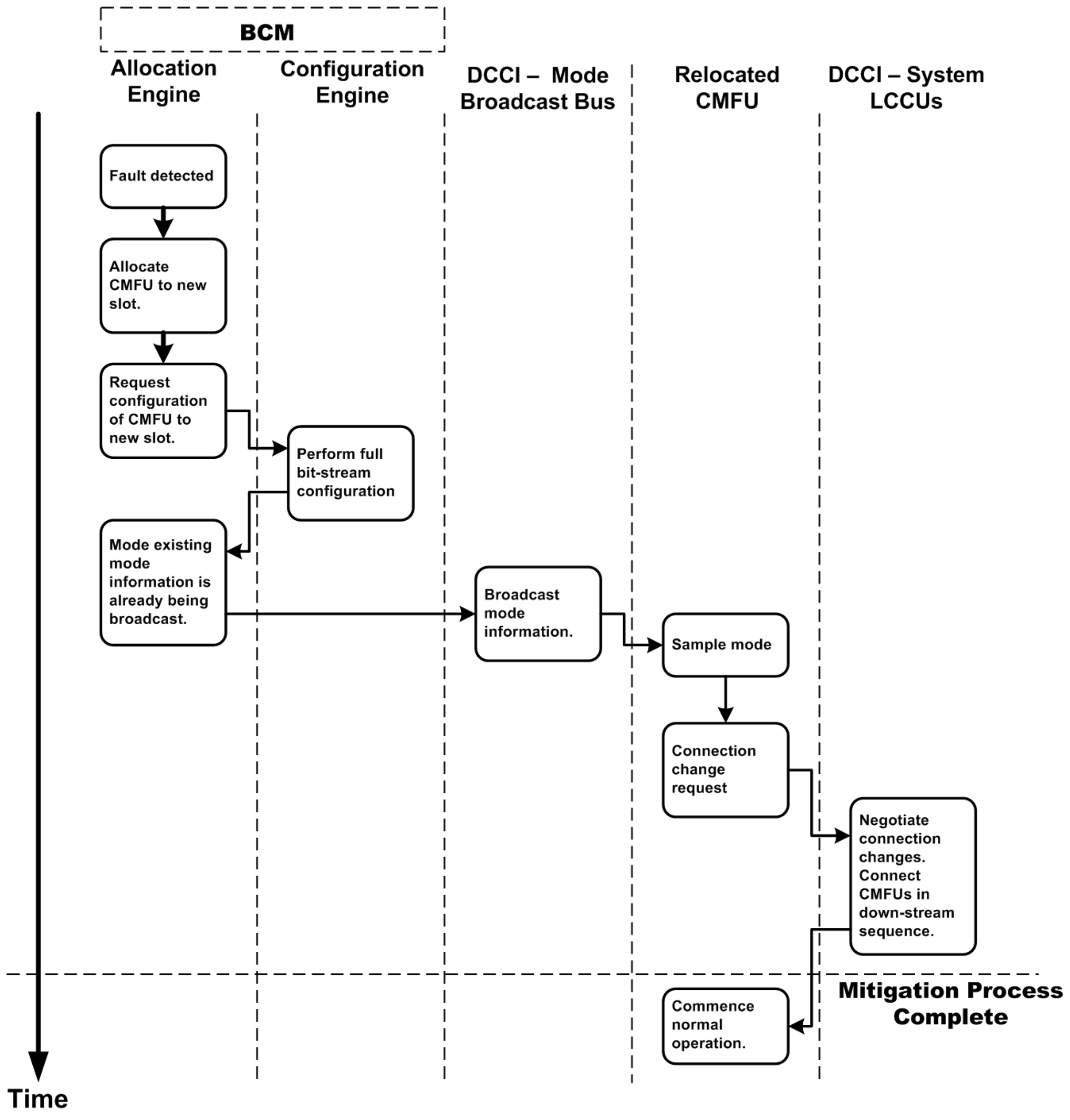

BCM
Allocation Engine
Configuration Engine
DCCI – Mode Broadcast Bus
Relocated CMFU
DCCI – System LCCUs
Fault detected
Allocate CMFU to new slot.
Request configuration of CMFU to new slot.
Perform full bit-stream configuration
Mode existing mode information is already being broadcast.
Broadcast mode information.
Sample mode
Connection change request
Negotiate connection changes. Connect CMFUs in down-stream sequence.
Mitigation Process Complete
Commence normal operation.
Time

Figure 6.7: Relocation-Based Mitigation Procedure

cast Network (if aggregation elements are used). The type of BIST circuit and testing process used would be largely application-dependent and would be selected from existing approaches in this field, such as those found in [38].

The inclusion of BIST circuits would allow for fault detection in individual system elements; however, these faults would have to be reported to allow mitigation to take place. In the MACROS Framework architecture, fault reporting could be accomplished using the Control Data Broadcast Network; the connectivity information transferred through the network could be expanded to include fault information. This fault information should be encoded using Error-Detection Codes, such that faults in the control data broadcast network will not lead to undetected faults. The fault information thus encoded would be transmitted to all other LCCUs in the system, as well as the BCM. LCCUs would use the error information to *ignore* all connectivity data coming from the LCCU associated with a fault. The BCM would use the fault information to identify faulty slots/regions and perform mitigation operations.

#### BCM and Mode-Mapper Hardening Procedures

The structure of MACROS Framework systems lends itself well to fault tolerant design due to the distributed nature of the system architecture. However, two system elements are not distributed in nature: the BCM and the Mode-Mapper; furthermore, both components are critical for system operation. Therefore, to ensure the fault resilience of the rest of the system, these two elements would have to be hardened. A straight-forward way of undertaking this hardening process would be to implement both elements using a separated external device, which would be, itself, hardened (either via shielding or radiation hardening); this approach would be similar to what is suggested in [156].

If the Mode-Mapper and BCM are implemented as internal elements then alternative hardening methods would have to be derived. The Mode-Mapper could be hardened via module redundancy (duplex and TMR); alternatively, the mode-mapper could be treated as a CMFU, and be relocated to specialized reserved slots. The BCM could, likewise, be hardened using module redundancy. However, unlike the Mode-Mapper, this approach would depend on the architecture of the FPGA used. If multiple on-chip configuration interfaces exist, then multiple copies of the BCM could be implemented, each targeting a different configuration interface.

## 6.4 Summary

The preceding chapter has presented a description of MACROS Framework systems run-time behavior, based on the architecture introduced in Chapter 5. This behavior primarily consists of run-time architecture adaptation procedures, and can be undertaken in response to changes in mode or the detection faults. For this reason, the behavioral description was divided into two parts, the first dealing with mode-specific ASP generation, while the second dealt with fault-induced CMFU relocation; both processes rely on the MACROS Framework building blocks of CMFUs, DCCI and BCM interacting together.

Chapters 5 and 6 together form a complete description of the MACROS Framework, incorporating both architecture and behavior. This complete description can now be used to guide the implementation of test systems which can be used to verify the choice of mechanism and architecture made in Chapter 4. Thus, Chapter 7 will use the information presented so far to introduce a number of experiments, including two full processors based on the MACROS Framework; Chapter 8 will then analyze the results of these experiments.

# Chapter 7

# MACROS Framework Experiments

## 7.1 Introduction

The MACROS Framework purports to allow run-time system assembly, architecture adaptation, and fault mitigation via CMFU relocation; these supposed capabilities must be verified. Additionally, the framework makes use of specific mechanisms as part of its structure as discussed in Chapter 4; these mechanisms must not only be verified, to ensure they offer the requisite behavior, but must also be analyzed and compared with alternative approaches, to determine their strengths and weaknesses. In order to undertake verification of the framework, as well as analysis, a number of experiments were derived and conducted. These experiments are described in the following sections; in each case, the aim of the experiment as well as the process undertaken will be described. The results of these experiments, as well as associated analysis will be presented in Chapter 8.

## 7.2 General Experimental Set-Up

All experiments described have as aim the determination of performance, power and resource use parameters for potential architectures and procedures; to obtain accurate measurements, the architectures were implemented as device bit-streams, and deployed using FPGA devices. Implementation procedures were undertaken using the Xilinx 14.7 CAD tool-chain [31, 169], as well as the PlanAhead tool for DPR implementations [170]. Resource costs were obtained from the reports generated by the Map tool [31]; circuit performance (in the form of maximum inter-register

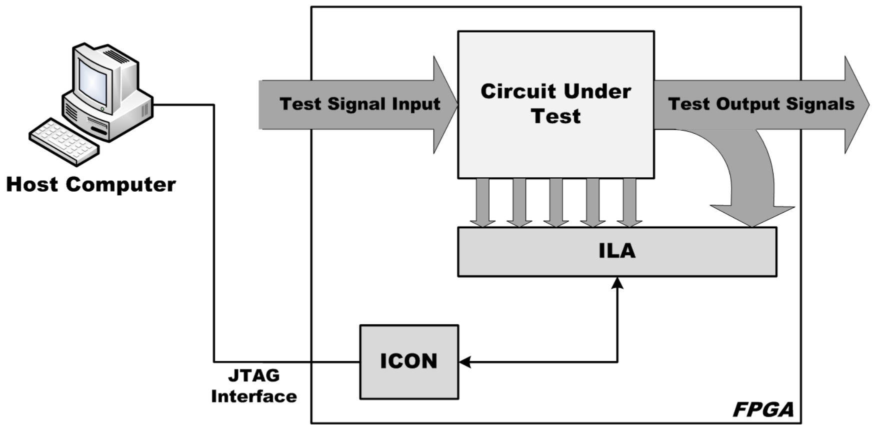


Figure 7.1: Experimental Setup Using an Integrated Logic Analyzer

combinational delays) were obtained from post-place-and-route timing analysis performed by the TRACE tool [31]. Finally, power estimates were obtained using the XPWR analysis tool [31], using fully placed and routed designs.

All experiments were conducted using the MARS reconfigurable platform [171], the KC705 evaluation board [172] and the ZedBoard [173]; the three devices used were a Virtex 4 XC4VLX160 FPGA, a Kintex 7 XC7K325T FPGA and a XC7Z020 programmable SoC, respectively. To obtain functional validation and performance measurements, test systems were deployed onto target FPGAs and measurements were obtained from these deployed and operating systems. To observe the behavior of various system circuits, the Chip Scope integrated logic analyzer [174] was utilized in conjunction with a host computer, as shown in Figure 7.1. Some of the experiments described also relied on the generation and processing of uncompressed video streams; for these systems functional validation was also accomplished via observation of the connected display device.

## 7.3 Experiment: Framework Verification and Performance Measurements

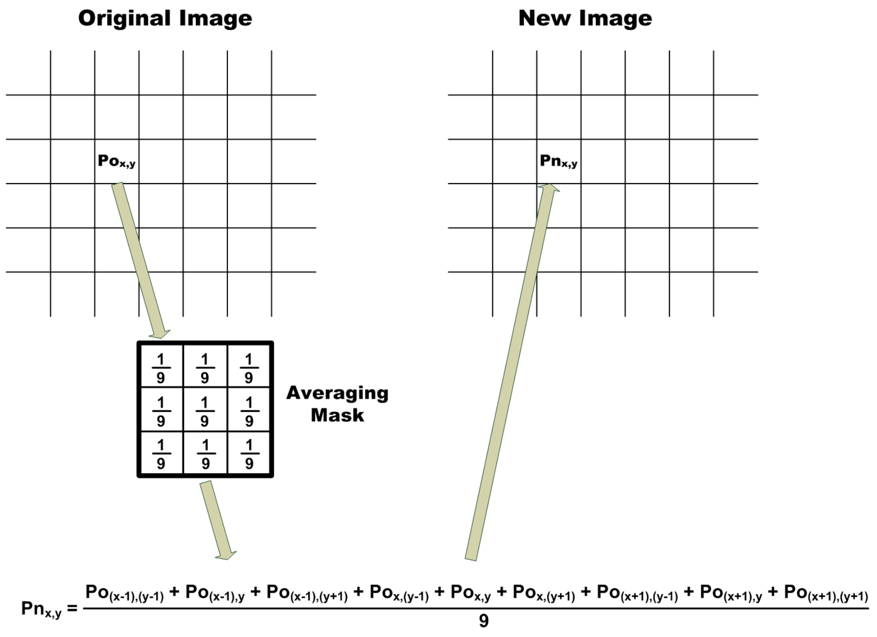
The MACROS Framework principle of operation is based on the interaction of a number of infrastructure and processing elements. In order to accurately verify the behavior of these interacting elements as well as obtain measurements of timing characteristics, two complete systems were built based on a test application. Because the emphasis is placed on high data rate stream processing, uncompressed video operations were selected as the target application. This application was then implemented as two MACROS Framework system variants, based on the two connectivity change procedures introduced in Chapters 4 and 5, and deployed using the KC705 platform.

These two systems are used to verify the capabilities of the MACROS Framework, thus ensuring that run-time ASP generation can be successfully undertaken. In addition, the systems are used to measure and analyze the timing behavior of the distributed control architecture being used. The configuration time, as well as the assembly and integration times are measured. Furthermore, general timing characteristics are extrapolated based on the observed behavior of these systems, which show how the framework scales with system size.

### 7.3.1 Experimental Application

The application being implemented is based on the concept of spatial color masking operations [175], which describes a set of operations performed on static images or frames of a video stream. These operations can achieve a number of effects, such as edge extraction, smoothing or sharpening by using neighborhood operations; using such operations, pixels in a frame are altered based on the values of their neighbors, through convolution with an operation matrix. An example of this process is shown in Figure 7.2, for a $3 \times 3$ averaging operation which results in a reduction of high frequency image elements [175]; this process, when applied to all three color channels of an image, will result in color masking.

An application was constructed based on the spatial color masks described above. This application consist of applying a number of color masks to a generated video stream and displaying said stream using a video interface. The main application tasks are shown in Figure 7.3, and consist of video pattern generation, a variable number of spatial color masks and, finally, video display. The video pattern generated takes the form of a 720p60 video stream [157] with a data rate of 1.15 Gb/s. For the test application, 4 mask types were selected, as well as a salt-and-pepper noise injection operation [175]. All mask types are $3 \times 3$ color masks, meaning 9-pixel neighborhoods are used; the mask library consists of averaging, median filtering, Sobel edge extraction and Laplacian (sharpening) [175]. Figure 7.4 shows the convolution matrices used for the sharpening, averaging and edge extraction masks. The median mask returns the median of a 9-pixel neighborhood; the noise injector selects pixels based on a counter value and alternatively sets all color values to 0x00 or 0xFF.



Original Image
New Image
$Po_{x,y}$
$Pn_{x,y}$
1/9
Averaging Mask
$Pn_{x,y} = \frac{Po_{(x-1),(y-1)} + Po_{(x-1),y} + Po_{(x-1),(y+1)} + Po_{x,(y-1)} + Po_{x,y} + Po_{x,(y+1)} + Po_{(x+1),(y-1)} + Po_{(x+1),y} + Po_{(x+1),(y+1)}}{9}$

Figure 7.2: Spatial Masking Procedure

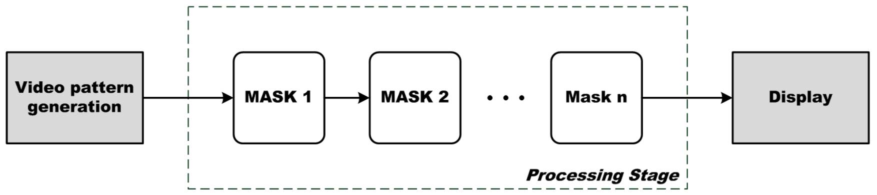


Figure 7.3: Video Test Application

For the purposes of the experiment, 10 different modes were defined and used to test the run-time architecture adaptation procedures, as shown in Figure 7.5. An additional 3 modes were defined for the purposes of testing fault mitigation in MACROS Framework systems, and are shown in Figure 7.6; these modes describe three ASP variations, but both the Sobel edge extractor and Median mask can be relocated.

### 7.3.2 Experimental Architecture

The experimental application was transformed into two systems based on the MACROS Framework; the two systems were differentiated by the connectivity change process implemented, and by the intended testing procedures. One system was implemented using the Minimal LCCU architecture and was oriented towards fault mitigation tests. The other system was implemented using the Canonical LCCU architecture, and was oriented towards run-time architecture adaptation tests.

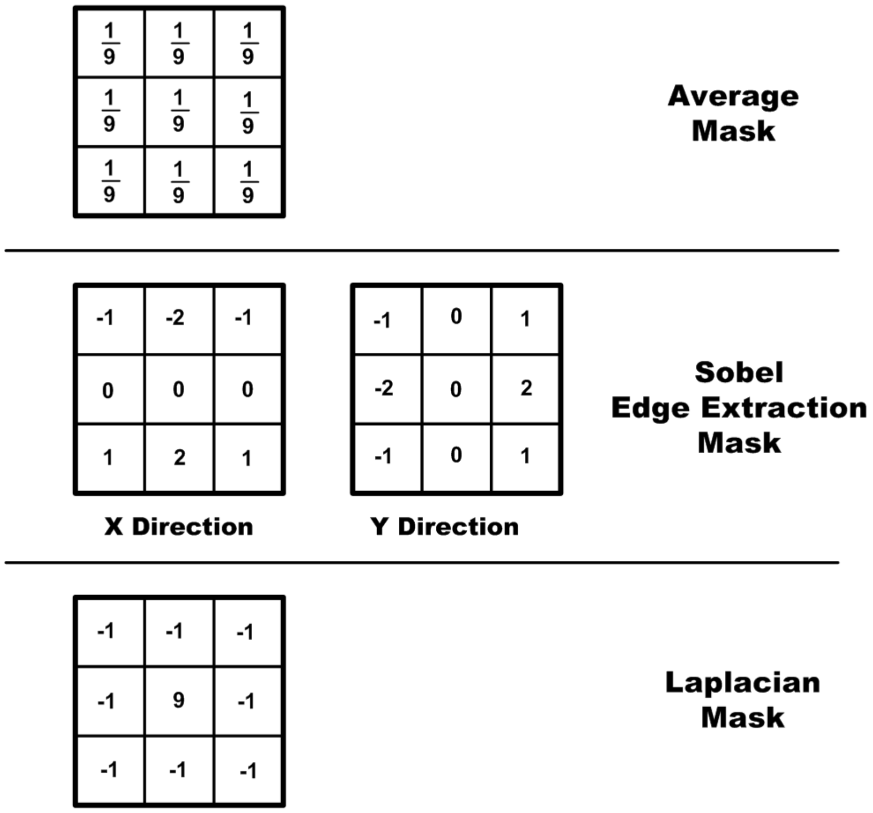


Figure 7.4: Application Spatial Masks

Both systems were implemented using the KC705 development platform and the XC7K325T FPGA. The systems share the same basic architecture, shown in Figure 7.7; the basic architecture consists of three static CMFUs, a number of system slots and the Distributed Communication and Control Infrastructures; each system slot was sized to contain 300 slices and 10 RAM blocks. A host computer acts as system BCM; for the purposes of functionality testing, all configuration operations were performed using the JTAG configuration interface [49]. This is where partial bit-streams for the system CMFU library reside. Mode information is received from a set of on-board switches. The Mode-Mapper and its connection to the BCM are omitted from this experiment, due to the fact that the Mode-Mapper presents little uncertainty in terms of behavior and implementation characteristics.

The system contains three static CMFUs; these CMFUs are always present in the system, since the functionality they offer is required for every mode. The video pattern generator CMFU is responsible for the creation of the test video stream being processed. The display unit CMFU is responsible for acting as a video interface to an on-board High-Definition Multimedia Interface

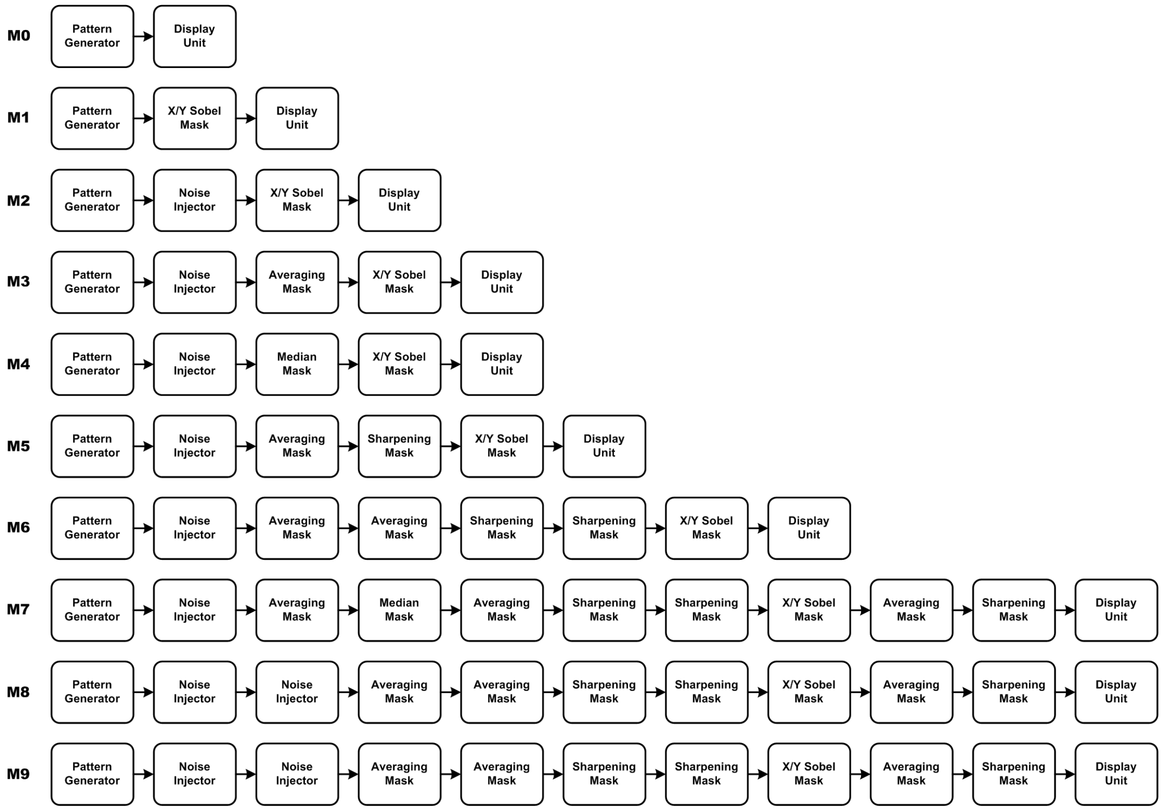


Figure 7.5: Modes for Run-time Architecture Adaptation Tests

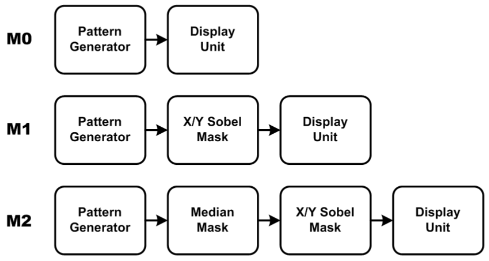


Figure 7.6: Modes for Run-time Fault Mitigation Tests

(HDMI) transmitter. Finally, due to the use of the HDMI interface and the board architecture, video data must be converted to a 4:2:2 format [176]; this task is accomplished by the color space converter CMFU.

The system display unit does not include a full frame buffer for display purposes; only one line of video data is buffered prior to being sent to the HDMI transmitter. To accommodate this behavior, all system CMFU operate on the full video frame, including blanking sections of the frame. Because of this, the safe period for all masks is equal to the blanking period of the last line in each frame [157], which is equal to 3.233 $\mu s$ (including mask latency). As part of this implementation, the two-

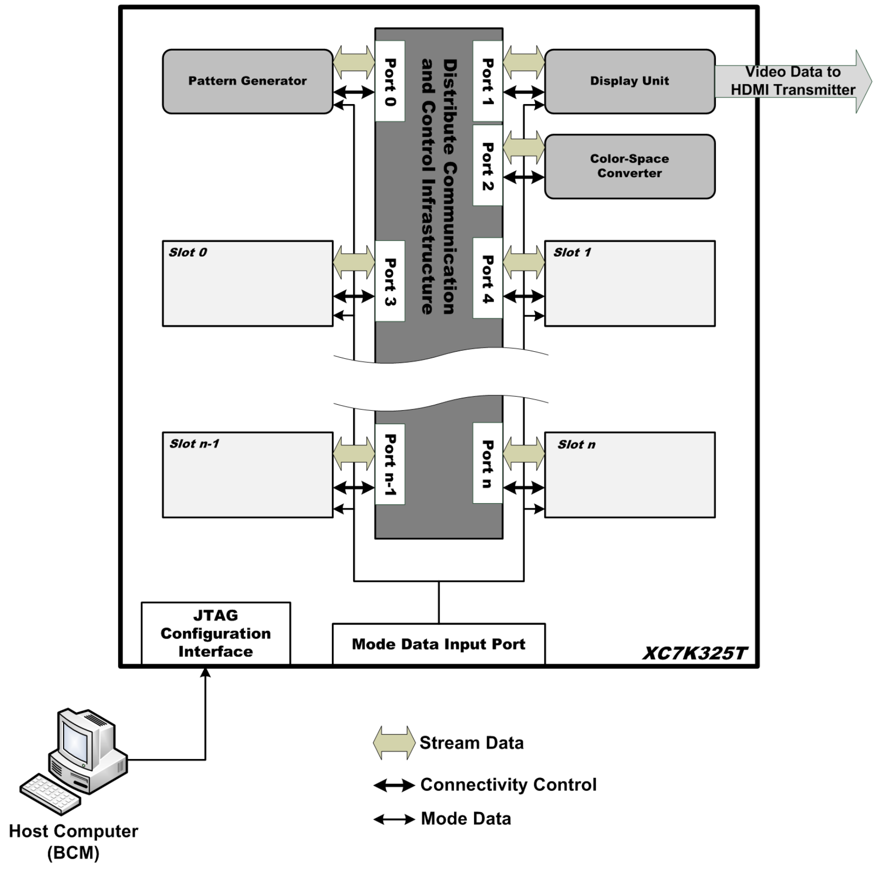


Figure 7.7: MACROS Framework Test Architecture

line latency associated with a mask (necessary in order for the construction of $9 \times 9$ neighborhood) is ignored.

The system using the Canonical LCCU architecture makes use of individual parallel links for the control data broadcast network. This system was implemented to contain 9 system slots, and is aimed towards mode-based architecture adaptation analysis; the arrangement of slots on the device is shown in Figure 7.8. For such a system, the total number of potentially supported ASPs ($N_{ASP}$) is listed in the equation below, where $N_{CMFUType}$ represents the number of CMFU types in the system and $N_{slots}$ represents the number of slots in the system:

$$N_{ASP} = (N_{CMFUType} + 1)^{N}_{slots} - (N_{CMFUType} \times (N_{slots} - 1)) \tag{7.1}$$

In the above equation, the number of CMFUs is expanded by one to account for blank CMFUs used in the system; the latter part of the equation accounts for systems which contain only one

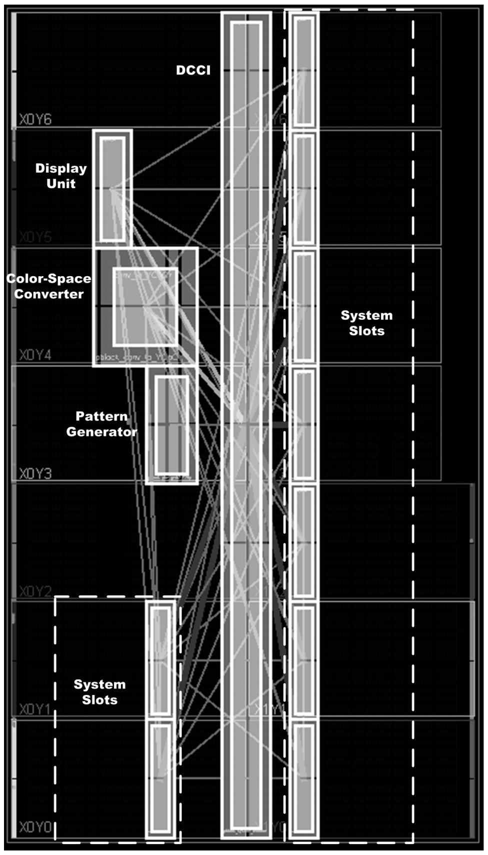


Figure 7.8: System On-Chip Layout for Test System Using Canonical LCCUs

CMFU mapped to different slots. Given the application being implemented, the system can create up to 10,077,656 ASP combinations. However, for the purposes of the experiment, only the 9 modes described in the previous section are encoded into the CMFUs.

The system using the Minimal LCCU architecture uses a control data broadcast network consisting of a single aggregation element. This system was implemented to contain only 4 system slots, with the aim of performing fault mitigation tests. In this system, all elements of the system are implemented as PRMs: static and dynamic CMFUs, LCCUs and the crossbar, control data broadcast network and mode broadcast bus. The arrangement of PRRs (including system slots) is shown in Figure 7.9. The CMFUs deployed to this system were encoded with the 3 fault mitigation modes described above.

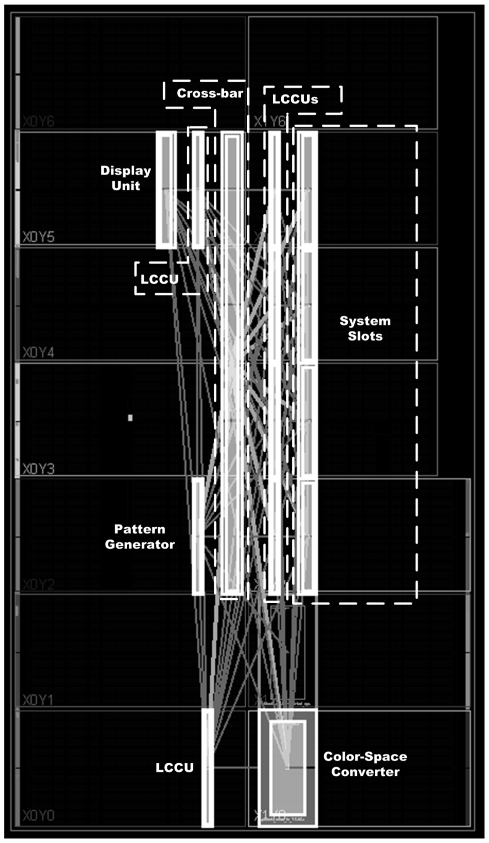


Figure 7.9: System On-Chip Layout for Test System Using Minimal LCCUs

## 7.4 Experiment: Monolithic System Implementation Effort Measurement

One of the main aims of the MACROS Framework is to allow modular functionality implementation in streaming systems, with the aim of reducing the implementation effort for these types of systems. An experiment was constructed, with the aim of A) determining how implementation time increases in relation to system size (for both monolithic and modular implementation), and B) comparing how modular implementation compares with monolithic implementation for various sized systems. The first part of the experiment consisted of collecting the implementation times for the two MACRO Systems described above, as reported by the NGDBuild, MAP and PAR programs [31].

In order to obtain implementation times for traditional (monolithic) systems, a number of static systems were constructed and implemented. These systems were based on an extrapolation of the MACROS Framework systems described in Section 7.3.2 and aim to match the functional

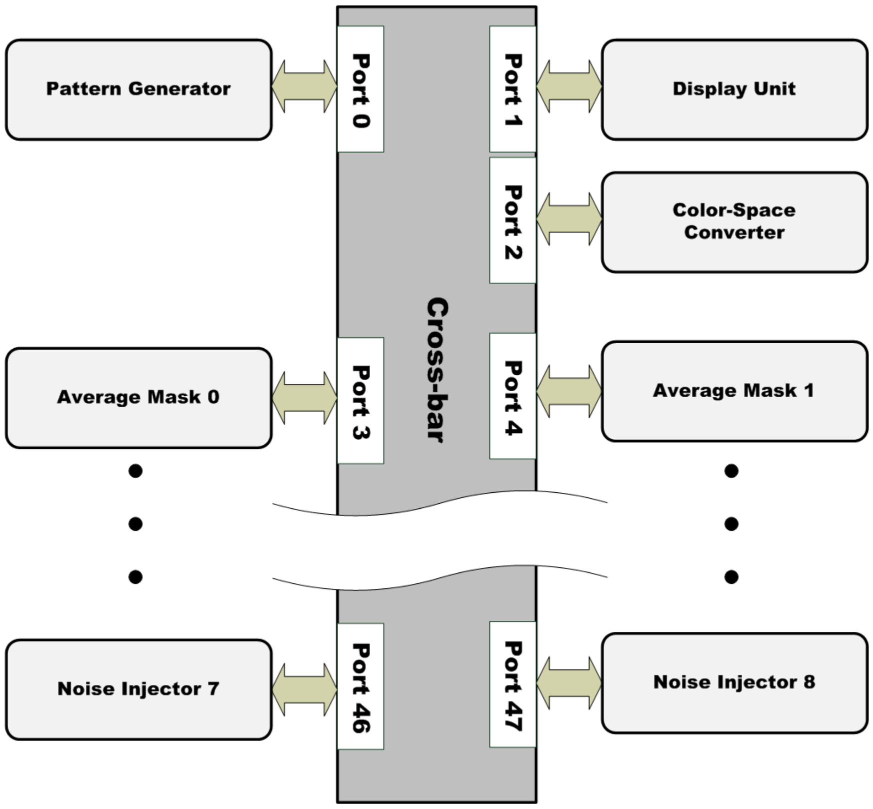


Figure 7.10: Static Test System

capabilities of the implemented MACROS Framework test processors. As already established, the 9-slot system described above can implement up to 10,077,656 modes thanks to various combinations of components in system slots. To accomplish the same functionality, a static system must contain 9 copies of each module, and a 48-port crossbar which allows any combination of component arrangements. This basic system is shown in Figure 7.10.

In order to determine the implementation cost associated with larger designs, three additional static systems were created by replicating the system shown in Figure 7.10 2, 3 and 4 times, respectively. This increase would be equivalent to an increase in the potential functionality of the system, and would allow for the generation and analysis of large systems. These systems were mapped to three different devices - XC7K160T, XC7K325T, XC7V980T [10] - and in each case the implementation time was measured. All implementation tasks were conducted on a computer using an Intel Core 2 Quad Q8400 CPU operating at 2.66 GHz, 4 GB of RAM and the Windows 7 64-bit operating system.

A number of CAD systems aimed at System on Programmable Chip (SoPC) development support Hierarchical Design [177], which divides a system into *partitions*; once implemented, partitions can be reused so long as they remain unchanged. This process will not reduce the initial implementation effort associated with a given system, but it can reduce subsequent implementations. This process can be used for off-line mode-specific ASP generation, whereby an initial version of the system is built using full implementation effort, and subsequent versions of the ASP are implemented using Hierarchical Design.

To determine the efficiency of this approach, static versions of the MACROS processors described above were implemented. These static systems were built using the MACROS Framework infrastructure, but all system slots were populated with a CMFU; these systems are not functionally equivalent to the MACROS processors, given that all slots hold static CMFUs. To support multiple operating modes, multiple versions of these systems must be implemented, where only portions of the design change; in this way, the Hierarchical Design process supported by the Xilinx PlanAhead 14.7 CAD was tested to determine what type of time cost savings could be obtained.

## 7.5 Experiment: Frameworks Resource Cost Measurements

The MACROS Framework offers a specific functionality (run-time architecture adaptation in stream processors) via a prescribed general architecture and behavior. The architecture imposes resource overhead onto the pure functionality of the system IP Cores, in the form of Co-Op unit, DCCI and to a lesser extent the BCM. By implementing the above test systems, this overhead can be measured for these system instances. However, the proposed framework aims to be applied to a whole class of systems; thus, more general resource usage information had to be obtained.

A number of measurements were made to determine the resource cost of the various system building blocks as the system size (in terms of number of system slots) changes. To obtain these measurements, multiple variants of the two LCCU architectures, the control data broadcast network and the system crossbar were designed; each variant was aimed at a specific system size. All variants were then implemented, and resource costs for each were recorded, as reported by the MAP tool [31]. Thus, the variance in overhead in relation to the number of system slots could be determined.

## 7.6 Experiment: Central Versus Distributed Control System Analysis

The MACROS Framework implements both its assembly and integration mechanism using distributed control architectures, embedded in Co-Op units and LCCUs. As Chapter 4 discussed, these mechanisms can also be implemented using a central approach. In order to validate the adoption of a distributed architecture as part of the MACROS Framework, comparisons must be made to central implementations of these mechanisms. Of particular interest are the resource overhead associated with each approach, as well as the achievable performance.

Central control mechanisms can be implemented using either dedicated circuitry, or a programmable, instruction-based sequential processor. In order to compare dedicated central control systems with distributed ones, the problem of scheduling was used as a basis for this experiment. Two types of scheduling circuits were built for a simple system model; the system in question consists of a sequence of functional units connected in sequence, and forming a pipeline. The job of the scheduling circuits is to activate the processing pipelines of each functional unit in sequence. In the case of the central scheduling circuit, this behavior consists of waiting for the correct combination of input control data (referred to as input indicators in the diagram), trigger functional units in sequence (using a counter to account for the latency of each functional unit) and generate output control data (referred to as an output indicator), to be used by downstream processors. The distributed scheduling waits for the correct input indicator, triggers the local pipeline, waits a fixed time period and generates an output indicators (to be used by other scheduling circuits down-stream). In both cases, the indicator signals and pipeline control signals (triggers) where assumed to be 1-bit wide for simplicity. The behaviors of the two control circuits are shown in Figure 7.11.

Multiple versions of the central scheduling circuit were implemented, for systems of different sizes; the distributed scheduling circuit was also implemented. In each case, the circuits being considered were implemented on a Xilinx XC4VLX160 device [7]; the results being considered are the logic resource requirements of the circuits (characterized by the number of 4-input look-up tables used to implement them) and the reported minimum clock periods the circuits can support.

The central scheduling circuit was further altered from the base behavior shown in Figure 7.11,

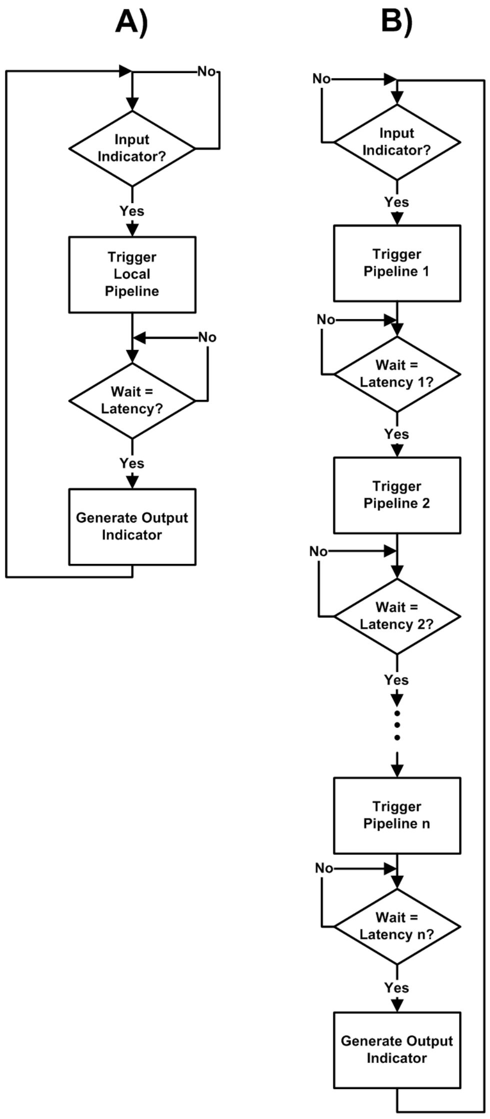


Figure 7.11: Test Circuit Behavior for Central and Distributed Schedulers

to take into account multi-mode operation, and the fact that one or more of the functional units may or may not be present in the system for each mode. For each functional unit, the scheduler must check a record indicating if the functional unit is present and it should be triggered or if it is absent and skipped (the record is assumed to be an input, and is not modeled). No alterations were needed to the distributed scheduling circuit, as its behavior does not change in multi-mode situations; if the functional unit (and scheduler) are present, then the schedule will take place.

As establish in Chapter 2, the most popular method of control implementation is the use of a central, instruction-based processor; the MicroBlaze soft-core processor [164] is often adopted for this task. In order to compare the proposed distributed framework with this type of central control

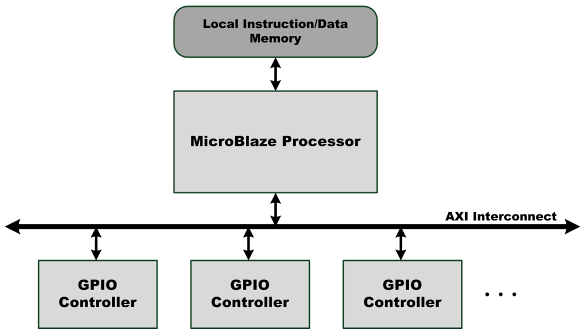


Figure 7.12: Single Instruction-Based Processor Test System

solution, a number of test systems were developed and used to obtain performance and resource data, using both the MicroBlaze processor as well as one of the two ARM Cortex A9 cores present in the Xilinx Zynq XC7Z020 device [178]. The basic architecture of the MicroBlaze systems is shown in Figure 7.12: the system consists of the processor proper, an 8 KB local memory and a number of General-Purpose Input-Output (GPIO) interface modules, connected to the processor via the Advanced eXtensible Interface (AXI) interconnect [179]; the system also contains an Integrated Logic Analyzer (ILA) module used for timing measurements. The processor and all sub-systems are operating using a 100 MHz clock frequency.

This system was used to determine the resource cost associated with this solution (as reported by the CAD flow), as well as the achievable performance. In order to determine the capabilities of the MicroBlaze processor to act as a control system, it was programmed to issue write commands to the included GPIO interfaces; the rate at which data could be written to these interfaces was measured using the integrated ILA module.

A second, extended version of the system was also implemented, containing two processors, as shown in Figure 7.13; this duplex system was implemented to determine the resource cost associated with a more fault-tolerant version of the control system. Once again, the resource cost was obtained from the reports generated by the MAP tool.

The Zynq test systems consisted of one of the ARM Cortex A9 cores present in the Zynq Processing System [178] executing test code, augmented with a timing counter, deployed in the Programmable Logic [178] section of the device; the ARM processor used an operating frequency of 667 MHz, while the timer used a frequency of 100 MHz. Using this approach, execution times for various operations could be measured and reported to a host computer via a serial interface.

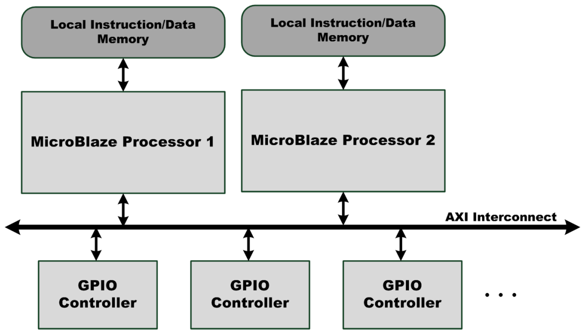


Figure 7.13: Duplex Instruction-Based Processor Test System

A combination of basic memory read, write and logic operations were used to simulate the type of control operations associated with system assembly.

## 7.7 Experiment: Communication Architecture Performance Measurements

As discussed in Chapter 4, the MACROS Framework relies on the use of a fully-connected crossbar as the medium of communication between functional units. This solution offers non-blocking communications (a highly desirable feature in pipe-lined and data-flow systems) as well as ease of re-programming in response to run-time architecture changes. However, as stated in Chapter 4, fully connected crossbars can result in a large resource overhead in large systems. A number of alternative architectures have been proposed and used in on-chip and reconfigurable systems; however, the closest competitor, in terms of achievable performance is the Network-on-Chip.

A number of existing research works have proposed the use of NoC as potential communication architectures for high performance partially reconfigurable systems (see Chapter 2). To determine the capabilities and cost of the NoC approach in comparison to the crossbar, a comparison was conducted. The NoC architectures used for the comparison were the 2D Mesh NoC presented in [92], the SoC Wire NoC [88, 89] and the Star-Wheels NoC [95]. These NoC architectures were chosen due to the availability of information regarding their resource cost and performance in scientific publications. For comparison purposes, 7 and 12-port versions of the crossbar (used in the two systems introduced above) were implemented using the same device type reported in these resource works (the Virtex 5 FPGA), and resource costs were obtained.

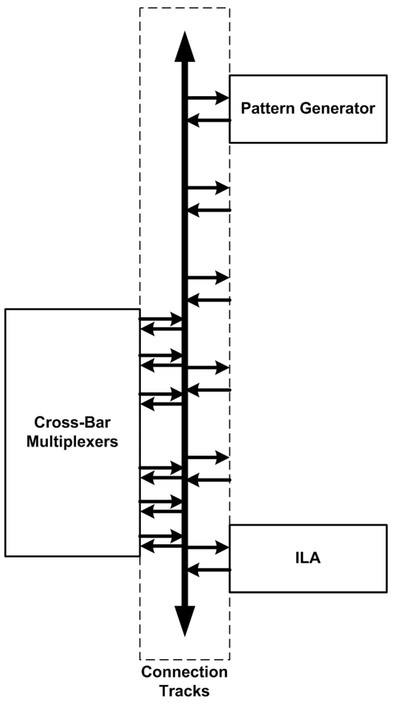


Figure 7.14: Crossbar Performance Test System

In addition, as discussed in Chapter 4, signal delay variations can be converted into pipeline latency variations. The aim of this approach is to decouple the system clock frequency from the on-chip distribution of slots, and allow higher operating frequencies to be achieved. This approach relies on the controlled placement of registers on-chip, such that wire delays are kept under a target value. This placement can be done via manual placement and routing, for instances where high clock periods are needed.

A test system was built to test the feasibility of this approach, as well as the maximum operating frequency achievable. The architecture of the test system is shown in Figure 7.14, and was implemented using the XC4VLX160 FPGA present on the MARS platform. The system consists of a crossbar, a pattern generator connected to one input port, and an ILA core connected to the furthest output port; the crossbar is divided into a multiplexer block (where connectivity control is implemented) and a set of connection tracks, used to connect the block to slots in the system. An on-chip Digital Clock Manager (DCM) was used to control the system clock; multiple versions of the system were implemented, using different system clock frequencies. For each frequency, the ILA was used to determine if data was received correctly or not; in this way, the maximum supported frequency was determined.

## 7.8 Summary

This chapter has described the experiments conducted to validate the framework being proposed; these experiments have three aims. The first, and most fundamental is verification of the functionality offered by the framework being proposed; thus, systems were implemented based on the MACROS Framework, and their capability to adapt their architecture at run-time (including CMFU relocation) was verified. The second aim of these experiments was to determine the performance and resource characteristics of the framework, via various measurements. Finally, based on these measurements, the chosen mechanisms and implementation methods were compared with alternative existing approaches in order to validate or repudiate their selection.

# Chapter 8

# Results and Analysis

## 8.1 Introduction

This chapter represents the final stage in the description of the MACROS Framework. Previous chapters have introduced the basic principle of operation of MACROS Framework systems, the mechanisms that support this principle, and the architecture and behavior which implement these mechanisms. Finally, this chapter validates the decisions made as to the various mechanisms used and their implementation. Chapter 7 described the experiments used to verify the functionality of the MACROS Framework, determine its characteristics and perform comparisons to other existing approaches to system design. This chapter presents the results obtained in said experiments and their analysis.

The presented results are organized into sections, each addressing a specific aspect of the framework. Section 8.2 describes the validation process for the MACROS framework. Once it has been established that the framework can implement run-time architecture adaptation, Section 8.3 presents the timing characteristics of this process. Section 8.4 validates the use of a distributed control architecture by providing an analysis of the performance and resource cost of the proposed approach, as well as a comparison of distributed and central control architectures.

Section 8.5 presents an analysis of the MACROS Framework communication architecture and compares it to recent NoC implementations to determine its suitability in stream processing reconfigurable systems. Section 8.6 presents an analysis of implementation times, which validates the assertion that the MACROS framework offers modular implementation capabilities and a concomitant reduction in implementation cost. Given that one of the main goals of the framework is to provide improved fault mitigation capabilities in stream processing systems, Section 8.7 analyzes the fault mitigation capabilities of the system, and compares them to existing approaches. Finally, Section 8.8 presents a complete analysis between existing system architectures and the proposed MACROS Framework architecture and design method.

## 8.2 MACROS Framework Verification

To validate the methods incorporated in the MACROS Framework, two video processors were devised as experimental instruments and were presented in Chapter 7. Using these two systems, the principles of operation of the framework were verified. Static initial versions of the systems were configured into the test devices, and the correct interaction between CMFUs and the DCCI were observed, as evidenced by the observed video output. Following this, additional CMFUs were deployed into system slots via partial reconfiguration; once again, the correct interaction was observed between CMFUs and DCCI as various mode values were sent to the system.

Secondly, relocation capabilities were verified in both systems; in each case, the mode was changed to omit one or more CMFUs, after which these CMFUs were relocated (via partial bitstream configuration targeting a different system slot), and the mode reverted back. In both cases, the original system functionality (as observed prior to relocation) was regained. Finally, scrubbing operations were undertaken for individual LCCUs and the system crossbar using the smaller of the two systems, which was designed for fault mitigation tests; once again, correct operation was resumed after these operations.

These initial experiments, while not very quantitative in nature, are, nonetheless, fundamental, as they establish the the two main features of MACROS Framework systems: run-time architecture adaptation via CMFU deployment, self assembly and self-integration and fault mitigation via CMFU relocation, self-assembly and self-integration. Whatever other results may be obtained, this first experiment ensures that the MACROS Framework, as proposed, is functionally correct.

## 8.3 Multi-Mode Run-Time Architecture Adaptation Timing Analysis

Having functionally verified the MACROS framework, more detailed timing parameters can be obtained. These timing parameters would be necessary to characterize any framework but are of particular importance to the MACROS Framework, which is aimed explicitly at systems with strict timing requirements. Thus, a detailed timing analysis of the run-time architecture adaptation process is presented in this section. The analysis concludes with a derivation of the stream interrupt factor $F_{si}$ for the two test video processors, as a means of showing the capabilities of the framework.

As established in Chapter 3, run-time architecture adaptation procedures (i.e. the generation of mode-specific ASPs) when transitioning from mode i to mode j consists of three timing costs: the time needed to select the architecture to implement $Tsel_{i-j}$, the configuration time proper $Tconf_{i-j}$, and the assembly and integration time cost $Tai_{i-j}$. $Tsel_{i-j}$ is the result of the architecture decoding and allocation process in the BCM; it is omitted from the current analysis, as the BCM was implemented using a host computer. $Tconf_{i-j}$ is the result of configuration activities performed by the BCM. Finally, $Tai_{i-j}$ is the result of the LCCU-based control system which handles link establishment via connectivity change procedures. As discussed in Chapters 4 and 5, self integration of CMFUs happens automatically as control data is received by the Co-Op Unit; because of this, in the current analysis $Tai_{i-j}$ does not contain an integration component.

The configuration time is derived based on the configuration interface being used, and the size of the bit-stream being configured. Table 8.5 lists bit-stream sizes for both full and partial bit-streams, as well as configuration times for both the JTAG interface at 10 MHz, and the SelectMAP 32 interface operating at its maximum supported speed, 100 MHz; the configuration times are computed based on the data rate of the interface and the bit-stream size.

Table 8.1: Bit-Stream Sizes Configuration Times

| | Size (KB) | JTAG Configuration (s) | SelectMAP32 Configuration (ms) |
|---|---:|---:|---:|
| Full Bit-stream | 11,176 | 9.154 | 28.6 |
| Slot Partial Bit-stream | 366 | 0.299 | 0.94 |

The assembly time cost $Tai_{i-j}$ was obtained for each LCCU type using direct measurement via ILA modules. Figures 8.1 and 8.2 show the assembly procedures for the Canonical and Minimal

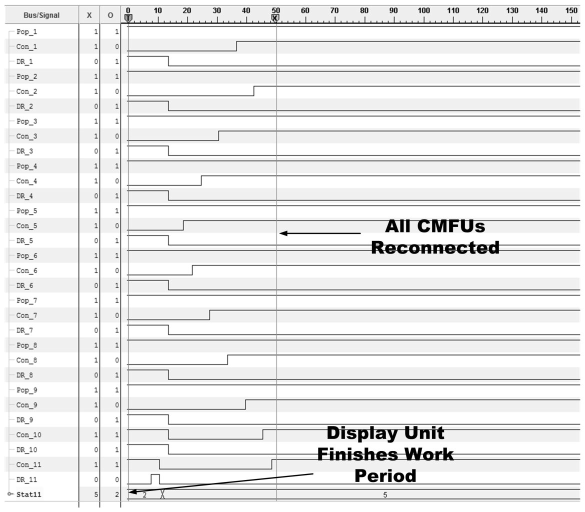


Figure 8.1: Canonical Connectivity Change Procedure Timing Results

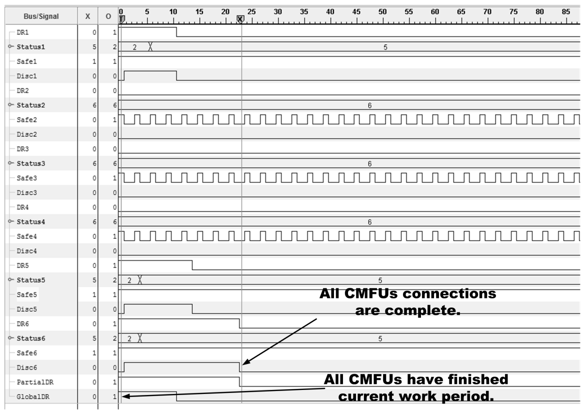


Figure 8.2: Minimal Connectivity Change Procedure Timing Results

connectivity change processes, respectively, excluding configuration times. The complete connectivity change process required 51 clock cycles for the Canonical system, and 25 clock cycles for the Minimal system; given that the clock used is the pixel clock of the 720p video stream (74.25 MHz), this leads to connectivity change procedures of 686.86 ns and 336.7 ns for the Canonical and Minimal processes, respectively.

Based on the measured values, complete ASP generation times $Tagen_{i-j}$ can be computed. These times will always depend on the number of missing CMFUs that must be configured; thus a best- and worst-case $Tagen_{i-j}$ can be derived for a system. Table 8.2 lists $Tagen_{i-j}$ values for situations where one CMFU is missing from the system, as well as the worst-case scenario, where all system slots are empty and must be filled; the listed values assume the use of the SelectMAP

32 interface for configuration purposes. As can be seen, the time costs are dominated by the configuration time for CMFUs and thus almost identical for the two methods when only one CMFU is missing.

Table 8.2: ASP Generation Cost Values for the Canonical and Minimal System

| | Canonical System | Minimal System |
|---|---:|---:|
| $Tagen_{oneCMFUmissing}$ (ms) | 0.9407 | 0.9404 |
| $Tagen_{allCMFUsmissing}$ (ms) | 8.4607 | 1.8804 |

Given these parameters, the stream interrupt factor $F_{si}$ can be computed for the presented video processor. The specifications for the 720p60 video stream dictate a 60 hz refresh rate, meaning 16.67 ms per frame; of this time, 0.667 ms is defined as the blanking period (no active information) [157]. Based on the timing information reported above, $F_{si}$ values were computed for both systems, and are shown in Table 8.3. Two versions of the stream interrupt factor were computed: one assumes the existence of spare system slots (and allows for the overlapping of processing and configuration activities) while the other does not. The worst-case $Tagen_{i-j}$ values reported above were used for these computations. The $T_w$ values were computed based on the worst-case latencies of these systems. The system safe period ($T_p - T_w$) is 3.233 $\mu s$, as discussed in Chapter 7; this value is the blanking period of one line, and is dictated by the chosen video standard [157]. As shown, both versions of the system have $F_{si} = 0$ if spare slots are available. If no spare slots are available, $F_{si} = 1$ regardless of the number of CMFUs that must be changed; this is due to the fact that all configuration activities for either system can fit within one video frame.

Table 8.3: Stream Interrupt Factors for the Canonical and Minimal System

| System | Canonical System | Minimal System |
|---|---:|---:|
| $T_p - T_w$ ($\mu s$) | 3.233 | 3.233 |
| $Tagen_{allCMFUsmissing}$ ($\mu s$) | 8,460.7 | 1,880.4 |
| $Tai_{allCMFUsmissing}$ ($\mu s$) | 0.687 | 0.337 |
| $F_{si}$ (spare slots) | 0 $((T_p - T_w) > Tai)$ | 0 $((T_p - T_w) > Tai)$ |
| $F_{si}$ (no spare slots) | 1 $(Tagen < T_p)$ | 1 $(Tagen < T_p)$ |

Given the presented analysis, it is clear that the implemented MACROS Framework systems can not only accommodate the regular operations of video processors, but can also implement seamless run-time architecture changes in said video processors. This is accomplished via the use of spare system slots, and rapid implementation of control operations for link establishment and self-integration of CMFUs.

## 8.4 Control System Performance and Resource Cost Analysis

The second aspect of the MACROS Framework that must undergo analysis is the distributed control infrastructure being proposed. This analysis must address both the timing characteristics of the infrastructure, as well as the performance cost. The performance of the control infrastructure as implemented in the two test systems was described in Section 8.3. The self-integration process occurs automatically at run-time, in the form of self-scheduling, and adds a 1 clock cycle delay to processing operations. The link establishment process is more involved, and imposes a cost of 52 and 25 clock cycles for the Canonical and Minimal connectivity change procedures, respectively; this translates into 686.86 ns and 336.7 ns, respectively.

The time cost of both connectivity change processes is dependent on system-specific parameters. In the case of the Canonical connectivity change process, the worst-case time cost for the complete connectivity change is dependent on the number of CMFUs that are affected at any one time; the smaller the number of CMFUs that must be disconnected, the faster the connectivity change process. In the case of the Minimal connectivity change process, all connectivity changes occur in parallel once all CMFUs are in their safe period (between work periods); however, this cost may be increased by delays associated with data traversing the control data broadcast network. In the current case, the aggregation element being used has an associated cycle time per data item sent; thus more data items lead to larger time costs. This problem can be eliminated by using fully parallel links, as described in Chapter 5.

Figures 8.3 shows the variations in time for the connectivity change procedures as the number of system slots changes; the presented data is derived from the measurements described in Section 8.3. The Canonical connectivity change time cost is based on the worst-case situation, where all CMFUs in the system must be disconnected and reconnected. For the Minimal connectivity change process, the increase in time is attributed to the use of aggregation architectures; if parallel links had been used, the time cost would remain constant at 8 clock cycles or 108 ns.

For the implemented systems, the resource cost for each control element is shown in Table 8.4; the table shows costs for the Co-Op Unit, LCCU variations and a system slot (for comparison

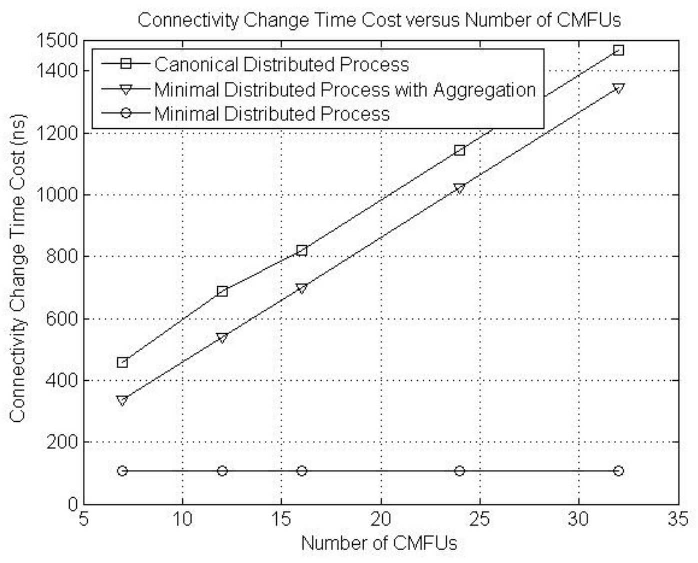


Figure 8.3: Connectivity Change Time Cost for Various System Sizes

purposes). Given that all slots were sized to hold all CMFUs, CMFU costs are not listed. All costs are presented in terms of Kintex 7 slices [32] and RAMB36 blocks [33].

Table 8.4: MACROS Test System Resource Cost

| Element | Slices | RAMB36 |
|---|---|---|
| Co-Op Unit | 14 | 0 |
| LCCU (Canonical) | 72 | 0 |
| LCCU (Minimal) | 36 | 0 |
| System Slot | 300 | 10 |

The MACROS Framework control overhead takes the form of Co-Op units, LCCUs, control data broadcast network and mode data broadcast bus. The mode broadcast bus consists of only a set of source registers and wires, and will not add any significant overhead to a system. The control data broadcast network will like-wise not add much in the way of resource overhead. If it is implemented as parallel links, the overhead will consist only of wires; if a central aggregation element is used, it will consist of one counter, a set of storage registers, one multiplexer and one control FSM, and will amount to no more than 110 slices for the largest system. The largest sources of overhead in the system are the Co-Op and LCCU units; these elements are added to the system in direct proportion to the number of slots present. Table 8.5 lists the control overhead (Co-Op unit, LCCUs and control data broadcast network) for both system types, as the number of system slots increases.

Table 8.5: Control Overhead in the MACROS Framework

| Number of CMFUs | Canonical System Overhead (slices) | Minimal System Overhead (slices) |
|---|---|---|
| 7 | 399 | 371 |
| 12 | 1032 | 632 |
| 16 | 1616 | 822 |
| 24 | 3408 | 1310 |
| 32 | 5664 | 1710 |

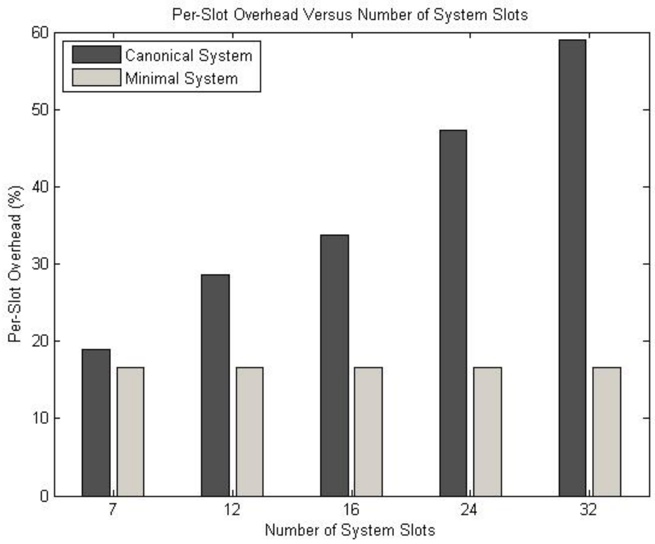
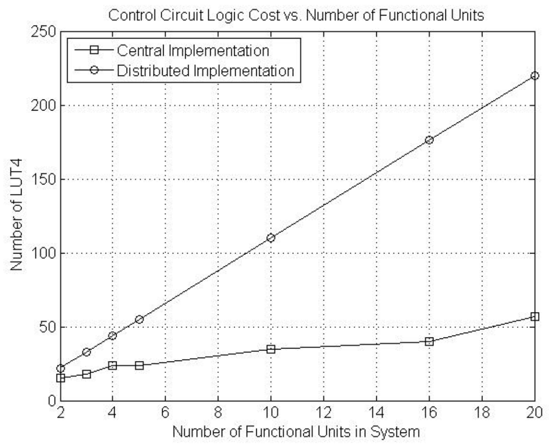


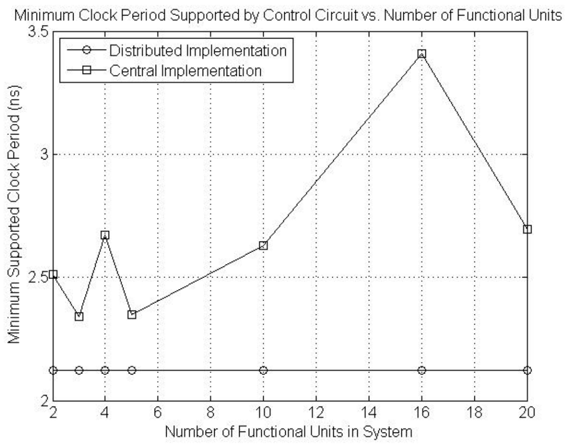
Figure 8.4: Per-slot Overhead in Different Sized Systems

As the table shows, the control overhead associated with the Canonical system increases more rapidly as the number of system slots is increased. This is due to the use of fully parallel decoders in the Canonical LCCU; this overhead can be reduced by implementing sequential decoder architectures, at the cost of reduced performance (slower decoding process). This overhead can also be treated on a per-slot basis, to determine what the expected overhead may be for other system sizes; Figure 8.4 lists the per-slot overhead as a percentage of one slot (300 slices). As expected, the Canonical system leads to larger per-slot overhead. These results further enforce the statement made in Chapter 4 and 5: if the conditions are present ($T_p > T_w$) then the minimal system should be used, as it leads to reduced system overhead.

The MACROS Framework control infrastructure implements part of the system assembly mechanism, as well as the totality of the integration mechanism. In addition to determining the performance and resource cost of the control infrastructure, said infrastructure must also be compared



(a) Logic Resources



(b) Combinational Delay

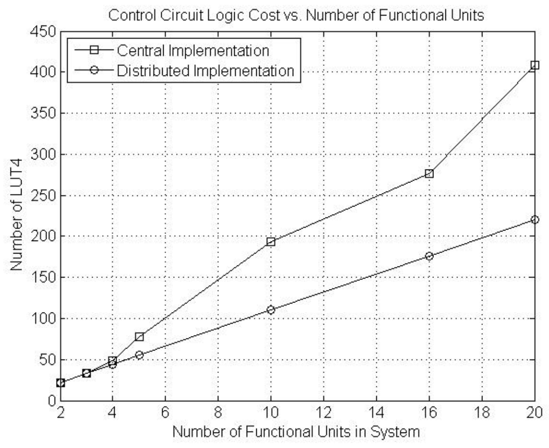
Figure 8.5: Comparison Between Central and Distributed Control

with alternative implementations, to determine the validity of the chosen architecture. The two potential alternatives to control implementation are using a central dedicated circuit or a central sequential, instruction-based processor.

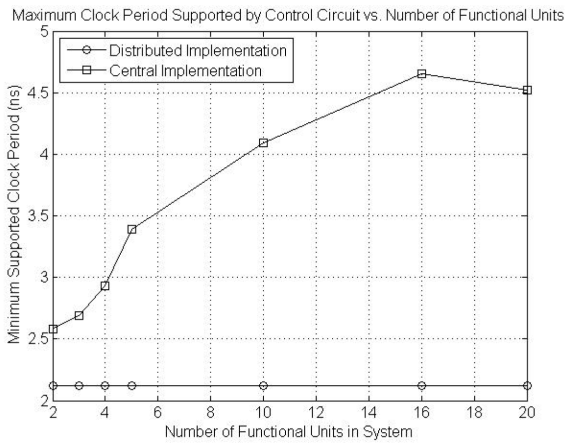
The distributed control system is first compared with central dedicated implementations; as described in Chapter 7, this comparison is based on implementations of scheduling circuits. The resource requirements for various systems sizes (meaning number of functional units) and worst-case combinational delay are listed in Figures 8.5(a) and 8.5(b), respectively. These figures show that the central circuit is uniformly more resource efficient for most system sizes; the distributed scheduling solution, however, supports a smaller minimum clock period (0.3 ns lower, on average).

The above results suggest that central scheduling can be more efficiently implemented using central architectures, as opposed to distributed ones. However, the current discussion is centered around multi-mode systems, where functional units may or may not be present in the system. Thus, Figures 8.6(a) and 8.6(b) list the resource and timing results for the two scheduling approaches, once multi-mode operation is considered; as stated in Chapter 7, the central scheduling circuit was altered to support this functionality. If multi-mode behavior is considered, the central approach becomes less efficient than the distributed approach when more than 3 components are being controlled; in addition, the minimum supported clock frequency increases more dramatically in the central circuit implementation.

To obtain a better understanding of why this drastic difference occurs, Figure 8.7 shows the



(a) Logic Resources



(b) Combinational Delay

Figure 8.6: Comparison Between Central and Distributed Control in Multi-Mode System

number of FSM transitions per control circuit implemented. The distributed solution stays constant, at 4 transitions; on the other hand, the number of transitions in the central scheduling circuit increases quickly with the number of functional units in the system. This increase is representative of the general increase in complexity in a control circuit when the complexity of the needed operations, and the number of elements that must be operated upon, increases. The associated increase in circuit size would be still larger in a real central scheduler implementation if functional unit distributions and latency change between modes had to be considered (in essence, the circuit shown in Figure 4.22 must be implemented).

The second approach to control system implementation is the use of a central, sequential instruction-based programmable processor. As already stated, a large number of research works rely on this approach to configuration, link establishment and scheduling, whether they be general computing systems [52, 51, 110, 75], high-performance computing systems [106, 80], or dedicated multimedia processors [76, 56, 77, 78, 104, 105, 99]. Thus, a comparison is conducted between these two approaches to system control design.

As described in Chapter 7, test systems were implemented to determine the characteristics of control solutions based on both the soft-core MicroBlaze processor and the ARM Cortex A9 core present in the Zynq SoC. Figure 8.8 shows the resource cost of the MicroBlaze system, as well as the cost for both canonical and minimal control method being proposed; the resource costs used are those reported in Table 8.5, consisting of Co-Op Units, LCCUs and control data broadcast

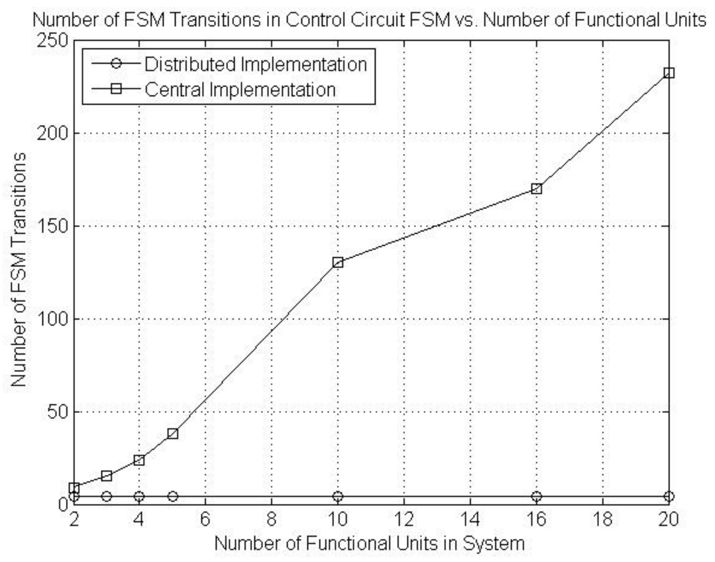


Figure 8.7: FSM State Transition Comparison Between Central and Distributed Control in Multi-Mode Systems

network. The MicroBlaze system cost shown also includes the cost of Co-Op units, based on the fact that the central control system cannot undertake both connectivity change procedures and scheduling simultaneously; the MicroBlaze-based control system is only responsible for connectivity changes. As the figure shows, the distributed canonical approach can be more costly than the central processor approach as the number of functional units in the system increases past 14 system slots. The distributed minimal system becomes more expensive as the number of system slots increases past 27. No resource cost is listed for the Zynq-based control solution, as this approach results in the use of a separate, additional device; thus, the cost of this approach is the device cost, plus all additional support circuits needed (power, decoupling networks, oscillators, etc.)

When considering fault-tolerant applications, the central control processor is a point of failure in the system and must be hardened in some way. The most basic such hardening method would be the use of a duplex system, which would allow for detection of faults in the control system. Figure 8.9 lists the resource overhead for a control system using two MicroBlaze processors, as well as the resource overhead for the two distributed MACROS framework control systems. In such an instance, the canonical distributed control method is less costly for systems up to 13 slots, while the minimal system is less costly up to 32 system slots. In the case of the Zynq solution, the ZC7Z020 device incorporates two ARM cores, so these can be treated as a duplex system, with no added

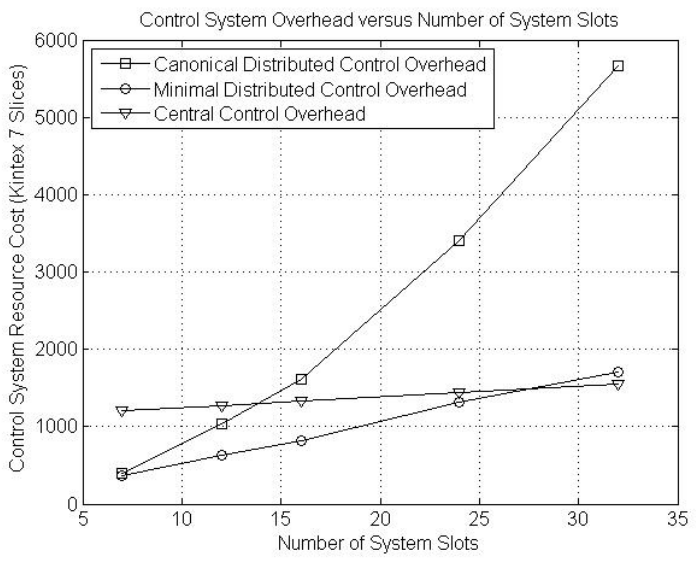


Figure 8.8: Control System Overhead Comparison for Various Sized Systems

cost; thus, a Zynq-based control system will have the same cost (the device itself) whether one or two processing cores are used.

These results are not surprising, given that instruction-based processors uses time multiplexing to offer control operations to all functional units using the same circuit; the two distributed control approaches use separate resources for each functional unit. However, to offer this reduced resource cost, the instruction-based processor solution must sacrifice performance. In the experiments conducted, it was assumed that the processors would perform connectivity change operations by writing control values to the crossbar control registers, *and nothing else.*

The MicroBlaze control system was found capable of writing one control value every 39 clock cycles of a 100 MHz clock. The ARM Coretex A9 Core in the Zynq device was capable of performing one control operation in 4.3 clock cycles of a 100 MHz clock. Based on these measurements, Figure 8.10 shows the worst-case time cost associated with connectivity change procedures for both distributed approaches (including two variants of control data broadcast network for the Minimal system) as well as the MicroBlaze and Zynq approaches, as the number of functional units controlled increases.

The MicroBlaze solution exhibits a much larger connectivity change time cost as compared with the distributed approach. Worst, however, is the observation that this time cost increases at a high rate in relation to system size. The Zynq solution has a time cost comparable to the Canonical

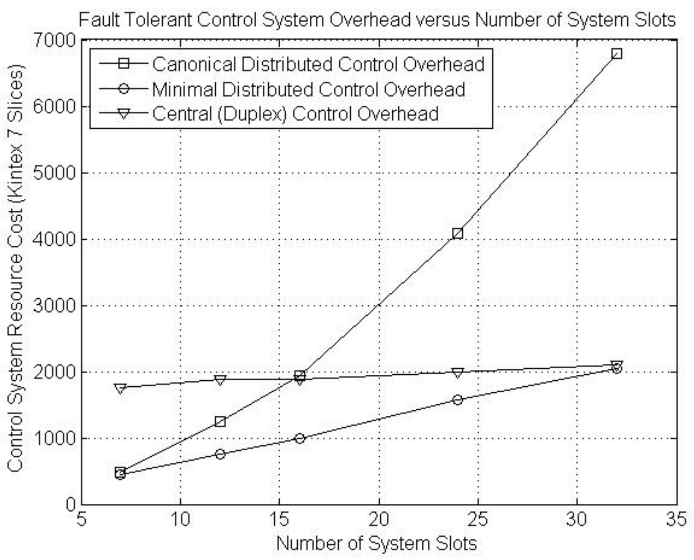


Figure 8.9: Fault Tolerant Control System Overhead Comparison for Various Sized Systems

Process; this performance is accomplished by an increase in operating frequency, from 100 MHz to 667 MHz. It should be noted that these results are somewhat idealized: here, both the MicroBlaze and ARM Cortex A9 processors are responsible only for writing control values from an array; array index computations are omitted and the highest compiler optimization levels are used. If more operations were required as part of the connectivity change process (which is to be expected) then the time cost would increase still further.

The time cost associated with system control imposes limitations on the type of systems that can be implemented; this observation applies to tasks such as scheduling, synchronization or assembly procedures. In the above example, the MicroBlaze-based central control mechanism exhibits a much larger $Tai_{i-j}$ parameter; given that seamless mode changes can only be implemented if $(T_p - T_w) > Tai_{i-j}$, an increase in the $Tai_{i-j}$ parameter requires an increase in $T_p$ (and thus a reduction in the stream data rate), or a reduction in $T_w$ (which may not be achievable). The same problem emerges if such a processor is used for scheduling operations; for this reason, a number of stream processing IP cores can self-activate based on stream control signals [167]. A potential solution to this problem would be to drastically increase the operating frequency of the central control system, thus reducing the $Tai_{i-j}$ parameter. This approach is shown in the above results, where the ARM processor present in the Zynq device can offer performance comparable to the distributed architecture adopted in the MACROS framework. This performance is obtained at a

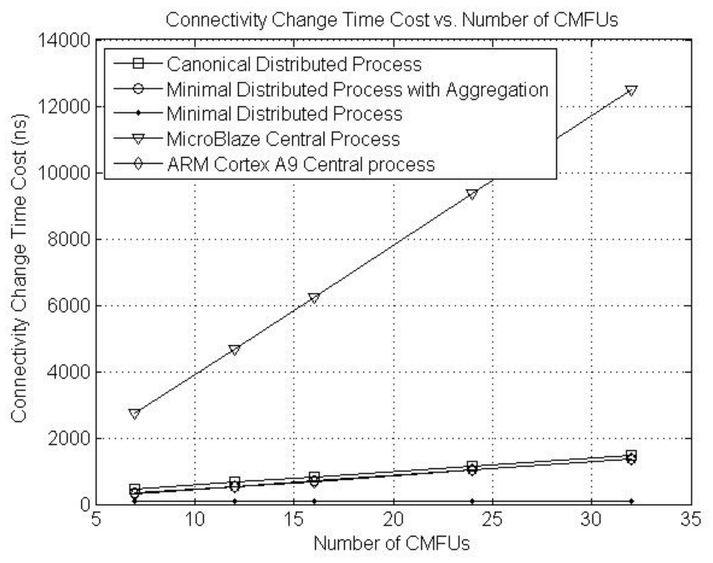


Figure 8.10: Connectivity Change Time Cost Comparison Between Central and Distributed Approaches

cost, however, since the ARM processor is operating at 667 MHz, whereas the MACROS control systems offer the same performance while operating at 74.25 MHz.

In deference to the impact that control operations can have on a system as the number of functional units increases, the MACROS Framework proposes a distributed control architecture for connectivity control and scheduling operations. The results presented in this section validate this design approach by showing that central control implementations can be either too costly, either in terms of area or power consumption, or limited in performance.

## 8.5 Communication Architecture Performance and Resource Cost Analysis

The MACROS Framework makes use of the fully-connected crossbar as its medium of communication; the main reason for this selection is the fact that the crossbar offers non-blocking communications, which are considered mandatory in pipe-lined and data-flow applications. Nonetheless, this choice must be validated and compared with available alternatives to determine its strengths and weaknesses.

As discussed in chapter 2 and 4, the crossbar resource cost is expected to grow in a non-linear fashion in relation to the number of ports it contains. Table 8.6 lists sizes for multiple crossbar implementations, in terms of Kintex 7 slices; the first two entries correspond to the crossbars used in the two video processing systems. The table also lists the device utilization for the XC7K325T devices (the one used for these experiments); this is done to show what percentage of the device is available for CMFU deployment when different number of slots are used.

Table 8.6: Crossbar Size in Relation to Number of Ports

| Number of Ports | Crossbar Resource Cost (slice) | Utilization (XC7K325T) (%) |
|---|---|---|
| 7 | 463 | 0.91 |
| 12 | 1040 | 2.05 |
| 16 | 968 | 1.90 |
| 24 | 5854 | 11.49 |
| 32 | 7852 | 15.42 |

The performance associated with the crossbar is based on the link width and clock rate used in the system. For the systems presented above, the per-port supported data rate is 1.79 Gbit/s; thus the 7 and 12-port crossbars support 12.53 and 21.48 Gbit/s, respectively. As discussed in Chapter 4, variations in CMFU deployment can lead to variations in the wire delay between various data ports; in the MACROS Framework, this variation is converted into a latency variation. In the presented video processors, the operating clock frequency (74.25 MHz) was low enough to permit the system to operate correctly using only input and output registration of crossbar ports. However, as discussed in Chapter 7, experiments were conducted to determine what performance could be obtained in an FPGA-based crossbar with controlled delay mitigation. The implemented 8-port crossbar, utilizing manual placement for part of its implementation, was capable of correct operation at clock frequencies of up to 425 MHz using a Virtex 4 VC4LX160 FPGA.

As stated in Chapter 7, the crossbar was compared with three existing NoC architectures: the 2D Mesh NoC presented in [92], the SoC Wire NoC [88, 89] and the Star-Wheels NoC [95]. Table 8.7 lists the communication capabilities and resource cost for the three NoCs and the fully connected crossbar; the numbers listed are those reported by the authors. All listed resource costs are based on the Virtex 5 family of devices; to obtain crossbar resource values, the 7 and 12 port crossbars from the two video processing systems described above where mapped to Virtex 5 architectures and implemented (the XC5VLX330 device was targeted). In the case of the SoC Wire NoC, the reported resource cost consisted of numbers of look-up tables and D flip-flops; these numbers were converted to Virtex 5 slices for comparison purposed. This conversion is not completely representative of the results obtained from a CAD system, but was undertaken so that some form of comparison can be performed. In each case, the tested operating frequency (as reported) is listed as well as the data-rate per port. In the case of the Star-Wheels NoC, the data rates are reported in terms of links between processing elements, and are based on the circuit-switched version of the system; the variation in data rate is based on the distance between the two processing elements (and thus the number of hops between them).

Table 8.7: NoC and Crossbar Comparison

| System | Element | Bit-width | Frequency (MHz) | Data-rate (Gbits/s) | Size (Slices) |
|---|---|---|---|---|---|
| 2D Mesh | Real-Time Router (48 element mesh) | 64 | 365 | 2.64 per port | 115 |
| SoC Wire | 4-port Switch | 32 | 170 | 5.06 per port | 635 |
| Star-Wheels NoC | Subswitch | 32 | 100 | 1.655 - 2.35 per link | 433 |
| | Superswitch | 32 | 100 | 1.655 - 2.35 per link | 1105 |
| | Rootswitch | 32 | 100 | 1.655 - 2.35 per link | 772 |
| Crossbar | 7-port | 26 | 74.25 | 1.79 per port | 317 |
| | 12-port | 26 | 74.25 | 1.79 per port | 616 |

The table shows that most presented communication architectures offer similar performance, in excess of 1.5 Gbit/s per port. What is immediately interesting is the fact that the crossbar, despite its reputation of leading to large system overhead, is no more costly than most presented switch and router architectures; even the 12 port crossbar is similar in size to SoC Wire 4-port switch, and smaller than both the Rootswitch and Superswitch in the Star-Wheels NoC.

The Real-Time router presented in [92] has the smallest resource foot-print, due to the simple time-division access scheme it implements. However, this approach requires two features to work. The first is that each functional unit must be able to accommodate the temporal distribution of data in the system (due to time-division multiplexing of multiple data streams). Secondly, to accommodate multiple high data-rate communication links, the operating frequency of the network must be higher than that of individual elements in the system. Given this discussion, each communication architecture can be analyzed in terms of the data-rate offered per MHz; Table 8.8 lists this parameter for each communication architecture. Here, the crossbar and the SoC Wire switch offer the best results, meaning they are best suited for high data rate systems. However, the SoC

Wire architecture is more costly than a regular crossbar.

Table 8.8: Data-Rate per MHz for Various Communication Architectures

| Architecture | Mbit/s/MHz |
|---|---|
| 2D Mesh | 0.115 |
| SoC Wire | 0.952 |
| Star-Wheels NoC | 0.752 |
| Crossbar | 0.949 |

The above analysis shows that the crossbar offers better performance while reducing the communication infrastructure overhead in the system, when compared with existing NoC architectures. At the same time, the crossbar offers guaranteed non-blocking communications, regardless of the distribution of functional units in the system; thus, the crossbar is guaranteed to offer the same level of performance regardless of deployment or relocation activities taking place in the system.

The above discussion dealt with the performance and cost of the compared architectures, and ignored any potential difficulties introduced by the dynamic nature of the proposed system types. As discussed in Chapter 4, both shared buses and crossbars would require additional support to track the presence of functional units in the system (similar to what the LCCUs and CMFUs accomplish in a MACROS System). The problem of link contention discussed in Chapter 4 is ignored here, as no data is provided for an analysis; nonetheless, such a problem is expected to occur in non-blocking architectures and must be addressed in order to be able to guarantee the data-rate needed by the application. Based on the assembled results, the crossbar emerges as the solution best-suited for high data-rate processing systems, and thus the MACROS Framework; this is further enforced by the crossbar's non-blocking capability which eliminates the problem of communication interference in dynamic systems.

## 8.6 Modular Implementation Effort Analysis

Having established that MACROS Systems can offer the desired functionality, a number of quantitative aspects of these systems must be investigated, starting with their implementation cost. The MACROS Framework was proposed in large part as an answer to the problem of increasing system and implementation complexity. This section analyzes the differences in implementation effort (expressed as time cost) between the two MACROS Framework systems and static systems with equivalent functionality.

For both MACROS video processors, implementation times were recorded, as reported by the CAD Tool-Chain used (PlanAhead 14.7). As described previously, the process used to generate a partial bit-stream for a given functional unit is to map its net-list to a given PRR and then translate, map and place-and-rout the net-list. In the case of MACROS Framework systems, each CMFU net-list is mapped to all system slots to obtain a bit-stream for the CMFU targeting each slot. In the used tool-chain, a given mapping of net-lists to PRRs is referred to as a *run* [35]; thus, for a given MACROS Framework system, one run is needed for each CMFU.

Tables 8.9 and 8.10 list the implementation times for all runs used to implement all CMFUs for the two systems; all data is listed in seconds. The Canonical system makes use of all five mask types. For most masks used, three versions of the CMFU are implemented, with different ID and connectivity specifications; in addition, a version of the system with all slots left blank (equivalent to Mode 0) is also implemented, leading to a total of 13 runs. The Minimal system only uses 2 of the five mask types, as the main emphasis of this system was characterization of the Minimal LCCU architecture and fault mitigation tests; thus only three runs are listed.

Table 8.9: Implementation Times for Canonical System

| Run | Translate Time (s) | MAP Time (s) | PAR Time (s) | Total (s) |
|---|---|---|---|---|
| Average 1 | 118 | 1,440 | 482 | 2,040 |
| Average 2 | 120 | 2,340 | 401 | 2,861 |
| Average 3 | 141 | 1,143 | 554 | 1,838 |
| Median 1 | 121 | 452 | 434 | 1,007 |
| Median 2 | 123 | 556 | 468 | 1,147 |
| Median 3 | 122 | 539 | 447 | 1,108 |
| Noise 1 | 80 | 292 | 353 | 725 |
| Noise 2 | 77 | 241 | 341 | 659 |
| Sharpening 1 | 137 | 602 | 448 | 1,187 |
| Sharpening 2 | 137 | 629 | 439 | 1,205 |
| Sharpening 3 | 137 | 624 | 439 | 1,200 |
| Sobel 1 | 127 | 598 | 423 | 1148 |
| Blanks | 69 | 610 | 336 | 1015 |

The implementation time per run is based on the complexity of the system being implemented, in terms of system slots and size of mapped net-lists. Systems with more slots will result in longer runs; like-wise, generating partial bit-streams for a larger net-list will result in longer runs. Thus, runs for the Canonical system are, on average, longer than runs for the Minimal system. At the

Table 8.10: Implementation Times for Minimal System

| Run | Translate Time (s) | MAP Time (s) | PAR Time (s) | Total (s) |
|---|---|---|---|---|
| Blanks | 21 | 154 | 497 | 672 |
| Median | 102 | 399 | 450 | 951 |
| Sobel | 106 | 989 | 419 | 1,514 |

same time, there are variations in runs for different CMFUs, based on the complexity (size) of the net-list being implemented.

Static system implementation times were obtained by implementing the systems described in Chapter 7; these systems will be referred to as Static 1X, 2X, 3X and 4X (depending on the number of times the base system was replicated). Table 8.11 lists the result of implementing Static 1X on three different FPGA devices. To determine the effect that constraints and net-list structure have on the implementation effort, Table 8.12 lists the implementation time for the 1X system using the XC7VX980T device and three different settings: A) no net-list hierarchy and no area constraints, B) enforced net-list hierarchy but no area constraints and C) enforced net-list hierarchy and area constraints. Finally, Table 8.13 lists the results of implementing Static 1X, 2X, 3X and 4X on the XC7VX980T device; for these tests, the systems were implemented using an enforced net-list hierarchy and no area constraints.

Table 8.11: Static 1X Implementation Time on Various FPGA Devices

| | XC7K160T | XC7K325T | XC7VX980T |
|---|---|---|---|
| Slice Capacity | 25,350 | 50,950 | 153,000 |
| RAMB36 Capacity | 325 | 445 | 1,500 |
| Slice Utilization (%) | 90 | 44 | 18 |
| RAMB36 Utilization (%) | 94 | 68 | 20 |
| Implementation Time (s) | 2,835 | 1,779 | 3,253 |

Table 8.12: Static 1X Implementation Time Using Constraints (XC7VX980T)

| Constraint | Implementation Time (s) |
|---|---|
| None | 45,899 (Failed) |
| Net-list Hierarchy | 3,268 |
| Net-list Hierarchy and Area Constraints | 63,219 |

By using three different devices of different size, three utilization levels were obtained. The results show that the higher utilization leads to a higher implementation effort (as represented by

Table 8.13: Static 1X, 2X, 3X and 4X Implementation Times on XC7VX980T FPGA

| System | 1X | 2X | 3X | 4X |
|---|---|---|---|---|
| Slice Utilization (%) | 18 | 29 | 46 | 63 |
| RAMB36 Utilization(%) | 20 | 39 | 59 | 80 |
| Implementation Time (s) | 3,253 | 23,830 | 36,629 | 126,904 |

the implementation time) when comparing the XC7160T and XC7325T devices. However, this trend does not continue when the utilization is reduced still further, by using a XC7V980T device (which results in a utilization of 20%); rather, here the implementation time increases, due to the much larger size of the FPGA being used, and the increased potential search space for mapping, placement and routing.

The results of Table 8.12 show that the fastest implementation results are obtained if the net-list hierarchy is maintained, but area constraints are not enforced, at least for the CAD tools used. If the net-list was completely flattened (no hierarchy), the implementation process failed during the place-and route phase, after 12 hours of operation. If area constraints are used in conjunction with a hierarchical net-list, the implementation process was found to complete but with a large effort (in excess of 16 hours for the place-and-route task). Table 8.13 shows that as the utilization of a large device continues to increase, so does the implementation time; this increase is not linear, unfortunately. The results show that implementation effort can be reduced dramatically (more than 24 hours of processing) by using smaller FPGAs for implementation purposes.

To compare the implementation efforts of monolithic systems with that of a system using modular implementation (such as a MACROS Framework System), Table 8.14 was assembled. This table lists the implementation time for the four static system, as well as total implementation efforts for equivalent MACROS Framework Systems. Such equivalent systems can be obtained by adding CMFUs to the existing system; this is accomplished by undertaking additional implementation runs, one per added CMFU.

Table 8.14: Static 1X, 2X, 3X and 4X Implementation Times on XC7V980T FPGA

| Equivalent Functionality | Static Implementation Time (s) | MACROS Implementation Time (s) |
|---|---|---|
| $9 \times 5$ Functional Units | 3,253 | 17,140 |
| $2 \times (9 \times 5)$ Functional Units | 23,830 | 23,280 |
| $3 \times (9 \times 5)$ Functional Units | 36,629 | 29,280 |
| $4 \times (9 \times 5)$ Functional Units | 126,904 | 35,280 |

The results show that, as the system complexity increases, the modular implementation approach that the MACROS Framework allows yields shorter implementation times. This can be attributed both to the reduction in the complexity of the mapping, placement and routing tasks (as they are performed on a functional-unit basis), as well as the fact that a smaller FPGA can be used to accomplish the same functionality; as Table 8.13 showed, the smaller the device used, the lesser the implementation effort, with some variation due to utilization. The most striking feature of the modular implementation approach being proposed is the fact that increases in system complexity and size result in a linear increase in implementation effort. These results demonstrated that modular implementation, as implemented in MACROS Framework Systems, leads to a reduction in implementation effort and associated time cost, as was proposed in the introduction of this work.

Finally, Table 8.15 lists the time costs when Hierarchical Design [177] is used in static systems. The table lists the implementation time for static versions of the MACROS Processors discussed above. The table lists initial implementation time costs, as well as re-implementation time costs, when one of the system modules is changed, while the others remain the same (and are re-used).

Table 8.15: Implementation Time Cost Using Hierarchical Design

| System | Hierarchical Design Initial Implementation Time (s) | Hierarchical Design Re-Implementation Time (s) |
|---|---|---|
| 4-Slot System - Static Version | 627 | 557 |
| 9-Slot System - Static Version | 740 | 600 |

As the results show, using the Hierarchical Design process, the re-implementation time can be reduced somewhat; however, the reduction is small (less than 20 % in either case). The off-line implementation times listed above are in the minutes range, as opposed to the MACROS approach, which uses on-line synthesis to generate ASP architectures in milliseconds (see Section 8.3). These results are not surprising, given that the goal of the Hierarchical Design process is *not* to reduce the implementation effort for various systems, but rather to improve design consistency and potentially help with timing closure [168].

## 8.7 Fault Mitigation Analysis

As discussed in Chapter 7, mitigation procedures in MACROS Framework systems consist of relocation of faulty components to spare slots, followed by potential scrubbing and testing operations to determine if the mitigated fault was transient or permanent. However, only the relocation process has an impact on processing operations; once a faulty CMFU has been relocated, additional testing procedures can occur in parallel with regular operations. The relocation procedure is identical to the process of loading a new CMFU to the system. Thus, the mitigation time cost is equivalent to that of ASP generation ($Tagen_{i-j}$), when only one CMFU must be added to the system. Based on the values presented above, relocation can be accomplished within 0.9407 ms. The fault will always be mitigated within two frames; if it is detected more than 0.9407 ms before the end of a frame, the mitigation process can be completed within that same frame.

The MACROS Framework can address both transient and permanent faults; in this it differs from pure scrubbing-based approaches [131, 132, 133, 134], which cannot mitigate permanent system faults. The mitigation process being proposed is based on functional unit relocation using partial bit-streams. The only limit imposed on the relocation process occurs if functional units are connected to I/O interfaces; otherwise, functional units can be moved freely between *all* slots. This differs from a number of existing approaches which use partial bit-streams, such as [137, 138], where single slots are reserved for specific system components only. Compared with existing approaches based on module relocation using partial bit-streams ([151, 152]), the mitigation mechanism proposed by the MACROS Framework is faster, as it permits module relocation using one configuration action. In [151], the relocation process for a single module requires multiple configuration steps to complete; likewise, the process presented in [152] requires multiple configuration steps for module relocation (one for the module proper, and one for the communication links in the system).

An alternative approach, adopted in a number of works [139, 156, 150], consists of relocation via full bit-stream reconfiguration. This approach can be used to mitigate multiple permanent faults, depending on the distribution of resources in the system. However, the time cost associated with a full reconfiguration is much larger than the cost of relocating only one functional unit via partial configuration; this fact is illustrated in Table 8.5, where the configuration time associated with both full and partial bit-streams are given for the XC7K325T device. To illustrate the advantage

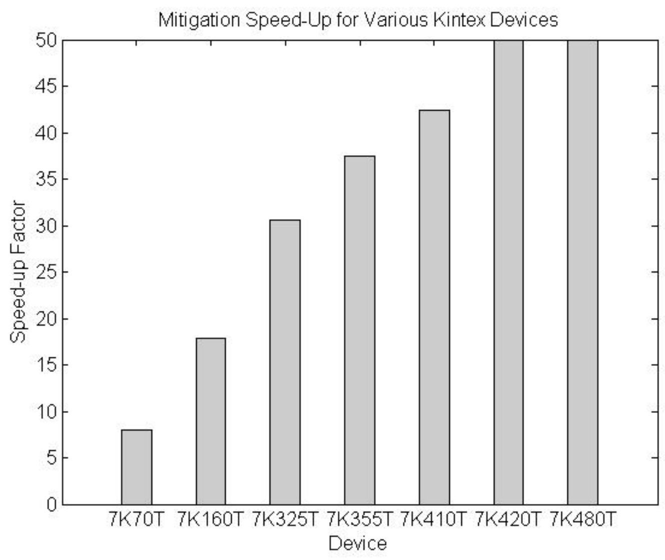
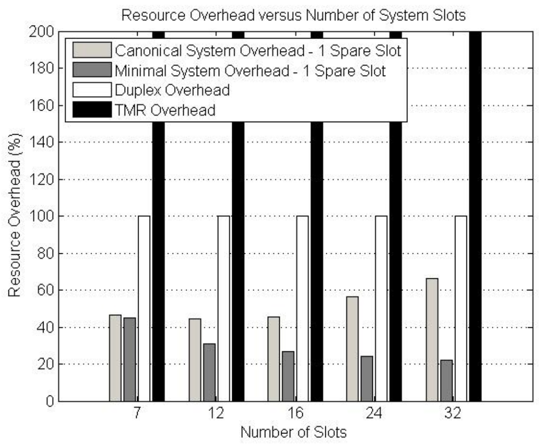


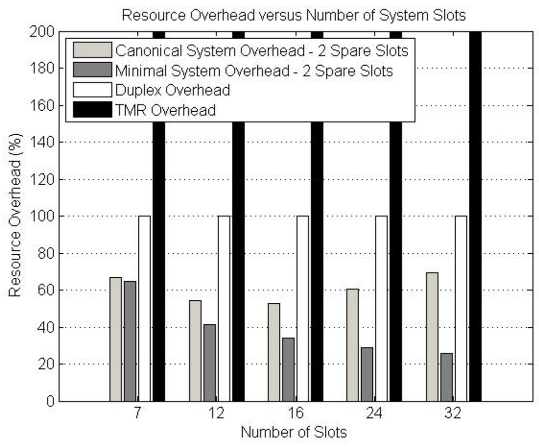
Figure 8.11: Full Versus Partial Bit-stream Component Mitigation Speedup

offered by partial bit-stream relocation, Figure 8.11 lists the increase in mitigation speed which results from using partial bit-streams instead of full ones for various Kintex and Virtex 7 devices; the reported data is based on the use of the slot size used in the presented video processors (300 slices, 10 RAMB36 blocks). The figure illustrates the fact that using partial bit-streams leads to a significantly faster mitigation time; this improvement increases in proportion to the size of the FPGA being used. In addition to this speed-up in mitigation time, a second feature of implementing relocation using partial bit-streams is the fact that the system does not undergo a global reset as part of the mitigation procedure, which does happen when the complete device is reconfigured.

A further comparison can be made between MACROS Systems and traditional Duplex and TMR systems. The MACROS Framework, through its associated overhead, allows fault mitigation via the usage of spare slots; the spare slots and control circuitry can be considered the cost for allowing the proposed mitigation method. TMR and Duplex systems also exhibit an overhead, of more than 100 and 200%, respectively, due to the reproduction of processing elements. Figures 8.12(a) and 8.12(b) show the overheads for different system sizes, for the Canonical and Minimal control methods, respectively, compared with Duplex and TMR systems; Figure 8.12(a) lists the overhead associated with a system with one spare slot, while Figure 8.12(b) lists the overhead for a system with two spare slots. In both cases, the overhead consists of spare system slots and control circuits; the crossbar is not considered an overhead, as it is required to support of multi-mode



(a) 1 Spare Slot



(b) 2 Spare Slots

Figure 8.12: Resource Overhead Comparison for Fault Mitigation Methods

operation and is, therefore, considered an unavoidable cost. It must be stressed that these figures list the overhead to the computation circuitry only, and only address the mitigation (repair) task. Fault detection via testing is not a topic of research in the presented work, and is not considered here.

Although the MACROS Framework adds overhead to a system, this overhead is, in the worst case, 80% (if a Canonical LCCU architecture with fully parallel decoders is used). In the case of the Minimal system, the overhead drops to under 30% for systems of 16 slots or more. Both approaches (even the expensive Canonical approach) still require less resources than a duplex system; compared with TMR, the resource overhead is reduced by more than half, even in this worst case situation. As discussed in Chapter 6, the MACROS Framework reduces the need for traditional hardening methods to only a small sub-set of the system (the Mode-Mapper and the BCM). As a result of this, fault repair capabilities can be obtained at a reduced resource cost; this reduction can be more than 80% when comparing TMR with the Minimal system architecture. Thus, not only does the MACROS Framework permit fault repair (via relocation of functional units), but it offers this repair capability at a greatly reduced cost when compared with traditional solutions based on redundant circuits.

## 8.8 Complete Architectural Comparisons

Previous sections of this chapter have analyzed individual aspects and components of the MACROS Framework, and, where possible, have compared them with existing approaches. In this section, a more holistic analysis shall be attempted, where complete architectures are compared. It must be noted that such comparisons are difficult, as complete information regarding *all* aspects of a proposed architecture is not easily found in published conference and journal works; as a result, the presented comparisons will be more qualitative than quantitative in nature.

A large number of examples of reconfigurable system architectures haven been presented in various research works over the past 10 years. However, the overwhelming majority follow the processor accelerator model initially outlined in [70, 71], where an instruction-based processor is augmented by dedicated accelerators, some of which are dynamically loaded into the system. Two examples of architectures which differ significantly from this approach are the Run-time Adaptive Multi-Processor System-on-Chip (RAMPSoC) [109] and systems based on the RAPTOR 64 architecture [82, 83].

The architecture of the Run-time Adaptive Multi-Processor System-on-Chip (RAMPSoC) is presented in [109, 110, 111]. The system consists of a variable number of on-chip processing elements connected via an NoC. The individual processing elements can be either software processors (MicroBlaze processors), possibly augmented by accelerators, or dedicated macro-function processors. Depending on the demands of the deployed applications, processing elements can be added or removed from the system. Initially, system control was accomplished using a static, software based processor running a dedicated Configuration Access Port Operating System (CAP-OS) [120]; this operating system was responsible for scheduling tasks, allocating said tasks to resources (including partial configuration to load additional processing elements if necessary) and establishing communication links between tasks. This operating system was further updated into a virtualization environment in [128], in the form of RAMPSoCVM. The presented environment was based on an embedded variant of Linux, and had the aim of abstracting away architecture details from the application designer; as before, a central processor was used to control task deployment using a server-client approach. In RAMPSoC systems, communications between processing elements are accomplished via the Star-Wheels NoC [94, 95]; for a more detailed discussion of this architecture, please consult Chapter 2.

A multi-FPGA architecture is presented in [82, 83], based on the RAPTOR 64 platform. The system consist of three modules connected together, one Communication Module and two Processing Modules. The communication module contains various communication interfaces, a central controller based on the LEON2-FT CPU, and communication controllers based on Spartan FPGA. The processing modules consists of Virtex 4 devices and local DRAM memories. The on-chip architecture of the processing modules consists of a MicroBlaze controller, configuration controller, memory controller and reconfigurable region; these components communicate via a shared bus, and also use dedicated links to the memory controller. Communication between functional units is accomplished via a shared local bus, as well as point-to-point links between adjacent modules.

Both these architectures are, nominally, general computing architectures, in that they make no explicit assumptions about the type of applications they will house. Likewise, they implicitly assume that instruction-based sequential processors will be used for both system control as well as processing tasks, with accelerators being deployed strategically for certain applications only. This approach does not lend itself well to high data rate stream applications, where instruction-based sequential processors are not capable of sustaining the needed level of processing throughput. To illustrate this fact, the Zynq system introduced in Chapter 7 was used to implement one of the masking operations described above; the mask was limited to only a single color channel, and omitted data transfer considerations. The resulting implementation could sustain a frame-rate of 7.2 frames per second, while operating at a frequency of 667 MHz. In contrast, mask CMFUs based on custom architectures could maintain a frame-rate of 60 frames per second while operating at a frequency of 74.25 MHz.

Examples of reconfigurable architectures aimed at pipe-lined systems using custom architectures do exist, primarily in the form of dynamically reconfigurable video processing systems. Two types of reconfigurable embedded video processing architectures are presented in [180], one based on the Erlangen Slot Machine [50, 51], and the other based on the Autovision system [114]. In both cases, the main processing path consists of dedicated acquisition and processing circuits, with the general processor playing a secondary management role in the system.

An on-chip DPR architecture is presented in [99] which consists of a general processor, plus one or more dynamic regions where modules can be placed. The communication between the processor and dynamic modules is accomplished via shared bus architectures. In addition, an I/O Bar (similar in structure to a crossbar) is included, which allows high data rate communications inside the processing path as well as to and from select I/O interfaces (video input and output interfaces). Finally, [115] shows a multi-point tracking embedded video system; once again, the implemented system relies heavily on custom hardware data-paths and a dedicated communication infrastructure. As with previous video processing architectures, the presented systems make use of central control elements, in the form of instruction-based processors (although the majority of processing operations are implemented as macro-function modules). As well, these works do not elaborate how processing and configuration activities are integrated, or how the reconfiguration aspect affects system behavior.

The above video applications can be considered closest to the MACROS Framework architecture, due to their emphasis on stream processing, and implementation of pipe-lined systems. However, they differ in two primary aspects. First of all, the presented architectures rely, in most cases, on central, instruction-based processors for system control and configuration; while the configuration task is, itself, sequential by nature, other control tasks are not, and can benefit from a distributed architecture, as was shown above. Thus, a central control solution using a sequential, instruction-based controller may provide inferior performance. Secondly and more importantly, none of the presented architectures explicitly attempt to integrate the architecture adaptation process (configuration, link establishment) with the stream processing tasks of the system. No mechanisms are provided to control and, if possible, minimize the impact that the adaptation process may have on the processing task. Finally, the use of a central control architecture and the lack of explicit support for functional unit relocation makes these systems less fault resilient that the proposed MACROS Framework architecture.

## 8.9 Summary

The preceding chapter concludes the description of the MACROS Framework, by validating the operating principles underpinning the framework, as well as the mechanisms which permit said operation. The framework was shown capable of implementing run-time mode-based architecture adaptation, as well as fault mitigation via functional unit (or CMFU, in this case) relocation.

The implementation effort associated with MACROS Framework systems was shown to be linearly proportional to the functionality of the implemented system, and smaller than that of similar static systems. The fault mitigation procedures being proposed were, likewise, shown to offer faster mitigation capabilities, with a reduced resource overhead.

Secondly, the distributed control architecture proposed as part of the MACROS Framework was shown to offer better performance than central solutions. The use of a crossbar communication architecture was, likewise, found to offer a better combination of performance and resource overhead when compared with existing NoC architectures. Thus, all aspects of the framework were validated.

# Chapter 9

# Conclusion

## 9.1 Problems in Modern Digital System Design and the Concept of Modular Dynamic Systems

Modern digital processing systems are becoming increasingly more complex: both the application complexity and volume of data processed are increasing, while system constraints are becoming more severe. This increased complexity is most evident in modern multi-mode applications, which exhibit multiple algorithm variations depending on the various operating and environmental conditions. At the same time, modern FPGA devices are increasingly being used as implementation mediums for complex applications, due to their reduced development cost. In response to this, modern FPGAs have shown a constant increase in capacity, and look set to continue this increase into the further. However, the increased size of digital systems and the FPGAs that house them has lead to a rapid increase in the implementation effort associated with these systems, in particular in the mapping, placement and routing tasks. As both the application complexity and device capacity continue to rise, this increased complexity threatens to be come a development bottle-neck.

As stated above, FPGAs are increasingly adopted as implementation mediums for modern digital processing systems; one of the fields which has seen an increase in FPGA use is that of space-borne computing system. Due to the increased radiation exposure associated with this environment, FPGA-based systems can undergo system faults, both transient (SEUs and SEFIs), or permanent. Until the present, these fault types have been limited to space-borne applications;

however, as the manufacturing process for CMOS devices improves and the associated feature size of transistors continues to decrease, these fault types may extend to terrestrial applications. This phenomenon can already be seen in DRAM memory systems [29, 30], where high-energy particle impacts cause data corruption. Thus, fault tolerance and mitigation methods will become more central to successful system designs. Unfortunately, traditional fault tolerance and mitigation relies on resource redundancy; this approach cannot be pursued indefinitely as system complexity continues to increase, due to the problem of implementation effort discussed above.

A potential solution to the above two problems is the concept of a modular and dynamic system. Such a system has the capability of changing is architecture at run-time in response to various conditions, via different combinations of functional units. These functional units implement macro-function operations meaning that the functionality of the system changes at the macro-function (as opposed to the micro-function level). Such a system is assembled at run-time from individual functional units; thus, implementation tasks are applied to functional units as opposed to a complete system, which reduces the implementation complexity and effort. By changing its own architecture at run-time in response to mode changes, the system can re-used the same underlying logic resources between modes; this further reduces the size of the system, and minimizes the problem of implementation complexity. At the same time, a system that supports run-time architecture changes can mitigate permanent system faults by ensuring that the faulty region is avoided. Thus, such a system will address both of the problems discussed above.

## 9.2 The MACROS Framework: Run-time Architecture Adaptation in Stream Processors

The preceding work has introduced the Multi-Mode Adaptive Collaborative Reconfigurable self-Organized System (MACROS) Framework, which permits the implementation of the modular dynamic system concept described above using dynamic partially reconfigurable FPGAs. The MACROS Framework is aimed primarily towards multi-mode high data-rate stream processing applications, although it can be applied to any multi-mode application, where the functionality of the system is dictated by the current mode of operation. The framework consists of a general system architecture, and an associated system behavior; together these two elements allow for run-time architecture adaptation in MACROS Framework systems.

MACROS Framework systems are composed of Collaborative Macro-Function Units (CMFU), macro-function processing elements which exhibit semi-autonomous operations, supported by a Distributed Communication and Control Infrastructure (DCCI) and a Bit-stream and Configuration Manager (BCM). MACROS Framework Systems have the capability of assembling and integrating Application-Specific Processors (ASP) from CMFUs, and can generate new ASP architectures at run-time in response to changes in mode. In addition, MACROS Framework systems have the capability to relocate CMFUs in the system, without affecting the functionality of the being implemented.

The functionality of the MACROS Framework has been tested via multiple video processing system implementations, as described in Chapters 7 and 8. The implemented MACROS Framework system video processors were able to self-assemble and self-integrate into working processors at run-time. The systems were also shown to support the generation of new ASP architectures at run-time, via deployment of new CMFUs, re-establishment of system links and self-integration. Equally important, the relocation of CMFUs was successfully implemented; this feature allows faulty regions of the device to be avoided at run-time without affecting the functionality of the system.

The ability of MACROS Framework systems to self-assemble and self-integrate into ASPs at run time through the use of CMFU partial bit-streams allows for modular system implementation. The implementation process for MACROS Framework systems relies on the mapping of net-lists into system slots, and performing mapping, placement and routing on a per-CMFU basis; in this instance, the scope of these tasks is limited, allowing for faster implementation. This feature was shown in Chapter 8, where MACROS Framework systems implementation times were found to be up to 65 % smaller when compared with equivalent static systems. More important was the fact that, in a MACRO System, the implementation time is linearly dependent on the system functionality as represented by number of CMFUs. Thus, the proposed framework offers a method of addressing the problem of increasing system complexity and implementation effort introduced in Chapter 1 and re-iterated above.

By allowing CMFU relocation, MACROS Framework systems also allow mitigation of both transient and permanent faults. This leads to better fault coverage than scrubbing-based solutions.

At the same, time, CMFU relocation is accomplished via partial bit-streams as opposed to full system bit-streams, which leads to a drastic improvement in mitigation time when compared to existing approaches; partial bit-streams lead to a mitigation time speed-up of at least 10 times in small Kintex devices, and as much as 50 times when considering large devices.

The MACROS framework allows for fault mitigation via relocation, thanks to the inclusion of distributed control circuitry; this control circuitry is an added cost to the system. Traditional approaches to fault mitigation already exist, through the use of resource duplication, in the form of Duplex systems and Triple Module Redundancy (TMR). However, the MACROS Framework overhead, in the absolute worst case scenario will reach 80% in large system, if using the Canonical connectivity change procedure and parallel decoders; in smaller systems, and when using the Minimal connectivity change procedure, this overhead drops below 40%, and can be as low as 28%. In contrast, duplex and TMR approaches add 100 and 200 % overhead, respectively. Thus, the MACROS Framework offers fault mitigation capabilities at a significantly reduced overhead cost when compared with traditional approaches.

The MACROS Framework is aimed at high-performance stream processing applications, where large volumes of data must be processed in limited time-frames; such applications offer limited windows where the ASP generation process may be implemented. By allowing configuration activities to be overlapped with processing activities, MACROS Framework Systems were found capable of seamless architecture adaptation; seamless here is defined to mean that the adaptation process would not interfere with the application workload. This feature was made possible thanks to the use of a distributed control architecture which allowed for parallel implementation of system-wide control operations; in the presented experiments, link establishment and integration took less than 1 $\mu s$ to complete, and could be as low as 120 ns. This same level of performance would not have been possible using existing implementation approaches, which rely on the use of central control elements using sequential, instruction-based processors. It was shown that MACROS Framework control operations could be implemented up to 10 times faster compared to a MicroBlaze processor; more importantly, the time cost for the MicroBlaze processor was found to grow much more rapidly as system size increased. This difference in time cost allows the MACROS framework to accommodate systems with much larger data rates, where the windows for control operations are limited. Alternatively, when comparing the MACROS Framework control process with central control solutions based on the ARM Cortex A9 processor core, similar levels of performance could be obtained, while suing a much smaller operating frequency (7 times smaller).

The MACROS Framework has direct application to any multi-mode stream processing application, and offers particular benefits to high data-rate systems. In particular, this framework can help reduce the complexity and implementation cost of networking and telecommunications infrastructure components, such as high data-rate routing and switching equipment, wireless base-stations and satellite transceivers. The listed systems all process large volumes of data and exhibit behavior variations in response to various conditions. Likewise, the MACROS Framework can offer increased fault tolerance to digital computing systems deployed in hazardous environments; examples include various space-borne applications (such as telecommunication and exploration), as well as monitoring and control of hazardous environments on the Earth's surface, such as nuclear power generation facilities.

## 9.3 Future Work

Based on the foundation presented in this work, a number of further developments can be made to the MACROS framework which will further increase its usefulness as a system design approach. The first such development, and one of the most important, is the development of better support for data-flow applications. More specifically, more detailed methods are needed for the synchronization of data in such systems, such that functional units with various latencies and cycle times can be supported. Currently, the MACROS Framework imposes specific requirements on the behavior of the functional units (generally IP Cores) being used, such that cycle times match throughout the system, and latencies match at a given level (See Chapter 4). Alternative solutions should be investigated, including the possibility of additional control signaling, or the inclusion of interface units into system links.

On a related note, further investigations are required in support for multi-clock domain systems. As it stands, the MACROS Framework makes no explicit concessions towards the existence of multiple clocks; the communication and control infrastructure both assume the existence of a single unifying synchronization signal. However, a number of modern systems make use of multiple clock domains; thus, the framework should be expanded to incorporate clock domain boundary crossover mechanisms for both data and control signals. Likewise, the system currently assumes one stream structure used throughout the system. However, applications exist where the stream structure may change (for example, as a result of compression operations or parallelization for processing purposes). Thus, the framework will require expansion to support stream structure transformations.

Finally, The MACROS Framework primarily targets high performance, high data-rate systems, and as a result, eschews all methods of resource time sharing among active units. However, in a multi-task scenario, different tasks may require different performance. For such systems, the MACROS framework should be expanded to support time-division multiplexing of system slots among functional units. In this way, systems can be built which can offer high performance capabilities to the sub-set of tasks that require it, while allowing all other tasks to exhibit reduced performance; such systems can lead to further reductions in deployment costs.

# Bibliography

[1] *Digital Video Broadcasting (DVB); Framing Structure, channel coding and modulation for Satellite Services to Handheld devices (SH) below 3 GHz*, ETSI Std. EN 302 583, 2012.

[2] B. P. Lathi and Z. Ding, *Modern Digital and Analog Communication Systems*, 4th ed. New York, New York, USA: Oxford University Press, 2009.

[3] *3GPP2: Physical Layer Standard for cdma2000 Spread Spectrum Systems, Revision D*, 3GPP2 Std. C.S0002-D, 2005.

[4] *Infrastructure of audiovisual services Coding of moving video*, ITU-T Std. H.264, 2013.

[5] *Virtex 2.5 V Field Programmable Gate Arrays (v4.0)*, Xilinx, 2013.

[6] *Virtex-II Platform FPGAs: Complete Data Sheet (v4.0)*, Xilinx, 2014.

[7] *Virtex-4 Family Overview (v3.1)*, Xilinx, 2010.

[8] *Virtex-5 Family Overview (v5.0)*, Xilinx, 2009.

[9] *Virtex-6 Family Overview (v2.4)*, Xilinx, 2012.

[10] *7 Series FPGAs Overview (v1.15)*, Xilinx, 2014.

[11] *Stratix Device Handbook, Volume 1*, Altera Corporation, 2005.

[12] *Stratix II Device Handbook, Volume 1*, Altera Corporation, 2007.

[13] *Stratix III Device Handbook, Volume 1*, Altera Corporation, 2010.

[14] *Stratix IV Device Handbook, Volume 1*, Altera Corporation, 2012.

[15] *Stratix V Device Overview*, Altera Corporation, 2014.

[16] *Virtex-4 FPGA User Guide (v2.6)*, Xilinx, 2008.

[17] *Virtex-5 FPGA User Guide (v5.4)*, Xilinx, 2012.

[18] "International technology roadmap for semiconductors, 2013 edition," ITRS, Tech. Rep., 2013.

[19] A. Corporation. (2014) Altera achieves industry milestone: Demonstrates fpga technology based on intel 14 nm tri-gate process. [Online]. Available: http://newsroom.altera.com/press-releases/nr-14nm-device.htm

[20] A. Sherwani, N., *Algorithms for VLSI Physical Design Automation.* Hingham, MA, USA: Kluwer Academic Publishers, 1998.

[21] Y. Sankar and J. Rose, "Trading quality for compile time: Ultra-fast placement for fpgas," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA.* ACM, 1999, pp. 157–166.

[22] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for fpgas," in *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA.* ACM, 2000, pp. 203–213.

[23] A. Singh, G. Parthasarathy, and M. Marek-Sadowska, "Efficient circuit clustering for area and power reduction in fpgas," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 4, pp. 643–663, 2002.

[24] N. H. E. Weste and D. Harris, *CMOS VLSI Design: a Circuits and Systems Perspective*, 3rd ed. Boston, Massachusetts, USA: Pearson Education Inc., 2009.

[25] S. Zhang and H. Liu, "Synthetical analysis on space radiation tolerance techniques in asics and fpgas," in *Proceedings of the International Conference on System Science, Engineering Design and Manufacturing Informatization (ICSEM), 2011, Guiyang, China*, vol. 2. IEEE, 2011, pp. 305–310.

[26] G. Allen, *Virtex-4VQ dynamic and mitigated single event upset characterization summary.* Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2009.

[27] C. Carmichael and C. Tseng, "Xapp1088: Correcting single-event upsets in virtex-4 fpga configuration memory (v1.0)," Xilinx, Tech. Rep., Oct. 2009.

[28] F. Smith and S. Mostert, "Total ionizing dose mitigation by means of reconfigurable fpga computing," *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 1343–1349, 2007.

[29] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1. ACM, 2009, pp. 193–204.

[30] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: positional effects in dram and sram faults," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, USA.* ACM, 2013, pp. 1–11.

[31] *Command Line Tools User Guide (v14.7)*, Xilinx, 2013.

[32] *7 Series FPGAs Configurable Logic Block (v1.6)*, Xilinx, 2014.

[33] *7 Series FPGAs Memory Resources (v1.10.1)*, Xilinx, 2014.

[34] *7 Series DSP48E1 Slice (v1.7)*, Xilinx, 2014.

[35] *Partial Configuration User Guide - UG702 (v14.5)*, Xilinx, 2013.

[36] V. Kirischian, "The methodology of synthesis of dynamically reconfigurable computing systems with temporal partitioning of homogeneous resources," Ph.D. dissertation, Ryerson University, Toronto, Canada, 2010.

[37] *HDL Coder - Data Sheet*, MathWorks, 2014.

[38] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital systems testing and testable design.* Hoboken, New Jersey, USA: Wiley-IEEE Press, 1990.

[39] L. Kirischian, V. Dumitriu, and P. W. Chun, "Virtualization of computing resources in rcs for multi-task stream applications," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs, 2009, Cancun, Mexico.* IEEE, 2009, pp. 368–373.

[40] L. Kirischian, V. Dumitriu, P. W. Chun, and G. Okouneva, "Mechanism of resource virtualization in rcs for multitask stream applications," *International Journal of Reconfigurable Computing*, vol. 2010, p. 8, 2010.

[41] V. Dumitriu, D. Marcantonio, and L. Kirischian, "Run-time component relocation in partially-reconfigurable fpgas," in *Proceedings of the International Conference on Computational Science and Engineering, 2009, Vancouver, Canada*, vol. 2. IEEE, 2009, pp. 909–914.

[42] V. Dumitriu and L. Kirischian, "A framework of embedded reconfigurable systems based on re-locatable virtual components," *International Journal of Embedded Systems*, vol. 4, no. 3, pp. 182–194, 2010.

[43] ——, "Soc self-integration mechanism for dynamic reconfigurable systems based on collaborative macro-function units." *Proceedings of the International Conference on Reconfigurable Computing and FPGAs, 2013, Cancun, Mexico*, pp. 1–7, 2013.

[44] V. Dumitriu, L. Kirischian, and V. Kirischain, "A framework for adaptive reconfigurable space-borne computing platforms for run-time self-recovery from transient and permanent hardware faults," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2012, Erlangen, Germany.* IEEE, 2012, pp. 280–287.

[45] V. Dumitriu, L. Kirischian, and V. Kirischian, "Decentralized run-time recovery mechanism for transient and permanent hardware faults for space-borne fpga-based computing systems," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2014, Leicester, UK.* IEEE, 2014, pp. 47–54.

[46] *7 Series FPGAs SelectIO Resources (v1.4)*, Xilinx, 2014.

[47] *7 Series FPGAs Clocking Resources (v1.10)*, Xilinx, 2014.

[48] *Spartan-3AN FPGA Family Data Sheet (v4.2)*, Xilinx, 2014.

[49] *7 Series FPGAs Configuration User Guide - UG470 (v1.7)*, Xilinx, 2013.

[50] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The erlangen slot machine: increasing flexibility in fpga-based reconfigurable platforms," in *Proceedings of the IEEE International Conference on Field-Programmable Technology, 2005, Singapore*, 2005, pp. 37–42.

[51] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The erlangen slot machine: A dynamically reconfigurable fpga-based computer," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 47, no. 1, pp. 15–31, 2007.

[52] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, "Dynamic and partial fpga exploitation," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 438–452, 2007.

[53] D. Llamocca, M. Pattichis, and G. Alonzo Vera, "A dynamic computing platform for image and video processing applications," in *Proceedings of the Forty-Third Asilomar Conference on Signals, Systems and Computers, 2009, Pacific Grove, California, USA*. IEEE, 2009, pp. 412–416.

[54] A. Rao, S. Nandy, H. Nikolov, and E. F. Deprettere, "Usha: Unified software and hardware architecture for video decoding," in *Proceedings of the IEEE 9th Symposium on Application Specific Processors (SASP), 2011, San Diego, California, USA*. IEEE, 2011, pp. 30–37.

[55] C. Trabelsi, S. Meftali, and J. Dekeyser, "Distributed control for reconfigurable fpga systems: a high-level design approach," in *Proceedings of the 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012, York, UK*. IEEE, 2012, pp. 1–8.

[56] D. Llamocca and M. Pattichis, "A dynamically reconfigurable pixel processor system based on power/energy-performance-accuracy optimization," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 3, pp. 488–502, 2013.

[57] H. Kalte, M. Porrmann, and U. Ruckert, "System-on-programmable-chip approach enabling online fine-grained 1d-placement," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004, Santa Fe, New Mexico, USA*. IEEE, 2004, pp. 141–148.

[58] J. Hagemeyer, B. Kettelhoit, and M. Porrmann, "Dedicated module access in dynamically reconfigurable systems," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium, 2006, Rhodes Island, Greece*. IEEE, 2006, pp. 8–pp.

[59] F. Ferrandi, M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto, "Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead," in *Proceedings of the International Symposium on System-on-Chip, 2006, Tampere, Finalnd*. IEEE, 2006, pp. 1–4.

[60] M. Hubner, C. Schuck, M. Kuhnle, and J. Becker, "New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits," in *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006, Karlsruhe, Germany*. IEEE, 2006, pp. 1–6.

[61] M. Hubner, C. Schuck, and J. Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for virtex-ii fpgas," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium, 2006, Rhodes Island, Greece*. IEEE, 2006, pp. 1–8.

[62] C. Schuck, M. Kuhnle, M. Hubner, and J. Becker, "A framework for dynamic 2d placement on fpgas," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2008, Miami, Florida, USA*. IEEE, 2008, pp. 1–7.

[63] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in virtex fpgas," in *IEE Proceedings - Computers and Digital Techniques*, vol. 153, no. 3. IET, 2006, pp. 157–164.

[64] Y. E. Krasteva, A. B. Jimeno, E. de la Torre, and T. Riesgo, "Straight method for reallocation of complex cores by dynamic reconfiguration in virtex ii fpgas," in *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping, 2005, Montreal, Canada.* IEEE, 2005, pp. 77–83.

[65] S. Corbetta, M. Morandi, M. Novati, M. D. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 11, pp. 1650–1654, 2009.

[66] C. Rossmeissl, A. Sreeramareddy, and A. Akoglu, "Partial bitstream 2-d core relocation for reconfigurable architectures," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, 2009, San Francisco, California, USA.* IEEE, 2009, pp. 98–105.

[67] M. Koester, W. Luk, J. Hagemeyer, M. Porrmann, and U. Ruckert, "Design optimizations for tiled partially reconfigurable systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 1048–1061, 2011.

[68] C. Hong, K. Benkrid, X. Iturbe, A. Ebrahim, and T. Arslan, "Efficient on-chip task scheduler and allocator for reconfigurable operating systems," *IEEE Embedded Systems Letters*, vol. 3, no. 3, pp. 85–88, 2011.

[69] X. Iturbe, K. Benkrid, A. Ebrahim, C. Hong, T. Arslan, and I. Martinez, "Snake: An efficient strategy for the reuse of circuitry and partial computation results in high-performance reconfigurable computing," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2011, Cancun, Mexico.* IEEE, 2011, pp. 182–189.

[70] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A self-reconfiguring platform," in *Field Programmable Logic and Application.* Springer, 2003, pp. 565–574.

[71] B. Blodget, S. McMillan, and P. Lysaght, "A lightweight approach for embedded reconfiguration of fpgas," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003, Munich, Germany.* IEEE, 2003, pp. 399–400.

[72] M. Ullmann, M. Hübner, B. Grimm, and J. Becker, "An fpga run-time system for dynamical on-demand reconfiguration," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004, Santa Fe, New Mexico, USA.* IEEE, 2004, p. 135.

[73] M. Huebner, T. Becker, and J. Becker, "Real-time lut-based network topologies for dynamic and partial fpga self-reconfiguration," in *Proceedings of the 17th Symposium on Integrated Circuits and Systems Design, 2004, Porto de Galinhas, Pernambuco, Brazil.* IEEE, 2004, pp. 28–32.

[74] F. Ferrandi, M. D. Santambrogio, and D. Sciuto, "A design methodology for dynamic reconfiguration: The caronte architecture," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, 2005, Denver, Colorado, USA.* IEEE, 2005, pp. 1–4.

[75] A. Donato, F. Ferrandi, M. Redaelli, M. D. Santambrogio, and D. Sciuto, "Caronte: a complete methodology for the implementation of partially dynamically self-reconfiguring systems on fpga platforms," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2005, Napa, California, USA*. IEEE, 2005, pp. 321–322.

[76] C. Claus, J. Zeppenfeld, F. Müller, and W. Stechele, "Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system," in *Proceedings of the Design, Automation & Test in Europe Conference and Exhibition, 2007, Nice, France.* EDA Consortium, 2007, pp. 498–503.

[77] X. Zhang, H. Rabah, and S. Weber, "Auto-adaptive reconfigurable architecture for scalable multimedia applications," in *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems, 2007, Edinburgh, Scotland, UK.* IEEE, 2007, pp. 139–145.

[78] ——, "Cluster-based hybrid reconfigurable architecture for auto-adaptive soc," in *Proceedings of the 14th IEEE International Conference on Electronics, Circuits and Systems, 2007, Marrakech, Morocco.* IEEE, 2007, pp. 979–982.

[79] K. Schleupen, S. Lelaich, R. Mannion, Z. Guo, W. Najjar, and F. Vahid, "Dynamic partial fpga reconfiguration in a prototype microprocessor system," in *Proceedings of the International Conference on Field Programmable Logic and Applications, 2007, Amsterdam, Netherlands.* IEEE, 2007, pp. 533–536.

[80] X. Zhang, Y. Ding, Y. Huang, and X. Dong, "Design and implementation of a heterogeneous high-performance computing framework using dynamic and partial reconfigurable fpgas," in *Proceedings of the IEEE 10th International Conference on Computer and Information Technology (CIT), 2010, Bradford, West Yorkshire, UK.* IEEE, 2010, pp. 2329–2334.

[81] V. Rana, M. Santambrogio, D. Sciuto, B. Kettelhoit, M. Koester, M. Porrmann, and U. Ruckert, "Partial dynamic reconfiguration in a multi-fpga clustered architecture based on linux," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2007, Long Beach, California, USA.* IEEE, 2007, pp. 1–8.

[82] J. Hagemeyer, A. Hilgenstein, D. Jungewelter, D. Cozzi, C. Felicetti, U. Ruckert, S. Korf, M. Koester, F. Margaglia, M. Porrmann *et al.*, "A scalable platform for run-time reconfigurable satellite payload processing," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2012, Erlangen, Germany.* IEEE, 2012, pp. 9–16.

[83] L. Sterpone, M. Porrmann, and J. Hagemeyer, "A novel fault tolerant and runtime reconfigurable platform for satellite payload processing," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1508–1525, 2013.

[84] G. DeMicheli and L. Benini, *Networks on Chips*, 1st ed. Burlington, Massachusetts, USA: Morgan Kaufmann Publishers, 2006.

[85] R. Koch, T. Pionteck, C. Albrecht, and E. Maehle, "An adaptive system-on-chip for network applications," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium, 2006, Rhodes Island, Greece.* IEEE, 2006, pp. 1–8.

[86] T. Pionteck, C. Albrecht, and R. Koch, "A dynamically reconfigurable packet-switched network-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe, 2006, Munich, Germany.* European Design and Automation Association, 2006, pp. 136–137.

[87] T. Pionteck, R. Koch, and C. Albrecht, "Applying partial reconfiguration to networks-on-chips," in *Proceedings of the International Conference on Field Programmable Logic and Applications, 2006, Madrid, Spain.* IEEE, 2006, pp. 1–6.

[88] B. Osterloh, H. Michalik, B. Fiethe, and K. Kotarowski, "Socwire: A network-on-chip approach for reconfigurable system-on-chip designs in space applications," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, 2008, Noordwijk, The Netherlands.* IEEE, 2008, pp. 51–56.

[89] Y. Dadji, B. Osterloh, and H. Michalik, "A middleware aided robust and fault tolerant dynamic reconfigurable architecture," in *Proceedings of the ASME/IFToMM International Conference on Reconfigurable Mechanisms and Robots, 2009, London, UK.* IEEE, 2009, pp. 572–579.

[90] L. Devaux, S. Ben Sassi, S. Pillement, D. Chillet, and D. Demigny, "Draft: Flexible interconnection network for dynamically reconfigurable architectures," in *Proceedings of the International Conference on Field-Programmable Technology, 2009, Sydney, Australia.* IEEE, 2009, pp. 435–438.

[91] L. Devaux, S. Pillement, D. Chillet, D. Demigny *et al.*, "R2noc: Dynamically reconfigurable routers for flexible networks on chip," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs, 2010, Cancun, Mexico.* IEEE, 2010, pp. 376–381.

[92] J. Strunk, J. Hiltscher, W. Rehm, and H. Schick, "Communication architectures for run-time reconfigurable modules in a 2-d mesh on fpgas," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs, 2010, Cancun, Mexico.* IEEE, 2010, pp. 49–54.

[93] L. Moller, P. Fischer, F. Moraes, L. S. Indrusiak, and M. Glesner, "Improving qos of multi-layer networks-on-chip with partial and dynamic reconfiguration of routers," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2010, Milano, Italy.* IEEE, 2010, pp. 229–233.

[94] D. Gohringer, B. Liu, M. Hubner, and J. Becker, "Star-wheels network-on-chip featuring a self-adaptive mixed topology and a synergy of a circuit-and a packet-switching communication protocol," in *Proceedings of the International Conference on Field Programmable Logic and Applications, 2009, Prague, Czech Republic.* IEEE, 2009, pp. 320–325.

[95] D. Gohringer, O. Oey, M. Hubner, and J. Becker, "Heterogeneous and runtime parameterizable star-wheels network-on-chip," in *Proceddings of the International Conference on Embedded Computer Systems (SAMOS), 2011, Samos, Greece.* IEEE, 2011, pp. 380–387.

[96] P. Lysaght and D. Levi, "Of gates and wires," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004, Santa Fe, New Mexico, USA.* IEEE, 2004, pp. 132–137.

[97] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S. Fekete, and J. Van der Veen, "A practical approach for circuit routing on dynamic reconfigurable devices," in *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping, 2005, Montreal, Canada*, 2005, pp. 84–90.

[98] J. Strunk, T. Volkmer, W. Rehm, and H. Schick, "An on chip network inside a fpga for run-time reconfigurable low latency grid communication," in *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, 2009, Patras, Greece.* IEEE, 2009, pp. 539–546.

[99] A. Oetken, S. Wildermann, J. Teich, and D. Koch, "A bus-based soc architecture for flexible module placement on reconfigurable fpgas," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2010, Milano, Italy.* IEEE, 2010, pp. 234–239.

[100] R. Ferreira, J. Vendramini, and M. Nacif, "Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga," in *Proceedings of the 9th IEEE International Conference on Industrial Informatics (INDIN), 2011, Lisbon, Portugal.* IEEE, 2011, pp. 810–815.

[101] M. He, Y. Cui, M. H. Mahoor, and R. M. Voyles, "A heterogeneous modules interconnection architecture for fpga-based partial dynamic reconfiguration," in *Proceedings of the 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012, York, UK.* IEEE, 2012, pp. 1–7.

[102] S. Koh and O. Diessel, "Communications infrastructure generation for modular fpga reconfiguration," in *Proceedings of the IEEE International Conference on Field Programmable Technology, 2006, Bangkok, Thailand.* IEEE, 2006, pp. 321–324.

[103] ——, "The effectiveness of configuration merging in point-to-point networks for module-based fpga reconfiguration," in *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines, 2008, Palo Alto, California, USA.* IEEE, 2008, pp. 65–76.

[104] K. F. Ackermann, B. Hoffmann, L. S. Indrusiak, and M. Glesner, "Enabling self-reconfiguration on a video processing platform," in *Proceedings of the International Symposium on Industrial Embedded Systems, 2008, La Grande Motte, France.* IEEE, 2008, pp. 19–26.

[105] J. Huang and J. Lee, "A self-reconfigurable platform for scalable dct computation using compressed partial bitstreams and blockram prefetching," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 11, pp. 1623–1632, 2009.

[106] C. Foucher, F. Muller, and A. Giulieri, "Exploring fpgas capability to host a hpc design," in *Proceedings of NORCHIP, 2010, Tampere, Finland.* IEEE, 2010, pp. 1–4.

[107] D. A. Rafi Budruk and T. Shanley, *PCI Express System Architecture.* Colorado Springs, Colorado, USA: MindShare Inc., 2003.

[108] *Universal Serial Bus 3.1 Specification, Revision 1.0*, Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, Renesas Corporation, ST Ericsson, and Texas Instruments Std., 2013.

[109] D. Gohringer, M. Hubner, V. Schatz, and J. Becker, "Runtime adaptive multi-processor system-on-chip: Rampsoc," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2008, Miami, Florida, USA*. IEEE, 2008, pp. 1–7.

[110] D. Gohringer and J. Becker, "High performance reconfigurable multi-processor-based computing on fpgas," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, Atlanta, Georgia, USA*. IEEE, 2010, pp. 1–4.

[111] S. Werner, O. Oey, D. Gohringer, M. Hubner, and J. Becker, "Virtualized on-chip distributed computing for heterogeneous reconfigurable multi-core systems," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, Dresden, Germany*. IEEE, 2012, pp. 280–283.

[112] P. Cvek, T. Drahonovsky, and M. Rozkovec, "Gnu/linux and reconfigurable multiprocessor fpga platform," in *Proceedings of the IEEE 11th International Workshop of Electronics, Control, Measurement, Signals and their application to Mechatronics (ECMSM), 2013, Toulouse, France*. IEEE, 2013, pp. 1–5.

[113] T. D. Nguyen and A. Kumar, "Pr-hmpsoc: A versatile partially reconfigurable heterogeneous multiprocessor system-on-chip for dynamic fpga-based embedded systems," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, Munich, Germany*. IEEE, 2014, pp. 1–6.

[114] C. Claus, W. Stechele, and A. Herkersdorf, "Autovision–a run-time reconfigurable mpsoc architecture for future driver assistance systems," *Information Technology*, vol. 49, no. 3, pp. 181–187, 2007.

[115] M. Rummele-Werner, T. Perschke, L. Braun, M. Hubner, and J. Becker, "A fpga based fast runtime reconfigurable real-time multi-object-tracker," in *Proceedings of the 2011 IEEE International Symposium on Circuits and Systems, Rio de Janeiro, Brazil*. IEEE, 2011, pp. 853–856.

[116] J. Resano, D. Mozos, D. Verkest, and F. Catthoor, "A reconfiguration manager for dynamically reconfigurable hardware," *IEEE Design & Test*, vol. 22, no. 5, pp. 452–460, 2005.

[117] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, 2008, Munich, Germany*. IEEE, 2008, pp. 519–522.

[118] F. Redaelli, M. Santambrogio, V. Rana, and S. Ogrenci Memik, "Scheduling and 2d placement heuristics for partially reconfigurable systems," in *Proceedings of the International Conference on Field-Programmable Technology, 2009, Sydney, Australia*, 2009, pp. 223–230.

[119] A. Ahmadinia, "Integrated scheduling and configuration caching in dynamically reconfigurable systems," in *Proceedings of the International Conference on Intelligent and Advanced Systems (ICIAS), 2010, Kuala Lumpur, Malaysia*. IEEE, 2010, pp. 1–6.

[120] D. Gohringer, M. Hubner, E. N. Zeutebouo, and J. Becker, "Cap-os: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures," in *Proceedings of the IEEE International Symposium on Parallel & Distributed*

*Processing, Workshops and Phd Forum (IPDPSW), 2010, Atlanta, Georgia, USA.* IEEE, 2010, pp. 1–8.

[121] A. Al-Wattar, S. Areibi, and F. Saffih, "Efficient on-line hardware/software task scheduling for dynamic run-time reconfigurable systems," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012, Shanghai, China.* IEEE, 2012, pp. 401–406.

[122] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[123] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for fpga-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, 2007.

[124] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, 2007, Nice, France.* IEEE, 2007, pp. 1–6.

[125] J. Angermeier and J. Teich, "Heuristics for scheduling reconfigurable devices with consideration of reconfiguration overheads," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2008, Miami, Florida, USA.* IEEE, 2008, pp. 1–8.

[126] H. Kooti, E. Bozorgzadeh, S. Liao, and L. Bao, "Transition-aware real-time task scheduling for reconfigurable embedded systems," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, Dresden, Germany.* IEEE, 2010, pp. 232–237.

[127] M. Murgida, A. Panella, V. Rana, M. D. Santambrogio, and D. Sciuto, "Fast ip-core generation in a partial dynamic reconfiguration workflow," in *Proceedings of the International Conference on Very Large Scale Integration, 2006, Nice, France.* IEEE, 2006, pp. 74–79.

[128] D. Gohringer, S. Werner, M. Hubner, and J. Becker, "Rampsocvm: runtime support and hardware virtualization for a runtime adaptive mpsoc," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2011, Chania, Greece.* IEEE, 2011, pp. 181–184.

[129] R. Graczyk, M. Stolarski, and P. Cormery, "Exploratory study about the use of new reconfigurable fpgas in space," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2011, San Diego, California, USA.* IEEE, 2011, pp. 220–226.

[130] J. L. Nunes, "Improving the dependability of fpga-based real-time embedded systems with partial dynamic reconfiguration," in *Proceedings of the 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W), 2013, Budapest, Hungary.* IEEE, 2013, pp. 1–4.

[131] M. Straka, J. Kastil, and Z. Kotasek, "Modern fault tolerant architectures based on partial dynamic reconfiguration in fpgas," in *Proceedings of the IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010, Vienna, Austria.* IEEE, 2010, pp. 173–176.

[132] ——, “Fault tolerant structure for sram-based fpga via partial dynamic reconfiguration,” in *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), 2010, Lille, France.* IEEE, 2010, pp. 365–372.

[133] S. Yousuf, A. Jacobs, and A. Gordon-Ross, “Partially reconfigurable system-on-chips for adaptive fault tolerance,” in *Proceedings of the International Conference on Field-Programmable Technology (FPT), 2011, New Delhi, India.* IEEE, 2011, pp. 1–8.

[134] M. M. Ibrahim, K. Asami, and M. Cho, “Reconfigurable fault tolerant avionics system,” in *Proceedings of the IEEE Aerospace Conference, 2013, Big Sky, Montana, USA.* IEEE, 2013, pp. 1–12.

[135] L. Sterpone and A. Ullah, “On the optimal reconfiguration times for tmr circuits on sram based fpgas,” in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2013, Torino, Italy.* IEEE, 2013, pp. 9–14.

[136] M. Psarakis, A. Vavousis, C. Bolchini, and A. Miele, “Design and implementation of a self-healing processor on sram-based fpgas,” in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014, Amsterdam, The Netherlands.* IEEE, 2014, pp. 165–170.

[137] S. Di Carlo, A. Miele, P. Prinetto, and A. Trapanese, “Microprocessor fault-tolerance via on-the-fly partial reconfiguration,” in *Proceedings of the 15th IEEE European Test Symposium (ETS), 2010, Prague, Czech Republic.* IEEE, 2010, pp. 201–206.

[138] M. Psarakis and A. Apostolakis, “Fault tolerant fpga processor based on runtime reconfigurable modules,” in *Proceedings of the 17th IEEE European Test Symposium (ETS), 2012, Annecy, France.* IEEE, 2012, pp. 1–6.

[139] H. Pham, S. Pillement, and S. Piestrak, “Low overhead fault-tolerance technique for dynamically reconfigurable softcore processor,” *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1179–1192, 2013.

[140] H.-M. Pham, S. Pillement, and D. Demigny, “A fault-tolerant layer for dynamically reconfigurable multi-processor system-on-chip,” in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs, 2009, Cancun, Mexico.* IEEE, 2009, pp. 284–289.

[141] H.-M. Pham, L. Devaux, and S. Pillement, “Re2da: Reliable and reconfigurable dynamic architecture,” in *Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011, Montpellier, France.* IEEE, 2011, pp. 1–6.

[142] J. M. Emmert, C. E. Stroud, and M. Abramovici, “Online fault tolerance for fpga logic blocks,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 2, pp. 216–226, 2007.

[143] *ORCA TM Series 2 Device Datasheet*, Lattice Semiconductor Corporation, 2010.

[144] F. Lahrach, A. Abdaoui, A. Doumar, and E. Chatelet, “A novel sram-based fpga architecture for defect and fault tolerance of configurable logic blocks,” in *Proceedings of the IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010, Vienna, Austria.* IEEE, 2010, pp. 305–308.

[145] F. Lahrach, A. Doumar, E. Châtelet, and A. Abdaoui, "Master-slave tmr inspired technique for fault tolerance of sram-based fpga," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2010, Lixouri, Kefalonia, Greece.* IEEE, 2010, pp. 58–62.

[146] F. Lahrach, A. Doumar, and E. Chatelet, "Fault tolerance of multiple logic faults in sram-based fpga systems," in *Proceedings of the 14th Euromicro Conference on Digital System Design (DSD), 2011, Oulu, Finland.* IEEE, 2011, pp. 231–238.

[147] F. Lahrach, A. Doumar, and E. Châtelet, "Fault tolerance of sram-based fpga via configuration frames," in *Proceedings of the IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2011, Cottbus, Germany.* IEEE, 2011, pp. 139–142.

[148] R. Salvador, A. Otero, J. Mora, E. de la Torre, L. Sekanina, and T. Riesgo, "Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2011, Cancun, Mexico.* IEEE, 2011, pp. 164–169.

[149] A. Gallego, J. Mora, A. Otero, R. Salvador, E. de la Torre, and T. Riesgo, "A novel fpga-based evolvable hardware system based on multiple processing arrays," in *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013, Boston, Massachusetts, USA.* IEEE, 2013, pp. 182–191.

[150] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, C. Braun, M. E. Imhof, H.-J. Wunderlich, and J. Henkel, "Module diversification: Fault tolerance and aging mitigation for runtime reconfigurable architectures," in *Proceedings of the IEEE International Test Conference (ITC), 2013, Anaheim, California, USA.* IEEE, 2013, pp. 1–10.

[151] L. Miculka, M. Straka, and Z. Kotasek, "Methodology for fault tolerant system design based on fpga into limited redundant area," in *Proceedings of the Euromicro Conference on Digital System Design (DSD), 2013, Los Alamitos, California, USA.* IEEE, 2013, pp. 227–234.

[152] S. Di Carlo, G. Gambardella, P. Prinetto, D. Rolfo, P. Trotta, and A. Vallero, "A novel methodology to increase fault tolerance in autonomous fpga-based systems," in *Proceedings of the IEEE 20th International On-Line Testing Symposium (IOLTS), 2014, Platja d'Aro, Girona, Spain.* IEEE, 2014, pp. 87–92.

[153] C. Bolchini and A. Miele, "Design space exploration for the design of reliable sram-based fpga systems," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems, 2008, Boston, Massachusetts, USA.* IEEE, 2008, pp. 332–340.

[154] C. Bolchini, A. Miele, C. Sandionigi, N. Battezzati, L. Sterpone, and M. Violante, "An integrated flow for the design of hardened circuits on sram-based fpgas," in *Proceedings of the 15th IEEE European Test Symposium (ETS), 2010, Prague, Czech Republic.* IEEE, 2010, pp. 214–219.

[155] C. Bolchini, A. Miele, and C. Sandionigi, "A novel design methodology for implementing reliability-aware systems on sram-based fpgas," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1744–1758, 2011.

[156] ——, "Increasing autonomous fault-tolerant fpga-based systems' lifetime," in *Proceedings of the 17th IEEE European Test Symposium (ETS), 2012, Annecy, France.* IEEE, 2012, pp. 1–6.

[157] *A DTV Profile for Uncompressed High Speed Digital Interfaces*, CEA Std. CEA-861-E, 2008.

[158] *Virtex-4 FPGA Configuration User Guide (v1.11)*, Xilinx, 2009.

[159] *Virtex-5 FPGA Configuration User Guide (v3.11)*, Xilinx, 2012.

[160] *Virtex-6 FPGA Configuration User Guide (v3.8)*, Xilinx, 2014.

[161] V. Dumitriu and G. N. Khan, "Throughput-oriented noc topology generation and analysis for high performance socs," *IEEE Transactions on, Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 10, pp. 1433–1446, 2009.

[162] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[163] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits and Systems Magazine*, vol. 4, no. 2, pp. 18–31, 2004.

[164] *MicroBlaze Processor Reference Guide (v14.7)*, Xilinx, 2013.

[165] *Constraints Guide (v14.5)*, Xilinx, 2013.

[166] *Platform Flash XL High-Density Configuration and Storage Device (v3.0.1)*, Xilinx, 2010.

[167] *RGB to YCrCb Color-Space Converter v7.1 LogiCORE IP Product Guide*, Xilinx, 2014.

[168] *Floorplanning Methodology Guide (v14.5)*, Xilinx, 2013.

[169] *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices (v14.7)*, Xilinx, 2013.

[170] *PlanAhead User Guide (v14.6)*, Xilinx, 2013.

[171] V. Kirischian, V. Geurkov, and L. Kirischian, "Cost effective reconfigurable architecture for stream processing applications," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering, 2008, Niagara Falls Ontario, Canada.* IEEE, 2008, pp. 541–546.

[172] *KC705 Evaluation Board for the Kintex-7 FPGA User Guide - UG810 (v1.4)*, Xilinx, 2013.

[173] *ZedBoard (Zynq Evaluation and Development) Hardware Users Guide - Version 2.2*, Avnet Electronic Marketing, 2014.

[174] *ChipScope Pro Software and Cores (v14.3)*, Xilinx, 2012.

[175] R. C. Gonzales and R. E. Woods, *Digital Image Processing*, 2nd ed. Upper Saddle River, New Jersey, USA: Prentice-Hall Inc., 2002.

[176] *ADV7511 - 225 MHz, High Performance HDMI Transmitter with ARC*, Analog Devices, 2010.

[177] *Hierarchical Design Methodology Guide (v14.5)*, Xilinx, 2013.

[178] *Zynq-7000 All Programmable SoC Technical Reference Manual (v1.9.1)*, Xilinx, 2014.

[179] *LogiCORE IP AXI GPIO v2.0 Product Guide*, Xilinx, 2014.

[180] C. Claus, W. Stechele, M. Kovatsch, J. Angermeier, and J. Teich, "A comparison of embedded reconfigurable video-processing architectures," in *Proceedings of the International Conference on Field Programmable Logic and Applications, 2008, Heidelberg, Germany.* IEEE, 2008, pp. 587–590.

# Glossary

**3GPP2** Third Generation Partnership Project 2. 2

**APSK** Amplitude and Phase Shift Keying. 2

**ASIC** Application-Specific Integrated Circuit. 3, 5, 28, 29, 34

**ASP** Application-Specific Processor. xi, xii, xiv, 10, 12, 14, 15, 40–49, 52, 57, 66, 74, 75, 84, 89, 127, 128, 131, 132, 143, 144, 147, 148, 151, 160–162, 178, 179, 188, 189

**AXI** Advanced eXtensible Interface. 154

**BCM** Bit-stream and Configuration Manager. ix, xiv, 84–86, 88, 92, 98, 101, 113–116, 120, 124, 125, 128–131, 134–137, 139, 140, 145, 151, 160, 181, 188

**BIST** Built-In Self-Test. 40, 134, 136–138

**CAD** Computer-Aided Design. 5, 9, 19, 21, 53, 55, 57, 84, 110, 141, 150, 151, 154, 175, 177

**CMFU** Collaborative Macro-Function Unit. viii, xiv, 84–107, 114–117, 119–121, 125, 127–137, 139–141, 145–148, 151, 157, 159, 161–165, 172, 174–177, 179, 183, 184, 188, 189

**CPU** Central Processing Unit. 29, 37

**CRC** Cyclic Redundancy Check. 18, 56

**DCCI** Distributed Communication and Control Infrastructure. viii, ix, xiii, xiv, 84–92, 94–101, 110, 111, 113, 125, 128, 130, 133–135, 140, 151, 159, 188

**DCM** Digital Clock Manager. 156

**DPR** Dynamic Partial Reconfiguration. 12, 18, 19, 21, 25, 35, 53, 55, 133, 134, 141, 183

**DRAM** Dynamic Random-Access Memory. 8, 183, 187

**DVB-SH** Digital Video Broadcasting - Satellite to Hand-held. 2

**ECC** Error-Correction Code. 7, 134

**FAR** Frame Address Register. 22, 56

**FEC** Forward Error Correction. 2

**FIFO** First-In First-Out. 116, 118, 125, 130

**FPGA** Field-Programmable Gate Array. vi, vii, xi, 4–7, 9–13, 16–20, 22, 28, 30–32, 34–37, 39, 42, 45, 47, 48, 52, 53, 61, 64, 86, 99, 114, 116, 124, 125, 128, 132, 139, 141, 145, 155, 156, 172, 176–178, 180, 183, 186, 187

**FSM** Finite-State Machine. xv, 76, 77, 94, 96, 102, 104, 106, 107, 112, 118, 120, 123, 125, 164, 167, 168

**GPIO** General-Purpose Input-Output. 154

**HDMI** High-Definition Multimedia Interface. 145, 146

**HPC** High-Performance Computing. 31

**I/O** Input/Output. 19, 22, 27, 28, 35, 56, 84, 86, 92, 93, 111, 124, 184

**ILA** Integrated Logic Analyzer. 154, 156, 160

**IP** Intellectual Property. 13, 89, 90, 93–98, 130, 151, 190

**JTAG** Joint Test Action Group. 18, 145, 160

**LCCU** Local Connection Control Unit. xiv, xv, 99, 100, 102–108, 110, 111, 128, 130, 131, 134, 136, 137, 139, 144, 147–149, 151, 152, 159, 160, 163–165, 167, 174, 175

**LUT** Look-Up Table. 4, 9

**MACROS** Multi-mode Adaptive Collaborative self-Organized System. vii–xi, xv, 13, 15, 38–40, 42, 43, 49–52, 56–58, 62, 64, 65, 67, 69, 74, 75, 79, 83, 84, 86, 87, 89, 92, 98, 99, 102, 114, 125, 127, 129, 131, 133, 134, 136–144, 147, 149–152, 155, 157–160, 162–165, 168, 170–172, 174, 175, 177–182, 184, 185, 187–191

**MEU** Multiple-Event Upset. 8

**MPEG** Moving Pictures Experts Group. 2

**MPSoC** Multi-Processor System on Chip. 24, 31

**NoC** Network-on-Chip. xi, 24–26, 32, 33, 58, 60, 155, 158, 172–174, 182, 185

**OFDM** Orthogonal Frequency Division Multiplexing. 2

**PCI** Peripheral Component Interconnect. 28–30, 82

**PE** Processing Element. 37

**PLD** Programmable Logic Device. 4

**PRM** Partially Reconfigurable Module. 20, 35–37, 53, 108, 111, 113, 148

**PRR** Partially Reconfigurable Region. 20, 21, 35, 36, 52–54, 57, 61, 101, 108, 148, 175

**PSK** Phase Shift Keying. 2

**QAM** Quadrature Amplitude Modulation. 2

**QPSK** Quadrature Phase Shift Keying. 2

**RAM** Random-Access Memory. 23, 61, 150

**ROM** Read-Only Memory. 97

**SEE** Single Event Effect. 7

**SEFI** Single Event Functional Interrupt. 7, 132, 186

**SEU** Single Event Upset. 7, 36, 132, 133, 186

**SoC** System on Chip. 5, 27, 142, 155, 167, 172–174

**SoPC** System on Programmable Chip. 150

**SPH** Stream Packet Hierarchy. 43

**SRAM** Static Random-Access Memory. 28

**SSI** Stacked Silicon Interconnect. 4

**TDM** Time-Division Multiplexing. 2

**TID** Total Ionizing Dose. 7

**TMR** Triple-Module Redundancy. 36, 134, 180, 181, 189

**USB** Universal Serial Bus. 29, 82