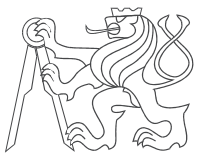
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering



Placement and Routing for Dynamic Reconfiguration in FPGAs

DOCTORAL THESIS

August 2010

Ing. Petr Honzík

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

Placement and Routing for Dynamic Reconfiguration in FPGAs

by

Ing. Petr Honzík

Supervisor: Ing. Jiří Kadlec, CSc

Dissertation submitted to the Faculty of Electrical Engineering of
Czech Technical University in Prague
in partial fulfillment of the requirements
for the degree of

Doctor

in the branch of study
Control Engineering and Robotics
of study program Electrical Engineering and Informatics
August 2010

To my family, grandfather i.m. and Kačka.

Preface

The presented thesis is a part of a long term joint research at the Department of Signal Processing of the Institute of Information Theory and Automation (UTIA) of the Czech Academy of Sciences of the Czech Republic, and the Department of Control Engineering, Faculty of Electrical Engineering of the Czech Technical University in Prague under the supervision of Ing. Jiří Kadlec, CSc. in the field of the development and implementation of advanced digital signal processing (DSP) algorithms for adaptive control and audio and video processing.

The common target platforms for the research are Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs). Matlab/Simulink is used to specify, model and verify algorithms that are later converted and synthesized in HW. As such specialized solutions are likely to be used in embedded systems, features that result in extremely fast execution, use small amount of memory, small chip area or low power consumption are also researched.

I joined the Department of Signal Processing at UTIA in 2003, when the EU RECONF2 project was intensively researched. Later in the IST project Æther I learned about self-adaptive systems and made use of my previous results from the previous projects. In the year 2005 a cooperation with the Atmel Corporation was established and allowed me to research the FPGA technology at the low level. The last project Scalopes allowed me to continue in the research oo self-adaptive networks.

The presented work deals with reconfigurable systems with self-adaptivity based on the FPGA technology. The thesis consists of three parts. The first part deals with partial dynamic reconfiguration of FPGA devices. The second part analyzes self-adaptive systems, their elements and features. The third part introduces a network on chip, and analyzes it in terms of communication, hardware cost and data stream processing.

First of all I would like to express my gratitude to my supervisor Jiří Kadlec for supporting me during all my research and becoming my supervisor, to Martin Daněk for introducing me to reconfigurable and adaptive systems and for useful discussions in all stages of my research, to Rudolf Matoušek for introducing me to FPGAs. My thanks also belong to all the people I have met during my research.

My deepest thanks belong to my family and grandfather, in memoriam, for supporting

me when I needed it.

This work was supported by the following institutions which I gratefully acknowledge:

- the European Commission under the FP6-IST projects Reconf IST-2001-34016, Æther IST-027611.
- the Artemis JU under the project Scalopes 2008-100029 and MSMT 7H09005.
- the Ministry of Education under the projects C-A-K LN00B096 and 1M05667

Petr Honzík

Prague, August 27, 2010

Placement and Routing for Dynamic Reconfiguration in FPGAs

Ing. Petr Honzík

Czech Technical University in Prague,
Institute of Information Theory and Automation of the ASCR

Supervisor: Ing. Jiří Kadlec, CSc
Institute of Information Theory and Automation of the ASCR

The presented work deals with reconfigurable systems with self adaptivity based on the FPGA platform. The thesis consists of three parts.

The first part deals with partial dynamic reconfiguration on the FPGA devices. The possibility of the dynamic reconfiguration in the reconfigurable systems and its space complexity is analyzed. The function density that expresses an application performance running in a dynamic module is presented. Further the text presents reconfigurable hardware platforms available on today's market and the methodology how to implement the reconfigurable flow and the reconfigurable hardware. It introduces a reconfiguration controller and its features necessary to control the reconfiguration process and store configuration bitstreams in an external memory. The problems with connections between a static and dynamic parts of the design during reconfiguration is presented. The two reconfigurable coprocessors with an identical function on Virtex from Xilinx and on AT94K FPSLIC from Atmel have been implemented. The comparison of these two implementations is done.

The second part analyses self adaptive systems, their elements and features. The analysis of the requirements of the self adaptive system is done with respect to the future implementation on the reconfigurable platforms based on the FPGA devices. An introduction of principles of the self adaptive element that is the basic building block of our adaptive system is done. There is a brief description of the four main blocks of the self adaptive element and their interaction with the environment. The next part describes

an implementation of a self adaptive ring network with four self adaptive elements. The simulation of the self adaptive features of the network is done.

The third part introduces a network on chip, analyzes it from the side of communication and hardware cost and data stream processing. The stress is put on restrictions due to the FPGA technology. The 2D-Mesh topology was chosen as the most suitable topology for the future simulation of the self adaptive system. Three routing algorithms and their impact to full loading network in the 2D-Mesh network are presented. The following text describes three placement algorithms and the Step-Adaptive Algorithm for improving the placement of running applications on the network and optimization criteria are defined. The simulation framework is used to test the features of the self adaptive system on test cases. The result of the simulations on the self adaptive system compares the Step-Adaptive Algorithm and the presented placement algorithms.

Contents

Preface **i**

Abstract **iii**

List of Acronyms **xiii**

1 Introduction **3**

1.1 Goals and Objectives of the Dissertation 5

1.2 Current State of the Art 9

1.3 Structure of the Disertation 13

2 Dynamic Reconfiguration Analysis **15**

2.1 State of the Dynamic Reconfiguration 16

2.2 Theory of the Dynamic Reconfiguration 17

2.3 Application Types 21

2.4 Thinking Dynamically From the Outset 25

2.5 Summary 25

3 Reconfigurable Hardware Platform **27**

3.1 Reconfigurable FPGA Devices 28

3.2 Implementation Issues 32

3.3 Reconfigurable FPGA Coprocessor 38

3.4 Comparison FPGA Coprocessor 45

3.5 Summary 47

4 Self-Adaptivity **49**
4.1 Requirements of a Self-Adaptive System 50
4.2 Architecture of a Self-Adapt Element 51
4.3 Function Block of a Self-Adapt Element 52
4.4 Modeling and Implementation 57
4.5 Summary 61

5 Network on Chip **63**
5.1 Network on Chip Topology Selection 63
5.2 2D-Mesh Topology 69
5.3 Network on Chip Characteristics 71
5.4 Summary 75

6 Placing Applications **77**
6.1 Placing Tasks to Nodes 78
6.2 Network Parameters Evaluation 86
6.3 Self-Adaptive Placement 88
6.4 Simulating the Self Adaptive Placement 93
6.5 Simulation Results 112
6.6 Summary 113

7 Conclusion **115**
7.1 Objectives Revisited 118
7.2 Summary of Author's Contribution 120

Appendix: **121**

A Network Traffic Visualization **121**
A.1 Network Notation 121
A.2 Network Parameters Expression 121

B Testing Applications **125**

C Progress Graphs **129**

Biblioghraphy **144**

List of Author's Publications **I**

Funding and Projects **V**

Vita **IX**

List of Figures

2.1 A typical stream-type application. 22
2.2 A dynamic reconfigurable stream type application 22
2.3 A typical control-type application. 24

3.1 Xilinx Virtex2 - configuration blocks. 30
3.2 Bus Macro element. 31
3.3 Atmel AT94K FPGA with AVR and peripherals. 32
3.4 Implementation-dependant flow. 33
3.5 Bitstream logical organization. 35
3.6 Possible port connections and routing 37
3.7 Synthesizing an invalid component. 37
3.8 Floating-point unit: design structure. 40
3.9 Microcontroller execution. 41
3.10 FP coprocessor on Virtex2 with MicroBlaze: block diagram 42
3.11 FP coprocessor on Atmel AT94K: block diagram 43

4.1 Self-Adapt Element block diagram 52
4.2 Reconfigurable computing engine 53
4.3 Implementation of Self-Adapt Element 56
4.4 Model of a Self-Adapt Element network 57
4.5 Ring network with four Self-Adapt Elements 58
4.6 Interpretation of simple packet structure 60
4.7 Consecutive processing data in network 61

5.1 The overview of hardware cost of topologies 68

5.2 The overview of communication bandwidth of topologies 68
5.3 Routing flits in 2D-mesh network 70
5.4 Random routing flits in a full load network 71

6.1 An example of a hop and unused node matrix 81
6.2 Result of First Node Placement algorithm 83
6.3 Result of Best Node Placement algorithm 84
6.4 Result of Multi-Best Node Placement algorithm 85
6.5 Best node algorithm example 85
6.6 Rules for moving tasks 91
6.7 Screeshot of the MeshViz simulation framework 95
6.8 Compare of average approximation 105
6.9 Evolution of six stream applications 107
6.10 Evolution of two stream applications 109

A.1 Network notation and IO ports 122
A.2 Node and ports vizualization 122
A.3 Network graf visualization 123

B.1 Example of test application 128

C.1 Network and applications path cost progress 131
C.2 Graphs of SA alg. for S1, S2, S3 with path cost method 132
C.3 Graphs of SA alg. for S1, S2, S3 with hops cost method 133
C.4 Graphs of SA alg. for S4, S5, S6 with path cost method 134
C.5 Graphs of SA alg. for S4, S5, S6 with hops cost method 135
C.6 Graphs of standard deviation with hop cost calculation 136
C.7 Comparison of improvements network parameters 137

List of Tables

2.1 Summary of the reconfiguration methods 20

3.0 Parameters that affect reconfiguration 29
3.1 Summarise the device characteristics 45
3.2 The resource usage in the FP coprocessor application 46
3.3 Data pertinent to dynamic reconfiguration 46

5.1 Comparison of the most common NoCs 67

6.1 Result of Step Adaptive alg. with path cost on sets S1, S2, S3 98
6.2 Approximation of Step Adaptive alg. with path cost on sets S1, S2, S3 . . . 98
6.3 Result of Step Adaptive alg. with path cost on sets S4, S5, S6 99
6.4 Approximation of Step Adaptive alg. with path cost on sets S4, S5, S6 . . . 99
6.5 Result of Step Adaptive alg. with hop cost on sets S1, S2, S3 100
6.6 Approximation of Step Adaptive alg. with hop cost on sets S1, S2, S3 . . . 100
6.7 Result of Step Adaptive alg. with hop cost on sets S4, S5, S6 101
6.8 Approximation of Step Adaptive alg. with hop cost on sets S4, S5, S6 . . . 101
6.9 Average of Step Adaptive alg. with path cost on sets S1, S2, S3 102
6.10 Average appr.of Step Adaptive alg. with path cost on sets S1, S2, S3 102
6.11 Average of Step Adaptive alg. with hops cost on sets S1, S2, S3 102
6.12 Average appr.of Step Adaptive alg. with hops cost on sets S1, S2, S3 103
6.13 Average of Step Adaptive alg. with path cost on sets S4, S5, S6 103
6.14 Average appr.of Step Adaptive alg. with path cost on sets S4, S5, S6 103
6.15 Average of Step Adaptive alg. with hops cost on sets S4, S5, S6 104
6.16 Average appr.of Step Adaptive alg. with hops cost on sets S4, S5, S6 104

6.17 List of applications run on the network 110

List of Acronyms

ASIC	Application–Specific Integrated Circuit
AVR	8-bit Microcontroller Unit from Atmel
BRAM	Block RAM
CLB	Configurable Logic Block
DFF	D–Type Flip Flop
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDK	Embedded Development Kit
FPGA	Field–Programmable Gate Array
fps	frames per second
FPSLIC	Field–Programmable System–Level Integrated Circuits
GCC	GNU C Compiler
HDL	Hardware Description Language
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IOB	Input/Output Buffer
IP	Intellectual Property
ISE	Integrated Software Environment
JTAG	Joint Test Action Group
LUT	Look–Up Table
MB	MicroBlaze soft–core processor

MIPS	Million Instructions Per Second
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
SoC	System on Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TRS	Technical Requirement Specification
TWS	Two–Wire Serial port
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High–Speed Integrated Circuit
VLSI	Very Large–Scale Integration
XPS	Xilinx Platform Studio
XST	Xilinx Synthesis Tool

Chapter 1

Introduction

The era of a single computer per one person became on era of pervasive computing with many computers per one person. The beginning of the era of pervasive computing started with an expansion of embedded systems and mobile systems. Current products have to process more and more complex tasks and fulfill requirements from users that put more demands on communication interfaces and cooperation among products.

The near future will increase the number of not only single computers, but mainly multiprocessors and systems on chip with heterogeneous hardware cores. These cores will have to process a huge amount of various data coming from the environment. Current cores can not fulfill requirements of the future multi-cores that will work independently of any central controller, and can adapt to environment that will change during life time of system.

The higher requirements on the future multi-cores will require much wider functionality with low power consumption and mainly reuse of the hardware resources. Today's computing systems have low hardware resource utilization, and the specific hardware pays for high performance by low utilization. This increases the cost of the hardware and power consumption. The future hardware has to have reconfiguration features that allow the change hardware functions of parts or the whole system without interrupting its run. This process will significantly increase the functionality and utilization of hardware resources and decrease power consumption.

The current mainly centralized systems depend on central arbiters or controllers that synchronize and control the whole system. It leads to inefficiency of utilization of hardware

and bottlenecks between parts of the system. The future systems and their parts have to implement some level of self adaptivity to avoid the central controlling mechanism. It will simplify the cooperation between parts of the system, and minimize problems with failures of part of the system.

An example of a system with the multi-core concept is the IBM Cell processor (Kahle et al., 2005) or the NVidia GPU with CUDA architecture (Zou et al., 2009). They implement a multiprocessor unit with several smaller processors on one chip that cooperate to finish the task. They still use a central arbiter to control the whole system and one communication bus, but some of the features became very near the requirements of the future systems on chip.

We will use programmable logic array devices known as FPGAs that allow to implement most of the future requirements on today's devices. They cannot be compared with multiprocessors like IBM Cell and NVidia GPU with CUDA architecture but they can be used as an ideal starting platform that can prove methodology and features we find desirable in future systems.

The FPGA devices allow fast prototyping of the hardware, and open fields for partial dynamic reconfiguration that can be used to change functions of parts of a system on chip. The current FPGA devices can contain several smaller microcontrollers with attached hardware accelerators that can act as a strong platform for future systems on chip with many of the features we described above. Unfortunately the speed of the FPGA devices cannot be compared with ASIC devices. The operating frequency of the current FPGA devices reaches 250MHz, in the case of the ASIC devices it is 20 times higher, but the functionality of the FPGA devices with partial reconfiguration is much higher than the functionality of the ASIC devices. The ASIC devices have a fixed structure designed during the factory process and cannot be changed during the life time of the device. More, the design of a new ASIC is extremely expensive and cannot fulfill various customer requirements for low volume production.

The future solution that can merge advantages of both device types can bring a new dimension to reconfigurable devices. A real example of the first solution of the combination of an ASIC device with an FPGA device is Atmel AT94K FPSLIC that has a hard core AVR type microcontroller connected to the programmable gate array. Another solution is the Xilinx Virtex device with the hard core Power PC processor attached to

the programmable gates array. The future will use this model on a single chip to multi cores with many elements containing microcontroller with reprogrammable hardware accelerators. We can call this multi-core concept reprogrammable basic computing element array. Such a reprogrammable array will be very close to our model of the future system on chip we discussed above. With the self-adaptive technology and scalability it can fulfil all requirements of future Multi-Processor Systems on Chip that will be able to adapt to their environment.

1.1 Goals and Objectives of the Dissertation

In our work we plan to use the partial dynamic reconfiguration technology on reconfigurable devices and use this technology to extend the functionality of the current commercially available devices. The extended functionality will be focused on the use of self-adaptive features that allow to design hardware with adaptive functions.

The current FPGA technology opens possibilities to implement reconfigurable devices that allow to change the function of part of devices without stopping the whole device. The design of reconfigurable devices still isn't well supported by commercial tools, and there isn't clear methodology that would lead to a successful partial dynamic reconfigurable design with acceptable cost and short time to market.

The work will analyze partial dynamic reconfiguration and describe main barriers that prevent to use this technology and its features widely. The work brings overview of the partial dynamic reconfigurable methodology and proposes solutions to the barriers. As a proof that the proposed solutions can lead to successful partial reconfigurable design the work will implement a reconfigurable platform with these features on two commercially available FPGA devices from different suppliers.

The analysis and proposed solutions with implementation of the two platforms is the first main contribution of the work. On this building block the work will continue on the level of self adaptivity that can fully use all features of partial dynamic reconfiguration and brings wide extension for future designs with self-adaptive features.

The current microcontrollers and CPUs became a bottleneck for parallel algorithms mainly in the case of embedded systems and mobile systems. The FPGA technology can bring solutions in this area because the FPGA devices offer parallel processing on the

hardware level. The increasing size of the memory will increase the logic size of the FPGA devices. The reconfigurable features bring to FPGA devices same extensions of functionality as task switching to CPU technology years ago. We can observe a combination of the hardware coprocessors and microcontroller on one chip in many System on Chip designs that became very useful in embedded systems. It is supposed that in the near future this combination of hardware accelerators with a microcontroller will become widely used and they will compose extensive systems containing hundreds of such hybrid microcontrollers cooperating as one unit facing outside as an adaptive hardware block that will be able to adapt its function according to the requirements from the environment. The first designs in this direction is the IBM Cell processor and the NVidia GPU with CUDA architecture.

The work will introduce the structure of a self-adaptive element that uses the combination of a hardware accelerator with a microcontroller and self adaptivity that are ensured by partial dynamic reconfiguration. Because the era of pervasive computing will become a reality in the near future, we will design the self-adaptive element with respect to connecting each other together and allowing them to cooperate as one complex system that propagates self adaptivity from the elements to the whole system.

Because of the size of the current FPGA devices and because of the implementation demand the last part of the work will be realized only in a simulation tool designed for this purpose. The work will design the most suitable network on chip topology that can connect self-adaptive elements together and offer scalability of the whole system.

The last part of the work will introduce an adaptive algorithm that can be used a basic self-adaptive feature of the network of the self-adaptive elements. The test and comparison of various parameters of the self-adaptive algorithm will be done to prove that systems with such features can bring improvement to a self-adaptive network.

The work will bring a new view on the use of partial dynamic reconfiguration combined with the self-adaptive features and implemented on the network with hybrid computing elements that can change their function according to the requirements from the environment, and they are able to process massive parallelism in hardware accelerators.

Summary of the Objectives

In this section, we summarize and specify the objectives of this thesis in detail. The list of tasks to be accomplished is as follows:

- To introduce a new technology based on partial dynamic reconfiguration we perform a low level analysis of the reconfiguration process and design a method that leads to designing reconfigurable hardware on reconfigurable devices.

- To increase variability and adaptability of the reconfigurable hardware we analyze the function of the partially reconfigurable hardware and its restrictions and extensions.

- To open a new reconfigurable platform the methodology to design hardware is modified to cover specific steps that allow to implement designs with partial reconfiguration.

- To validate the possibility of the using partial dynamic reconfiguration the reconfigurable coprocessors with reconfigurable features will be built on two commercially available hardware platforms that allow to implement the reconfiguration process.

- To increase the variability of the reconfiguration features the self-adaptive element will be designed as the basic building element of self-adaptive systems based on the reconfigurable hardware. The self-adaptive element will be designed as an independent core that will be able to adapt its function according to the requirements from the environment.

- To create a suitable environment for the self-adaptive elements the analysis of the current network on chip topologies suitable for reconfigurable hardware will be performed. According to this we will choose the best network topology for cooperative self-adaptive elements.

- To use the chosen network the basic parameters and placement algorithms have to be introduced. They ensure first placement of application in the network and its start and interaction with an already running applications in the network.

- To measure and evaluate the effectiveness of using the network cost functions will be set up. They will drive the adaptation process.

- To improve efficiency of the running network that is fragmented by placement and releasing applications an adaptive algorithm has to be designed. The adaptive algorithm will be part of each node in the network and guarantee that nodes will adapt

their function to the most suitable function for a given case in the possible range of the node's neighborhoods.

Scientific Originality and Innovation

The work focusses on embedded systems with FPGA devices used in System on Chip designs. The System on Chip became a progressive technology that solves problems with opposing requirements of high performance and low power consumptions for embedded and mobile systems.

The aim of this work is to put together partial dynamic reconfiguration on FPGA devices and self-adaptive features that are opened for FPGA devices with reconfiguration. The work builds on the fact that future FPGA devices will increase their logic space and contain a huge amount of cores with hardware accelerators. These cores will need to increase their functionality, and they will need a mechanism to adapt their function to the requirements of the environment.

The work introduces method of dynamic reconfiguration and uses this exploration in the area of reconfigurable FPGA devices for designing a structure of a universal self-adaptive element that will be the basic computing element in future computing networks on chip. These combinations bring new pieces of knowledge for the future approach to pervasive computing that is the next generation of the design era.

The particular innovative solutions include:

- Introduction of three methods of the dynamic reconfiguration and definition of their space complexity. Analysis of the function density that defines the application performance of a dynamic module.

- Design of a reconfigurable coprocessor for universal use in parallel computing. Implementation and verification of the reconfigurable coprocessor on two independent FPGA platforms.

- Structure of the self-adaptive element based on the reconfigurable coprocessor and its verification on the ring bus topology.

- Integration of the self-adaptive elements in a scalable network on chip with monitoring their interactions with an environment.

- Development of the Step Adaptive algorithm for improving parameters of the running network on chip with self-adaptive elements and the dynamic reconfiguration technology.

1.2 Current State of the Art

The current situation in reconfigurable hardware mainly on the FPGA devices is influenced by the need for high hardware resource reuse, high computing power and low power consumption. The last two factors go against each other, and can be solved by new trends in the silicon technology or by reusing hardware resources. Increasing hardware resource reuse leads to the reconfiguration process in hardware. Two main trends are preferred today: connected microcontrollers on one chip and connected heterogeneous cores on one chip. We will focus on heterogeneous cores on one chip, and their adaptivity to requirements that come from the environment. The adaptivity increases hardware resource reuse and increases the functionality of the whole system on chip.

The future designs that will build on many-core platforms and schemas will need to handle these cores as one unit working in parallel on many small tasks that come from outside of the unit. The self-adaptivity feature of each element in the unit is one approach how to manage the whole unit.

The following sections will discuss the research that is relevant to the presented work.

Reconfigurable Hardware

The current state of reconfigurable hardware comes mainly from the state of the FPGA devices. They are the main representative devices of the reconfigurable hardware design. Recently the design tools became open to the technology of partial dynamic reconfiguration. But still there is not a clear methodology that can lead to a sufficient reconfigurable design. As an example of the partial dynamic reconfigurable implementation is the Gecko design from IMEC, Belgium (Verkest, 2003) that uses tiles in an FPGA device as reconfigurable blocks with variable functions. An implementation of a notetaker for blind people (Daněk et al., 2005) is another example of partial dynamic reconfiguration of FPGA devices.

The last way of using reconfigurable hardware is a reconfigurable coprocessor with

high functional density attached to a processor. The hardware coprocessor works as a slave for an embedded microprocessor that calls services on the hardware coprocessor. They have local memory to share data and control signals. The reconfiguration process is hidden from the designer and it is covered by software functions. The designer can call functions that guarantee right function of the reconfigurable coprocessor. The paper (Huang and Hsiung, 2009) shows a reconfigurable system with virtualization of the partial dynamic reconfiguration. The paper (Danek et al., 2008) shows a reconfigurable hardware accelerator based on the FPGA device that increases the level of abstraction.

The reconfigurable hardware can be used to increase fault tolerance of hardware. The reconfiguration process can change a faulty part of the hardware. The paper (Straka et al., 2010) presents a modern fault tolerant architecture using reconfigurable hardware for increasing fault tolerance of the architecture. The paper (Kafka, 2008) analyses applicability of partial runtime reconfiguration in fault emulators based on FPGAs. It uses reconfiguration for loading emulators and injecting faults in the emulated circuit.

Another type of the reconfigurable hardware with a different approach to hardware is the plastic cell architecture PCA (Ito et al., 1998). The PCA combines the object model and communication on special hardware that allows to allocate part of the hardware matrix to objects and send communication messages through the hardware to another allocated object. The plastic cell contains small hardware that can implement basic gates like D, JK or LUT.

Multi-core Hardware

The current multi-core hardware can be split to two classes. The first class contains multi-core designs based on field of microprocessors connected by any type of communication bus. This concept comes from the multi-computer schema with several boards connected together. A typical example of an implementation of the multi processor design on chip is the IBM Cell processor (Kahle et al., 2005) or the NVidia GPU with CUDA processor (Zou et al., 2009). They use the model of several processors with local memory connected together by a communication bus. The bus is controlled from one place that creates a centralized mechanism for the whole design.

The second class contains heterogeneous multi-core designs based on computing elements that cooperate to increase the computing power. Such designs often contain some

type of a microcontroller and hardware accelerators that form computing elements. Many of these elements on a single chip are connected together by a communication bus. The communication bus has a scalable topology that tries to avoid any centralization and data overloading.

A typical heterogeneous multi-core design is a reconfigurable system on chip with the Morpheus architecture (Kuhnle et al., 2008). The Morpheus architecture contains heterogeneous reconfigurable engines and on chip memory connected together by the spidergon topology bus. The NEC Dynamically reconfigurable processor (DRP) (Suzuki et al., 2004) is a typical mesh heterogeneous multi-core design with scalable processing elements. The matrix of processing elements contains a special reconfigurable microcontroller connected from sides with horizontal and vertical memories.

Chip vendors open new fields of the custom ASIC devices with the possibility to change their basic functions inside the chip. The chip contains a scalable network with basic computing elements that can be chosen before its fabrication. The ST Microelectronics platform P2012 can offer such services.

Self-Adaptivity on Hardware

The main characteristic of the current self-adaptive system is the capability to determine its function or configuration at a given time in an autonomous and distributed way. The self-adaptive features have to be supported on the hardware level and by the architecture intended to be used as an efficient platform for self-adaptive systems.

The main benefit of such computing architectures against conventional architectures is the possibility to delegate part of the operational and functional specifications of an application to the computing resource itself. This enables the handling and managing of an increasingly complex environment of software, hardware and communication infrastructure. Therefore a self-adaptive computing device must embedded all the necessary facilities (hardware, software, communication) so as to autonomously perform the trade-offs and resource optimisation required by the adaptation process.

The self-placement and self-routing processes on the multi-core hardware allow to perform complex functionality changes in real-time. In the current FPGA devices these programmed changes consist of modifying the configuration of an FPGA configuration memory by means of a configuration manager.

The research on the FPGA and self-adaptivity described in (Casas et al., 2007) introduces a self-adaptive architecture based on the FPGA devices. It is an example of the implementation with self-adaptivity in hardware. Self-adaptivity opens fields for increasing fault tolerance of the hardware. The current implementation of self-adaptivity and fault tolerance in hardware can be found in (Soto et al., 2009).

Network on Chip

Interconnection became an important part of the design with the increasing number of cores on chip. Mainly the network on chip became frequently used with the development of the multi-core field. For this purpose the scalability and decentralization are the most important features of a network on chip. In current designs with a higher number of multi-cores, the mesh type of the network became most suitable. It offers a simple routing mechanism.

The basic paper in the network on chip is (Salminen et al., 2007) that defines the basic properties of the network on chip paradigm and compares network topologies suitable for FPGA devices.

The network topology strongly influences the parameters of the network and at the same time the connected computing elements. Mostly the topology is a compromise between the hardware cost and data throughput. When only several cores are connected together in most cases the ring topology is used as in the multi-processor IBM Cell (Kahle et al., 2005) that uses four independent rings to connect all computing and IO blocks.

Designs with a huge number of cores use mainly the mesh topology. The mesh topology offers cores and communication throughput scalability and are very efficient in point-to-point communication. The paper (Strunk et al., 2009) presents a reconfigurable mesh topology implemented on the FPGA device.

Recent designs use the mesh network to connect soft-core microcontrollers on FPGA devices to increase the computing power and reconfiguration. The paper (Giefers and Platzner, 2010) uses the soft-core Microblaze microprocessors and hybrid interconnect to realize a reconfigurable mesh network.

1.3 Structure of the Disertation

The dissertation thesis is divided to seven chapters. The chapters start with a low level analysis of dynamic reconfiguration in FPGA devices, continue with a design of a reconfigurable platform, and finish with a design of a reconfigurable network of self-adaptive elements that are able to adapt their function to the requirements of the environment.

Chapter 2. In this chapter dynamic reconfiguration of FPGA devices are analyzed. Dynamic reconfiguration extends their functionality and increases their function density. Further the chapter analyzes the space complexity of the dynamic reconfiguration. Three methods are presented for reconfiguration of a dynamic module. Later the text analyzes function density that expresses an application performance running in the dynamic module. The end of the chapter presents two types of applications. The stream-type application allows to process data in batches by different functions, and the control-type application changes the function in the dynamic module according to requests from external devices.

Chapter 3 This chapter presents reconfigurable hardware platforms available on today's market and the methodology implementing the reconfigurable flow and reconfigurable hardware. It introduces the reconfiguration controller and its features necessary to control the reconfiguration process. Further the chapter describes three ways to store the configuration bitstream in an external memory and its impact on speed of the reconfiguration process. Later the text introduces problems with connections between the static and dynamic parts of the design during reconfiguration. It describes how to solve problems with the floating connection lines and how to unify interfaces of different dynamic modules by using a wrapper module. Based on the analysis done before, the chapter implements two reconfigurable coprocessors with identical functions in Virtex from Xilinx and in AT94K FPSLIC from Atmel; these two implementations are compared.

Chapter 4 This chapter presents a self-adaptive system, its elements and features. The analysis of requirements of the self-adaptive system is done with respect to a future implementation in the reconfigurable platforms based on the FPGA devices. Further, the text introduces principles of the self-adaptive element that is the basic building block of our adaptive system. There is a brief description of the four main blocks of the self-adaptive element and their interaction with the environment. The end of the chapter describes the

details of the self-adaptive element and its implementation including the building blocks and reconfigurable parts.

Chapter 5 This chapter presented a network on chip analysis from the point of communication, hardware cost and video processing. The stress is put on FPGA restrictions. The 2D-Mesh topology is chosen as the most suitable for future simulation of the self-adaptive system. Three routing algorithms and their impact on a fully loaded 2D-Mesh network are presented. The end of the chapter describe details of the network we will use in future simulations. The definition of the following network parameters used to evaluate network performance during the simulation is presented: link capacity, communication latency and the move function delay.

Chapter 6 This chapter deals with the network model and application model. The placement process is presented and three placement algorithms are designed to inject applications in the network. Cost values are defined for measuring effectiveness of the network use. The text describes the Step-Adaptive Algorithm for improving the placement of applications running in the network and measures are defined. The simulation framework is used to test the features of the self-adaptive system on test cases. The result of the simulations of the self-adaptive system compares the Step-Adaptive Algorithm and placement algorithms.

Chapter 2

Dynamic Reconfiguration Analysis

This chapter analyzes the dynamic reconfiguration. The following text discusses the FPGA devices and their possibilities with respect to the dynamic reconfiguration and the present technology state with a brief overview of the dynamic reconfiguration parameters and their characteristics. At the end of the chapter the application type analysis is done.

The main part of the work in this chapter on dynamic reconfiguration was done by author and colleagues in the EU research project RECONF 2 (nr.IST-2001-34016), see description of the project on page V. The project was directly focused on design methodology and environment for dynamic reconfigurable FPGA and their implementation on FPGAs available on market.

The research in the area of dynamic reconfigurable FPGAs has several significant papers that define fundamental of the partial reconfiguration on FPGA. Some of them are (DeHon and Wawrzynek, 1999), (Banerjee et al., 2005) and (Liu and Wong, 1999). The partial dynamic reconfiguration on the FPGA brings many difficulties that must be solved for sufficient introduction technology to market. Author and colleagues describe and solve several of them in RECONF 2 project. The following chapter and papers (Daněk et al., 2004) and (Bartosinski, Daněk, Honzík and Matoušek, 2005*b*) describe suggested solutions and methodology for the partial dynamic reconfiguration.

2.1 State of the Dynamic Reconfiguration

The FPGA devices present an important direction in the evolution of the VLSI devices. They allow to design VLSI circuits effectively and quickly with minimal requirements for production costs. The FPGA devices have few advantages like short design cycle and reusability in the case of SRAM-based FPGA devices. On the other hand, there are several disadvantages when compared with application-specific integrated circuits – ASICs for example, worse performance, lower operating frequency and power consumption. Because of the production costs FPGAs are not used in productions that exceed 10000 pieces (Wu et al., 1998).

The FPGA devices have two major advantages. First, the FPGA technology follows the RAM process development curve that goes down much faster than the processor development curve. The performance of the FPGA devices grows faster than that of the processors. The second advantage relies on the dynamic reconfiguration. It can be used to increase the functional density compared to the ASIC devices, and decrease power consumption.

There is a group of applications that have to be solved in hardware because the data throughput to be processed is too big for a software solution. Such implementations open new possibilities for the FPGA devices and mainly for their dynamic reconfiguration. An example is a portable video device with low consumption, or general data streaming applications.

Complexity of the Dynamic Reconfiguration

The dynamic reconfiguration of the FPGA devices was introduced few years ago and a lot of research groups have worked on the connected problems. Many problems have been solved since the first attempt to dynamic reconfiguration was done (DeHon and Wawrzynek, 1999). But introducing such a technology to a wider engineer society and designing robust applications with dynamic reconfiguration are still hard problems, and a lot of barriers is on the way. Today's main problems are the lack of design tools provided by several companies and the absence of good guidelines for the dynamic reconfiguration design flow.

The actual state of the practical dynamic reconfiguration of the FPGA devices is

captured in few applications introduced by several research groups (see Section 3.4). These applications were designed in spite of the inconvenient design tools and guidelines. To offer dynamic reconfiguration to the wide engineering society a lot of work still has to be done.

The area of using dynamic reconfiguration is limited by the current size of the devices and the speed of the reconfiguration process. The present trend is to reconfigure a device less often because of the slow speed of the reconfiguration process (see Section 2.2). This can be solved by using multi-context FPGA devices that can switch between two configuration contexts in one clock cycle.

2.2 Theory of the Dynamic Reconfiguration

This section analyzes dynamic reconfiguration of the FPGA devices from the theoretical point of view, and shows its problems, limitations and possible solutions. Several possible solutions to the dynamic reconfiguration process and the organization of the configuration bitstreams are described. Further, it describes how functional density can be increased by dynamic reconfiguration and the limitations for two platforms available on the marked. The following notation is used for analyze parameters of the dynamic reconfiguration on FPGA:

s	=	The dynamic slot on the chip
t_s	=	The time required to set all bits in the slot s.
t_p	=	The time to change one dynamic module to another by a differential bitstream.
t_l	=	The time needed to set a new module from the empty module.
t_o	=	The time to set the empty module from the last module.
t_c	=	The configuration time of the dynamic module.
t_e	=	The execution time of the dynamic module.
A_s, A_d	=	The area of static A_s and dynamic A_d part of design.
C_A	=	The application cost.
D_A	=	The application density.
F_{max}	=	The maximal frequency of incoming data batches in stream-type applications.

Types of the Dynamic Reconfiguration

The biggest part of the FPGA device is a slot for dynamic modules. The slot can be organized in one big part or more smaller ones over the whole FPGA device. This allows to use more dynamic modules that run concurrently or to have just one big dynamic module over the whole free area inside the FPGA device (Banerjee et al., 2005), (Handa and Vemuri, 2004), (Liu and Wong, 1999). The dynamic reconfiguration process changes the FPGA configuration memory that belongs to the slot according to the configuration data of the required dynamic module. The dynamic modules are stored in an external memory. During the dynamic reconfiguration process the static part of the FPGA device keeps running. Only the processed slot that is reconfigured is disconnected from the static part of the FPGA device and from other slots if there are any. Loading of a new dynamic module consists of changing each bit over the whole slot to the required value. t_s is a time required to set all bits in the slot s. It is the worst case time for the reconfiguration of the slot s. There are lots of cases during reconfiguration processes when it is not necessary to change all bits in the slot, because many of them can have the same value like the previous dynamic slot. Unfortunately, each dynamic slot has a different set of unchanged bits associated with a specific dynamic module. That's why the new dynamic module has to rewrite all bits in the slot and it requires the worst time t_s. In the case of n dynamic modules the space complexity is $O_1(n)$.

There is another possibility to change a dynamic module in slot (Honzík, 2005). The previous text said that each dynamic module has a set of bits that have the same value in other dynamic modules. This suggests a differential configuration bitstream that describes just bits with different values from the last dynamic module in the slot. In some cases a differential bitstream can occupy very small memory space, but in other cases its size can be equal to the size of the whole slot. The time t_p is a time necessary to change one dynamic module to another by a differential bitstream. The relationship between t_s and t_p can be described as follows: $t_s \geq t_p > 0$. It is easy to find that t_p can not be 0, because it would mean a configuration to the same dynamic module. The strategy to change dynamic modules with differential bitstreams seems well-suited to decrease the time complexity of the reconfiguration, because the time t_p isn't worse than t_s, and in many cases it is better than t_s. But the space complexity is not so good. It is necessary to create differential bitstreams for reconfiguration among all n dynamic modules. In the

case of few modules this is possible, but with an increasing number of dynamic modules the space complexity increases too fast. In this case the space complexity for this case can be described as follows:

$$2(\sum_{k=1}^{n}(n-k)) \approx O_2(n^2)$$

The last possible way to solve the problem with dynamic modules is a combination of the two aforementioned methods. Such types of dynamic modules are acceptable from the point of view of the time and space complexity. This reconfiguration method requires an empty configuration bitstream (an empty module). The empty bitstream is a starting point for all other dynamic modules, so each new dynamic module loaded to a slot will be loaded over empty bitstream configuration of the slot. As a consequence, only one differential bitstream for each dynamic module is needed to be loaded over the last module in the slot. Before loading the new module, the slot has to be reconfigured bask to the empty module. For each new dynamic module a clear bitstream is needed. Which is a differential bitstream generated as a difference between the empty slot and the corresponding dynamic module.

The empty bitstream can be just a bitstream that changes all bits in the slot to the same value or a bitstream that sets the slot to the most often used dynamic module. To achieve an ideal state (= fastest reconfiguration) the empty bitstream may be formed as an average over all configuration bitstreams. To get the fewest changes during the reconfiguration process the empty module should minimize the Hamming distance between configurations bitstreams.

The time needed to set a new module from the empty module is denoted as t_l. Afterward t_l is defined in the same fasion as t_p so $t_s \geq t_l > 0$. The time to set the empty module from the last module is denoted as t_o. t_o is defined similary to t_l. So the time to change the last dynamic module to a new dynamic module is a sum of the time to set the empty module t_o and the new module to the empty module t_l. The time complexity can be up to $2t_s$ in the worst case. But we can say that on average it is equal to t_s. The space complexity can be worse than that in the first case, because there are two bitstreams for each dynamic module (the load bitstream and the clear bitstream). In the case of n modules the space complexity is $O_3(2n) \approx O_3(n)$.

To compare the time and space complexity of these three methods we will consider

Reconfiguration type	Space Complexity	Reconfiguration Time
Full bitstream	$O(n)$	t_s
Differential bitstream	$O(n^2)$	$t_s \geq t_p > 0$
Empty bitstream	$O(n)$	$t_s \geq t_l > 0$

Table 2.1: Summary of the reconfiguration methods for the dynamic module.

the worst case scenario. The time complexity comparison can be seen as $t_p = t_s = \frac{t_l + t_o}{2}$. From this it follows that the first method and the second method are equal and the last method is twice slower. Fortunately, in well-composed tasks the second method should be better than the first one, and the third method can perform similarly to the first and the second methods as follows: $2t_s \gtrapprox t_l + t_o \gtrapprox t_p$. The space complexity of the first and third methods is the same, but in almost all cases the bitstream for the whole slot is bigger than the differential bitstream. Unfortunately, two differential bitstreams are required by the third method. The space complexity for the second method is really very bad, and in the case of many dynamic modules it cannot be used. The time complexity of the second method is the best.

Finally, it is important to say that the previous analyses of the time and space complexity don't consider the time required for initialization and the space required to store additional information about each dynamic module. It is considered that such time and space is very small compared to the rest.

Performance Issues

The next very important perspective is the application performance. The dynamic reconfiguration increases the functional density, but the price paid is the decreased application performance. The designer must optimize these two parameters to satisfy the technical requirement specification (TRS).

The designer should always analyze the application in detail and try to estimate the resulting performance. The performance must satisfy the TRS, and in addition it must overcome other implementation technologies (DSP, uControllers, etc.) at least in one important parameter (power dissipation, size, weight, cost, design time, etc.). If this is not the case, the designer should probably select the other technology.

For reconfigurable applications there is an important fact that the application operational time is a sum of the execution time and the configuration time $t = t_e + t_c$. For the current FPGA devices, t_c lies in the order of milliseconds. Dynamically reconfigurable applications have t_c parameter less affected than the statically reconfigurable applications, because just a portion of the design is affected by the reconfiguration.
The best expression of the application performance is the functional density.

Definition 2.2.1 *The functional density is defined as an inverse of the application cost $C_A = At$, where A is the area and t is the total operational time, i.e. $D_A = \frac{1}{C} = \frac{1}{At}$. A simple modification can be done for dynamically reconfigurable applications:*

$$D_A = \frac{1}{A_s t_{es} + A_d(t_{ed} + t_{cd})},$$

where $A_s t_{es}$ is the cost of the static part, and $A_d(t_{ed} + t_{cd})$ is the cost of the dynamic part of the application. It is clear that the T_{cd} parameter strongly affects the functional density.
The maximal functional density of application can be achieved when the $\frac{t_{cd}}{t_{ed}}$ ratio approaches in the limit to 0, i.e. $t_{cd} \ll t_{ed}$. For a real application it means that the processing time of the dynamic part of the application must be much bigger than the time required to reconfigure it. If it is not, the performance will be enormously downgraded.

2.3 Application Types

This chapter will discuss classification of dynamically reconfigurable applications and their types according to their implementation. To design dynamically reconfigurable application the type of application has to be defined . It is divided to two main groups: stream-type applications and control-type applications.

The stream-type application operates on a continuous data stream. A good example would be audio or video processing. Applications in this group are typically designed as a pipelined data-path with input and output buffers. They process data in real time. Many applications in this group process the data in batches. A typical example is video processing based on video frames.

The control-type application can be understood as a complex sequential circuit that controls another process. A parallel port controller or a state machine that implements a set of protocols would be the best examples. The decomposition is typically used to overcome the drawbacks of a single state-machine like solution.

Stream-Type Applications

A typical stream-type application is shown in Figure 2.1. In fact, all the data processing is done in a data-path with several processing units. The units can be inserted or removed from the path to obtain the requested result. The input and output of such data-path is typically buffered to overcome latency problems in the data-path.

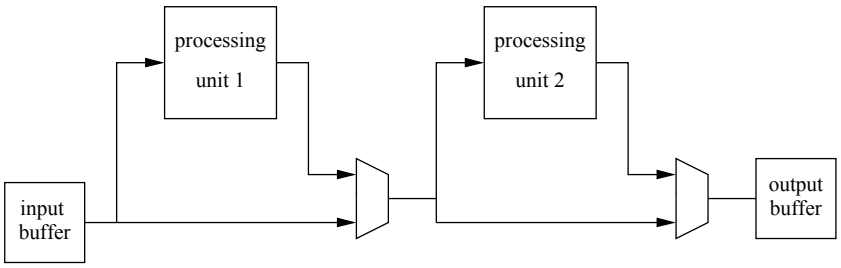


Figure 2.1: A typical stream-type application.

Figure 2.2 presents the same data-path implemented using the dynamic reconfiguration approach. The set of processing units was replaced by one dynamic module called super-macro that can be used to implement any of the data-path functions. The design is now significantly simplified.

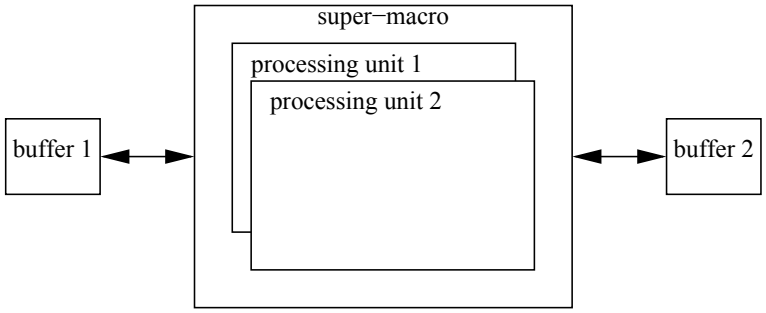


Figure 2.2: A stream-type application implemented using dynamic reconfiguration.

The input and output buffers were replaced by two buffers that are capable to handle data in both directions. The super-macro is a dynamic module that implements either

processing unit 1 or processing unit 2. Only one unit is loaded in the super-macro at one time and therefore there is significant decrease in the design size. The other unit is stored in an external storage as a bitstream. Units may be swapped upon request similarly to a replacement of an IC in a package holder. The transformed application contains in addition a reconfiguration controller that controls the reconfiguration process. It is not shown in Figure 2.2.

The application can now process the data in batches. First, a new data comes in the buffer 1. Depending on requested data-path configuration, the processing unit 1 is loaded in the super-macro. Once the whole batch is processed, it is stored in the buffer 2. If only one processing was requested, the buffer 2 becomes the output buffer. If not, the other unit replaces the processing unit 1 and the data batch is processed again, but from buffer 2 in the buffer 1. In this case, the buffer 1 becomes the output buffer.

This solution is well scalable. The more processing units are required in the data-path, the higher logic spare may be achieved. With increasing the size of data batch the reconfiguration overhead decreasing and the functional density increases.

In the case that only one processing unit is required, the design leads to a very efficient solution without any drawbacks. In other cases, the maximal frequency of incoming data batches must be lower than an inversion of a sum of processing and reconfiguration times of all requested processing units:

$$F_{max} \leq \frac{1}{\sum_{i=1}^{n} t_{ei} + t_{ci}} \leq \frac{1}{n(t_{emax} + t_{cmax})},$$

where n is the number of processing units involved in the computation, t_e is the execution time and t_c is the reconfiguration time of a particular unit.

Control-Type Applications

A typical control-type application is shown in Figure 2.3. It is the parallel port controller. It contains three state machines that implement different communication standards (SPP, EPP, ECP). Only one protocol is required for communication with another device and therefore the others may be unloaded from the FPGA fabric. This trick reduces the size of the application and the implementation can fit in a smaller FPGA device.

This approach uses the standard dynamic modules. In Figure 2.3, the unloaded dynamic modules are marked with a dash-dot line. The only loaded module is the ECP

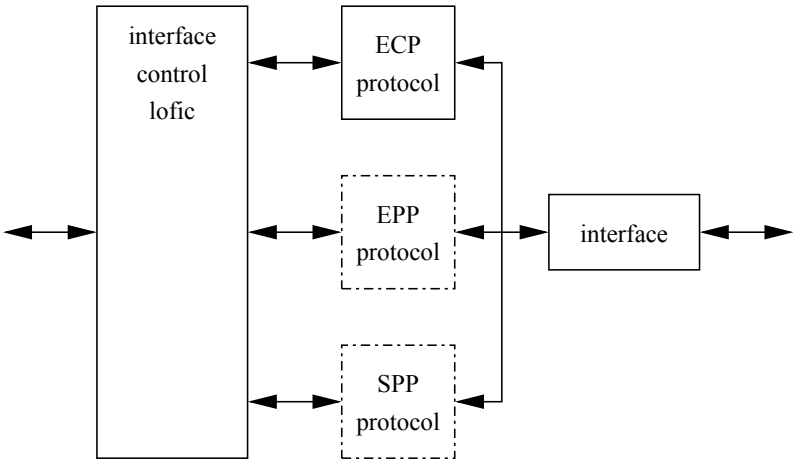


Figure 2.3: A typical control-type application.

protocol state machine. This example is well scaleable, because with increasing the number of dynamic modules the design size does not change. It depends only on the size of the biggest dynamic module. The functional density of this application is very good, because the execution time is long compared to the reconfiguration time.

The parallel port interface is a very simple example. The different protocol state machines are not required to communicate with each other, and they do not need to store any internal state. Once the host finishes the communication with the other side, there is enough time to switch the protocol controller.

In the case of more complex state machines, the designer must precisely analyze the application and search for temporal localities of well defined design partitions (configurations). The longer the application stays in one configuration, the bigger the functional density.

The other problem is the state storage. In the best case the loaded design partition requires only reset to enter the required state (our parallel port example). In other cases the data storage infrastructure must be implemented in the static part of the design. In our case, it is integrated with the reconfiguration controller. There exist FPGA prototypes that can handle the state storage automatically, but unfortunately they are not commercially available yet.

2.4 Thinking Dynamically From the Outset

The best results can be achieved when the design is being developed with the dynamic reconfiguration in mind. The design can be written in a way that it looks as an output from the partitioning tool. In this case, the designer is able to use all benefits of super-macros, reduce the amount of stored sequential states and optimize the storage elements. The functional density is typically higher, because the designer can use the knowledge of the application functionality to optimally partition the design.

The designer can start building the application directly as shown in Figure 2.2. The top-level entity includes only one instance of the super-macro. The buffers can be implemented in a different way to support bi-directional data transfers. The multiplexers shown in Figure 2.1 can be completely omitted due to the function of the reconfiguration controller.

The drawback of this approach lies mainly in the verification of the design. The current simulators do not support simulation of dynamically reconfigurable applications. Using the constraint file and the simulation tool, a special testbench environment that enables simulation can be set up.

2.5 Summary

The chapter analyzed the dynamic reconfiguration, the FPGA devices and their possibilities in the area. The FPGA technology has wide opportunity in the future that brings a memory technology and its variability, fast prototyping application and low cost development curve. The dynamic reconfiguration brings new features for the FPGA devices. It extends functionaly and increase function density.

The next part of the chapter analyze reconfiguration process, its possibility and space complexity. Three methods are presented for reconfiguration dynamic module from last to new. Each method has its advantages and disadvantages. The first metod rewrites all bits in the dynamic slot and has space complexity $O(n)$. The second method uses a differential bitstream that changes only necessary bits in the last dynamic module to get the new dynamic module. It is the fastest reconfiguration method but the space complexity is $O(n^2)$. The last method uses empty module. Each reconfiguration process is done in two steps. From the last module to the empty module and from the empty module to

the new module. This method has the space complexity $O(n)$ but the speed is in most of cases better than the first method.

Further the text analyzes function density D_a that express the application performance. The maximal functional density is done by a ratio between a reconfiguration time t_{cd} and execution time t_{ed}. From this ratio we can see that $t_{cd} \ll t_{ed}$. For a real application it means that the processing time of the dynamic part of the application must be much longer than the time required to reconfigure it. If it is not, the performance will be enormously downgraded.

The end of the chapter presents two types of applications. The stream-type application allows to process data in batches by different functions. The function is loaded in the dynamic module and data are processed in the dynamic module. The function in the dynamic module is changed and data are processed by the new function. The second type, control-type application changes function in dynamic module according requests from control devices. The function of the whole application is changed by changing the control part of the design.

Chapter 3

Reconfigurable Hardware Platform

This chapter will deal with available reconfigurable FPGA platforms and their support of dynamically reconfigurable applications (Daněk et al., 2004), (Horta et al., 2002). There are a few FPGA device types on the market. It is possible to meet pure FPGA devices like Spartan and Virtex from Xilinx (Xil, 2001), (Xil, 2000) and AT40K and AT6000 from Atmel (Atm, 1999*a*), (Atm, 2002*a*). On the other hand, there are several combinations of an FPGA with an ASIC microcontroller or microprocessor on one chip. An example is the Virtex II Pro family from Xilinx (Xil, 2003) or FPSLIC from Atmel (Atm, 2002*c*). The first decision is between a pure hardware solution implemented just in an FPGA, or a combination of software and hardware solutions implemented in the FPGA with an additional microcontroller. An embedded microcontroller in the FPGA opens a new posibility to manage the design from software. In designs with the dynamic reconfiguration the reconfiguration process can be managed more easily and effectively.

The main part of the work in this chapter on reconfigurable hardware platform was done by the author and colleagues in the EU research project RECONF2 (nr.IST-2001-34016), see the description of the project on page V, in the project C-A-K (nr.LN00B096), see the description of the project on page VII and the commercial cooperation with the FPGA chip supplier Atmel Corporation, see page VI. The project RECONF2 was directly focused on design methodology and environment for dynamically reconfigurable FPGA and their implementation on FPGAs available on the market. The project CAK was focused on cooperation of universities and industry. The final implementation of the reconfigurable platform was funded by FPGA chip supplier Xilinx and Atmel.

The research in the area of dynamically reconfigurable FPGAs has several significant papers that describe other possible implementations of the reconfigurable platform on FPGA. Some of them are (Verkest, 2003) and (Prophet, 2004).

The implementation of the reconfigurable hardware platform on the FPGA brings many difficulties and barriers that must be solved during implementation. The author and colleagues described and solved several of them inthe RECONF project and in cooperation with Atmel. The following chapter and papers (Bartosinski et al., 2005*a*) and (Honzík and Kafka, 2005) describe suggested solutions of implementations a hardware platform with a partial dynamic reconfiguration.

3.1 Reconfigurable FPGA Devices

There are several FPGA devices that allow to change the whole or a part of the configuration memory during the lifetime of the application. This features were not included in the first FPGA devices twenty years ago. To be reconfigurable, the FPGA devices have to have direct access to the SRAM-based configuration memory from external sources, an internal microcontroller, or directly from the programmable logic array.

The SRAM-based configuration memory is a special type of memory that affects the function of the FPGA device. Each element in the memory is connected with some element in the logic matrix or interconnection. By changing the contents of this memory element the element in the function of the logic matrix or interconnection will be changed. This way the design can change its internal structure and function while running. The change of one element in the configuration memory must not affect the other logic elements. In other words, the remaining elements have to work properly during addressing and changing the first element. This feature enables partial dynamic reconfiguration of the FPGA device.

The parameters of the configuration are affected by many things: The speed of the configuration interface and its data path width, the type of the communication protocol, or the size of the configuration memory. FPGA devices have several types of configuration modes. There are both master and slave configuration modes. The master mode allows the device to behave as a stand-alone unit that generates clock and addresses for the configuration data stored in an external memory device. The slave mode has to be managed by an external device, which can be a microcontroller or configuration automata. The

slave mode is suitable for partial reconfiguration. The configuration interface can be serial or parallel. Because of speed the parallel interface is more suitable for reconfiguration, but it employs more I/O pins in the package of the FPGA device.

Currently there are three types of FPGA devices widely available on the market that support dynamic reconfiguration. The following table describes important parameters of these FPGA devices. More information can be found in (Atm, 1999*a*), (Atm, 1999*b*), (Atm, 2002*a*), (Atm, 2002*c*), (R.Matoušek, 2003), (Xil, 1997), (Xil, 2001), (Xil, 2003), (Xil, 2010*a*), (Xil, 2010*b*), (Xil, 2010*c*), (Xil, 2010*d*). (Alt, 2007)

Device	**Array**	**Memory [bit]**	**LB Mem**	**Width [bit]**	**freq. [MHz]**	50% **[ms]**
AT40K40	48×48	336,504	139	16	33	0.32
AT94K40	48×48	815,852	219	16	33	0.77
XC2S200	28×42	1,335,840	977	8	50	1.67
XCV1000	64×96	6,127,744	977	8	50	7.66
XCV3200E	104×156	16,283,712	952	8	50	20.36
XC2V8000	112×104	26,194,208	2 199	8	66	24.79
XC2VP125	136×106	43,602,784	2 225	8	66	41.29
XC4VSX35	96×40	13,700,288	n/a	32	66	6,49
XC5VSX50T	120×34	12,556,672	n/a	32	66	2.97
XC5VLX110T	160×64	31,118,848	n/a	32	66	7.37
XC5VSX240T	240×78	79,610,368	n/a	32	66	18.85
XC6VLX760	360×166	184,823,072	n/a	32	66	43.76
XC6VSX475T	360×105	156,689,504	n/a	32	66	37.10

Table 3.0: Important parameters that affect reconfiguration - Column 1 describes the device type. Column 2 denotes size of the logic matrix. Column 3 shows the memory size needed to store one configuration of the whole device. Column 4 shows the memory size occupied by one logic block. Column 5 shows the configuration interface data path width. Column 6 denotes the maximal operating frequency of the configuration interface and column 7 shows the time it takes to configure 50% of the device through the described interface and frequency.

Xilinx Virtex2

Xilinx Virtex2 devices are today's high-end FPGA devices used widely in the industry and research (Xil, 2000), (Xil, 2003). They can be reconfigured by loading application specific configuration data into the configuration memory. They have the ability to be partially reconfigured by loading new data into a specified area of the chip without any affect of the rest of the chip, while the rest of the chip remains in operation. There are two kind of partial reconfiguration - static and dynamic. Static partial reconfiguration is done before the device is fully active or when the device is inactive; this can be accomplished by deactivating the chip select signal (CS) during configuration. For the partial reconfiguration to take place, the rest of the device is in the shutdown mode and is brought up again once the reconfiguration is completed. Partial dynamic reconfiguration is done when the device is active. Except during some inter-design communication, certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming.

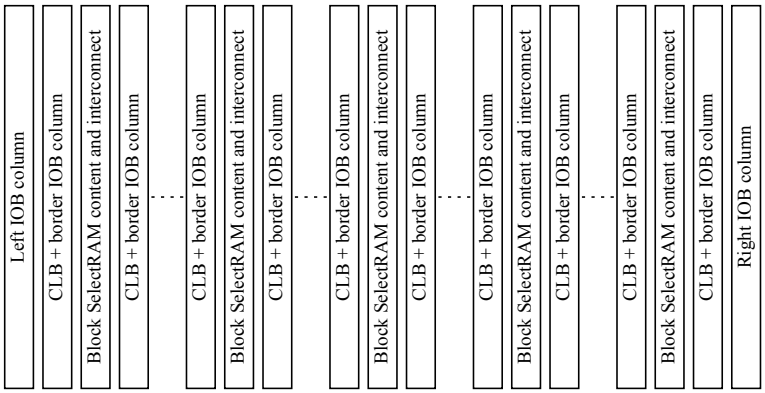


Figure 3.1: Xilinx Virtex2 - configuration blocks.

Partial reconfiguration of Virtex2 devices can be accomplished from an external source in either the Slave SelectMAP mode, or the Boundary-Scan mode, or from the internal logic resources by using the internal configuration access point (ICAP). The ICAP implements an internal connection to the SelectMAP port.

The internal configuration memory is partitioned into frames. Each frame is one bit wide and its size depends on the chip height. The number and size of frames varies according to the device size. The total number of configuration bits for a particular device is calculated by multiplying the number of frames by the number of bits per frame, and

adding the total number of bits. The device resources are organized into columns that are composed of several frames as Figure 3.1 shows. The number of frames in column depends on the kind of resources present in the column.

The configuration bitstream contains a synchronization word (each word is 32bits long) and two kinds of packets, header packet and data packet. There are two types of header packets: Type1 packet headers are used for register writes. A combination of Type1 packet and data packet are used for frame data writes. Frames are read and written sequentially with ascending addresses for each operation. Multiple consecutive frames can be read or written with a single configuration command. The smallest amount of data that can be read or written with a single command is a single frame. The entire CLB array plus the IOBs and SelectRAM block interconnect can be read or written by a single command. Each SelectRAM block contents must be read or written separately.

There are two methods to create the partial bitstream:

Difference-based method of partial reconfiguration is accomplished by making a small change to a design and then by generating a bitstream based on only the differences in the two designs. Switching the configuration of a module from one implementation to another is very fast as the bitstream differences can be smaller than the entire device bitstream.

Module-based method of partial reconfiguration is based on the modular design methodology. This flow requires the signals used as communication paths between or through reconfigurable modules use fixed routing resources, as shown in Figure 3.2. The Bus Macro is a fixed routing bridge with reliable communication; it is a pre-routed hard macro used to specify the exact routing channels that will not change from compilation to compilation. For independent modules bus macros are not needed.

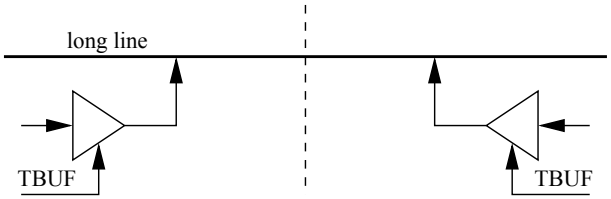


Figure 3.2: Bus Macro element.

Atmel AT40K/AT94K

Atmel AT40K is a 40K-gate FPGA device with logic cells organized in a 2D array (Atm, 2002*a*). The logic cell consists of two 3-input LUTs , two DFFs, and connection switches. The AT94K is an extended version that contains (see Figure 3.3) in addition a hard core of AVR 8-bit microcontroller connected to the east-side FPGA I/O pads, a hardware multiplier, two UARTs, a two-wire serial port (TWS), three counters, a watchdog timer, and a 36KB dual-port internal SRAM memory accessible both by the AVR and FPGA. The memory can be partitioned to a 20-32KB AVR program space and a 4-16KB AVR data space.

The AT40K (Atm, 2002*a*) and AT94K (Atm, 2002*c*) supports four configuration modes, out of which the simplest reconfiguration mode is the so-called Mode 4 (Atm, 1998), (Atm, 2001*a*) (simplest = principally does not require an external hardware). The mode 4 reconfiguration is based on four configuration registers denoted as FPGAX, FPGAY, FPGAZ, and FPGAD that are accessible by the user logic inside the FPGA device (AT40K), or serviced by the AVR (AT94K). The first three registers specify the address in the FPGA configuration memory, the last contains the configuration data. This means that the designer can change the configuration of any logic or routing element at any time without a need to consider other FPGA resources.

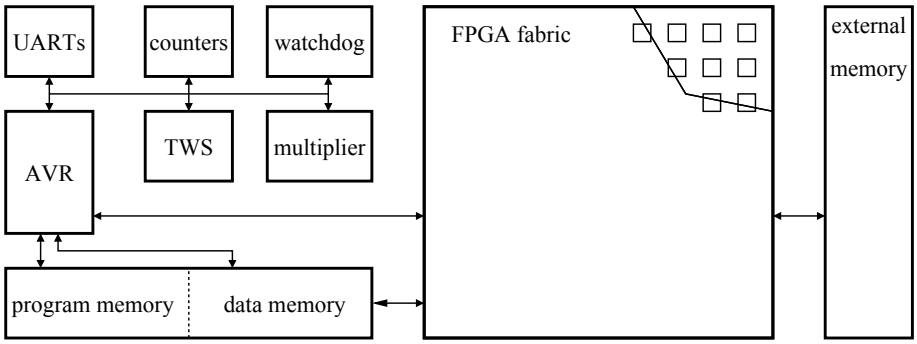


Figure 3.3: Atmel AT94K FPGA with AVR and peripherals.

3.2 Implementation Issues

The implementation issues described in this section deal with physical implementation of dynamic reconfiguration in hardware with all the necessary support circuits. The issues

are connected both with the implemented user design and with the necessary supporting designs (Daněk et al., 2004). The discussion is based on the experience of the author with the RECONF2 flow developed within the EU project RECONF2 for both Xilinx and Atmel FPGAs.

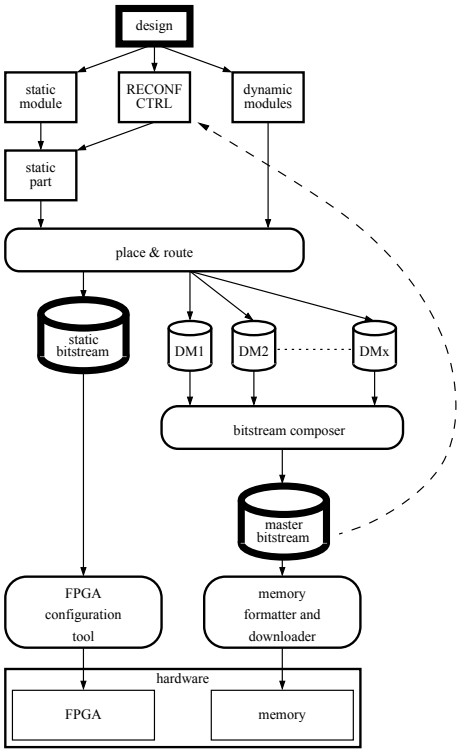


Figure 3.4: Implementation-dependant flow.

Figure 3.11 shows parts of the RECONF2 flow that are affected by a specific hardware implementation (Carvalho et al., 2004), (Robertson and Irvine, 2004), (Wu et al., 2002). The important inputs and outputs are marked with a thick line, these are denoted as idea, which can be replaced by the relevant front-end tools in the case of automated re-design for reconfiguration, *static bitstream*, which is the initial contents of the whole FPGA device, and *master bitstream file*, which, simply speaking, merges together all needed partial bitstreams that implement different contexts of the dynamic modules. The dashed line indicates a dependency between the organization of the master bitstream file and the reconfiguration controller, namely its address generation part.

Reconfiguration Controller

The reconfiguration controller is responsible for correct transfer of data between a storage that contains the configuration bitstreams, and the programmable fabric. Considering a general FPGA-based system-on-a-chip (SOC) platform, the controller can be implemented either in a microcontroller or in hardware. In the case of the microcontroller implementation the controller is likely to be specified as a sequential computer program in C, the hardware implementation will probably be specified in VHDL.

Both implementations share common structure: an external scheduler triggers the data transfer and on completion of the transfer the reconfiguration controller must signal this fact to the scheduler and the data management blocks. The controller basically consists of two parts:

- The bitstream starting address generation locates the required partial bitstream in the master bitstream file stored in an external memory.
- The responsible for proper timing and completeness of the transfer of the partial bitstream.

The structure of the first part depends on the selected organization of the master bitstream file, the second part consists of a memory address register for accessing the external memory and either an end-of-partial-bitstream-mark detection circuit, or a top address register that is loaded with the top address of valid data for each partial bitstream to be transferred.

Bitstream Organization

The bitstream organization is given mainly by the architecture of the used external memory and by the properties of the reconfiguration controller. This section will concentrate on the logical organization that affects the reconfiguration controller.
Figure 3.5 shows three possible organization schemes of a master bitstream file. The first scheme (in the left part) uses an index table at the beginning of the bitstream to redirect the reconfiguration controller to specific bitstreams as needed. The second scheme reserves a word at the beginning of each bitstream that contains the length of the following bitstream. The third scheme reserves fixed slots for all bitstreams and in addition it includes the length of the following bitstream at the beginning of each bitstream.

The first scheme is the most straightforward one. Its major advantage is that each bitstream can be accessed in a constant time, since two addressing operations are needed: the first retrieves the bitstream starting address from the index table and the second stores it to a bitstream address register. The drawback of this scheme is mainly the limited size of the bitstream index table, which determines the number of addressable bitstreams.

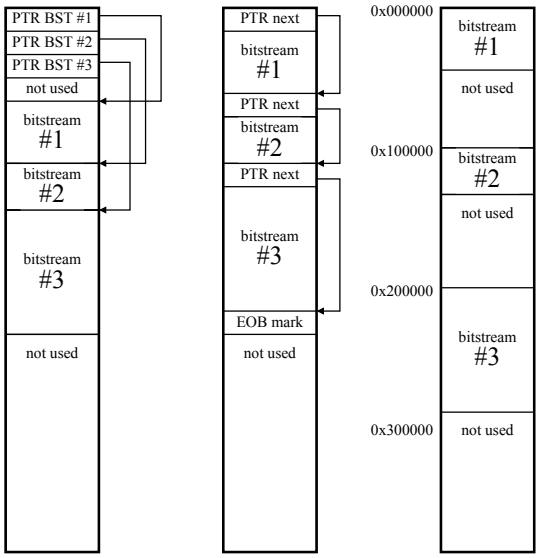


Figure 3.5: Bitstream logical organization.

The second scheme is based on the linked list structure. Its biggest advantages are the unlimited number of addressable bitstreams and a possibility of simple addition of new bitstreams at the end of the master bitstream file. The most important disadvantage is that the time required to retrieve a bitstream is not constant - to generate the starting address of the n-th bitstream n read/add operations are needed. Under usual circumstances we may limit the time by twice the time required to read the biggest bitstream contained in the master bitstream file.

The third scheme adopts the better of the two previous schemes to achieve an efficient hardware implementation. Its advantage is a very fast bitstream starting address generation with a constant bitstream access time and a very simple hardware implementation, but this is traded for wasted memory space due to padding bitstreams to a constant size.

Bitstream Storage

Probably in all marketed FPGA devices reconfiguration is simply a data transfer operation between a bitstream storage and special locations inside the FPGA. Given the usual sizes of configuration bitstreams, the necessary storage size exceeds the size of the available memory built in the FPGA devices and it is necessary to use an external bitstream memory.

The external memory is likely to have a classical address/data parallel interface, or it may use an SPI-like interface. In any case it is possible to reduce the required read/write controller to its simpler read-only version required for reconfiguration. The complete read/write controller is needed in a preparatory stage to download the master bitstream file to the memory; the end user will likely never modify the configuration data and the reconfigurable design can use the read-only controller to spare chip resources and increase design robustness.

The Need of Wrappers

Compared to the classical VHDL design the use of dynamic reconfiguration brings new requirements on synthesis of user macros. The designs are usually synthesized as several independent user designs that are packed together during placement and routing, i.e. when all valid FPGA configurations must be assembled to produce valid configuration bitstreams.

There are two main synthesis issues: net connectivity, and preservation of macro ports.

Net connectivity can be analysed as shown in Figure 3.6 parts *a*, *b* and *c*. Part *a* shows a situation where an output in the static part feeds inputs in the dynamic part, part *c* shows a reverse situation with an output in the dynamic part feeding inputs in the static part, and part *b* shows a mix of the two where an output in the static part feeds both an input in the dynamic part as well as static part.

The problem arising in the situation shown in part *a* is where the actual net branching occurs, and it is solved during placement and routing as is discussed later.

The problem with the situation in part *c* is that during reconfiguration or in contexts where the output is not present on the FPGA the inputs in the static part are floating, i.e. their value is not defined. This problem must be solved on the synthesis level, for example, by using an interface buffer in the static part that will be enabled only when the

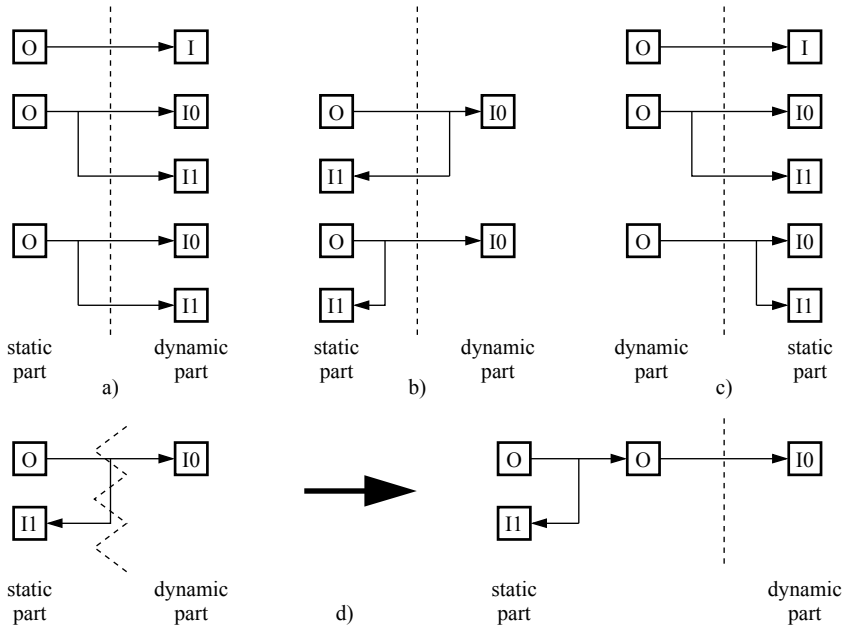


Figure 3.6: Possible port connections and routing between the static and dynamic parts.

corresponding dynamic logic is present.

Net connectivity is the most critical problem that arises in the most complex shown in part *b*. The problem is that the user assumes that an input in the static part connected to the output in the static part should always remain connected no matter which dynamic modules are present on the FPGA. As discussed later on this is not usually the case, since during reconfiguration the router removes all nets that have at least one port in the dynamic part.

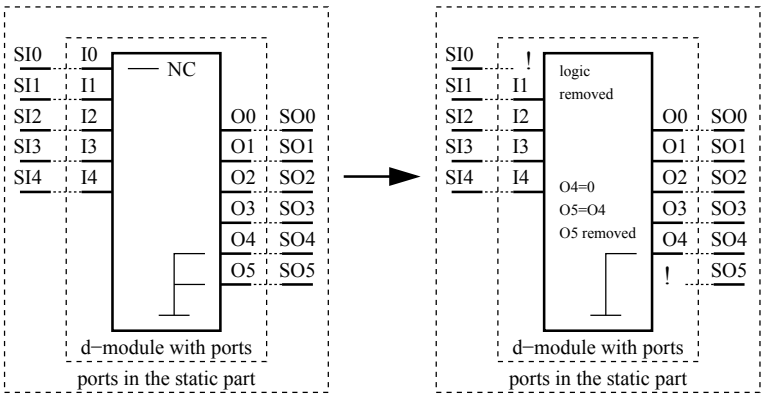


Figure 3.7: Synthesizing an invalid component.

A sample situation with macro ports is shown in Figure 3.7. The depicted user macro connects signal SI0 to a logic that is optimized away during synthesis, and the logic driving output signals O4 and O5 is also optimized away and replaced with a constant low. Since the top-level design must be synthesized as a separate design, it uses the dynamic logic instantiated as black-box components, which preserves all their interface inputs and outputs. On the other hand, when synthesizing the user macro, its interface ports are generated to reflect the actual inputs and outputs used by the logic. When integrating together such a macro with the top-level design during mapping, placement and routing, the tool will find an inconsistency between the static and dynamic part connections and generate an error. This usually does not happen in classical modular designs, since the design flow is the other way round, i.e. the user first defines all macros, gets their post-synthesis interface definitions, and instantiates them as black-box components in the top-level design. There is also no need to unify interfaces of different user macros as is the case with reconfigurable super-macros.

A systematic solution to these problems requires a new approach to logic synthesis and routing. At present, a viable workaround is to preserve all defined entity ports used in user macros, and to transform all connections with mixed inputs and outputs in the static and dynamic part to connections with no direct "static input to static output" connections as shown in Figure 3.6 *d*. This can be easily done by the use of interface buffers generated in a core generator, instantiated as black-boxes in the top-level design as well as in each dynamic module used.

A suitable solution for the RECONF flow is to introduce interface wrappers that decrease the degrees of freedom the designer has to tackle to a reasonable level (Honzík and Kafka, 2005), (Horta et al., 2002). A wrapper is a component that consists of several static-to-dynamic and dynamic-to-static connectors implemented either as buffers, or latches, or registers. In the presented concept always one static wrapper goes together with just one dynamic wrapper.

3.3 Reconfigurable FPGA Coprocessor

This section describes a realistic example that demonstrates how dynamic reconfiguration can be used to implement a reconfigurable hardware accelerator attached to a processor

core (Bartosinski et al., 2005*a*), (Daněk et al., 2004), (Kadlec et al., 2004), (Maruyama and Hoshino, 1999).

Systems with current FPGAs that use dynamic reconfiguration must use an external data memory to store the additional configuration bitstreams. An external FLASH memory can be used to store fixed bitstream data that are copied at different times to specific locations of the internal SRAM memory. The FPGA is divided to a static part that implements an access to the FPGA reconfiguration data, and to a dynamic part that contains user-defined designs; the designs can be reconfigured at run time. The special features of this design are:

- All implemented coprocessor cores have the same interface.
- All implemented coprocessor cores are self-contained function units without direct dependencies between them.
- The design is writen in VHDL and C language; it does design independent on used platform.

These facts make this example ideal for an implementation using super-macros. The accelerator cores are formed by a set of parameterised single-cycle floating-point units that perform basic arithmetic operations such as addition, multiplication, and square root. A possible integration with CPU-FPGA devices, such as the Atmel FPSLIC with the AVR hard core microcontroller, or the Xilinx Virtex2/2Pro with the MicroBlaze soft core or PowerPC hard core microprocessors, leads to a computationally powerful, yet simple SoC design that can be used for simple DSP applications.

The idea is to include a relatively complex unit in the FPGA and to reconfigure it on demand (Bartosinski et al., 2005*a*), i.e. when the nature of the required data processing changes. The proposed design is a study on implementing complex, self-contained, mutually exclusive units in a reconfigurable hardware available today.

Design Structure

The structure of the design is shown in Figure 3.8.The design consists of a wrapper that provides a data interface for the CPU, and of a reconfigurable functional block that holds

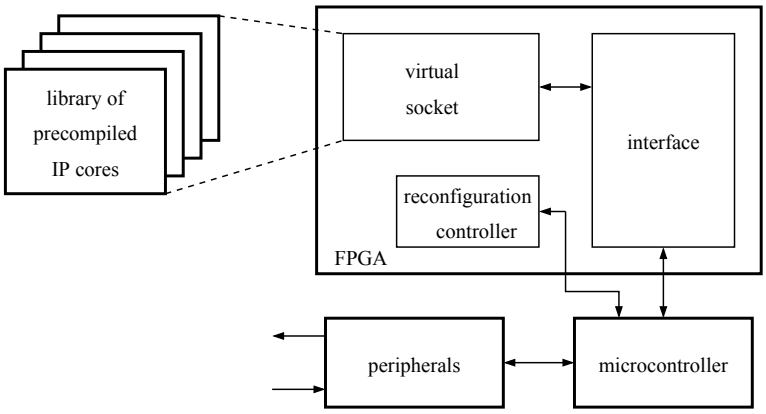


Figure 3.8: Floating-point unit: design structure.

different contents according to the desired floating-point function (denoted as Library of precompiled IP cores).

Data exchange between the FPGA and CPU is managed either through a dedicated 8-bit data bus. The design uses three main data exchange channels:

1. Between the CPU and the FPGA.
2. Between external memory and the CPU via the FPGA.
3. Between the static part of the FPGA and the replaceable floating-point unit.

The first and third channels are represented by the FPGA hardware. The second channel is implemented as a C function that uses a dedicated logic implemented in the FPGA static part.

Programming Model

The FPGA design implements a numeric floating point coprocessor for the CPU. The idea is that the software support built into the operating system makes both the data processing and the reconfiguration processes transparent to the programmer (Honzík, 2004*a*). FPGA execution happens when demanded by a user program. The compiler detects functions that are supported in hardware, and adds necessary function calls to routines that handle CPU-FPGA data transfer and FPGA reconfiguration (Wigley and Kearney, 2001).

The principle of the programming model can be seen in Figure 3.9, where the commands written in the C programming language call procedure FP_ADD and FP_DIV

(Honzík, 2004*b*). These two procedures hide calls to the reconfiguration controller that pass the identifier and parameters over to the dynamic module. In the case that a required dynamic module is not in the slot the reconfiguration process will be initiated. This approach is platform independent and it can be used on any platform with any programming language.

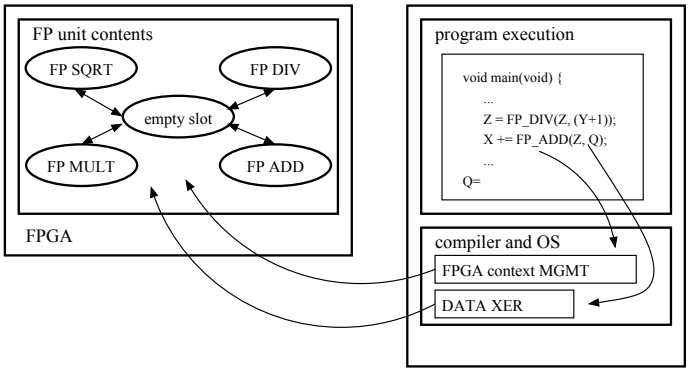


Figure 3.9: Microcontroller execution.

Implementation in Xilinx Virtex2

The design (shown in Figure 3.10) was implemented using the starter kit from Memec with Virtex2 XC2V1000-FG456-4. It consists of two main parts: 1) the static part with the MicroBlaze (MB) processor and its peripherals, 2) the dynamic part with the floating-point coprocessor modules.

The static part implements the MB processor (32bit RISC soft-core processor) with the following peripherals: internal 32 KB SRAM, UART, controller of external DDR SDRAM, the configuration controller connected to the ICAP, and the interface to the dynamic part, i.e. the communication ports to the FP coprocessor. The interconnection between the static part and dynamic part is defined by 2x24 output signals and 24 input signals.

The design was created only with using the Xilinx tools - the Embedded Development Kit (EDK) and Integrated Software Environment (ISE). The implementation of the design is based on module based partial reconfiguration flow.

The processor module was created in EDK as the first step in the implementation flow. The outputs of this step are separate netlists of the processor and its peripherals. The floating-point coprocessor modules were created in Handel-C, compiled into VHDL,

and synthesized with the Xilinx synthesis tool (XST). The next steps were the creation of the top-level designs for each combination of the interconnected modules. The top-level design was synthesized with modules instantiated as black-boxes.

The user constraint file describes the placement of the modules and BusMacro interconnections. Because the MB processor is provided from Xilinx as a macro, it has a fixed width, which means that in this case the processor must consume more than half the width of the used Virtex2 chip.

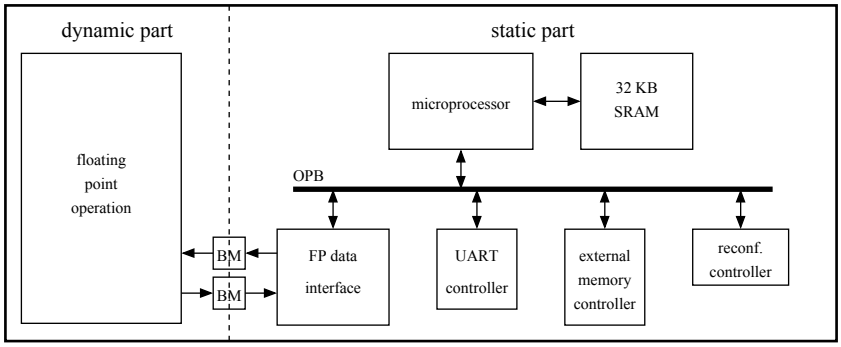


Figure 3.10: The FP coprocessor on Virtex2 with MicroBlaze.

The next performed step was mapping, placement and routing of each module separately with the information from the top-level design. Partial bitstreams for dynamic modules (FP coprocessor modules) were created in this step.

The top-level design had to be assembled from module netlists to create the initial full bitstream. The initial program for the processor is saved in an internal memory, and therefore it must be added into the initial full bitstream. The resulting full FPGA bitstream was imported into EDK, where the C program for the processor was compiled with GCC and included in the bitstream.

The sizes of the partial bitstreams are about 100kB each, but in the described implementation the dynamic area is used sparsely and could have been narrower. A program executed in MicroBlaze controls the communication with the FP coprocessor and its reconfiguration.

Implementation in Atmel AT94K

The FPGA is divided to a static part that implements an access to the FPGA reconfiguration data and the AVR program code, and to a dynamic part that contains the user-defined

designs that can be reconfigured at run time - in our case the different floating-point operations. Before the AT94K FPGA can be used as an AVR coprocessor, it is necessary to define data exchange schemes between the AVR and FPGA (Atm, 2001*b*). There are two possible schemes: to use registers implemented in the FPGA, or to use the internal SRAM; the design uses dedicated registers connected to the AVR-FPGA bus. The width of the FPGA registers is limited by the 8-bit AVR-FPGA data bus. The advantage of this scheme is its simplicity, a major drawback is its slow transfer rate: each AVR load or store instruction requires 2 clock cycles.

To be able to use dynamic reconfiguration in a reasonable way it is necessary to add an external FLASH memory to the AT94K chip that will store the additional configuration information. The static part of the FPGA must implement a DMA controller that will provide the AVR with the access to the configuration data. The most convenient implementation of the DMA controller uses one FPGA register for context (bitstream) selection and another for data passing. The static part of the FPGA implements an address register that consists of the context register (MS bits) and a counter. On writing to the context register the counter is reset. Each time the data register is read by the AVR the counter increments. When the top address specified in the bitstream header is reached, i.e. when the reconfiguration of the dynamic part is finished, the FPGA interrupts the AVR operation.

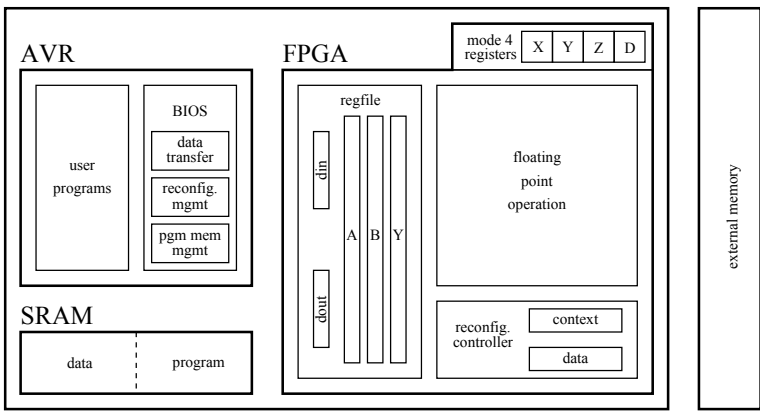


Figure 3.11: The FP coprocessor on AT94K with AVR.

When not considering different execution time due to reconfiguration, the reconfiguration process is transparent to the application software. The access to the FPGA coprocessor is implemented as a special AVR BIOS function, whose parameters are the required

operation and its operands passed either as direct values in the case of the register transfer, or as a starting address of their location and their number in the case of the SRAM transfer. When BIOS detects a request for an operation different than the one currently present in the FPGA, it calls a function that reconfigures the FPGA. This function first translates the requested operation to the context (address) of the corresponding bitstream and writes it to the context register. Then it sequentially reads 4-tuples of values and writes them to the four FPGA configuration registers (X, Y, Z, D) (Atm, 2001*b*), (Atm, 2002*b*) until the FPGA generates an interrupt.

The runtime reconfiguration of the AT94K FPGA fabric requires partial bitstreams. Such bitstreams reconfigure only a part of the chip while the rest is not affected in its operation. A new version of the Figaro design implementation tool provided by Atmel is meant to generate bitstreams that are suitable for partial reconfiguration of the FPGA. A special implementation procedure must be used to obtain such bitstreams. The idea is to get several complete bitstreams with all different coprocessor contexts that contain the same placement and routing of the identical static part.

The Figaro tool works with a system of macros stored in a design library. Any design component can be implemented as a macro and stored in a library. The top-level design may contain instances of such components as black boxes (i.e. without a description of their content). Figaro will then search project libraries for components that fit the instance interfaces.

All design configurations with different coprocessor contexts must contain the same placement and routing of the identical static part. To obtain partial bitstreams the complete bitstreams obtained in the previous step must be compared using the Figaro bitstream compression tool. The tool generates incremental changes that must be performed to switch from the configuration given by the base bitstream to the configuration given by the new bitstream. To be able to use the partial bitstreams for changing the coprocessor configuration, all possible coprocessor context swap combinations must be generated. A direct approach leads to space complexity $O(n^2)$ combinations (each to each), where n is the number of contexts. (see Section 2.2) A significant reduction of combinations is obtained by introducing a common reference coprocessor configuration, such as empty contents (empty bitstream) or the most frequently used function; this approach decreases the number of necessary bitstreams to space complexity $O(n)$. Both partial bitstreams

consist of 20000 32-bit configuration words, with their sizes being 100kB each.

3.4 Comparison FPGA Coprocessor

As can be seen, AT94K is suitable for applications where power consumption matters. The Virtex2 device is suitable for complex non-portable designs. Another observation is that the use of the AVR hard core in the AT94K device effectively increases its factual logic capacity when compared to Virtex2 designs with MicroBlaze. The reconfiguration times in both devices seem somewhat equivalent when the operating frequency is considered.

	Atmel AT94K40FPSLIC	**Xilinx XC2V1000**
Chip size	5K to 40K	1M
SRAM	18.4Kbits	720Kbits
Multipliers		40 18×18 bits
Processor	AVR 8-bit RISC hard core microcontroller (120K) 36KB SRAM, counters, UARTs FPGA ⇔ AVR interface	MicroBlaze 32-bit RISC soft core processor 32KBSRAM, UART OPB bus for peripheral connections
System clock	up to 25MHz	up to 100MHz

Table 3.1: Summarise the device characteristics of both FPGA platforms

Real Dynamically Reconfigurable Applications

The following text describes several applications that use the dynamic reconfiguration of the FPGA devices. There are many research papers about the dynamic reconfiguration and related work, but only a few of them really got down to the real hardware test bench. The others were just playing with this feature or they did a brief evolution. To get a functional dynamically reconfigurable design requires using any hardware platform and doing a lot of hardware tests. The following applications were carried out as far as industrial applications.

GIN - a notebook for blind people uses the dynamic reconfiguration to achieve an low power consumption and portability. Gecko is a dynamically reconfigurable platform that

	Atmel AT94K40FPSLIC	**Xilinx XC2V1000**
Static part	447 cells ≐ 19% FPGA	6400 cells ≐ 62.5% FPGA
Processor	AVR build-in ASIC macro	MicroBlaze 2053 cells ≐ 32% 3 18×18 Multipliers ≐ 10% 17 BlockRAM ≐ 57%
Reconf. part	1176 cells ≐ 51% FP ADD: 970 cells ≐ 42% FP MUL: 602 cells ≐ 26%	3840 cells ≐ 37.5 % FP ADD: 927 cells ≐ 24% FP MUL: 287 cells ≐ 7%
Free resources	327 cells ≐ 14 %	Static part: 4100 cells ≐ 64 % Reconf.part: 2822 cells ≐ 73 %
Power consumption	74mW	1.1W

Table 3.2: The resource usage in the FP coprocessor application for both implementations.

	Atmel AT94K40FPSLIC	**Xilinx XC2V1000**
Type	Mode4 reconfiguration	full columns reconfiguration
FPADD Bitstream Size	2x 14KB	116KB
FPMUL Bitstream Size	2x 7KB	110KB
Speed	4MHz ≐ 50ms 25MHz ≐ 8ms	25MHz ≐ 5.3ms 66MHz ≐ 2ms

Table 3.3: Data pertinent to dynamic reconfiguration of both paltforms

increases functionality of the Xilinx FPGA devices and creates a modular, easy to use platform. The last application uses the dynamic reconfiguration to increase reliability of a system by using triple modular redundancy with the ability to re-load the failed modules in the FPGA device.

3.5 Summary

This chapter presented reconfigurable hardware platform available on todays market and the methodology how to implement reconfigurable flow and reconfigurable hardware. In the beginning of the chapter two devices (Virtex2 from Xilinx and AT94K from Atmel) were presented as devices with reconfigurable features. The text described possible methods that can be used for reconfiguration process.

Further the chapter presented reconfiguration controller and its features necessary to control reconfiguration process. The reconfiguration controller as software in embedded microcontorller and pure hardware type reconfiguration controller were presented.

The chapter presented the way how to store configuration bitstreams. There are three types of storing configuration bitstreams in external memory. Each type has different features important for the reconfiguration process. The speed of the access to bitstream data and space occupied by the bitstream database influence dynamically reconfigurable hardware design. The following text presented problems between static and dynamic part of the design. The connection of these two parts can bring lots of problems during reconfiguration. The text describe how to solve floating connection lines and how to unify interfaces of different dynamic modules by using a wraper module.

Based on the analysis done before we implemented two reconfigurable coprocesors with identical function. The first implementation was done on Virtex 2 from Xilinx and the second implementation was done on AT94K FPSLIC from Atmel. Both implementation works as the CPU with attached reconfigurable floating-point unit that can be called by software services. The reconfigurable floating-point unit can change function by reconfiguration. The end of the chapter compares these two implementations. The Virtex2 implementation is better for non-portable more complex reconfigurable systems. The AT94K FPSLIC implementation is better for smaller portable reconfigurable systems with major stress on power consumption.

Chapter 4

Self-Adaptivity

The following chapter will discuss Self-Adaptive principles from the side of a Self-Adaptive systems based on an FPGA with dynamic reconfiguration. The previous text showed the possibility of the dynamic reconfiguration and platforms for its realization. The following text will define the basic Self-Adaptive architecture and its requirements. Further it will introduce main functionalities of a Self-Adaptive system. In the end of the chapter we will outline hardware parts of the Self-Adaptive system and its cooperation with its environment.

The main part of the work on Self-Adaptive systems and its implementation was done by the author and his colleagues in the EU research project ÆTHER (nr.FP6-2004-IST-4-027611) and Czech research project CAK (nr.1M0567). The project ÆTHER was directly focused on Self-Adaptive systems and pervasive computing and their collaborative work.

On the international scientific scene, some other projects are scientifically linked with the ÆTHER proposal at the hardware level. In the U.S., the High Productivity Computing Systems (HPCS) program (DARPA-HPCS, n.d.) of the U.S. DARPA agency aims at developing a broad spectrum of innovative computing system. These features will help critical systems such as military ones to achieve their goals even if they are damaged. The Architectures for Cognitive Information Processing (ACIP) program (DARPA-ACIP, n.d.) aims at designing a new generation of computing architectures in order to build "systems that know what they are doing". The MIT Oxygen project (MIT-Oxygen, n.d.) gathers academic labs and industrials around the topic of adaptable, efficient and powerful computing resources for pervasive applications. The T-Engine forum (T-Engine, n.d.),

(Krikke, 2005) funds by 22 technological firms aims at developing an open, standardised, real-time platform for future ubiquitous systems.

On the European side, several projects in the pervasive computing domain are under way. The RUNES project (EU-Runes, n.d.) aims at enabling the creation of large-scale, widely distributed, heterogeneous networked embedded systems that interoperate and adapt to their environments. The goal of the project is to ease the development of wide networks of embedded systems in order to enable the programmers to exploit all the benefits of the great number of computing resources.

4.1 Requirements of a Self-Adaptive System

The Self-Adaptivity can be defined as the ability of a system to adapt to an environment. It is done by allowing components to monitor environment and change their behaviour in order to preserve or improve the operation of the system according to some defined criteria. The environment of a system is defined by everything that interacts with this system. The adaptation of a system can occur at different levels, from hardware to software.

The software level adaptation takes a place in operation memory, stack and registers. The architecture of computing units based on microprocessors has hardware support for software changing without any interrupting of function. System of memory windows can change function of computing unit in one clock.

The hardware level adaptation takes a place in the architecture that must adapt its structure. The adapted hardware has to be reconfigurable in order to allow a deep change in the hardware structure.

The Self-Adaptive architecture also needs to efficiently monitor its environment and its behaviour in order to be aware of its performance and the possible related improvements. It also needs a decision taking mechanism to decide the moment of the adaptation. The combination of the monitoring process and the decision taking process provides the device with the ability to Self-Adapt. It can automously trigger adaptation process to improve or to keep its performance after a modification in its environment.

4.2 Architecture of a Self-Adapt Element

We will define Self-Adapt Element as architecture based on Self-Adaptive system. The Self-Adapt Element is a indivisible computing entity with local autonomous decision process occurs that affect its own operation. Figure 4.1 shows a functional view of the Self-Adapt Element that takes into account the definition given in the previous paragraph and the discussions about implications of Self-Adaptivity at the hardware and software level. Let us describe the different parts shown in figure 4.1:

- The computing engine processes data. It must provide the necessary processing power to handle future complex algorithms required by new standards and multimedia applications. It must also be reconfigurable to handle a wide spectrum of applications, and as the user needs may vary when the system is online, this reconfiguration process must be dynamic ("on the fly" reconfiguration). It is the most flexible block of the Self-Adapt Element.

- The observer is responsible for monitoring the computation process and other runtime parameters. It allows the Self-Adapt Element to sense its current environment and optimize its performance, which means both comparing its actual processing parameters with the required constraints given by the application designer, and monitoring communication parameters.

- The controller is in charge of actually taking all decisions regarding the ongoing computation task. Based on the observations of the current running task, the controller is responsible for changing the state of the Self-Adapt Element by loading any locally available task implementation. The computing tasks that are loaded can be pre-synthesized hardware tasks best seen as configurations for an FPGA-like fabric or pre-compiled code best seen as binaries for a CPU soft core, both implemented in the computing engine.

- The communication interface is responsible for the management of Self-Adaptive assemblies. It enables resources sharing and collaboration between Self-Adapt Elements as well as providing the Self-Adapt Element with goals to be reached.

These four main functionalities are embedded in a common box which is the Self-Adapt Element. This box has three links with the external world (which might be joined

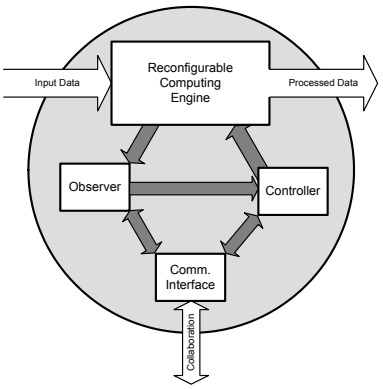


Figure 4.1: Self-Adapt Element block diagram with four main functionalities and three links with external world.

for implementation purposes): the input and the output links are directly related to data processing since the Self-Adapt Element has a dataflow model. The communication link is dedicated to communications among different Self-Adapt Elements. The Self-Adapt Element is basically a tightly coupled hardware/software entity with its computing engine seen as a hardware-based reconfigurable computing unit (e.g. an FPGA fabric), and the observer and the controller implemented as more software programmable CPU cores.

4.3 Function Block of a Self-Adapt Element

The implementation of Self-Adapt Element defined in the previous text leads to embedded devices with adaptive features. Embedded devices are the main resources for computing power and are not intended to follow the user. On the contrary, handheld devices are limited in size and weight since they belong to the user and follow him. They can delegate part of their computation in order to conserve power and they can perform a wide variety of functions thanks to reconfiguration.

The Self-Adaptive embedded device must be able to handle the broad range of applications that are required by the user. This implies that it must be able to provide both the necessary computing power and a large flexibility to efficiently handle a wide spectrum of algorithms. But this should also be small enough in order to be apt for pervasive and low power applications.

For the implementation of the Self-Adapt Element we will use knowledge from chapter 2. We will use hardware parts and principles from chapter 3. The reconfigurable

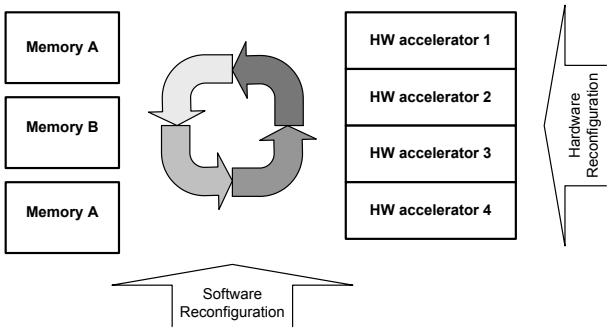


Figure 4.2: Possible implementation of reconfigurable computing engine and its interaction with sorrounding.

coprocessor introduced in the chapter 3 will be base building block for implementation of Self-Adapt Element.

Reconfigurable Computing Unit

The main part of the Self-Adapt Element is reconfigurable computing unit that executes incoming data according current configuration. The reconfiguration of execution part leads the use reconfigurable hardware. Another requirement derived from this first assumption. The system has to be able to quickly change the executed operation during runtime to follow the needs of the user and minimize latency. It implies that the system reconfiguration has to be relatively fast and dynamic.

The range of the possible implementations of the computing engine is broad and depends on the granularity of the unit being configured. One can envisage a range of architectures where various combinations of hardwired processors, soft processors and custom logic can be configured.

The two possible types of reconfiguration can be used to change the executed operation in computing unit. The hardware reconfiguration changer hardwired function in the hardware accelerators as in figure 4.2. The hardware reconfiguration changes elementary functions of the computing unit. The hardwired function can be seen as atomic dataflow functions like ADD, SUB or MAC. The hardwired functions can be viewed as a bitstream file that is loaded into an FPGA to program it.

Another method how to change function of the computing unit is software reconfiguration, see figure 4.2. The software reconfiguration will change the way how to use elementary hardwired function. It changes internal dataflow between memories and hard-

wired functions that change complex function of the Self-Adapt Element from external point of view.

We implemented reconfigurable unit as block of reconfigurable accelerators and three dual port RAMs. The unit is able to process incoming data stored in one of the RAM and result stores to another RAM. This cycle allows to process data in iterations by hardware accelerators, see figure 4.2. The control of the execution data is done by microcontroller. The microcontroller can switch dataflow in the computing unit according the internal program stored in two internal program memory, see figure 4.3.

Observer

To adapt to its environment or to improve the execution of loaded tasks, the system must be aware of its performance. This implies the use of an observation process that enables the system to know if it respects the required constraints. The existence of this observation process can close the loop inside the computing entity, thus enabling Self-Adaptation as it is shown in figure 4.1 by arrows between observer, controller and computing engine.

The observation task has two main objectives. On the one hand, it must embed the required sensors to perform the monitoring that is specified by the designer. On the other hand, it must filter and translate the monitored variables to compute a report to the controller. This enables the sensors to be calibrated and taken into account dynamically.

Since the task that is loaded in the computing engine will change during the life time of the device, it is highly probable that the monitored variables will also change. So the observer has to accept new parameters dynamically.

The observer is implemented as data monitor. It reads tags of the incoming data and create statistic table. The table contains number of packet of each passed or processed application in the adaptive system. Data form table are sent to microcontroller for decision of behaving of the Self-Adapt Element on environment.

Controller Unit

The controller's unit role is to initiate the reconfiguration process based on the information received from the observer and a goal assigned to the Self-Adapt Element. The controller unit compares the observed parameters with the constraints that are given by the application designer as part of the goal. If the observed parameters indicate that the

computation task is deviating from the goal/constraints, the controller takes a decision on the improvement of the computation process. It can then decide to change the implementation of the executed function in the reconfigurable computing unit so as to respect the given constraints. Constrains can be processing time, processing coefficients or power consumption.

The second role of the controller is to manage the different implementations of tasks that are stored in the local Self-Adapt Element memory. If a new implementation is needed by the computing unit to perform the mission, the controller either initiates an immediate task load if the task is locally available or sends a request for an appropriate task implementation.

The control unit is sequential automata and its implementation leads to microcontroller. We used PicoBlaze soft core microcontroller as building block. The Picoblaze has two program memories that allow to do software reconfiguration in one cycle by switching memories, see figure 4.3.

The PicoBlaze has three functions in the Self-Adapt Element. It controls dataflow unit and drives internal switches of computing unit to complete task. Further it receives data from observer and evaluates function of the Self-Adapt Element. If the PicoBlaze finds that function should changed the hardware or software reconfiguration is called. The last function of the PicoBlaze is communication with environment through PLB bus.

Communication Interface

The Self-Adapt Element is viewed as a unified computing model whether it is working as an on-chip computing model or as a multi-chip model. These entities must embed some mechanisms to enable a Self-Adapt Element to publish its internal resources and abilities and to discover the resources belonging to other Self-Adapt Elements. This sharing mechanism enables two Self-Adapt Elements to exchange their tasks or just to clone their states to another Self-Adapt Element, providing the system with self-healing capabilities and online optimization.

As the adaptive systems can be pervasive, the environment will often change during the lifetime, so the system has to optimize continuously to new applications, new users or new elements that may enter the computing area. So the communication channel can be the place of intensive one-to-one exchanges among elements since the system does not

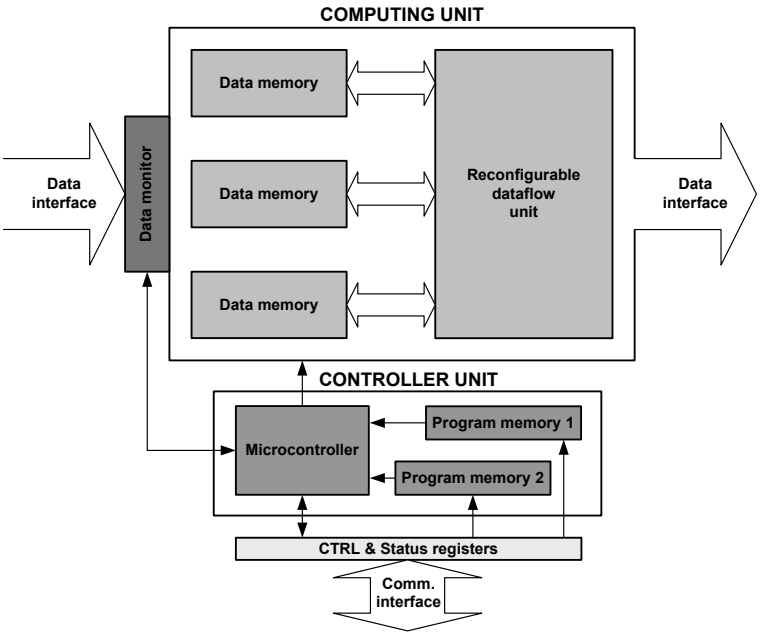


Figure 4.3: Possible implementation of Self-Adapt Element using reconfigurable hardware platform.

implement a centralized control.

The fact that a Self-Adapt Element has the ability to take local decisions will potentially dramatically decrease the need of intensive control communications among Self-Adapt Elements related to optimization, thus relaxing the constraints on this part of the system.

The adaptive system is a virtual processor composed of several computing units type of Self-Adapt Element that are aggregated during runtime. The Self-Adapt Element has the ability to cooperate with others by the means of a publish/discovery mechanism. On the one hand the Self-Adapt Element can publish its active state and the tasks that are stored in its memory. On the other hand it can listen to others in order to discover their state and abilities. This mechanism brings the possibility of tasks exchanges between Self-Adapt Elements or task cloning in order to parallelize an algorithm or for self-healing purposes.

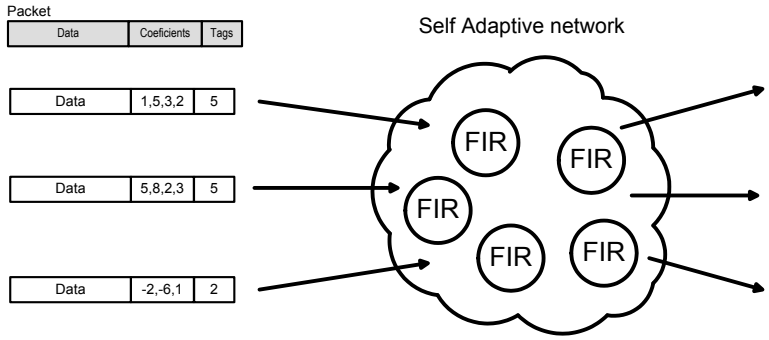


Figure 4.4: Model of a Self-Adapt Element network with the FIR filter function.

4.4 Modeling and Implementation

Imagine a network of Self-Adapt Elements discussed above that implement a family of FIR filters. The requirement is to share the resources among different data channels with different data throughput and filtering requirements, see Figure 4.4. The task of the network is to process data streams, i.e. perform different filtering operations on the input data and generate responses. The network operates on tagged data packets. Each packet is formed by a header and data part. The header specifies in some way operations needed to process the data part. In the example given in Figure 4.6, the first number in the header specifies the order of FIR required to process the data with the next numbers specifying the FIR weights. The header can be viewed in a more general way as a sequence of tags that specify a sequence of operations required to process the data.

A Self-Adapt Element reaction depends on tag values. The packet consists of five parts, see Figure 4.6:

- Packet length: the overall length of the packet.
- Array of tags: tags mark operations that are required to process the data array. In a way the array of tags defines the semantic meaning of the data in the system.
- End of tag array delimiter.
- Data array length.
- Data array: the data can be unprocessed input data, partially processed data, or the desired results.

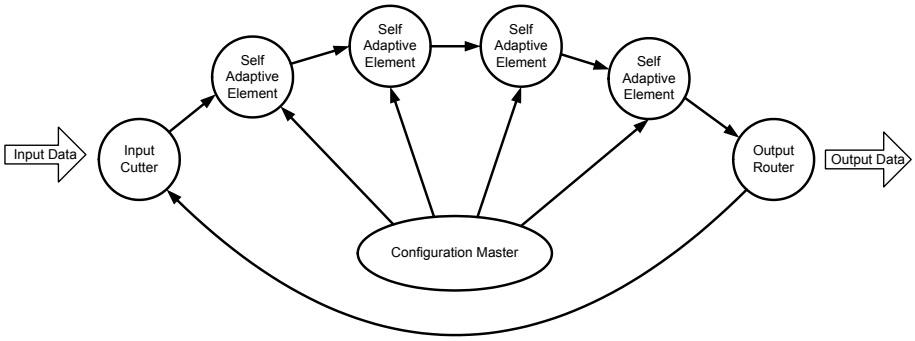


Figure 4.5: Ring topology network with four Self-Adapt Elements and infrastructure for data handling.

The whole network of FIR Self-Adapt Elements can be viewed as a linear chain of processing elements with a FIFO memory forming a loop to circulate data not completely processed, see Figure 4.5. The use of this topology (ring) enables us to concentrate on the Self-Adapt Element computation only, since ring can emulate any given network topology on a logical level. We are aware of the importance of a proper networking scheme, and it will be addressed in more advanced stages of this research.

The Self-Adapt Element network is modeled in the Matlab/Simulink environment and verified on the Celoxica RC10 boards acting as hardware in the loop. The idea is to use the Simulink environment for modeling, visualization and debugging. A major advantage of this approach is that it allows us to gradually move from the software model to the hardware implementation.

Another advantage is the possibility to use any FPGA platform provided it supports a standard simple data exchange protocols; this way we can easily mix implementations on different boards and with FPGAs from different families and manufacturers.

Model in Simulink

We created a model of the Self-Adapt Element network with four Self-Adapt Elements connected in the ring topology. The structure and basic functions of the Self-Adapt Element was discussed above. The network also contains blocks for handling packets in the network and the reconfiguration master controller. The model of the Self-Adapt Element network in Simulink uses on the following blocks:

Input Cutter The input cutter takes an input data stream and divides it into packets of

data. We assume packets with the same length, but this is not necessary in all cases. In addition the cutter prepares the packet headers for the data. In the simplest case the header contains an ordered array of tags that indicate operations that are required to process the data. In more advanced stages of this experiment the array of tags will merely indicate the current state of the data and its desired state, with the proper sequence of operations decided by the Self-Adapt Element network itself.

Output Router The output router is responsible for directing the processed data out of the network, and to direct partially processed data packets back to the network. The processed data are recognized by an empty array of tags. Partially processed data are then stored in a FIFO memory in the feedback loop.

Configuration Master The configuration master is responsible for managing the database of configuration bitstreams for the reconfigurable FPGA part and binary programs for the CPU part of the Self-Adapt Elements. The master sends configurations on demand from individual Self-Adapt Elements. The master by no means introduces central control to the Self-Adapt Element network since this would invalidate the distributed processing character of the experiment.

Self-Adapt Element The Self-Adapt Element monitors the character of data that pass through it, it processes data packets with tags that match its functionality. The distributed control of the network is implemented as individual local decisions by each Self-Adapt Element to change its internal configuration to match the majority of passing data tags and subsequent requests for new configurations to the configuration master.

We use the above building blocks to assemble a Simulink schema that models the Self-Adaptive network. Figure 4.5 shows a sample network with four Self-Adapt Elements, one input cutter, one output router and a configuration master. The whole network is modeled in Simulink. The Self-Adapt Elements are implemented in individual FPGA development boards (in our case Celoxica RC10) and connected to the Simulink environment as hardware in the loop. It allows to use advantages of the Matlab/Simulink environment and hardware platform together. The Matlab plays the role of the input cutter, output router and configuration master. Further it generates cycle-accurate test data that are compared with the output of the simulated network to check them.

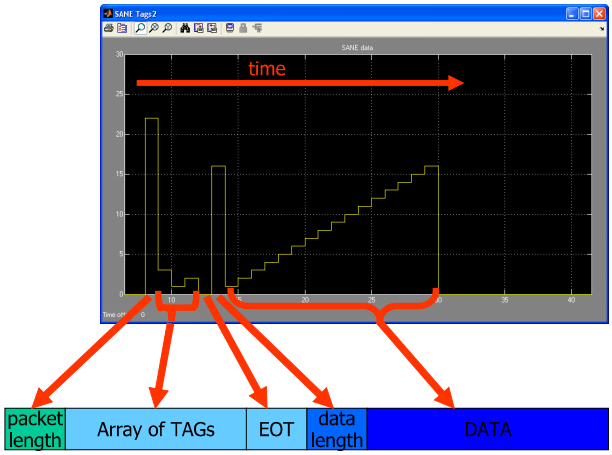
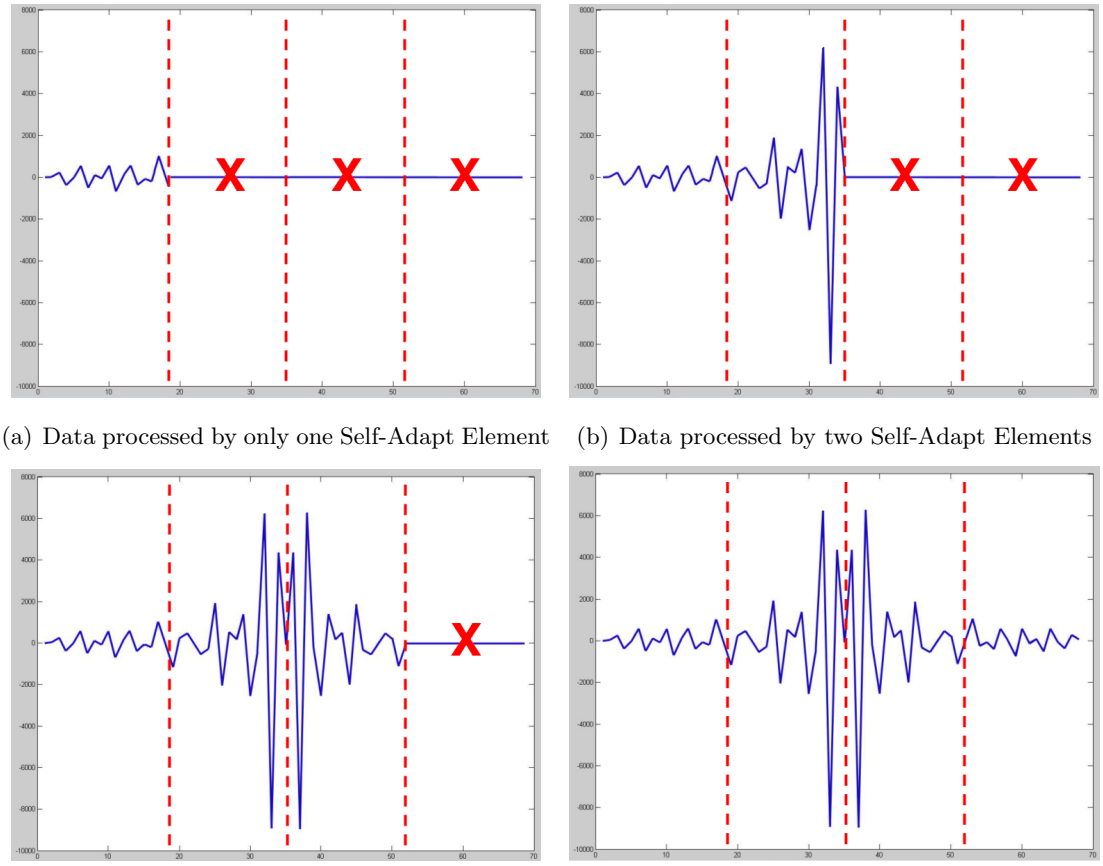


Figure 4.6: Interpretation of simple packet structure with signals plot.

As a test case we used audio data generated by a Matlab script and filtered in the Self-Adaptive network. At the beginning the Self-Adapt Elements have no function and just pass the data through. When the observer of the Self-Adapt Element decides on one of the available functions, the Self-Adapt Element gets configures and starts processing th appropriate data. During the run of the network all the Self-Adapt Elements become processing the data that cycle in the network.

The incoming data in the network consecutively set the function in each Self-Adapt Element and the computing power of the network increases. Figure 4.7 shows stages of the processed data in the network. Figure 4.7(a) shows a stage where only one Self-Adapt Element implements the function of a FIR filter and only the data with the first tag are processed. Other data only pass through the Self-Adapt Elements and cycle in the network. Figure 4.7(d) shows a stage when all the Self-Adapt Elements have been configured and data with all tags are processed in one pass through the network.

When new data come from environment with different tags that define a new function that is not served by any Self-Adapt Element, the data cumulate in the network and influence the decision of the observers. The observers change the functions of the Self-Adapt Elements, and the network adapts to the new type of data. This process tests the Self-Adaptive mechanism of the network with Self-Adapt Elements.



(a) Data processed by only one Self-Adapt Element

(b) Data processed by two Self-Adapt Elements

(c) Data processed by three Self-Adapt Elements

(d) Data processed by four Self-Adapt Elements

Figure 4.7: Consecutive processing data in network by Self-Adapt Elements.

4.5 Summary

This chapter presented a Self-Adaptive systems and its elements. At the beginning of the chapter the requirements of the Self-Adaptive system were analyzed. It was done with respect to a future implementation on reconfigurable platforms based on the FPGA devices. Further the text introduces principles of the Self-Adaptive-element that is the basic building block of our adaptive system. It countains a brief description of the four main blocks of the Self-Adapt Elements and their interaction with the environment. Then the chapter describes the details of the Self-Adapt Element and its implementation including the building blocks and reconfigurable parts. The implementation of the Self-Adapt Element is based on the previous knowledge presented in Chapter 2 and Chapter 3.

The end of the chapter presents the implementation of a model of the Self-Adaptive network with four Self-Adapt Elements. The model is implemented in the Matlab/Simuling environment, and the Self-Adapt Elements are implemented in FPGA boards connected to the Matlab/Simulink environment as hardware in the loop. To test the network we used a desing with a distributed computation of a FIR filter. The implementation of the network shows the adaptation process of the empty network to the fully configured network.

Chapter 5

Network on Chip

This chapter will discuss topology of network on chip suitable for the FPGA chips. The text introduces seven types of network topologies and compares networks topologies from their communication and hardware cost. We will choose the best topology for video and image algorithms suitable to implement on current FPGA chips.

Further the chapter introduces three basic routing algorithms for data transporting over the network. There are presented principles of the routing algorithms and their influence communication delay.

The last part of the chapter describes details of the chosen topology and defines basic parameters of the topology we chose for future simulation of a adaptive system. We will define parameters suitable for measuring the network communication and adaptation processes.

The paper (Salminen et al., 2007) defines network on chip and compares common topologies. The papers (Ni and McKinley, 1993), (A. Mello and Calazans, 2004), (Palesi et al., 2009) describe current state of the routing algorithms and their trends. Finally the (Kogel, T. et al., 2006) and (Foroutan et al., 2010) describe parameters and their analytical evaluations for networks implemented on SoC.

5.1 Network on Chip Topology Selection

Network-on-Chip (NoC) connects nodes into one computing engine on a single chip. Such system with computing elements and network connect them is called System-on-Chip

(SoC). The basic properties of the NoC paradigm discussed in (Salminen et al., 2007) are:

- separates communication from computation
- offers scalability in links and nodes
- variable link width and buffer sizes
- avoids centralized controller for communication
- offers varying properties for transfer
- allows multiple frequency domains

but commonly the name NoC is used for all communication networks that are targeted for a single chip implementation. There are several most important NoC topologies that can connect a number of nodes into the SoC. To compare them we chose the following topologies: *single bus, ring, tree, star, 2D-mesh, fully connected and hypercube topology*

The main application area that we will use the network for are video and graphic applications for analyzing pictures in real time. For this reason we will have strict constrains on data transfer capacity in the network. According to (Salminen et al., 2007) and (Sander, 2004) the comparison of the topologies below brings groups of suitable networks for graphic algorithms. The graphic algorithms for video processing brings the following constrains for NoC:

- HDTV standard resolution in 24 bit color depth
- Real-time constrain in image processing with 25fps in HDTV

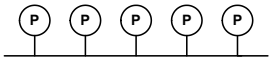
Further we will use self adaptivity in the network. This feature nodes accessible to each other to utilize node reconfiguration. The network must allow to measure parameters of running applications and communication parameters. These parameters identify success of the adaptive process.

There are two aspects of different NoC topologies that we should look at. The first is scalability and the second is the cost of communication and hardware implementation. We will look at the communication cost from three sides: communication one node to one node (one-to-one), communication one node to all nodes called broadcasting (one-to-all) and communication all nodes to one node (all-to-one). The parameter p is a number of nodes in the NoC topology.

Single Bus Topology

The single bus has a low hardware cost, all nodes are accessible to others but the shared bus is a bottleneck when increasing the number of nodes. This topology is suitable for systems with small communication requirements.

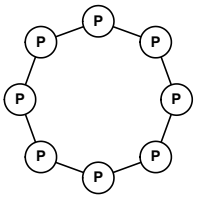
Hardware cost	1
One-to-One	1
One-to-All	1
All-to-One	p



Ring Topology

The ring topology partly solves the problem with bottleneck in a shared bus. It is suitable mainly for pipelined algorithms that can utilize connections between blocks. It will have communication bottleneck for collateral algorithms that work on the same data.

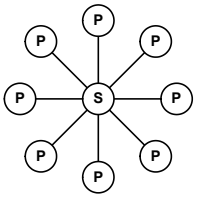
Hardware cost	p
One-to-One	$\lfloor \frac{p}{2} \rfloor$
One-to-All	$\lceil \frac{p}{2} \rceil$
All-to-One	$2\lceil \frac{p}{2} \rceil$



Star Topology

The star topology implements connections with only 2 hops between all nodes. The major problem can be found in the central switch. It uses a centralized control mechanism. When increasing a number of nodes the switch can be a bottleneck of the network. The solution can be found in connected several substar topologies to one star topology.

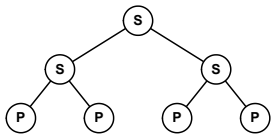
Hardware cost	$p-1$
One-to-One	2
One-to-All	$p-1$
All-to-One	$2(p-1)$



Tree Topology

The tree topology can bring a bottleneck in the top switch during more communication flows over the network. The solution is to add more wires in the top switch and create a fat tree. But more wires increase the hardware cost.

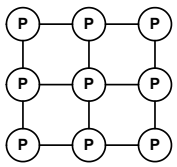
Hardware cost	$2p - 2$
One-to-One	$2\log_2 p$
One-to-All	$\log_2 p(1 + \log_2 p)$
All-to-One	$2\log_2 p(1 + \log_2 p)$



2D-Mesh Topology

The 2D-Mesh topology offers a good ratio between the hardware cost and communication between nodes. The problem can be found in a communication protocol and in communication swithes. The topology has a complicated communication protocol.

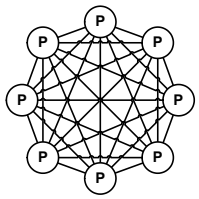
Hardware cost	$2(\sqrt{p})(\sqrt{p} - 1)$
One-to-One	$2(\sqrt{p} - 1)$
One-to-All	$2(\sqrt{p} - 1)$
All-to-One	$2\lceil\frac{\sqrt{p}}{2}\rceil$



Fully Connected Topology

The fully connected topology has an excellent throughput. The connection between nodes is one hop. Problem is with a hardware cost. When increasing the number of nodes, the hardware cost increases quadratically.

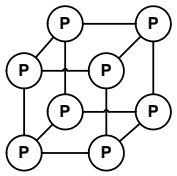
Hardware cost	$p\frac{(p-1)}{2}$
One-to-One	1
One-to-All	$\lceil\log_2 p\rceil$
All-to-One	$2\lceil\log_2 p\rceil$



HyperCube Topology

The hypercube topology is one of the most elegant designs, often used in multi-processor solutions. The throughput of this topology is sufficient for image processing. The problem is with implementation in hardware and its hardware cost with more nodes.

Hardware cost	$\frac{p}{2}\log_2 P$
One-to-One	$\log_2 p$
One-to-All	$\log_2 p$
All-to-One	$2\log_2 p$



We discussed the most common NoC topologies in the previous text. The parameters of the NoCs were described and we chose three main parameters: *scalability, communication cost* and *hardware cost*. From the FPGA side of view the hardware cost affect placement design into chip. We are restricted by the connection matrix in the FPGA chip and the size of the chip. The scalability became an important parameter in multiprocessor design. Scalability has to bring an easy way to resize a connection network, a number of processing nodes and a bandwidth of connection link. The communication cost is an important parameter in designs with high communication load such as video processing. Even more when the design has to satisfy real-time constraints.

Topology	Bus	Ring	Star	Tree	2D-Mesh	Fully	Hypercube
Scalability	+	+	0	+	+	-	-
Communication cost	-	-	-	-	+	+	+
Hardware cost	+	+	+	-	0	-	-

Table 5.1: Comparison of the most common NoCs from side of scalability, communication cost and hardware cost.

The short summary of these three parameters is shown in table 5.1. Each parameter is compared three categories from the best to worst $(+, 0, -)$. From the comparison of the NoC we will choose 2D-Mesh topology as the best NoC topology for image and video processing designs. It brings scalability; successful communication cost and the hardware cost fulfill restrictions that come from the FPGA connection matrix.

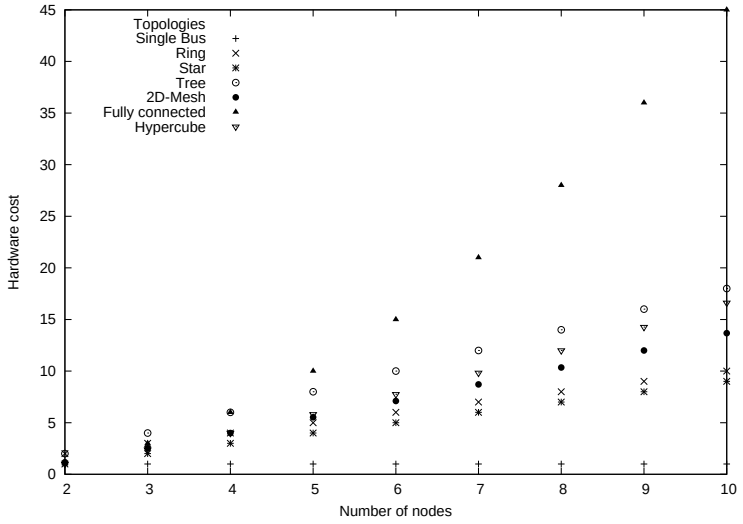


Figure 5.1: The overview of hardware cost of selected topologies

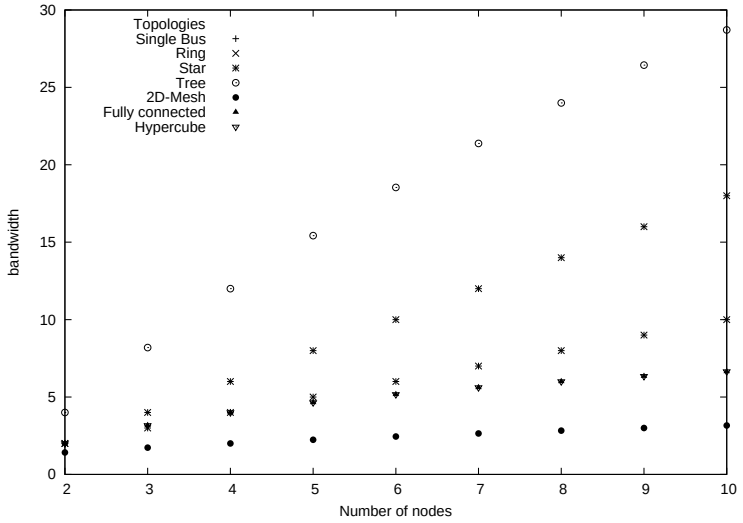


Figure 5.2: The overview of communication bandwidth of selected topologies

5.2 2D-Mesh Topology

We compared parameters of the NoC topologies in the previous text and we chose 2D-Mesh as the best topology. The 2D-Mesh brings scalability, suitable communication cost and hardware cost doesn't exceed FPGA restrictions.

The 2D-Mesh network has several modifications such as 2D-Wrap round mesh. The 2D-Wrap round Mesh connects nodes in the edge of network together in a vertical and a horizontal direction. The connection is direct. It solves the problem with high communication load in center of 2D-Mesh topology that can bring difficulties during processing. Unfortunately in the FPGA technology such a direct connection of the edges will be solved by long lines. In our case we use long line and half long lines for a connection of dynamic and static parts. Direct edge connections significantly decrease a number of the long lines. From that reason in the following text we will use the 2D-Mesh topology without direct edge connections.

Routing Packets in a Network

We have defined a network topology suitable for the FPGA chips and able to transfer data between each node. The performance of the 2D-Mesh network partly depends on the efficiency of a routing algorithm. The research about routing on the network is a huge challenge and there are a lot of papers about this problem. Because packet routing is not the main goal of this work, we will use the knowledge from the following papers (Ni and McKinley, 1993), (A. Mello and Calazans, 2004), (Palesi et al., 2009) and (Gindin et al., n.d.). The papers discussed problems from a side of successful delivery packets and flits like deadlock, livelock and starvation and from the way of routing like deterministic or adaptive routing and minimal and nonminimal path.

We chose three types of routing that can test future placement algorithms nodes on network. The types of the routing differ mainly in their routing rules.

Direct routing algorithm

The direct routing algorithm routes flits alternating in a X and Y direction until reaching the destination node. This creates a direct path from a source node to a destination node as we can see in Figure 5.3(a). This routing algorithm will have problems with

communication overload in the center of the network in a case of more long diagonal communication paths.

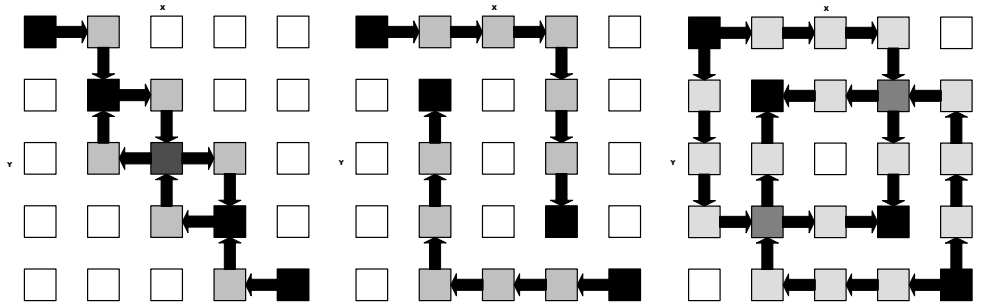
XY routing algorithm

The XY routing algorithm routes flits first in the X direction, until the flits reach the X coordination of the destination node. Afterward the flits are routed in the Y direction until reaching the destination node. The example of the communication is in Figure 5.3(b). This algorithm solves problems with communication overload in the center of the network.

Extended XY routing algorithm

The extended XY routing algorithm routes flits according to their order number. If the order number is odd, the flit is routed first in the X direction. If the order number is even, the flit is routed first in the Y direction. When they reach the first coordination the flits are routed in the second coordination until reaching the destination node. The example of the communication is in figure 5.3(c). These routing rules create two paths from the source node to the destination node. Each path has half the communication load than in the path with XY routing rules. It distributes the communication load into a bigger space in the network communication matrix.

To simplify routing process and hardware cost of routing blocks we will use deterministic distributing way of routing with a minimal path. The flit contains the address of the destination node and the order number of the flit.



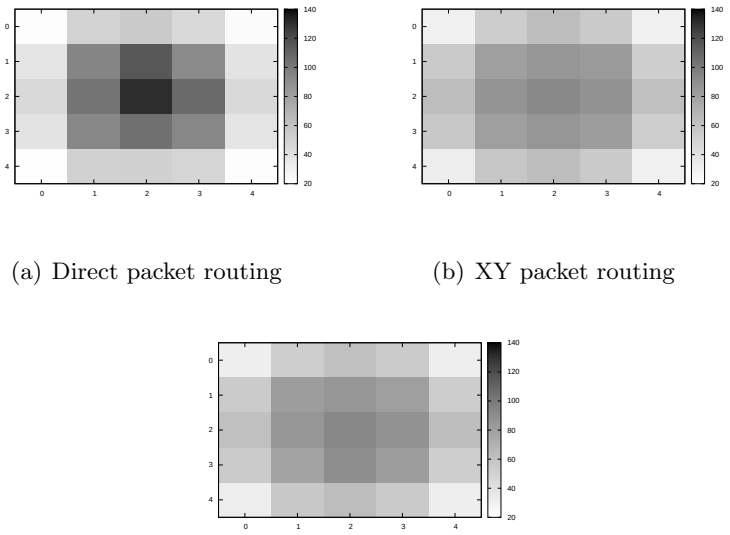
(a) Direct flits routing (b) XY flits routing (c) Extended XY flits routing

Figure 5.3: Three types of routing flits in 2D-mesh network (a), (b) and (c)

Our three different routing algorithms bring three types of communication load in

the network. The direct routing algorithm will move the communication load into the center of the network, see Figure 5.4(a) unlike the extended XY routing algorithm that will distribute the communication load into the whole network, see Figure 5.4(c). The comparison of the three algorithms can be seen in the random communication test in Figure 5.4. The random communication test proves the communication load of routing algorithm. All the nodes in the network are occupied by a random generator that randomly generates flits. Each generated flit is randomly directed into one of the nodes in the network.

From the heat graph in Figure 5.4 we can see the distribution of the communication load in the 5x5 2D-Mesh network. A dark node means more communication load than a white node.



(a) Direct packet routing

(b) XY packet routing

(c) Extended XY packet routing

Figure 5.4: Random routing flits in a full load network with three types of routing rules. (a), (b) and (c)

5.3 Network on Chip Characteristics

The previous text discussed the network topology and we chose the 2-D Mesh topology as the most suitable for our purpose. To measure network parameters and test the network we have to define parameters of the network. The main parameters like link capacity or

communication delay can be determined by an analytical method see (Foroutan et al., 2010). Other methods to evaluate parameters of the network are described for example in (Kogel, T. et al., 2006), (Guz et al., 2007) and (Chang and Yubai, 2006).

The network we chose in the previous text has a regular topology. We will use the 5x5 size mesh topology in our simulation. Links are connected by five port switches with names *west, north, east, south and process*. The links are full duplex. Each direction has a 32bit wide bus to connect the switches. The switch works as an automation with a packet switching mechanism. There is a fifo buffer for incoming data with the capacity of 6 flits.

To measure the performance of the network we have to define network parameters that describes actual network status and performance. The parameters come from single parts of the network like links, flit path, loading of switches and processors. We will use the network cost and the application cost as the main parameters that describe load of the network and its effeciency. Both parameters come from the basic parameters of the network described below.

The following notation is used to analyze the network on chip performance:

F	=	The set of all flows from every source module $1 \leq s \leq N$ to every destination module $1 \leq d \leq N$.
f^i	=	A flow from set F.
m^i	=	The mean packet length of flow $f^i \in F$.
λ^i	=	Average packet generation rate of flow $f^i \in F$ [packet/second].
T^f_{REQ}	=	The required mean packet delivery time for flow f.
C_j	=	Capacity of link j [bits/second].
C_{Lmax}	=	Maximal capacity of link j [bits/second].
t_{lat}	=	communication latency of single transaction.
t_{reconf}	=	time consumed by the reconfiguration process.
C_t	=	weight average of past rewards and the initial estimate C_0.
α	=	step size parameter (constant) $0 \leq \alpha \leq 1$.
r_t	=	reward from running a process in time t.

Flit Format

We suppose a flit communication format for communication in all the networks presented above. The flit is composed from a header, data and tail. The flit header contains

information about the destination and data format. The flit data contains data processed from the previous node. The flit tail contains the whole flit check sum. The size of the flit depends commonly on the size of the data. In our simulation we will use 1024 word of 32bit size flits including the header and tail.

Link Capacity

All the links in a NoC have the same link capacity. It is defined as:

Definition 5.3.1 *The capacity of the link C_{Lmax} is determined by a configuration of the communication resource bitwidth and by a clock frequency of the resource.*

$$C_{Lmax} = bitwidth * clockfrequency \tag{5.1}$$

The link capacity has to be higher than the required capacity required by the processes communicating over the link. The 100% utilization of the link can be reached only theoretically. The average utilization of the link should not exceed 50% when the considering communication delays. With increasing the utilization of the link the communication delays can increase significantly. The minimal link capacity is defined in (Guz et al., 2007) and can be formalized as follows:

$$\begin{aligned} \text{Given:} \quad & F \\ & \forall f \in F : m^f, \lambda^f, T^f_{req} \\ \text{Have:} \quad & \forall linkj, \text{ assign link capacity } (C_j) \\ & \forall f \in F : T^f \leq T^f_{REQ} \\ \text{Where:} \quad & \sum C_j \text{ is minimal link capacity} \end{aligned}$$

where F is a set of all flows, m^f is a mean packet length, λ^f is a average packet generation and T^f_{req} is a mean packet delivery time.

The theoretical capacity of a link in our network is 381MB/s in each direction in a case of 100MHz communication frequency. The minimal capacity for one HDTV video stream is 148MB/s, that is 39% of a capacity of a link.

Communication Latency

The communication latency is a time needed to transfer one flit from its source to destination. We will use the hop communication latency term. The term hop communication latency in this text means the latency with which the flit crosses one hop between two neighboring nodes. The IO buffer in network interface is time stamp for the hop communication latency.

The following notation is used in order to analyze the communication latency:

Definition 5.3.2 *The communication latency t_{lat} is a single transaction that correspond to the delay from start to completion one events on the link.*

$$t_{lat} = \Delta t_{pending} + \Delta t_{arbitrate} + \Delta t_{transfer} \tag{5.2}$$

where $t_{pending}$ is the time which the packet will reach the top of the internal switch fifo buffer. It's size is $size * 10,24\mu s$. The worst time in our network is equal to the size of the buffer. The arbitration time $t_{arbitrate}$ is the time from the request for transfer to neighbor and its answer. The time depends on the load of the neighboring switches. The worst case can be $51,2\mu s$ (five input streams share one output stream). The transfer time $t_{transfer}$ is equal to the size of the packet. We will use 1024x32bit size packets and buffers. The $t_{transfer}$ is $10,24\mu s$. In our network the worst communication latency between two switches is $112.64\mu s$.

Move Function Delay

The move function delay t_{reconf} in the network is thetime needed to change the function of the processor by partial dynamic reconfiguration. The process changes the hardware function of the module. The reconfiguration process is described in the Chapter 3. The speed of the reconfiguration process depends mainly on the reconfiguration method presented in section 2.2. The following notation is used in order to analyze the move function delay:

Definition 5.3.3 *The move function delay t_{reconf} is time consumed by reconfiguration process changing hardware function. It contains request time and reconfiguration time. The structure of reconfiguration time depends on reconfiguration method.*

$$t_{reconf} = t_{req} + t_o + t_c \tag{5.3}$$

where t_{req} is the time from the request function to the answer of the reconfiguration arbiter. The reconfiguration process contains two operations. The first operation is delete a reconfiguration module and takes time t_o, and the second is set a new module and takes time t_c. $t_o = 0$ in case when we reconfigure full module.

In our simulation we will use reconfiguration of the floating-point unit as the basic function of the processor. The reconfiguration process by the method with the full column bitstream takes $1.3ms$. The request time t_{req} depends on the architecture and speed of an external memory system that stores reconfiguration bitstreams.

Averaging Method

The evaluation of the network parameters and running application parameters shows effectivity of our placement algorithms. To calculate parameters we will use the averaging method published in (Sutton, S. R. and Barto, G. A., 1998). The method is suitable for non stationary problems. It uses a step size parameter that evaluates new values of our cost function in the next step. The step size parameter determines the size of new incoming reward that is added to the old weight average from the last step.

Definition 5.3.4 *The averaging method evaluates the weight average C_t of past rewards during the lifetime of the measuring process. The speed of evaluation is done by step size parameter α*

$$C_t = C_{t-1} + \alpha(r_t - C_{t-1}) \tag{5.4}$$

where C_{t-1} is the weight average from the last step. α is the step size parameter (constant) $0 \leq \alpha \leq 1$. r_t is a reward from running process in time t.

5.4 Summary

This chapter presented a network on chip analysis from the side of communication, hardware cost and video processing. The emphasis was done to FPGA restrictions. According to the presented parameters of the networks we chose the 2D-Mesh topology as the most suitable for our future simulation of the self adaptive system.

Further we presented three routing algorithms for routing flits in the 2D-Mesh network. The text described principles of routing algorithms and tests of their impact on full load network.

The end of the chapter described details of the network we will use in our simulation. We defined parameters of the network; like link capacity, communication latency and move function delay that we will use to evaluate network performance during the simulation of an adaptive network.

Chapter 6

Placing Applications

This chapter describes methods how to place and improve placement of applications tasks in a 2D-Mesh network implemented on an FPGA device. First we discuss parameters and constrains that have to be met during a placement process. Next we design three placement algorithms that can be used to reach placement constrains. Each of the designed algorithms has own specific features suitable for different case.

When we have a network with placed applications, we need some parameters for measuring effectiveness of the data processing. For this purpose we will use a cost function that will measure parts and the whole network. The cost will be based on communication in the network connection matrix to evaluate loading of each region of the network.

To improve effectiveness of the running network we will design an algorithm that adapts placement on the network according to the current communication load. The adaptive algorithm will be run in each node to work independently and without any central controller. As an input, the algorithm will use cost values of involved nodes. The success of the adaptation process will be measured by the network cost that shows effectiveness of the communication in the network.

The last part of the chapter will introduce a simulation framework built to test adaptation of the algorithms. We will choose three sets of applications that test a stream and a parallel types of applications and their improvement after adaptation process. To test the usefulness of adaptation placement in real life of the network, we will simulate process of placing and releasing applications on the network and their improvement during life time with incremental adaptation.

6.1 Placing Tasks to Nodes

The application that we want to run on the network has to be placed to it. The application is broken down to small tasks that can be processed by nodes in the network. The tasks are placed to nodes by a placement algorithm. The placement algorithm finds the best node for each task according to the number of nodes and network parameters.

The following notation is used for analyze the placing application on a network on chip:

p_x	=	Node in the network $G(P, L)$ where $p_x \in P$
l_{ij}	=	Link between two nodes where $l_{ij} \in L$
v_x	=	Task from an application $A(V, E)$ where $v_x \in V$
e_{ij}	=	Interaction between two tasks where $e_{ij} \in E$
$h_{min}(p_s, p_d)$	=	Minimal hop distance between source node $p_s \in P$ and destination node $p_d \in P$
$pred(v_k)$	=	Task $pred(v_k)$ is predecessor of task v_k
$dist(v_k, p_f)$	=	Distance between placed task v_k and free node $p_f \in P_f$
P_f	=	Group of free nodes from P
M_{dist}	=	Manhattan distance matrix with distances between all nodes in P
M_{hop}	=	Hop matrix contains number of hops between node $p_{00} \in P$ and node $p_{ij} \in P$
M_{unused}	=	Unused matrix with occupation of nodes P

Network Model

The network model is defined using a connection graph. The connection graph is a direct graph $G(P, L)$, where each vertex $p_i \in P$ represents a node in the mesh topology. The edge between vertices p_i and p_j denoted $l_{ij} \in L$ represents a connection between two nodes in the mesh network. l_{ij} is considered to be direct from node p_i to node p_j. To describe the parameters of the network we introduce hop distance H. $h_{ij} \in H$ represents the number of the edges between nodes p_i and p_j. C introduces the cost of path. $c_{ij} \in C$ represents the cost of the path between nodes p_i and p_j.

In the previous chapters we described NoCs and we have chosen the chooses mesh topology as the best topology for designs with dynamic reconfiguration. The mesh topology offers the best proportion between scalability and connection of nodes. We defined

the mesh network as a pair $G(P, L)$. On the other hand we have applications that we need to run on the mesh network. An application is defined by a pair $A(V, E)$.

Application Model

The application model is defined in the same way as the network model. The application model decomposes application A to tasks $v_i \in V$ and their interactions $e_{ij} \in E$. The application can be written as a pair $A(V, E)$. Each vertex $v_i \in V$ represents one task, and each edge e_{ij} represents an interaction between task v_i and task v_j. e_{ij} is considered as a direct interaction and reflects how task v_i influences task v_j.

Definition 6.1.1 *For the purpose of a future placement algorithm we define a predecessor of the node v_j as $pred(v_j)$ and a successor of the node v_i as $succ(v_i)$. The predecessor and the successor are defined as:*

$$\forall v_i; v_j \in V;\ (v_i, v_j) \in \exists \Leftrightarrow v_i \in pred(v_j) \wedge v_j \in succ(v_i) \tag{6.1}$$

Placement Algorithm

The placement algorithm is a process that assigns tasks $v_i \in V$ to nodes $p_j \in P$ in the network. A placed application is described by an application model, and the network is described by the network model. Placement is a projection of application tasks to nodes in the network.

Definition 6.1.2 *The placing process defined by the placement algorithm to places groups of tasks $v_i \in V$ to groups of nodes $p_j \in P$ according to constrains defined by the application area and the network topology.*

$$V \mapsto P : \forall v_i \in V, \exists p_j \in P; place(v_i) = p_j \tag{6.2}$$

Hop Distance

The hop distance is a distance between two nodes that shows how many links have to be passed to deliver a flit from the source node to the destination node. It is one method to define a cost of delivering flits between two tasks.

Definition 6.1.3 *The minimal hop distance $h_{min}(p_s, p_d)$ is the length of the shortest path from the source node p_s to the destination node p_d.*

The minimal hop distance is equivalent to the efficiency of the communication and energy consumption.

In the following definitions we will use two dimensional mathematic expressions because in chapter 5 we have chosen the 2D-Mesh network as the best network for the Self-Adaptive system. The implementation of the 2D-Mesh network allows us to restrict to only two dimensions.

Definition 6.1.4 *The distance $dist(v_s, p_f)$ is the number of the link in the path from the source task v_s placed on node to the destination node p_f.*

$$dist(v_s, p_f) = \begin{cases} 0 & \Leftrightarrow place(v_s) = \varnothing \\ |p_s(x) - p_f(x)| + |p_s(y) - p_f(y)| & otherwise \end{cases}$$

Manhattan Distance and Hop Matrix

We need to know a distance (number of hops) between two nodes to place tasks in nodes. To find a distance (i.e. the number of hops in the network) in a graph we can use the Manhattan distance matrix defined below.

Definition 6.1.5 *The Manhattan distance matrix $\mathbf{M_{dist}}$ shows the distance between two nodes in an orthogonal network.*

$$\mathbf{M_{dist}} = \begin{vmatrix} 0 & p_{ij} & \dots \\ p_{ji} & 0 & \dots \\ \vdots & \vdots & \ddots \end{vmatrix}$$

$$p_{xy} = |p_i(x) - p_j(x)| + |p_i(y) - p_j(y)|$$

where $p_{xy}(x)$ and $p_{xy}(y)$ are coordinates $[x, y]$ of node p_{xy} in the network P.

The distance values are used by the placement algorithm. The Manhattan distance matrix can be used, but its size increases quadratically when increasing the network size. This becomes a problem when we consider the memory limits in embedded systems. For this reason we define the hop matrix $\mathbf{M_{hop}}$. The elements of the hop matrix $\mathbf{M_{hop}}$ contain

$$\mathbf{M_{hop}} = \begin{vmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{vmatrix} \qquad \mathbf{M_{unuse}} = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{vmatrix}$$

(a) Hop matrix $\mathbf{M_{hop}}$ (b) Unused node matrix $\mathbf{M_{unused}}$

Figure 6.1: An example of a hop matrix 6.1(a) and unused node matrix 6.1(b) for a mesh network of 5x5 nodes.

for each node in the network its distance in hops from origin defined at the node $[0, 0]$. It can be easily implemented in hardware as a memory array. It leads to fast access to the distance information. An example of a hop matrix for a 5x5 network is in Figure 6.1(a).

Definition 6.1.6 *The hop matrix* $\mathbf{M_{hop}}$ *shows number of hops from the node at coordination* $[0, 0]$*. Subtractions of the two values on two elements we have number of hops between nodes.*

$$\mathbf{M_{hop}} = \begin{vmatrix} p_{00} & p_{10} & \cdots \\ p_{01} & p_{11} & \cdots \\ \vdots & \vdots & \ddots \end{vmatrix}$$

$$p_{xy} = |p_{xy}(x) - p_{00}(x)| + |p_{xy}(y) - p_{00}(y)|$$

where $p_{xy}(x)$ and $p_{xy}(y)$ are coordinates $[x, y]$ of node p_{xy} in the network P.

Unused Node Matrix

We need a fast method to find an occupation of node in network by any task from running application. For this purpose we define an unused node matrix that describes the occupation of all nodes in the network.

Definition 6.1.7 *The unused node matrix* $\mathbf{M_{unused}}$ *shows nodes that contain task* v_i *of any running application* $a_k \in A$*. If node* p_{xy} *executes task* v_i *the matrix* $\mathbf{M_{unused}}$ *contains 1 at position* $[x, y]$*, otherwise the matrix element is set to 0.*

$$\mathbf{M_{unused}} = \begin{vmatrix} p_{00} & p_{10} & \cdots \\ p_{01} & p_{11} & \cdots \\ \vdots & \vdots & \ddots \end{vmatrix}$$

$$p_{xy} = \begin{cases} 0 & \Leftrightarrow \exists v_i \in V, v_i \in p_{xy} \\ 1 & \Leftrightarrow \forall v_i \in V, v_i \notin p_{xy} \end{cases}$$

where p_{xy} is a node in the network P and v_i is a task from application A running in the network.

The unused node matrix $\mathbf{M_{unused}}$ is used by the placement algorithm. It contains information about the occupation of nodes by tasks. The implementation of the unused node matrix in hardware is done by a memory array. This guarantees fast access to the occupation information. An example of the unused node matrix for a 5x5 mesh network is in Figure 6.1(b).

First Node Placement

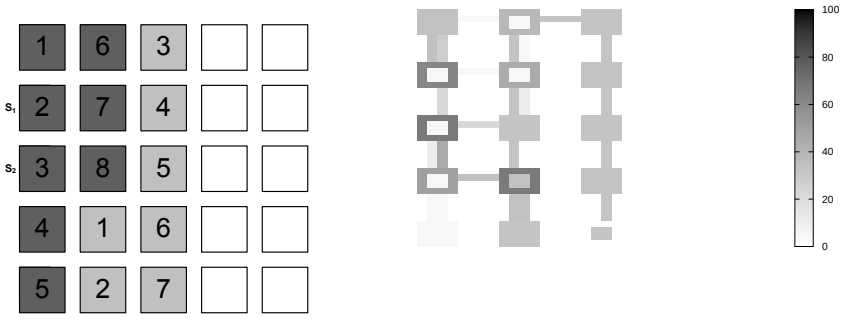
The first node placement algorithm is based on placing tasks v_i on the first free node p_j in the network. The first free node is chosen from the unused node matrix $\mathbf{M_{unused}}$. The first node placement algorithm is the fastest placement algorithm, and its hardware cost implementation is minimal. The most significant problem of the placing task is its inefficient use of the network and heavy communication load. The reason for using this algorithm is its short placing time on an low loaded network. Figure 6.2 shows an example of two applications placed on the network by a first node placement algorithm.

Best Node Placement

The best node placement algorithm is based on placing task v_k on a free node p_j with a minimal hop distance between placed taks v_s and free node p_j.

$$v_k \in V \; \exists p_j \in P_f \subset P; \; place(v_k) = p_j \wedge dist(p_j, v_s) = \min \tag{6.3}$$

where $dist(p_j, v_s)$ is the hop distance from the last placing task v_s and node p_j. P_f is a group of free nodes from P.



(a) An application placed on the network. The stream application is light grey, the parallel application is dark grey

(b) A graph of network utilization. The network path cost $C_{net} = 283$ and the network hop cost $H_{net} = 9$.

Figure 6.2: Two applications placed by First Node Placement algorithm.

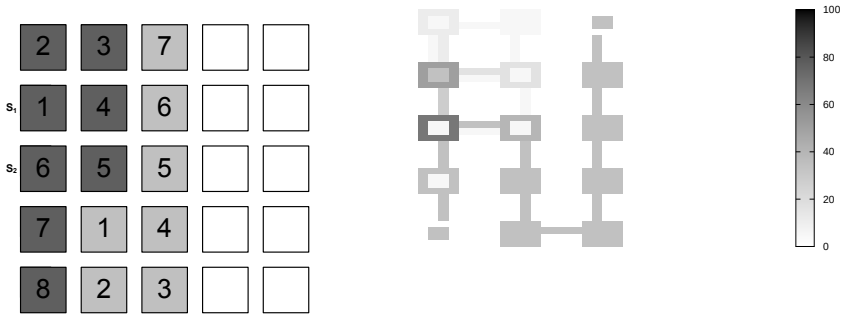
The placement algorithm finds the nearest free node p_j to the last placed task v_s. The hop distance is calculated from the hop matrix. The time to find the best node is equivalent to the number of nodes in network.

The best node placement algorithm is suitable for pipelined applications that can be broken down to more chained tasks. It generates an effective and fast placing of an application to the network. The hardware cost of the algorithm is low and it is suitable for loaded networks. Figure 6.3 shows an example of two applications placed on the network by the best node placement algorithm. We can see that the network path cost is better than in case of the first node placement.

Multi-Best Node Placement

The Multi-Best Node placement algorithm is based on placing a task v_k on a free node p_j that has a minimal distance from tasks in $N(v_k)$. The tasks in $N(v_k)$ have direct connections to v_k.

$$v_k \in V \exists p_j \in P_f \subset P;\ \forall v_s \in pred(v_k); \sum_{v_s \in pred(v_k)} dist(place(v_s), p_j) = \min \tag{6.4}$$



(a) An applications placed on the network. The stream application is light grey, the parallel application is dark grey

(b) A graph of network utilization. The network path cost $C_{net} = 223$ and the network hop cost $H_{net} = 7$.

Figure 6.3: Two applications placed by the Best Node Placement algorithm.

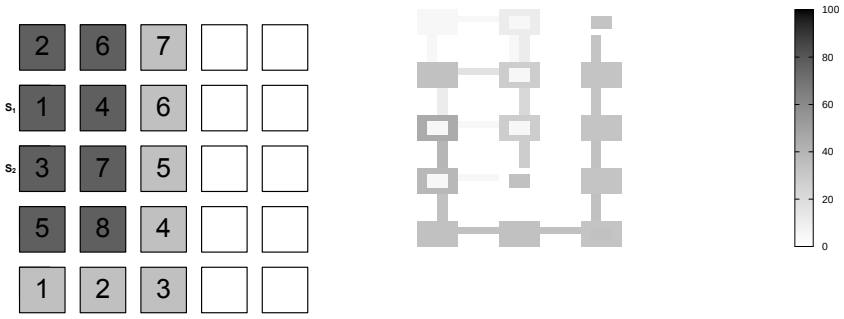
where $dist(place(v_s), p_j)$ is the hop distance from task v_s placed on node to node p_j. The group of tasks $N(v_k)$ are tasks palced on nodes that interact with task v_k. P_f is a group of free nodes and subset of nodes from P.

The Multi-Best Node placement algorithm is most suitable for parallel applications that can be broken down to more tasks that work concurrently. It generates an effective placement of such applications on the network. The hardware cost of the algorithm is acceptable for FPGA chips and it works in loaded network. Figure 6.4 shows an example of two applications placed on the network by the Multi-Best Node placement algorithm. We can see that the network path cost is better than in the case of first node placement and best node placement.

Get-Best Node Algorithm

The Get-Best Node algorithm finds the best position for task v_{best} in the network. The best position is determined by the hop distance from its predecessor tasks already placed on the network. The algorithm finds a node in a group of unused nodes only. The hop distance is counted from the hop matrix, see Figure 6.1(a).

The principle of Algorithm 1 is to count the distance from all nodes with the predecessor



(a) An application placed on the network. The stream application is light grey, the parallel application is dark grey

(b) A graph of the network utilization. The network path cost $C_{path} = 203$ and the network hop cost $C_{hop} = 6$.

Figure 6.4: Two applications placed by the Multi-Best Node Placement algorithm.

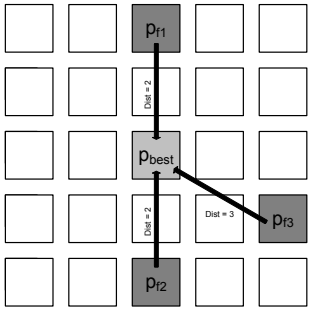


Figure 6.5: Example of the best node for task with three placed predecessors.

tasks of task v_{best} and all unused nodes in the network. The unused node with the minimal hop distance to the nodes with predecessor tasks is the best node p_{best} to place task v_{best} on it. Figure 6.5 shows the best node p_{best} with minimal distance from nodes p_f that contain predecessor tasks of task v_{best}.

To implement of the algorithm in hardware we will the use the hop matrix $\mathbf{M_{hop}}$, unused node matrix $\mathbf{M_{unused}}$, see Figure 6.1(b) and the list of predecessors of the task we want to place on the network.

To process all unused nodes takes n_{node} operations, to process all predecessors takes n_{prec} and to count distance $Dist_f$ takes two operations. The algorithm takes $2n_{node}n_{prec}$

Input: Unused nodes, placed predecessors of task v_{best}

Output: Best Node Position p_{best} for task v_{best}

```
1 foreach unused Node i do
2     foreach Predecessor of v_best placed on p_f do
3         Dist_f = Σ |p_i(x) − p_f(x)| + |p_i(y) − p_f(y)|
4     end
5     if Dist_best > Dist_f then
6         Dist_best = Dist_f
7         p_best = p_i
8     end
9 end
```

Algorithm 1: The algorithm finds the best position in the network for task v_{best}. The position is determined by the shortest distances from its predecessors placed in the network.

operations. The worst case for our network size 5x5 is 1200 operations $(2 * 25 * 24)$ and in an average case it requires 192 operations $(2 * 12 * 8)$.

6.2 Network Parameters Evaluation

The previous text introduced the network and application model and placement algorithms to place tasks of applications to the network. Now we have a running network with several applications, and we need to measure parameters of the network to assess effectiveness of processing data on the network. To assess the network we define a cost that measures effectiveness of parts of the network. The cost value can be used as a parameter for a future adaptation of the network placement and for increasing performance of the network. We define the following costs:

C_{link}	=	Link cost reflects loading of link by communication
C_{flit}	=	Flit path cost reflects cost of path passed by the flit from the source
H_{flit}	=	Flit hop cost reflects the number of links passed by the flit from the source
C_{net}	=	Net cost reflects cost of paths passed by all flits in the network
H_{net}	=	Net hop reflects the number of links passed by all flits in the network
L'	=	Sequence of links the flit passed during its delivery
α	=	a step size parameter in the range $0 \leq \alpha \leq 1$ for calculating a weight average of costs

Link Cost Evaluation

The link cost C_{link} reflects the use of a communication link between two nodes. It is a weighted average of all flits that passed through the link during a defined time. The link cost can be in the range $[0 - 100]$ where 100 means fully used a link and 0 is an unused link. Using a link with C_{link} near 100 can lead to its overloading and delaying flits in the network. The link cost parameter is used for calculating flit path cost during the adapting process of a node.

$$C_{link(t+1)} = C_{link(t)} + \alpha(f - C_{link(t)}) \tag{6.5}$$

where α is the evaluation speed parameter, f is reward from passing flits through the link. If the flit passes, $f = 100$, otherwise $f = 0$. t is time.

Flit Cost and Hops Evaluation

The flit path cost C_{flit} says how expensive it was to deliver the flit from a source task to the current task. It is a sum of all link costs C_{link} of links between the nodes that the flit passed through. The flit hop cost H_{flit} says how many links the flit passed during its delivery from a source task to the current task. The ratio of the flit path cost and flit hop cost indicates the quality of the path the flit passed.

$$C_{flit} = \sum_{l \in L'} C_{link}(l);\ L' = (l_1, l_2, l_3, ...l_n) \tag{6.6}$$

$$H_{flit} = \sum_{l_1 \in L'} 1 = |L'|;\ L' = (l_1, l_2, l_3, ...l_n) \tag{6.7}$$

where L' is the sequence of links the flit passed during its delivery.

Network Cost Evaluation

The network cost contains two parameters, the network path cost C_{net} and the network hop cost H_{net}. Both the parameters say how the network is fragmented compared to the theoretical minimum of the network cost for the current set of applications. The network path cost is a weighted average of all flit path costs of flits that reached their destination nodes over the whole network. The network hop cost is a weighted average of all flit hop cost of flits that reached their destination nodes over the whole network.

$$C_{net(t+1)} = C_{net(t)} + \alpha(\sum_{i \in A'} \sum_{j \in V'} \sum_{k \in T'} C_{flits}(i,j,k) - C_{net(t)}) \tag{6.8}$$

$$H_{net(t+1)} = H_{net(t)} + \alpha(\sum_{i \in A'} \sum_{j \in V'} \sum_{k \in T'} H_{flits}(i,j,k) - H_{net(t+1)} \tag{6.9}$$

where A' denotes all the running applications, V' denotes all the destination nodes in applications and T' denotes all flits that reached their destination node.

6.3 Self-Adaptive Placement

The running network accommodates and releases lots of applications during its life. The applications are accommodated irregularly according to incoming requests from the environment. When an application processes all data, the network releases it. These two operations cause fragmentation of tasks in the network. A fragmented network has higher power consumption and longer data processing.

We developed an algorithm that improves parameters of the network without interrupting its work. The algorithm improves the fragmentation of tasks. The Step-Adaptive Algorithm is based on statistical information available at each node that contains a task. An autonomous and independent algorithm without a centralized unit was the main requirement when developing the algorithm. The algorithm tries to adapt the current placement of a task on nodes by removing tasks independently step-by-step. We call the algorithm *Step-Adaptive Algorithm*, see Algorithm 2. The algorithm incrementally adapts task placement on the network according to the current communication.

The following notation is used to analyze the communication on a network on chip:

C_{port} = a port cost reflects the communication on a port of a node $p_i \in P$

α = a step size parameter in the range $0 \leq \alpha \leq 1$ for calculating a weight average of costs

$\beta_{magnetic}$ = a coefficient that increases the port cost of direct neighbour nodes v_i and v_j with $H_{min}(v_i, v_j) = 1$

Principle

The Step-Adaptive Algorithm look for the best placement of all running applications on the network. The network hop cost H_{net} and network path cost C_{net} are two parameters used for measuring the adaptive process of the network as a whole. The algorithm uses a port cost C_{port} to measure local parameters influenced by the process. Further we use the flit path cost C_{flit} and flit hop cost H_{flit} that were introduced in Section 6.2.

The main principle of the algorithm is to move a task from its current node to a neighbour node with the biggest port cost C_{port} of the current node. The task moves in the same direction as the data come from. If the task meets a task with the biggest port cost C_{port}, it tries to move around. If the task meets the source task, the algorithm stops.

Each node has an independent counter with a random seed that invokes the Step-Adaptive Algorithm for the node. The node needs to know only its port cost and port cost and occupation of its neighbour nodes. It allows to adapt a node independently of any central unit and central information.

Definition 6.3.1 *The port cost calculates loading of each port by incoming flits that are processed in the node. It show the direction of incoming data. We have two types of the port cost:*

Path method: *port cost calculates from flit path cost C_{flit} that says how expensive it is to deliver flits from the source to the node.*

$$C_{port(t+1)} = C_{port(t)} + \alpha(\beta_{magnetic} C_{flit(t)} - C_{port(t)})$$

Hop method: *port cost calculate from flit hop cost H_{flit} that says the distance from the source to the node that the flits have to pass to get delivered.*

$$C_{port(t+1)} = C_{port(t)} + \alpha(\beta_{magnetic} H_{flit(t)} - C_{port(t)})$$

where $C_{flit(t)}$ or $H_{flit(t)}$ is a flit value to passed port l in time t. α is a positive step size parameter in the range $0 \leq \alpha \leq 1$. $\beta_{magnetic}$ is a constantly increasing relation between direct neighbours. Each port of the node has its own port cost C_{port}.

We use only flits that are processed in the current node to calculate the port cost. Flits that only pass through the node do not influence the port cost directly. They influence only the link cost C_{link}. The link cost influences the flit path cost C_{flit}. The link cost doesn't influence the flit hop cost H_{flits}.

Definition 6.3.2 *The coefficient $\beta_{magnetic}$ increases the port cost of the node when the source task is a direct neighbour node of the current node. The size of the coefficient depends on the type of the method used to calculate the port cost.*

$$\beta_{magnetic} = \begin{cases} 1 & \text{if } H_{flit} > 1 \text{ for path and hop method} \\ < 1 - 1.9 > & \text{if } H_{flit} = 1 \text{ for path method} \\ < 1 - 3 > & \text{if } H_{flit} = 1 \text{ for hop method} \end{cases}$$

The coefficient $\beta_{magnetic}$ influences the relation between nodes during adaptation. It reflects that source and destination tasks placed to neighbour nodes have a bigger port cost than when they are far away from each other.

Rules

The Step-Adaptive Algorithm moves tasks according to basic moving rules. Moving rules define behaving of tasks during adaptation. The rules are classified in two classes according to the number of incoming streams of flits to a task. We have rules for one-stream tasks and for multi-stream tasks.

The rules are in figure 6.6. Rules A, B, and C are valid for both one-stream and multi-stream tasks. Rules D and E are valid only for one-stream tasks and rules F and G are valid for multi-stream tasks.

Algorithm 2 applicates rules for adapting tasks positions. The following text describes details of each rule and its implementation in the algorithm.

Rule A defines a behaviour when a task reaches its source, see the algorithm line 7. In this case the port cost of the node is increased by multiplying it with the magnetic coefficient $\beta_{magnetic}$. Than other nodes must have a higher port cost to remove the

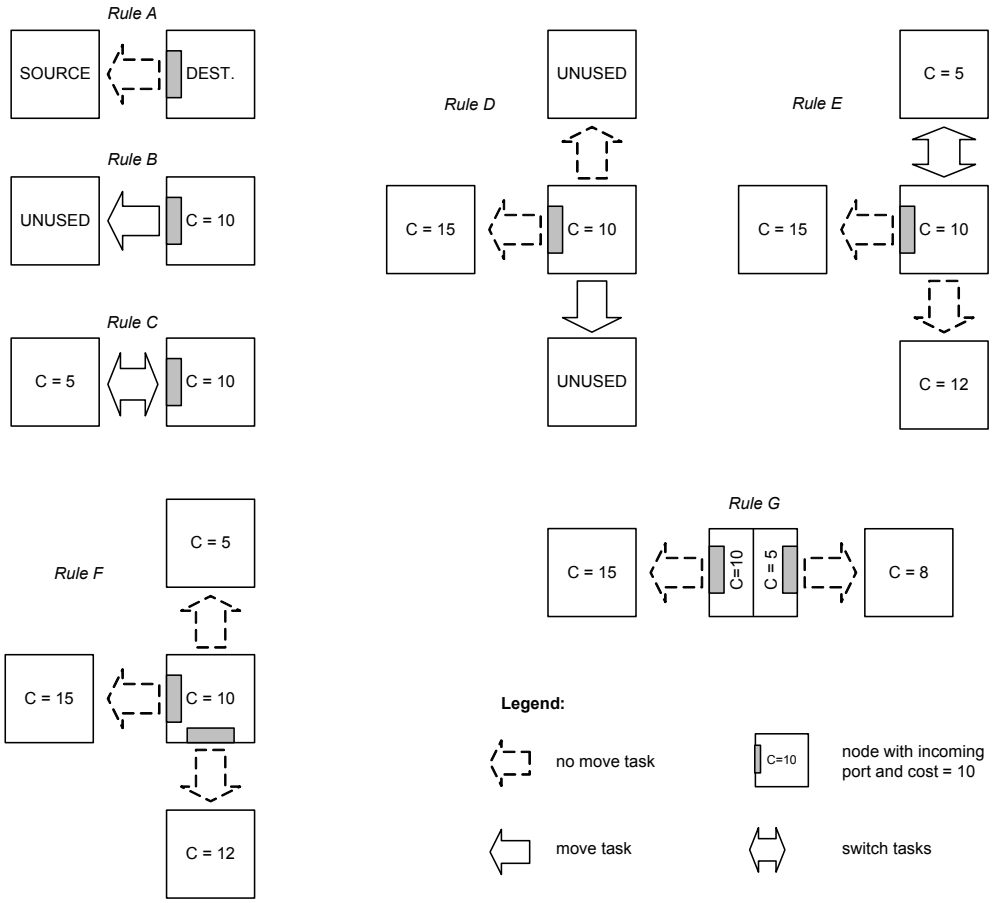


Figure 6.6: Rules for moving tasks across nodes during Step-Adaptive Algorithm processing.

task. When the task reaches its external source from outside of the network, the magnetic coefficient is 2β.

Rule B defines a behaviour when a node moves its task to a free node, see the algorithm line 4. This move is always done independently of its port cost.

Rule C defines a behaviour when a node wants to move a task to an occupied node, see the algorithm line 9. If the port cost $Cost_{best}$ of the node is higher than the port cost $\max(Cost_{next})$ of the occupied node, the tasks at the nodes are swapped.

Rule D defines a behaviour when the neighbour node p_{next} has a higher port cost than the current node, see the algorithm lines 18 and 20. In this case the task at the node is moved to the right node p_{right} or to the left node p_{left}.

Rule E defines a behaviour when the neighbour node p_{next} has a higher port cost and the left and right nodes are occupied, see the algorithm lines 22. The algorithm tests both the max port cost of the left node p_{left} and the right node p_{right}. The task is swapped with the task of the node with a smaller port cost than the current port cost. If both the costs are higher, then the current port cost task stays at its position.

Rule F defines a behaviour of a node with two and more streams. The algorithm chooses the highest port cost $Cost_{best}$ of the current node, and tests if the neighbour node has a higher port cost $max(Cost_{next})$. If the $Cost_{best}$ is smaller, the algorithm chooses the next port with $Cost_{best} > 0$.

Rule G defines a behaviour of a node with two streams coming from opposite sides. When one of the right or left node's port has a lower cost than the cost of the connecting port of the current node, the algorithm moves the task to the neighbour node. The algorithm chooses the first port of the current node with the highest cost $Cost_{best}$.

Algorithm

The pseudo code of the Step-Adaptive Algorithm is on page 94. The algorithm is implemented in each node of the network. The input of the algorithm is node p_n with task v_n, and an actual port cost C_{port} for each port of node p_n. We can split the algorithm to three parts.

The first part from line 1 to line 11 solves an interaction with the neighbour node p_{next} in the direction we want to move the task in the current node. It can moves a task, swap a task with a task in node p_{next}, or escape the algorithm without moving task v_n

The second part from line 12 to line 14 solves nodes with more input streams. The code invokes a loop with the first part to the next port of the node.

The last part from line 16 solves walking around a node with a high port cost. It tries to move the task to the right node p_{right} or the left node p_{left} from the current node. If the node doesn't satisfy any rule, it stays at the same position.

The result of the algorithm improves the position of the task v_n in the case that any of the rules described above was applicable to the node p_n with task v_n. In other cases

task stays in the current node p_n.

6.4 Simulating the Self Adaptive Placement

The previous text introduced the Placement and Step-Adaptive Algorithm for a network on chip. These algorithms were designed with respect to the implementation on the FPGA devices. The FPGA technology restricts mathematical operations mainly to integer types. From this reason we will respect restrictions in the implementation of the algorithms in the simulation framework.

We have built the simulation framework in Visual C++ as a program for simulation of the network. The simulation framework will prove the network architecture introduced in chapter 5 and the placement and the Step-Adaptive Algorithm.

Simulation Framework

The simulation framework *MeshViz*, see screenshot in Figure 6.7, is built in the object oriented C language as a program for personal computer. It implements the simulation tool for a network with 5x5 nodes. The internal clock of the network is tuned by the software timer, which allows to change the speed of the simulation and synchronization of all processes in the network.

The simulation framework contains tools for placing tasks to the network by the *First Node Placement*, *Best Node Placement* and *Multi-Best Node Placement* algorithms introduced in Section 6.1.

A node can contain one task of an application that is running on the network. Applications are placed to the network by one of the placement algorithms. The task and node are identified by coordination $[x, y]$, see Appendix A. A task can migrate from one node to another. This process is equivalent to the partial dynamic reconfiguration. The migration process is driven by the Step-Adaptive Algorithm.

Each node has a small process we denoted in Section 4.3 as an observer. This process monitors incoming and passing flits, and creates statistics for each port. These statistics are of two types. The observer creates the path cost statistic that is influenced by the flit path cost C_{flit} and hop cost statistic that is influenced by flit hop cost H_{flit} of each flit incoming to the destination node. It informs the node which port is loaded by the

Data: Node p_n with task v_n

Result: New node p_{new} for task v_n

```
1  Get max Cost_best of ports from node p_n
2  Neighbour node p_next of node p_n where l_best(p_next, p_n)
3  Get max Cost_next of ports from node p_next
4  if p_next does not contain a task then
5  |  Move task v_n to node p_next
6  else
7  |  if p_next contains source of v_n then
8  |  |  End of the algorithm
9  |  else if l_next < l_best then
10 |  |  Swap task v_n and v_next
11 |  else
12 |  |  if p_n has more inputs then
13 |  |  |  Cost_best = 0;
14 |  |  |  New iteration
15 |  |  else
16 |  |  |  Get max Cost_right of ports from right node p_right
17 |  |  |  Get max Cost_left of ports from left node p_left
18 |  |  |  if p_right has not task then
19 |  |  |  |  Move task v_n to node p_right
20 |  |  |  else if p_left has not task then
21 |  |  |  |  Move task v_n to node p_left
22 |  |  |  else if Cost_right > Cost_left then
23 |  |  |  |  if Cost_best > Cost_left then
24 |  |  |  |  |  Swap task v_n and v_left
25 |  |  |  |  else if Cost_best > Cost_right then
26 |  |  |  |  |  Swap task v_n and v_right
27 |  |  |  |
28 |  |  |  end
29 |  |  end
30 |  end
31 end
```

Algorithm 2: Step-Adaptive Algorithm

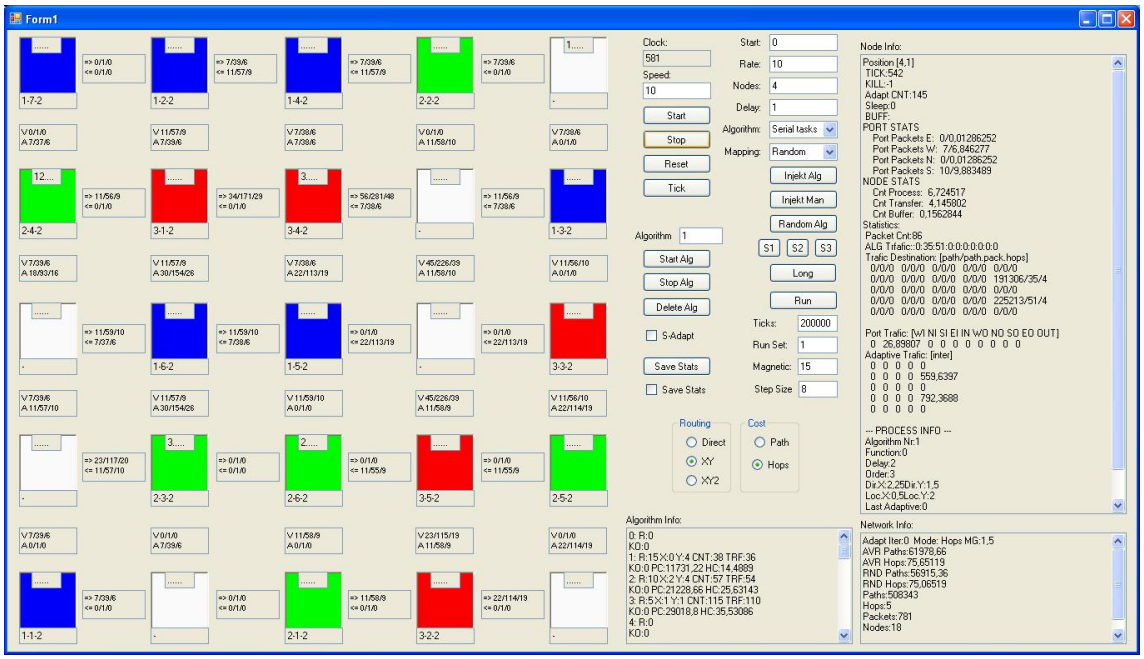


Figure 6.7: Screeshot of the MeshViz simulation framework with a 5x5 mesh network.

incoming flits and the direction of the source nodes. The Step-Adaptive Algorithm can use one of the port costs to migrate tasks in the network.

Placement an Application to the Network

The network can contain maximal 25 different applications. For simulation purposes we can place an application by three placement algorithms and by a random inject. To simulate different cases of the applications we have serial and parallel types of applications, for more details see Appendix B.

The application panel of the simulation framework allows to define a type of an application, placement algorithms and parameters of the application like the number of nodes, delay of nodes and rate of flits for application. For easier comparison we defined six sets of applications that cover the stream type, parallel type and mixed type algorithms, see Appendix B.

The application panel allows to start, stop and release each application separately. By this function we can simulate the life time of the network including introducing a new application invoked by an external event and releasing a finished application from the

network.

Routing Flits

The simulation framework offers three types of routing methods described in Section 5.2. The routing method significantly influences loading of the connection matrix and the Step-Adaptive Algorithm. To simulate the test sets we will use the XY routing algorithm.

Simulation process

The simulation process is controlled by the simulation panel on the right side of the simulation framework. It allows to set the speed of the simulation, start and stop the simulation and manually tick the internal clock. Each node contains an information about its tasks and its basic settings. Each application has its own color that identifies its nodes. The lower line shows an application number, task order number and delay of the task. The upper line shows the content of the node buffer. Each flit in the buffer is represented by its application number.

Measuring the Network

The simulation framework implements a measuring process for each node and link in the network. The measuring process is used to improve the algorithm and calculate the cost of flits going through parts of the network. Each link has its link cost C_{link} that is displayed on each side of the node. The passing flits add the link cost value to its flit path cost and increment their flit hop cost value.

We have a local and global flit path cost and hop cost. The local flit path cost and local hop cost is used only between the source node and the destination node of the flit. The local costs of the flit influence the port cost of the destination node. The global flit path cost is counted over the whole application and it is a sum of all local flit path costs. It gives the application cost and hop. The application cost and hop shows the effectivity of a placed application in the network. The sum of costs of all running applications evaluates the network cost C_{net} and network hop H_{net}.

The simulation framework can make a snapshot of the actual traffic over nodes and links. The mesh graph and its explanation can be found in Appendix A.2. Form this

graph we can see loading of the network parts. The simulation run can save the network costs and application costs in a file to see the improvement process and its results. The sample rate for saving is 500 clock ticks.

The Step-Adaptive Algorithm implemented in the simulation framework uses the cost of network parts to drive the improving process. The algorithm can be set in the panel. We can choose from two port statistics: the port cost and port hop. It is set in the panel. The hop cost is easier to implement in hardware and the path cost is more accurate to identify loading of the ports.

The second parameter that influences the Step-Adaptive Algorithm is the magnetic. It sets the magnetism of two neighbor nodes that form a source-destination pair. The following sections will discuss the impact of the parameter to the Step-Adaptive Algorithm. The value of 1 is equivalent to the value 0.1 defined in Definition 6.3.2.

The third parameter is the stepsize. The stepsize influences evaluation of the cost value for links and network. The value of 1 is equivalent to the value 0.01 for parameter α in Equation 5.4.

Adaptivity Based on the Path Method

The network with six application sets S1 to S6 has been simulated. The Step-Adaptive Algorithm was driven by the path cost (later called path method) to test its attributes. We did 150 runs for each set of applications to get the behaviour of the Step-Adaptive Algorithm. Each 25 runs from a total of 150 runs have a different magnetic coefficient to observe its impact to the Step-Adaptive Algorithm. The application set can be split to two groups. First, application sets S1, S2 and S3 test applications with different communication loads. Second, application sets S4, S5 and S6 have the same communication load. Graphs of the simulation progress for the first group is in Figure C.2 and for the second group in Figure C.4. Graphs on the left side show the network hop cost and graphs on the right side show the path cost of the running application set. The optimal placement application set on the network is represented by the line *Minimal network cost*. The weighted average of each run with the same magnetic coefficient is represented by the same color. We have three magnetic coefficients for the path method. Magnetic coefficient 10 is small enough not to influence the Step-Adaptive Algorithm, magnetic coefficient 15 already influences short paths, and magnetic coefficient 19 doubled the magneticity between two

nodes.

The simulation for the first group should test behaviour of applications with a low and high communication load. We want to know if the magnetic coefficient can influence the Step-Adaptive Algorithm and see its impact on high communication load applications as well as low communication load applications.

From Graphs C.3(a) and C.3(b) of the set S1 it can be seen that different magnetic coefficients can generate better results. The run with the magnetic coefficient 10 has a result than runs with higher magnetic coefficient. This result was supported by another measurement on the set S1. From the graphs C.3(c) and C.3(d) of the set S2 it can be seen that different magnetic coefficients cannot improve results of the Step-Adaptive Algorithm. From Graphs C.3(e) and C.3(f) of the set S3 it can be seen that a higher magnetic coefficient cannot improves result of the Step-Adaptive Algorithm.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	27.5/254	20.6/254	74.6/94	36.1/233	45.8/132	29.8/133
$\beta = 15$	29.1/434	22.6/591	72.5/55	36.2/272	47.9/277	32.4/349
$\beta = 19$	27.6/303	21.3/831	84.0/134	38.3/134	47.2/322	31.0/322

Table 6.1: Minimal network hop and path cost of the Step-Adaptive Algorithm with the path method on the test sets S1, S2, S3 and the iteration when it reached minimum.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	1.000	1.001	1.102	1.056	1.006	1.023
$\beta = 15$	1.058	1.099	1.070	1.058	1.053	1.112
$\beta = 19$	1.003	1.033	1.241	1.119	1.037	1.066

Table 6.2: Approximation of network hop and path cost of the Step-Adaptive Algorithm with the path method on the test sets S1, S2, S3. Value 1 means minimal cost.

From Table 6.1 we can see the best network cost and the iteration it was reached in by the Step-Adaptive Algorithm with different magnetic coefficients. From Table 6.8 we can see that all the sets can get very near to the minimal network cost. For the set S1 it can reach the minimal network cost.

The simulation on the second group should test the behaviour applications with same communication load and impact of magnetic coefficient to the Step-Adaptive Algorithm. From Graphs C.4(a) and C.4(b) of the set S4 we can see that the magnetic coefficient can significantly improve result of the Step-Adaptive Algorithm. This is caused by the elimination of power that holds neighbour nodes in case of M=10. Without the magnetic coefficient the stream applications strongly influence themselves and they cannot get near to the minimal network cost. From Graphs C.4(c) and C.4(d) of the set S5 and Graphs C.4(e) and C.4(f) of the set S6 we can see that the magnetic coefficient cannot influence the results. It is because tasks in parallel applications have more input streams from different sides, and cannot migrate to only one source. In such cases the magnetic coefficient cannot be used.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	53.1/988	33.5/747	20.2/193	20.3/253	64.2/48	35.0/392
$\beta = 19$	48.0/288	30.0/288	21.2/110	21.6/244	71.5/34	36.4/129
$\beta = 29$	48.0/597	30.0/597	20.2/385	20.2/203	68.2/317	37.7/324

Table 6.3: Minimal network hop and path cost of the Step-Adaptive Algorithm with the path method on the test sets S4, S5, S6 and the iteration when it reached this minimum.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	1.106	1.118	1.000	1.014	1.174	1.166
$\beta = 19$	1.000	1.000	1.019	1.080	1.306	1.213
$\beta = 29$	1.000	1.000	1.000	1.012	1.247	1.256

Table 6.4: Approximation of network hop and path cost of the Step-Adaptive Algorithm with the path method on the test sets S4, S5, S6. Value 1 means minimal cost.

Adaptivity Based on the Hop Method

Another simulation was done with different input parameters of the Step-Adaptive Algorithm with the same six groups of application sets as in the simulation above. The

Step-Adaptive Algorithm was driven by the hop cost (later called the hop method) to test its attributes. We performed from 150 to 600 runs for each set of the application as before to get the behaviour of the Step-Adaptive Algorithm driven by the hop cost. Each group of runs have a different magnetic coefficient.

Graphs of the simulation progress for applications with different communication loads is in Figure C.3 and for applications with the same communication load in Figure C.5.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	27.5/351	20.6/356	68.8/186	34.7/310	47.5/227	29.6/349
$\beta = 20$	27.5/341	20.6/341	73.7/17	37.8/248	50.2/384	29.5/172
$\beta = 30$	28.7/913	22.6/477	77.4/33	36.2/177	47.6/244	31.1/346

Table 6.5: Minimal network hop and path cost of the Step-Adaptive Algorithm with the hop method on the test sets S1, S2, S3 and the iteration when it reached this minimum.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	1.000	1.001	1.016	1.014	1.045	1.016
$\beta = 20$	1.000	1.001	1.088	1.105	1.104	1.015
$\beta = 30$	1.043	1.098	1.144	1.060	1.047	1.068

Table 6.6: Approximation of network hop and path cost of the Step-Adaptive Algorithm with the hop method on the test sets S1, S2, S3. Value 1 means minimal cost.

From the simulation on the first group we can see a small difference in the progress of the stream type applications, in Figures C.3(a) and C.3(b). Other two sets don't show significant differences between various magnetic coefficients when we use the hop method for driving the Step-Adaptive Algorithm. From Graph C.3 and Tables 6.7 and 6.8 we can reach the same conclusions as for simulations with the path cost method.

The simulations on the second group of application sets don't bring any differences between the hop method and the path method.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	48.0/567	30.0/567	20.3/376	21.9/377	58.8/140	32.5/373
$\beta = 30$	50.0/395	31.6/403	20.3/268	21.1/214	55.2/383	30.9/383
$\beta = 50$	53.3/262	33.3/262	22.1/116	22.8/159	60.5/131	32.8/328

Table 6.7: Minimal network hop and path cost of the Step-Adaptive Algorithm with the hop method on the test sets S4, S5, S6 and the iteration when it reached this minimum.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	1.000	1.000	1.003	1.093	1.075	1.083
$\beta = 30$	1.042	1.055	1.007	1.053	1.009	1.030
$\beta = 50$	1.110	1.110	1.096	1.141	1.106	1.092

Table 6.8: Approximation of network hop and path cost of the Step-Adaptive Algorithm with the hop method on the test sets S4, S5, S6. Value 1 means minimal cost.

Step Adaptive Algorithm Comparison

We introduced two methods for driving the Step-Adaptive Algorithm and test them on six application test sets to find all features specific to each method. We will split the simulation results to two groups. Group one will contain test sets S1, S2 and S3 with different communication loads. Group two will contain test sets S4, S5 and S6 with the same communication load. To compare the simulation results we count the average network cost for each test set and for each magnetic coefficient. The average cost is not count over all iterations of the improvement process, but after the network cost stabilized.

Test sets S1, S2 and S3

Table 6.9 contains average network costs calculate by the path method and Table 6.11 contains average network costs calculate by the hop method. From these two tables we can see that both methods have very similar results, and we cannot exactly say which of the methods works better. Tables 6.10 and 6.12 contain a proportion between the minimal network cost and average network cost. Both the methods are able to reach a proportion

between 1.5 and 1.2.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	34.3	25.9	105.6	42.1	65.9	39.2
$\beta = 15$	41.3	30.2	107.6	42.3	64.1	37.9
$\beta = 19$	42.4	31.2	109.4	42.8	64.1	38.7

Table 6.9: Average of the network hops and path cost of Step Adaptive algorithm with the path method on the test sets S1, S2, S3.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	1.246	1.259	1.560	1.232	1.448	1.349
$\beta = 15$	1.502	1.464	1.589	1.238	1.409	1.301
$\beta = 19$	1.542	1.515	1.616	1.252	1.410	1.331

Table 6.10: Average approximation of the network hops and path cost of Step Adaptive algorithm with the path method on the test sets S1, S2, S3. Value 1 means minimal cost.

Graphs on pages 132 and 133 show the network cost progress during the Step-Adaptive Algorithm. The graphs contain the minimal network cost and network cost reached by the Multi-Best Node placement in an empty network (MBN Placement). Improving network placement by Step adaptive algorithm is not able to reach at least the same network cost as by the MBN placement. It is caused by one dominant application with high communication load being present in each test set.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	39.0	28.0	97.6	41.8	69.2	40.3
$\beta = 20$	32.6	25.2	97.1	41.6	62.7	37.8
$\beta = 30$	39.5	29.6	97.7	42.2	65.5	39.2

Table 6.11: Average of the network hops and path cost of Step Adaptive algorithm with the hops method on the test sets S1, S2, S3.

	Set1 C_{net}	Set1 H_{net}	Set2 C_{net}	Set2 H_{net}	Set3 C_{net}	Set3 H_{net}
$\beta = 10$	1.418	1.358	1.442	1.221	1.522	1.383
$\beta = 20$	1.186	1.223	1.434	1.216	1.379	1.298
$\beta = 30$	1.436	1.439	1.443	1.234	1.440	1.346

Table 6.12: Average approximation of the network hops and path cost of Step Adaptive algorithm with the hops method on the test sets S1, S2, S3. Value 1 means minimal cost.

Test sets S4, S5 and S6

Table 6.13 contains average network costs gained by the path method on test sets S4, S5 and S6 Table 6.15 contains average network costs gained by the hop method on the same test sets. From Tables 6.13 and 6.15 we can see that both the methods have problem with improving the network placement without the magnetic coefficient M=10. From Graphs C.4(b) and C.5(b) we can see that a small magnetic coefficient can cause oscillations in the network cost with the same stream applications.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	110.7	59.0	26.1	25.1	104.4	48.1
$\beta = 19$	60.6	36.6	26.8	25.8	97.8	47.8
$\beta = 29$	68.5	40.7	26.2	25.1	98.3	47.3

Table 6.13: Average of the network hops and path cost of Step Adaptive algorithm with the path method on the test sets S4, S5, S6.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	2.306	1.967	1.294	1.255	1.908	1.603
$\beta = 19$	1.262	1.220	1.327	1.288	1.788	1.594
$\beta = 29$	1.427	1.356	1.295	1.256	1.798	1.576

Table 6.14: Average approximation of the network hops and path cost of Step Adaptive algorithm with the path method on the test sets S4, S5, S6. Value 1 means minimal cost.

The main difference between the methods can be seen for applications of the parallel

type (set S5). The result of the path method is two times better than for the hop method. Sets S4 and S6 show similar results for both methods.

The result of the Step-Adaptive Algorithm is close to the Multi-Best Node placement algorithm for both methods. It is caused by the same communication load of applications, the MBN Placement algorithm places applications sequentially and the second application already cannot occupy the best nodes for itself. The network cost of the MBN Placement strongly depends on the placing order of applications mainly with different communication loads.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	104.0	56.0	34.0	28.6	88.8	44.5
$\beta = 30$	68.9	39.8	33.3	27.8	93.7	44.7
$\beta = 50$	67.1	40.7	34.2	28.2	89.0	43.0

Table 6.15: Average of the network hops and path cost of Step Adaptive algorithm with the hops method on the test sets S4, S5, S6.

	Set4 C_{net}	Set4 H_{net}	Set5 C_{net}	Set5 H_{net}	Set6 C_{net}	Set6 H_{net}
$\beta = 10$	2.167	1.865	1.682	1.430	1.624	1.485
$\beta = 30$	1.436	1.328	1.649	1.389	1.713	1.491
$\beta = 50$	1.397	1.358	1.694	1.412	1.628	1.434

Table 6.16: Average approximation of the network hops and path cost of Step Adaptive algorithm with the hops method on the test sets S4, S5, S6. Value 1 means minimal cost.

From the previous simulations and comparisons of both the methods and from the comparison graph in Figure 6.8 we can conclude that for applications with different communication loads we can use both the methods with similar results. For applications with the same communication loads we need the magnetic coefficient and the path method can have better results for parallel applications.

The graphs on page 136 show a standard deviation for all the test sets and the hop method. The graphs indicate that the Step-Adaptive Algorithm has stable results for all test sets.

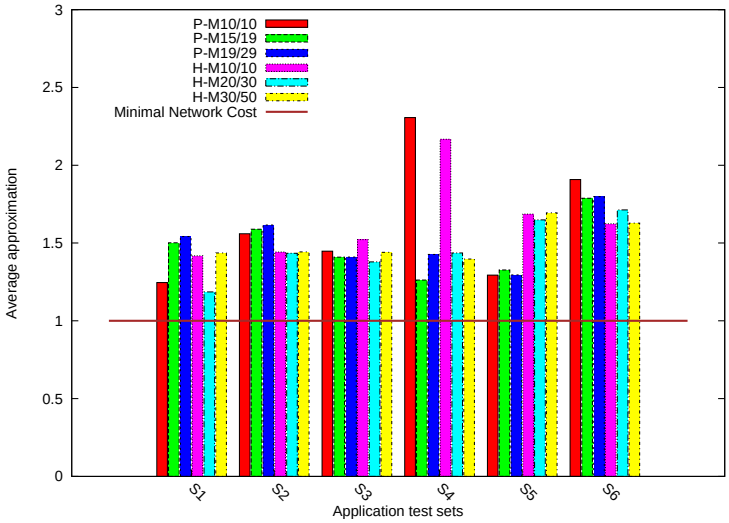


Figure 6.8: Bar graph compares average approximation of application test sets.

Network Cost Influenced by Applications

To choose the right composition of applications in the network and the right magnetic coefficient we need to know how parameters of the running applications influence the network cost. Figure C.1 shows an impact of application costs on the network cost during progress of the Step-Adaptive Algorithm. The network cost is mainly influenced by applications with higher communication load. The low communication load applications have only a small impact on the network cost. From this we can say that when the application with a higher communication load reaches a cost near to the minimal cost, the improvement process can be stopped. Longer runs do not bring about any improvement. The designs that have to reduce task migration can improve only the placement of high communication load applications.

Stream Application Tests

Stream type applications are most used in image processing and computer graphics. Parallelism of these applications can be seen on more stream applications that work concurrently in one network. From this reason we want to know the behavour of the Step-Adaptive Algorithm in the network with only stream applications. As two test cases we chose five stream applications, each with four nodes, and two stream applications, each with nine

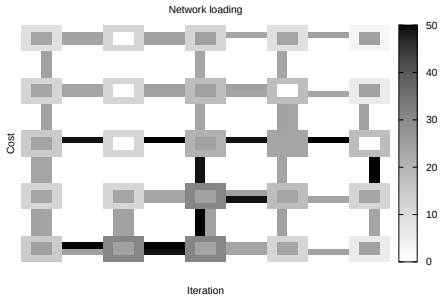
nodes. All the applications have the same rate of incoming flits. We want to see the interaction of the applications as the placement, and their effectiveness in terms of use of the network resources us.

As in the first test case we run five equal stream applications in the network, see Figure 6.9. We use two placement algorithms to inject applications to the network. To simulate a real situation when applications are injected in the network that already processes a number of applications we use random placement. The loading graph of the network with the random placement algorithm is in Figure 6.9(a). We can see an unfolded communication in the whole network and the network path cost C_{net} is 595, and the network hop cost H_{net} is 165. The minimal network path cost in this case is 121.3, and the network hop cost is 50. This placement reached the to minimal network cost after 350 iterations.

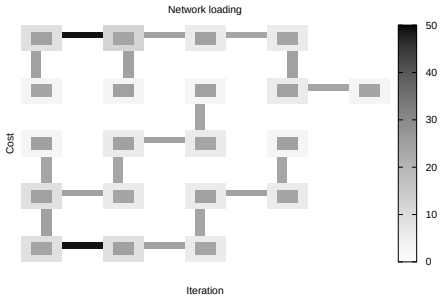
To simulate network cost improvement after injecting five stream applications in an empty network we used the Multi-Best Node Placement Algorithm. This placement algorithm reaches a better network cost than random placement. The network starts with the network path cost C_{net} equal to 194, and the network hop cost H_{net} equal to 64. This placement reached the minimal network cost after 145 iterations. The minimal cost is reached faster than with the random placement because the initial network costs are smaller. The Step-Adaptive Algorithm for this test case generates the minimal network cost. The result of the network loading is in Figure 6.9(c). We can see that all applications were replaced to chains to optimize their communication. Figure 6.9(d) shows the improving progress of the network hop cost and the network path cost with the random placement (Rand.Place.) and with the Multi-Best Node Placement (MBN Place.). The Step-Adaptive Algorithm used the magnetic coefficient equal to 20.

The comparison of the two different starts of the network shows that both reached the minimal network cost. The run with the Multi-Best Node placement reached the minimal network cost two times faster than with the random placement. The improving Multi-Best Node placement can downgrade the network cost for a short time, but the final result gets improved by 37%. In the case of the random placement the improvement is 79.6%, but because the starting network cost is three times worse. Comparing Graphs 6.9(a) and 6.9(b) we can see that the Multi-Best Node placement can eliminate the problem with overloaded node links.

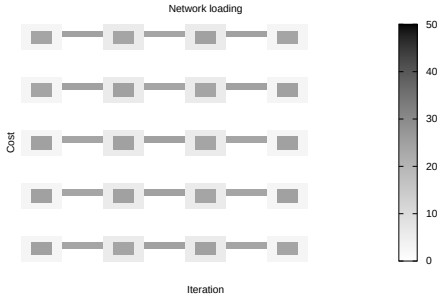
As the second test case we run two equal stream applications in the network, see Figure



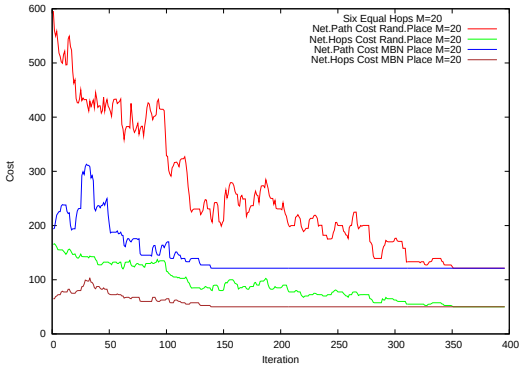
(a) Network load after placement an application with the Random Node Algorithm



(b) Network load after placement an application with the Multi-Best Node Alagorithm



(c) Network load after improving a placement of an application by the Step-Adaptive Algorithm



(d) Improving network cost C_{net} and H_{net} with the Step-Adaptive Algorithm with the Random Node placement and Multi-Best Node placement at the start of the network.

Figure 6.9: Evolution of network parameters when using the Step-Adaptive Algorithm for six equivalent stream applications.

6.10. We use two placement algorithms as in the first test case to inject applications to the network. The loading graph of the network with the random placement algorithm is in Figure 6.10(a). We can see unfolded communications in the whole network, and network path cost C_{net} is 450, and the network hop cost H_{net} is 156. The minimal network path cost for this test case is 68.8, and the network hop cost is 36. The random placement reached the minimal cost after 660 iterations.

We injected two equal stream applications by the Multi-Best Node placement algorithm to simulate network cost improvement by the Step-Adaptive Algorithm. The network starts with the network path cost C_{net} with 76.4 and the network hop cost H_{net} with 40.

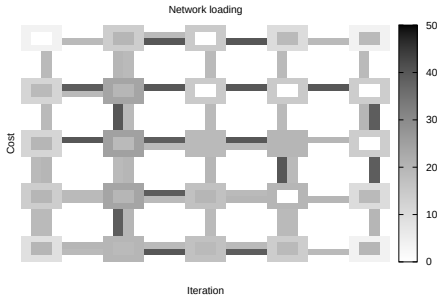
From the simulation results we see that behaviour of applications during the improving process depends on the magnetic coefficient. When the magnetic coefficient is smaller than 30, the applications oscillate, and the network cost can increase as we can see in Figure 6.10(d), blue and violet lines. When the magnetic coefficient is higher than 30, the magneticity between the neighbor nodes is big enough to generate placement as best as with the Step-Adaptive Algorithm, see Figure 6.10(d), light blue and yellowlines.

We can see that all applications in both the tests were reshaped to chains that are near to most efficient in terms of communication. The conclusion of the tests for stream applications is that the improvement process leads a near optimal placement. From the results of the second test case we can see that, a higher magnetic coefficient produces better results with the of Step-Adaptive Algorithm, and it can prevent bad placement by the Multi-Best Node placement algorithm.

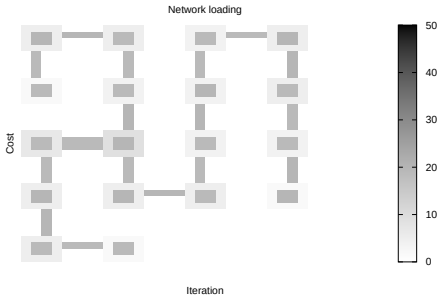
Comparing the Placement and Self Adaptive Algorithm

We simulated the Step-Adaptive Algorithm with different values of the magnetic coefficient and the measuring method in the previous text. All these tests were done on the test sets from Appendix B. The test sets test the interaction of applications with each other during the improvement process. The test sets cannot prove the behavour of the improvement process when placing new applications to the network.

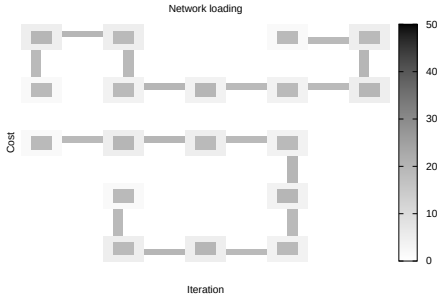
This section will test the Step-Adaptive Algorithm in real network life when new applications are placed to the network and finished applications are released from the network. For placing a new application we use the Multi-Best Node placement algorithm and the random placement algorithm that places an application to a running network.



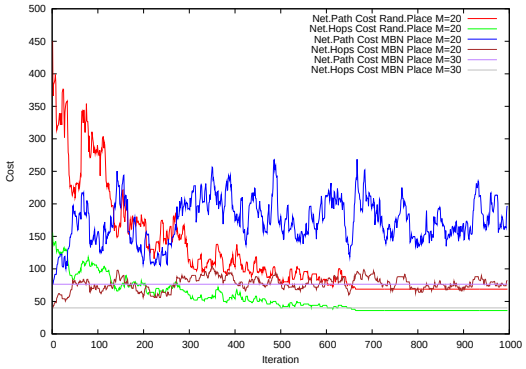
(a) Network load after placement an application with the Random Node algorithm



(b) Network load after placement an application by the Multi-Best Node algorithm



(c) Network load after improving the placement of applications by the Step-Adaptive Algorithm



(d) Improving network cost C_{net} and H_{net} with the Step-Adaptive Algorithm with the Random Node placement and Multi-Best Node placement at the start of the network.

Figure 6.10: Evolution of network parameters when using the Step-Adaptive Alagorithm for two equivalent stream applications.

Type	Place	Release	Rate	Nodes
Stream	77	237	12	2
Stream	77	1197	10	2
Stream	77	1197	8	2
Stream	77	237	15	2
Stream	77	877	4	2
Stream	77	1197	6	2
Stream	77	877	4	2
Parallel	557	2797	8	9
Parallel	957	1517	3	8
Parallel	1197	2157	15	6
Stream	1837	2477	15	4
Stream	1837	2477	3	4

Type	Place	Release	Rate	Nodes
Parallel	2317	2797	15	3
Parallel	2317	2477	4	3
Stream	2637	3434	15	2
Stream	2637	-	10	2
Stream	2637	-	3	2
Stream	2637	-	15	2
Stream	2637	-	10	2
Stream	2637	-	3	2
Stream	3117	-	15	2
Stream	3117	-	10	2
Stream	3117	-	3	2
Stream	3434	-	3	4

Table 6.17: List and parameters of applications placed and released on the network to test impact of the Step-Adaptive Alagorithm to the network cost.

The Multi-Best Node placement algorithm improves positively the cost of applications by choosing the best free node for each task of the application.

Table 6.17 summarizes applications and their parameters that we use to test the Step-Adaptive Algorithm in real life of the network. Each line of the table represents one algorithm that is placed to the network. The algorithm is described by its type, placing time and releasing time. Further important parameters are the rate of the flits processed by an application and the number of nodes that are occupied by an application. A closer description of the stream and parallel application types is in Appendix B.

First, we test only application placing by the Multi-Best Node placement algorithm without any adaptation by the Step-Adaptive Algorithm. Figure C.7(c) shows the network hop cost (blue line). Figure C.7(d) shows network path cost (blue line). Each change of the value is caused by placing or releasing an application. We can see cases with low communication load and high communication load of the network communication that is equivalent to high and low network cost.

To test if the Step-Adaptive Algorithm can bring a better network cost in time than just placing applications to the network, we measured the network cost during the life

time of the network with applications like in the first test case. In this case we start the Step-Adaptive Algorithm with the hop method and the magnetic coefficient equal to 20. The result of the test is in Figure C.7(c) that shows the network hop cost (red line) and Figure C.7(d) that shows the network path cost (red line). From these graphs we can see that in almost all cases the Step-Adaptive Algorithm improved the network costs after any change in the applications running in the network.

When we study the graphs in Figure C.7, we can note the placement process changes the network cost. The Step-Adaptive Algorithm starts improving the placement and the network cost that was changed by a new application. In the case of an application with a heavy communication load the improvement became better than in the case of application with low a communication load.

An application with a heavy communication load is placed to free nodes in the network. But these nodes cannot be really the most suitable nodes, and the application can significantly increase the network cost. A typical case is when we place several low communication load applications, and later we place heavy communication load applications. The heavy communication load application cannot be placed near the data source because of the previous application. It will increase mainly the network path cost. The Step-Adaptive Algorithm starts moving tasks with a heavy communication load to their source and push low communication load tasks out from their nodes.

When we compare the types of applications running in the network, we can see that the stream applications reached better improvement by the Step-Adaptive Algorithm than the parallel applications. It is due to heavy transfer between nodes with serial connections.

Another conclusion is that more small applications have better improvements than fewer bigger applications. It is caused by the independency of the small applications. The tasks of the small applications can be moved easier than large applications. Large applications create long walls from tasks tied together by their communications.

From the test of real application cases in the network we can conclude that the placement algorithm can bring some improvement compared to the random node placement, but there is still enough space to improve the network cost. This improvement can be done by the Step-Adaptive Algorithm. The Step-Adaptive Algorithm brings improvements in most of the cases from 20% to 40% compared to the network cost gains by the Multi-Best Node Placement Algorithm.

6.5 Simulation Results

The previous sections introduced the simulation framework with the Step-Adaptive Algorithm and two methods for driving the improvement process in the running network. With these tools and methods we simulated six types of application sets that cover many real but different application cases. From the simulation we can assess the application behaviour in the network and their improvement.

We introduced two methods for driving the Step-Adaptive Algorithm. The complex path cost method that respects the communication load of the links in the network and the easier hop cost method that uses only the distance between a source task and a destination task. In the current state both methods have very similar results, and they can be used for driving improvement process successfully. The implementation cost of the path method is much higher than the hop method, we can conclude that the hop method is better for the self-Adaptive network on an FPGA device. Probably the path method can have much better results in connection with the routing algorithm that will open another dimension for routing and placing to the network.

We designed the Step-Adaptive Algorithm for improving task placement in the network. With this we introduced the magnetic coefficient that holds neighbor tasks together. From the simulation we can see that the magnetic coefficient can significantly influence stream type applications with the same communication load. Generally the magnetic coefficient can control all stream type applications placing. We can use the magnetic coefficient to force an application to group tasks in a chain and decrease delays in a concrete application. In the case of parallel applications the magnetic coefficient doesn't work because the task works with more input streams.

The Step-Adaptive Algorithm is designed to improve placement of applications in a running network. Injecting and releasing an application from the network increase task fragmentation in the network, and we need the Step-Adaptive Algorithm for task defragmentation. The Step-Adaptive Algorithm improves network fragmentation even when we use the Multi-Best Node placement algorithm to inject applications in the network. From the simulation of a real life time of the network we can see that the Step-Adaptive Algorithm can significantly improve the placement. An average improvement during the life time of the network is between 20% and 40%. It strongly depends on the type of the

application and on the communication load of the application.

We simulated a self-Adaptive network in the simulation framework to find suitable network topology and improvement algorithm. All restrictions for simulation come from the FPGA platform that is our target platform for the self-Adaptive network. They reguire us to find an undemanding topology and algorithms that can be implemented on the FPGA devices. From this reason we prefer the Step-Adaptive Algorithm with the hop method for implementation. The result of the hop method is similar as the path method, but the implementation of the cost calculation hardware block is much easier than for the path method.

6.6 Summary

The chapter discussed application placing to a network and continuous improvement of placed running applications. At the beginning of the chapter we defined a network model and an application model and a placing process that places applications to the network. We introduced a matrix that contains information about the current placement on the network and its implementation in hardware. For the placing application we designed three placement algorithms. Each placement algorithm is suitable for a different application type and can handle the placing process with different speed and effectivity of placing tasks to the network.

The next part of the chapter introduced measuring efficiency of processing data in the network. We defined cost functions of links, paths and network that can be measured by two methods, by the path cost that reflects the cost of delivering packets from a source node to a destination node, and the hop cost that reflects the distance between a source node and a destination node. The cost values are inputs to the adaptation process and can explain its effectivity and successfulness.

The self-Adaptive placement section introduces the Step-Adaptive Algorithm and its principles. The Step-Adaptive Algorithm is a mechanism that moves tasks in the network and increases the effectivity of processing data in the network.

The end of the chapter test the Step-Adaptive Algorithm on six sets of testing applications that cover stream type applications, parallel type applications and mixed type applications. There are two groups of test runs. First there are tests of the Step-Adaptive

Algorithm with the cost function based on the path cost, and second we run tests with the hop cost function. The text continues with a comparison of these two groups of test runs. Finally we present a test on a real network life with consecutive injection of new applications to the network and releasing finished applications. These tests are done only for the placement algorithms and for runs with placement algorithm and the Step-Adaptive Algorithm to see if the improvement can be achieved on the running network.

Chapter 7

Conclusion

The basic analysis of the partial dynamic reconfiguration of FPGA devices has been done to the start of the thesis. Three different reconfiguration types have been introduced and their complexity was determined. The *full bitstream reconfiguration* with $O(n)$, *differential bitstream reconfiguration* with $O(n^2)$ and *empty bitstream reconfiguration* with $O(n)$. Each of the reconfiguration types is suitable for different applications.

Stream type applications and control type applications have been identified as two basic classes of reconfigurable applications. It opens a new dimension in FPGA devices, increases variability and adaptability of the hardware. For this reason the functional density has been defined and the basic condition for maximizing the function density has been specified as *reconfiguration time* $\ll$ *execution time*

On the previous knowledge the methodology for dynamic reconfiguration has been prepared and the technological barriers have been solved. The two commercially available FPGA platforms have been presented - Virtex from Xilinx Inc. and FPSLIC from the Atmel Corporation, both have features that allow to implement partial dynamic reconfiguration. The reconfiguration controller and the wrapper to connect dynamic modules with the static part has been designed. With the platforms and the basic blocks the way to implement a reconfigurable coprocessor has been opened. We implemented two reconfigurable coprocessors on the FPGA platforms. The comparison of the coprocessors shows that the FPSLIC platform is suitable for small and portable devices with low power consumption and Virtex is suitable for complex and large designs with the main demand on device speed.

To open new possibilities of the partial dynamic reconfiguration self-adaptivity has been determined as an important principle for future reconfigurable devices. The requirements of self-adaptive systems have been researched, and a self-adaptive element has been designed as the basic building block of the self-adaptive systems based on the FPGA devices. The basic building blocks of the self-adaptive element have been defined: *Reconfigurable computing unit*, *Observer*, *Controller unit* and *Communication interface*. To test the design of the self adaptive element a ring topology network with four elements with FIR filter functions has been implemented. The results of the tests on the ring network show that the principle of the self adaptive element works. We found important to store the past state in the elements and not to transport them in packets together with data.

The analysis of the networks on chip has been done to find the most suitable connection network to connect self-adaptive elements in a self-adaptive system. Seven network topologies have been presented and compared from the side of the hardware cost and the communication cost. As the most suitable network for self-adaptive elements implemented on the FPGA devices we chose the 2D-Mesh network. The scalability and regular shape has been found as important features for the self-adaptive system.

Because of huge implementation demand in hardware of a network on chip with self-adaptive elements we only simulated the parts of the self-adaptive systems. We verified their function and a possibility of a real implementation of the partial dynamic reconfiguration on the FPGA devices and the function and structure of the self-adaptive element. Parameters gained from the previous implementations we used in the simulation of the future self-adaptive systems. The model of the network and application has been defined. The placement process has been described by three placement algorithms suitable for stream and parallel application types. The placement algorithms have been designed with respect to the restrictions of the FPGA technology and the possibility to simulate real application scenarios.

To adapt functions of the network and improve the parameters of the running network, parameters of the network have been defined and the Step-Adaptive Algorithm has been designed. The Step-Adaptive Algorithm uses features of the partial reconfiguration and the function of the observer block in the self-adaptive element to move tasks across nodes in the network and improve the network parameters.

To verify the Step-Adaptive Algorithm and network topology the simulation framework

has been developed on the personal computer platform. Six application test sets have been composed to cover all possible real cases. We defined two measuring methods for the Step-Adaptive Algorithm: The *Path cost method* that is more complex, but it has a higher hardware cost, and the *Hops cost method* that covers only distances, but its hardware cost is lower. We have been simulating runs for both methods and for all six application test sets to find the features of the Step-Adaptive Algorithm in the mesh network. From the simulation we can see that both methods have the same results in the case of concurrently running applications with different communication load. In the case of applications with the same communication load we found that stream applications need the magnetic coefficient to get better results and parallel applications have better results with the path cost method. From the results of the simulation we can conclude that the path cost methods have not enough. Better results compared to its hardware cost, and we can say that the hop cost method with the magnetic coefficient from 20 up can bring the best ratio between the implementation demands and improving the results in the mesh network.

At the end a simulation of a real case of the self-adaptive network has been tested. A set of 24 different applications has been continuously placed and released to/from the network by the Multi-Best Node placement algorithm with and without the Step-Adaptive Algorithm. The result was that the self-adaptivity can bring about 20% to 40% improvement. The bigger improvement can be gained for applications with high communication load.

The contribution of the thesis is the analysis of the possibilities of partial dynamic reconfiguration and its composition with self-adaptivity. The self-adaptive elements connected in a scalable and parameterized network opens new possibilities for implementation of stream processing designs with high functional density. It decreases the cost of the hardware by increasing its functional density. The thesis introduced placement and self-adapting algorithms with respect to the restrictions of the FPGA technology. With an increasing size of the FPGA devices a future implementation of huge networks with self-adaptive elements can manage future complex multi-core designs.

The future work will focus on implementing knowledge in modern FPGA devices and preparing a self-adaptive platform to accommodate a wide range of stream applications.

7.1 Objectives Revisited

In this section, the dissertation objectives presented in introduction of the thesis are briefly reviewed and the achieved results are presented:

- To introduce a new technology of partial dynamic reconfiguration we performed a low level analysis of the reconfiguration process and the design methods that lead to designing reconfigurable hardware on reconfigurable devices.

  The analysis of the complexity of dynamic reconfiguration has been done and two classes of dynamically reconfigurable designs have been introduced in Chapter 2.

- To increase the variability and adaptability of the reconfigurable hardware we analyzed the functionality of the partial reconfigurable hardware and its restrictions and extensions.

  The increasing functionality of the reconfigurable design has been presented and the ratio between the reconfiguration time and the processing time has been presented as the key parameter of the reconfiguration design in Chapter 2.

- To open a new reconfigurable platform the methodology of the design. Hardware has been modified to cover specific steps that allow to implement design with partial reconfiguration.

  An analysis of two commercially available FPGA platforms has been presented. A methodology for designing reconfigurable designs has been introduced and configuration bitstream organization, the wrapper module and the reconfiguration controller have been presented in Chapter 3 as the key parts of a reconfigurable design.

- To validate the possibility of using the partial dynamic reconfiguration the reconfigurable coprocessors with reconfigurable features will be built on two commercially available hardware platforms that allow to implement the reconfiguration process.

  Two reconfigurable FPGA coprocessors based on the presented platforms have been designed and implemented to verify the theoretical knowledge presented in Chapter 2 and Chapter 3. Both reconfiguration coprocessors have been compared.

- To increase the variability of the reconfiguration features the self-adaptive element will be designed as the basic building element of self-adaptive systems based on reconfigurable hardware. The self-adaptive element will be designed as an independent IP core that will be able to adapt its function according to the requirements ot the environment.

  Self-adaptivity and the key factor and requirements have been presented in Section 4.1. The self-adaptive element and its basic building block have been designed in Section 4.2 as the basic element of future self-adaptive systems. An implementation of a self-adaptive system with self-adaptive elements has been done in Section 4.4 to prove the design of the element and self-adaptibility of the whole system.

- To create a suitable environment for the self-adaptive elements the analysis of the current network on chip topologies suitable for reconfigurable hardware will be done. According to this we will choose the best network topology to connect self-adaptive elements.

  The analysis of the network on chip topologies has been done in section 5.1 to find best topology for the self-adaptive system. The 2D-Mesh topology has been chosen in section 5.2 as the most suitable topology to connect self-adaptive elements in a self-adaptive system.

- To use the chosen network the basic parameters and placement algorithms have to be introduced. They ensure first the injection of an application to the network and its start and interaction with already running applications in the network.

  The network and application model have been introduced. Three placement algorithms have been designed in Section 6.1.

- To measure and evaluate the effectiveness of using the network the cost functions will be set up. They will be the input parameters to the adaptation process.

  The network cost parameters have been set up in Section 6.2 to measure the successfulness of the adaptation process.

- To improve the effectiveness of the running network that is fragmented by injecting and releasing applications an adaptive algorithm has to be designed. The adaptive

algorithm will be part of each node in the network, and it will guarantee that nodes will adapt their function to the most suitable function for a given case in the possible range of node's neighborhoods.

The Step-Adaptive Algorithm has been designed in section 6.3. The two cost methods have been used to verify the Step-Adaptive Algorithm and the methods have been compared. A real example of a self-adaptive system has been simulated to show how an application placement improves due to the adaptation process in the network.

7.2 Summary of Author's Contribution

The author's contribution consists of:

- Analysis of the complexity of the reconfiguration process.
- Cooperation on implementation of the reconfigurable coprocessors based on the Atmel FPSLIC platform and the reconfigurable methodology for the Atmel FPSLIC platform.
- Implementation of the self-adaptive element and the self-adaptive system on a ring network to verify the design of the self-adaptive concept on FPGA devices.
- Analysis of network topologies to find the most suitable topology for connecting the self-adaptive elements in one system.
- Development of three placement algorithms for the network on chip.
- Development of the Step-Adaptive Algorithm for improving placement on the running network and ways for measuring successfulness of the placing process.
- Implementation of the simulation framework to verify the self-adaptive system based on the mesh network and test the parameters of the self-adaptive system.

Appendix A

Network Traffic Visualization

All the previous chapters describe a network on chip (network) and algorithms that run on it. For a better explanation of the network parameters and their characterization this appendix explains the notation and parameters we used.

A.1 Network Notation

We use a 2D-mesh network with 25 nodes organized in a matrix of 5x5 in all the examples and descriptions, see Figure A.1. The notation of nodes starts from the top left corner. The top left node has coordinates $[0, 0]$ and the bottom right node has coordinates $[4, 4]$. The axis X represents columns and axis Y represents rows in the network mesh.

The Network has five inputs and five outputs. The inputs are connected to the left side of the network through the west port W_{in}. The outputs are connected to the bottom side of the network through the south port S_{out}.

A.2 Network Parameters Expression

The logical model of the node is in Figure A.2. The node has an execution unit P that processes incoming data and a transfer unit T that routes incoming data to the desired output or to the execution unit. The transfer unit can be seen as a network router. Each node except edge nodes has four input ports and four output ports. Ports are called North, East, South and West according to the direction from the node. They connect

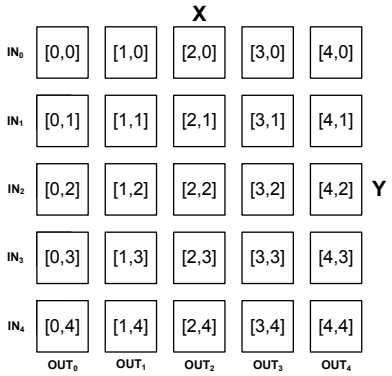


Figure A.1: Numbering of nodes in the network, and input/output ports for communication with environment.

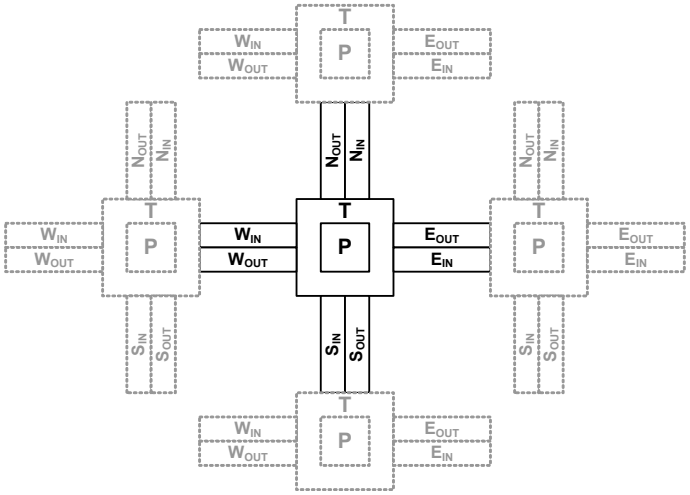
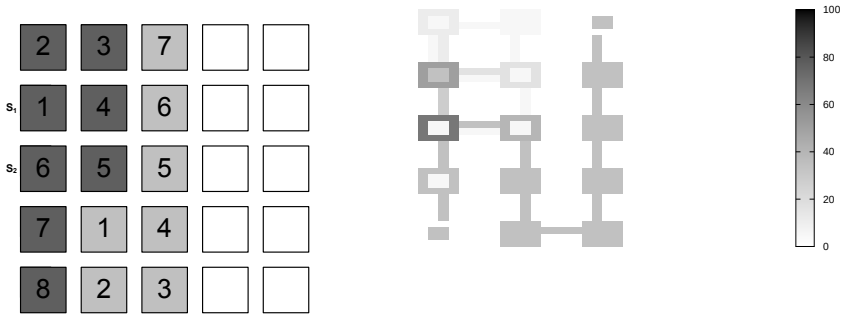


Figure A.2: Explanation of the NoC graph with dataflow and data processing at a node and ports. Port names: North, East, South and West.

nodes together and create the network connection matrix.

The model from Figure A.2 corresponds to the network graph. Each element of the network graph visualizes one node with its ports. The node and ports are filled by a gray color that represents use of the visualized part. If the execution unit or other parts of the node are used near to 100%, it is filled with a dark gray or black color. If the execution unit or other parts of the node are unused, it is filled with a white color, they don't appear on the network graph. The scale legend is on the right side of the graph and the scale is in percents unless stated otherwise. Figure A.3(b) shows the network graph with two running applications.



(a) An example of two applications placed on the network. A stream application is light grey, a parallel application is dark grey. S_1 and S_2 are data sources for the applications.

(b) An example of a NoC graph with two applications and its dataflow and data processing.

Figure A.3: Two applications placed on the NoC and their dataflow and data processing graph.

Figure A.3(a) shows applications placed on the nodes in the network. The color denotes groups of nodes that form one application. The number in the node identifies the task in the application. The sources of the running applications are represented by S_x on the left side of the network. The outputs of the network are connected to nodes at the bottom of the network. They are represented by O_x.

Matrix A.3(a) and heat graph A.3(a) identify tasks and their dataflow in the network. The placed applications are described in Appendix B. For the test purpose we use a serial and a parallel application model. Tasks of the serial application model are numbered sequentially. In the parallel application the concurrently working nodes are numbered in interval $< 2, n-1 >$. The first and last nodes in the parallel application distribute and pick up packets to/from the nodes that work concurrently.

Appendix B

Testing Applications

We use sets of the basic testing applications to prove the parameters of placement and improvement algorithms. The testing applications are of two types. The first is the stream type, see Figure B.1(a), that contains several tasks that communicate in a serial way. The second type of application contains nodes that work concurrently, see Figure B.1(b). The first node prepares and sends flits to the concurrently working nodes. The last node collects flits from the concurrently working nodes. This application models for parallel applications.

We created six sets of applications that contain the application types described above. These sets prove our network for a wide range of applications including their concurrent execution in the network. All the tests in this thesis have been done with the following sets of applications

Stream Type Set

The test set S1 contains three stream applications with different flit rates and with the following parameters. It models the stream-type graphic algorithms like filter, erosions or subtractions. The minimal number of hop for the set is 20.6, and the minimal path for the set is 27.5.

Type	Nodes	Rate	Start	Delay	Hops	Path
Stream	5	5	1	2	10	19.2
Stream	6	10	3	2	6	5.5
Stream	7	15	5	2	4.6	2.8

The test set S4 contains three stream applications with the same flit rates and with the following parameters. It models stream-type graphic algorithms like filter, erosions or subtractions. The minimal number of hop for the set is 30, and the minimal path for the set is 48.

Type	Nodes	Rate	Start	Delay	Hops	Path
Stream	5	5	1	2	8.3	13.3
Stream	6	5	3	2	10	16
Stream	7	5	5	2	11.7	18.7

Parallel Type Set

The test set S2 contains two parallel applications with the different flit rates and with the following parameters. The set models parallel-type graphic algorithms like coloring or AdaBboost. The minimal number of hop for the set is 34.2, and the minimal path for the set is 67.7.

Type	Nodes	Rate	Start	Delay	Hops	Path
Parallel	8	2	1	4	21.2	52.4
Parallel	10	5	3	4	13	15.3

The test set S5 contains two parallel applications with the same flit rates and with the following parameters. The set models parallel-type graphic algorithms like coloring or AdaBoost. The minimal number of hop for the set is 20, and the minimal path for the set is 20.8.

Type	Nodes	Rate	Start	Delay	Hops	Path
Parallel	8	4	1	4	10	11.3
Parallel	10	4	3	4	10	9.5

Mixed Type Set

The test set S3 contains one parallel application and two stream applications with different flit rates and with the following parameters. The set models concurrent execution of parallel and serial types of graphic algorithms like coloring and erosions. The minimal number of hop for the set is 29.1 and the minimal path for the set is 45.5.

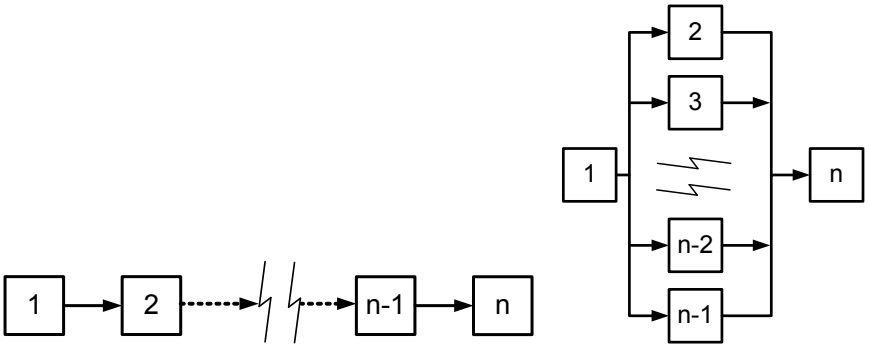
Type	Nodes	Rate	Start	Delay	Hops	Path
Parallel	8	3	1	4	15.1	26
Stream	5	5	3	2	10	17.2
Stream	5	15	5	2	4	2.3

The test set S6 contains one parallel application and one stream application with the same communication load and with the following parameters. The set models concurrent execution of parallel and serial types of graphic algorithms like coloring and erosions. The minimal number of hop for the set is 30, and the minimal path for the set is 54.7.

Type	Nodes	Rate	Start	Delay	Hops	Path
Parallel	9	2	1	4	22.9	50.4
Stream	9	14	3	2	7.1	4.3

All the sets have been designed to model different types of applications executed in the network. They occupy 72% of the nodes in the network. The rate of the flits for each application was chosen in a wide range to test the behaviour of different application load on the network and their influence on each other during improving the network. The sets test the placement behaviour and mainly the improvement algorithms during their run. The tables of the sets show the parameters of the applications used in the set. Each task in the application have delay parameter that specifies cycle delay for one flit processed inside. The parameters hops and path present minimal placement of the application. The value can be influenced by other applications running on the network concurrently. These parameters were found by the random placement with selection of the best result. We cannot test all possibilities for the reason of the huge number of combinations 25!.

The following table shows the minimal network cost and the network cost reached by the Multi-Best Node placement (MBN Place) into an empty network.



(a) An example of a typical stream application with n tasks.

(b) An example of a typical parallel application with n tasks.

Figure B.1: Two types of test applications used in the sets.

	S1	S2	S3	S4	S5	S6
$min(C_{net})$	27.5	67.7	45.5	48	20.2	54.7
$min(H_{net})$	20.6	34.2	29.1	30	20	30
$MBNPlaceC_{net}$	30	96.5	62.9	61.6	37.3	76.7
$MBNPlaceH_{net}$	22.7	40.3	34.9	35	26.4	32.1

Appendix C

Progress Graphs

This appendix contains progress graphs for simulations of the test sets in the simulation framework. The progress graphs are part of Chapter 6. They show the behaviour of the network and its parameters when improving the placement of the running applications. Brief information about the content of the graphs is in the caption under each graph.

Application comparison The graphs show behaviour of each application in the test set. The impact on the network cost can be seen from the graphs. See Figures on page 131.

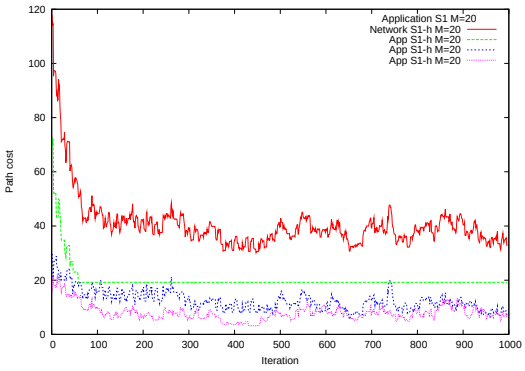
Improving the Placement with the Path Method shows the progress of improving the application sets S1, S2 and S3 placed by the Step-Adaptive Algorithm with the path method as input parameters of algorithm. See Figures on pages 132 and 136.

Improving the Placement with the Hops Method shows the progress of improving the application sets S1, S2 and S3 placed by the Step-Adaptive Algorithm with the hop method as input parameters of algorithm. See Figures on pages 133 and 136.

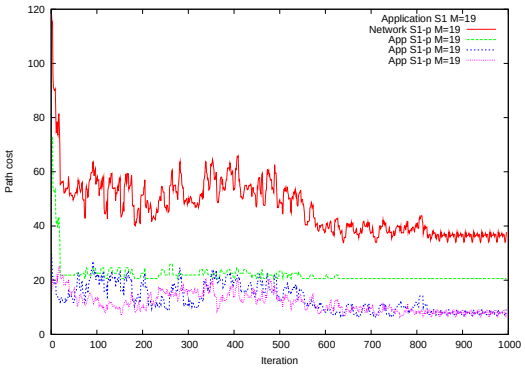
Improving the Placement with the Path Method shows the progress of improving the application sets S4, S5 and S6 placed by the Step-Adaptive Algorithm with the path method as input parameters of algorithm. See Figures on pages 134 and 136.

Improving the Placement with the Hops Method shows the progress of improving the application sets S4, S5 and S6 placed by the Step-Adaptive Algorithm with the hop method as input parameters of algorithm. See Figures on pages 135 and 136.

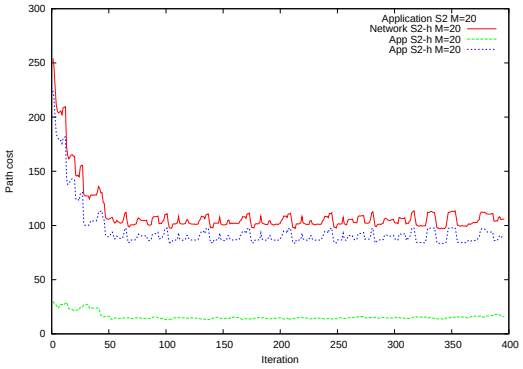
Real Example of Improving Placement shows the progress of improving the application placement by the Step-Adaptive Algorithm with the hop method as input parameters of algorithm. The graph shows injecting and releasing applications during the life time of the network. See Figures on page 137.



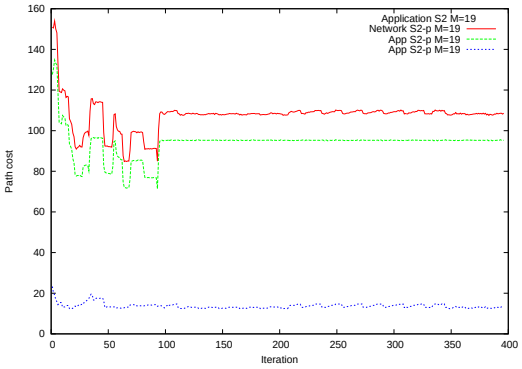
(a) Network and S1 applications path cost progress. The Step-Adaptive Algorithm with the path method.



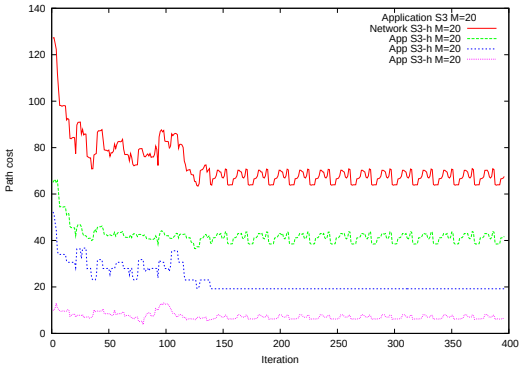
(b) Network and S1 applications path cost progress. The Step-Adaptive Algorithm with the hops method.



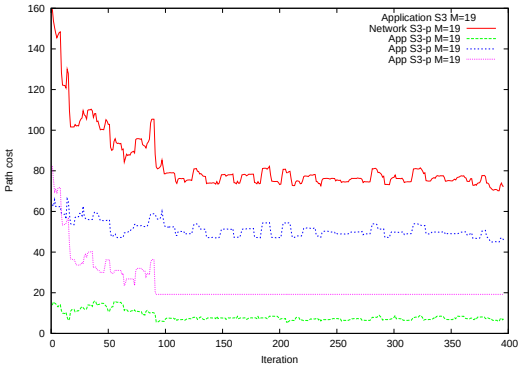
(c) Network and S2 applications path cost progress. The Step-Adaptive Algorithm with the path method.



(d) Network and S2 applications path cost progress. The Step-Adaptive Algorithm with the hops method.

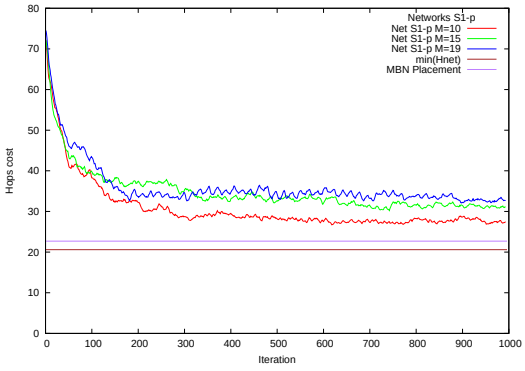


(e) Network and S3 applications path cost progress. The Step-Adaptive Algorithm with the path method.

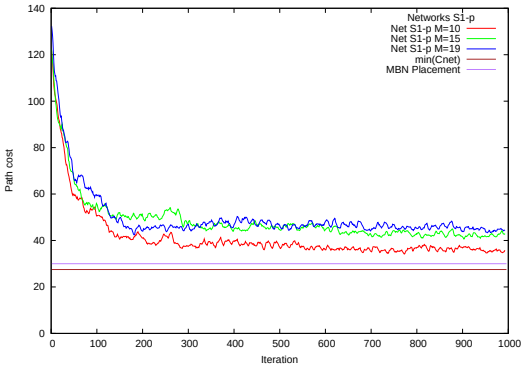


(f) Network and S3 applications path cost progress. The Step-Adaptive Algorithm with the hops method.

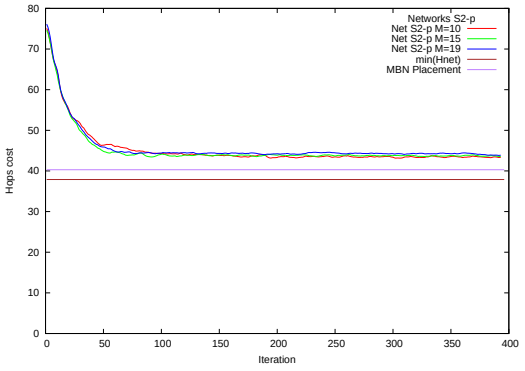
Figure C.1: Network and application path cost progress for the Step-Adaptive Algorithm.



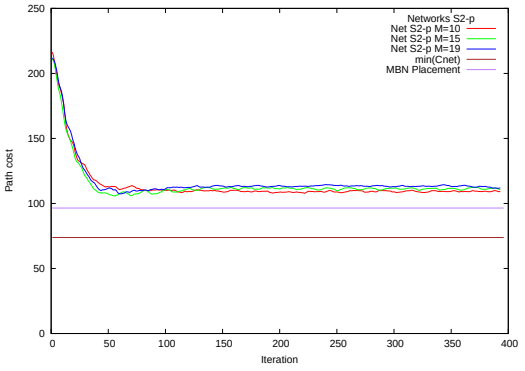
(a) Network hops cost H_{net} for application S1 running for 1000 iterations.



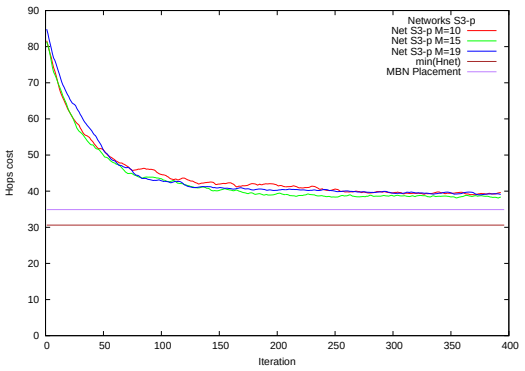
(b) Network path cost C_{net} for application S1 running for 1000 iterations.



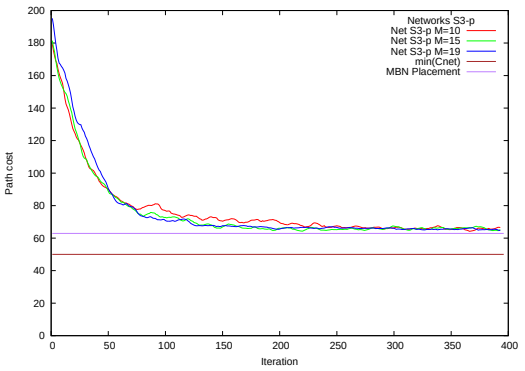
(c) Network hops cost H_{net} for application S2 running for 400 iterations.



(d) Network path cost C_{net} for application S2 running for 400 iterations.

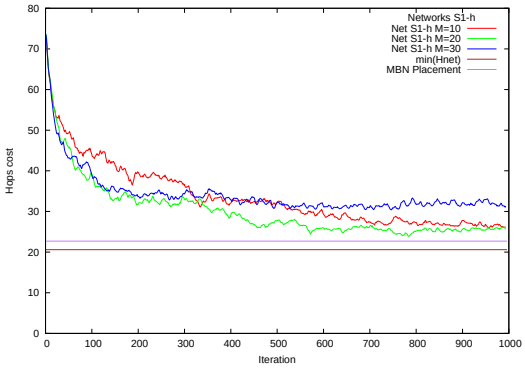


(e) Network hops cost H_{net} for application S3 running for 400 iterations.

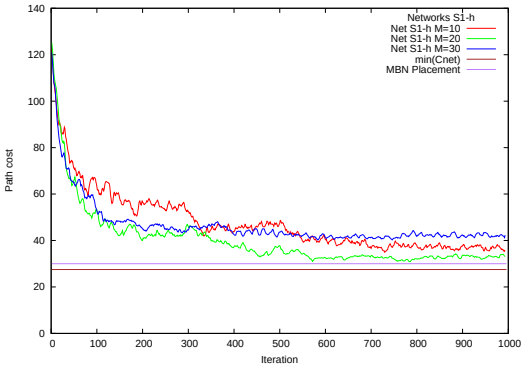


(f) Network path cost C_{net} for application S3 running for 400 iterations.

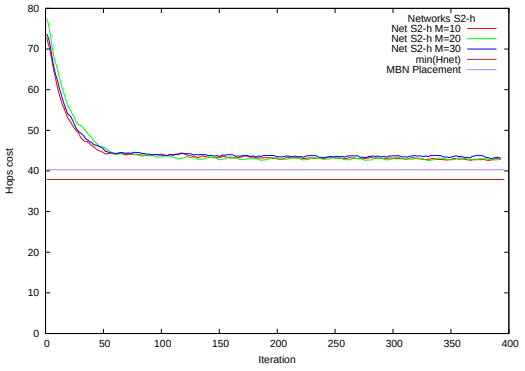
Figure C.2: The Step-Adaptive Algorithm progress for test sets S1, S2 and S3 with the path cost C_{flit}.



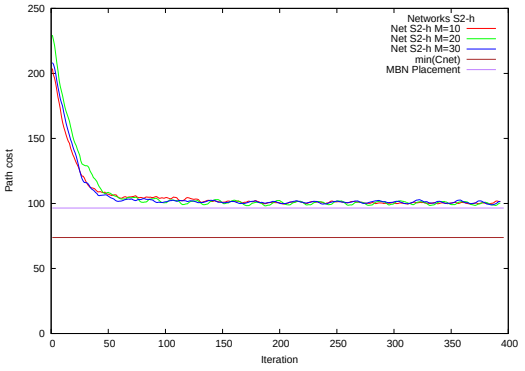
(a) Network hops cost H_{net} for application S1 running for 1000 iterations.



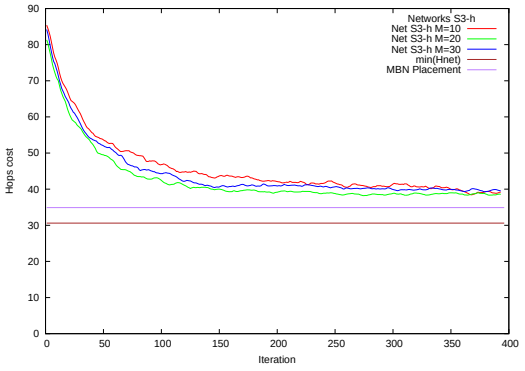
(b) Network path cost C_{net} for application S1 running for 1000 iterations.



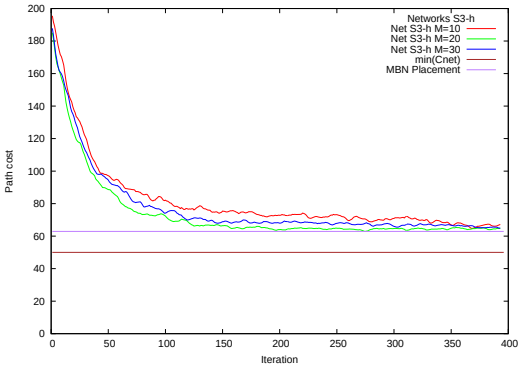
(c) Network hops cost H_{net} for application S2 running for 400 iterations.



(d) Network path cost C_{net} for application S2 running for 400 iterations.

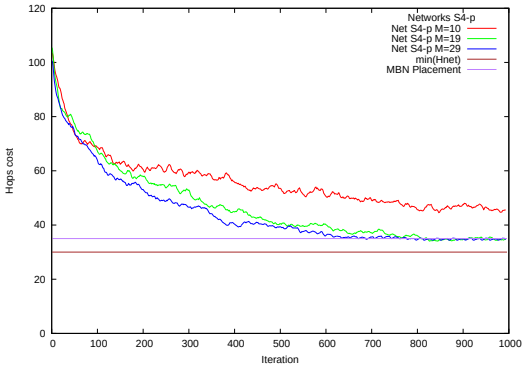


(e) Network hops cost H_{net} for application S3 running for 400 iterations.

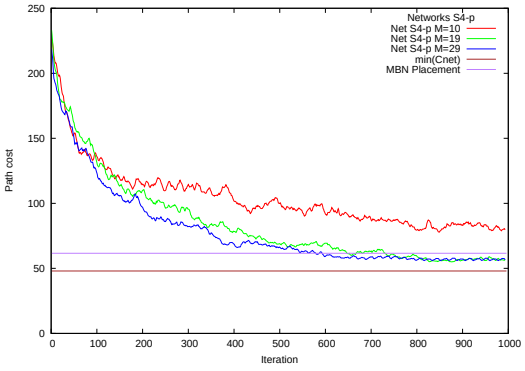


(f) Network path cost C_{net} for application S3 running for 400 iterations.

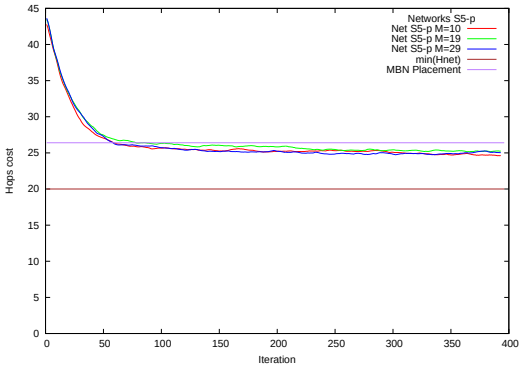
Figure C.3: The Step-Adaptive Algorithm progress for test sets S1, S2 and S3 with the hops cost H_{flit}.



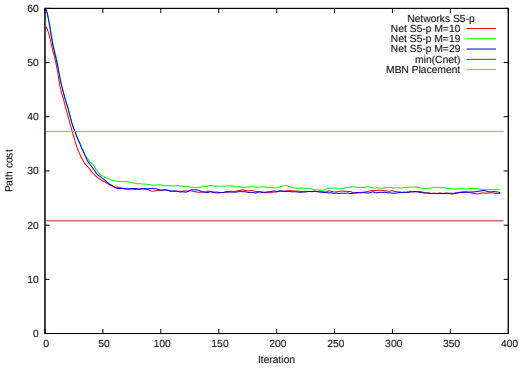
(a) Network hops cost H_{net} for application S4 running for 1000 iterations.



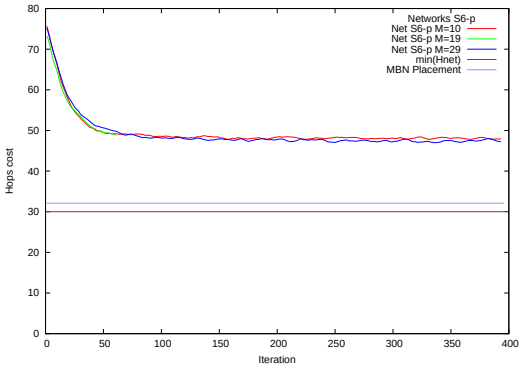
(b) Network path cost C_{net} for application S4 running for 1000 iterations.



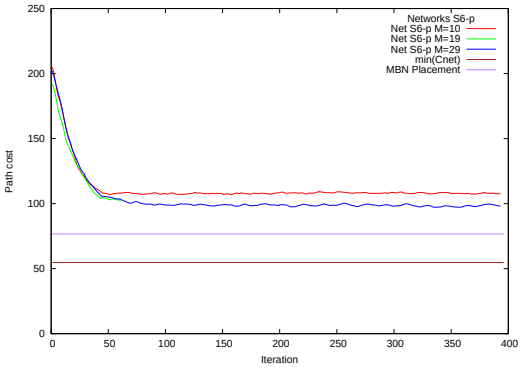
(c) Network hops cost H_{net} for application S5 running for 400 iterations.



(d) Network path cost C_{net} for application S5 running for 400 iterations.

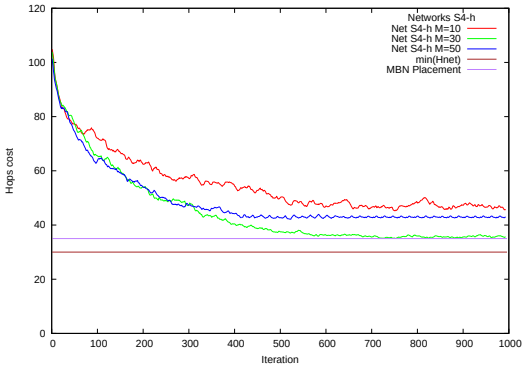


(e) Network hops cost H_{net} for application S6 running for 400 iterations.

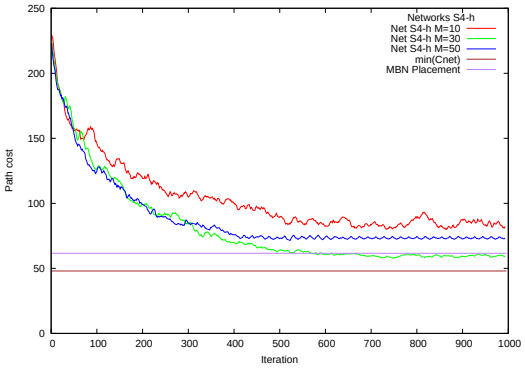


(f) Network path cost C_{net} for application S6 running for 400 iterations.

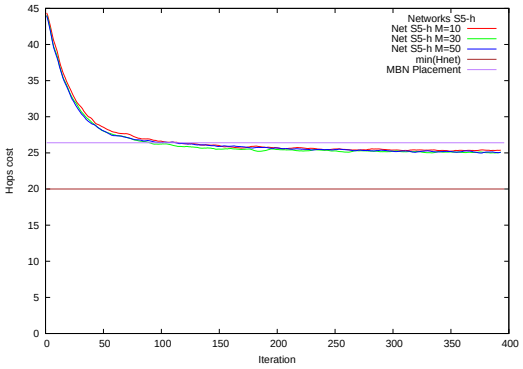
Figure C.4: The Step-Adaptive Algorithm progress for test sets S3, S4 and S5 with the path cost C_{flit}.



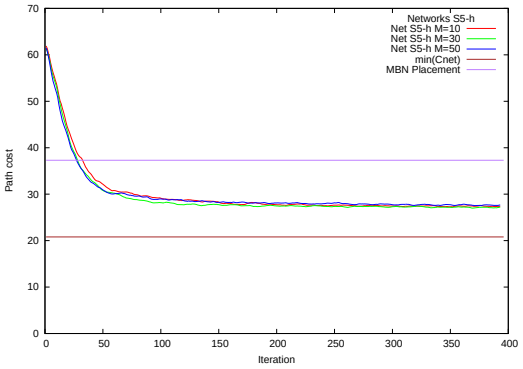
(a) Network hops cost H_{net} for application S4 running for 1000 iterations.



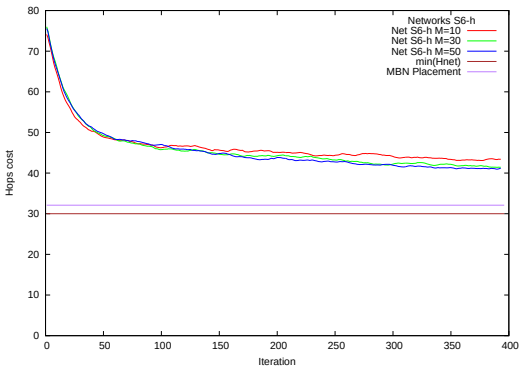
(b) Network path cost C_{net} for application S4 running for 1000 iterations.



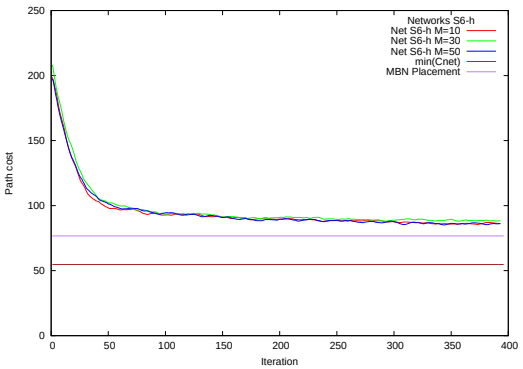
(c) Network hops cost H_{net} for application S5 running for 400 iterations.



(d) Network path cost C_{net} for application S5 running for 400 iterations.

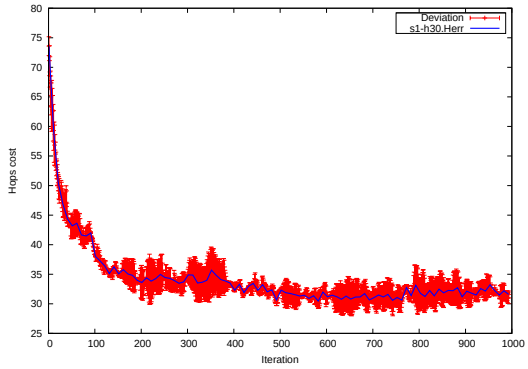


(e) Network hops cost H_{net} for application S6 running for 400 iterations.

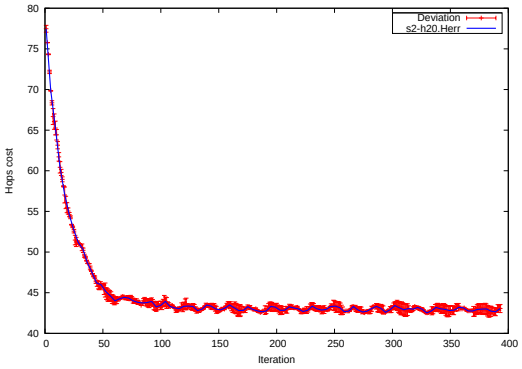


(f) Network path cost C_{net} for application S6 running for 400 iterations.

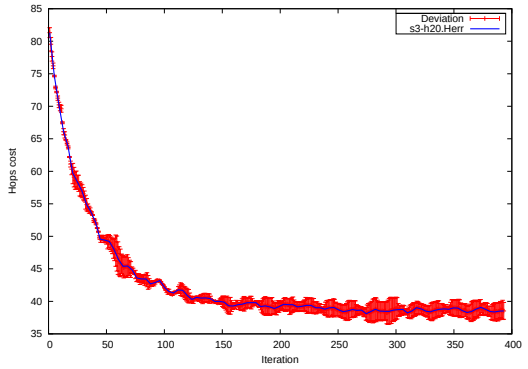
Figure C.5: The Step-Adaptive Algorithm progress for test sets S4, S5 and S6 with the hops cost H_{flit}.



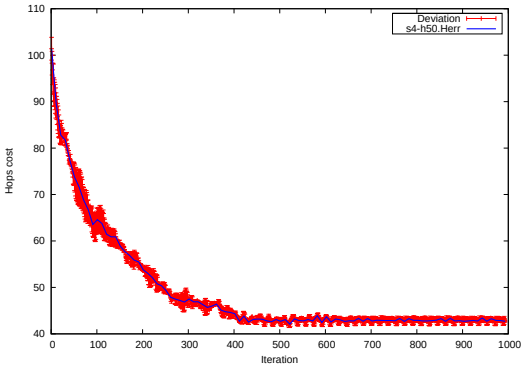
(a) Network hops cost H_{net} for S1, standard deviation



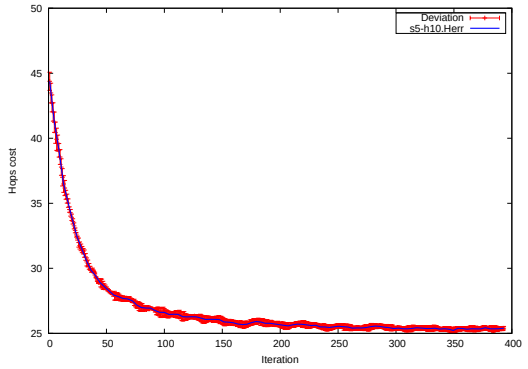
(b) Network hops cost H_{net} for S2, standard deviation



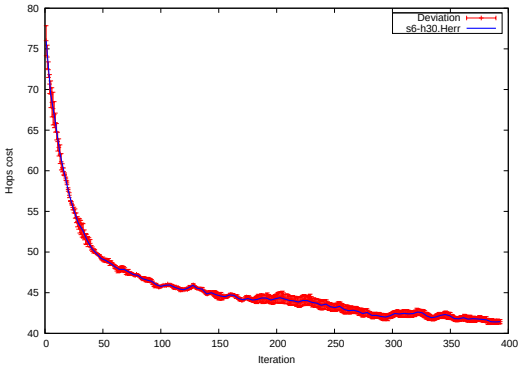
(c) Network hops cost H_{net} for S3, standard deviation



(d) Network path cost H_{net} for S4, standard deviation

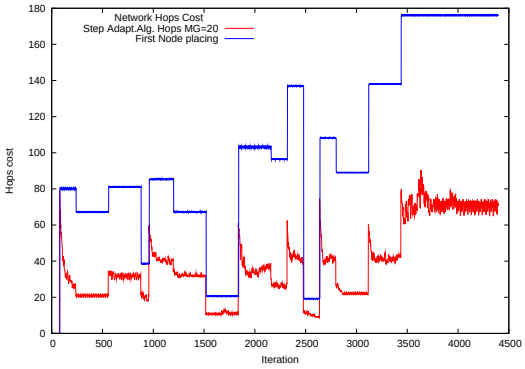


(e) Network path cost H_{net} for S5, standard deviation

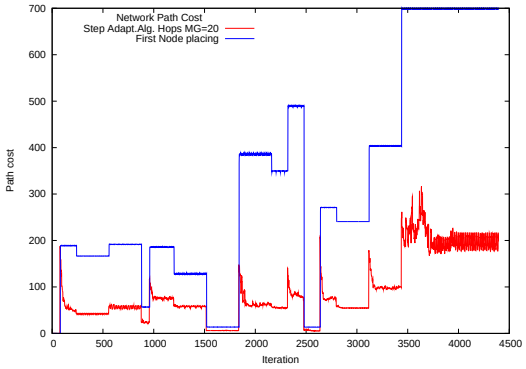


(f) Network path cost H_{net} for S6, standard deviation

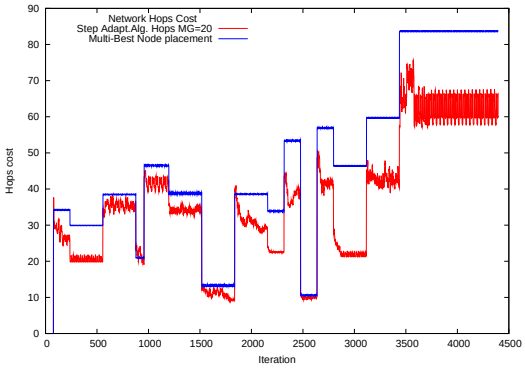
Figure C.6: The Step Adaptive algorithm progress for the hop cost calculation, standard deviation



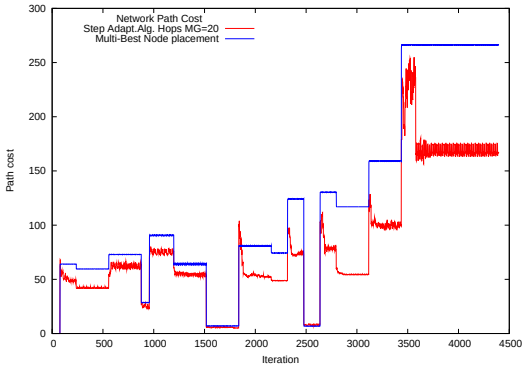
(a) Network hops cost H_{net} for the Step-Adaptive Algorithm, random node placement.



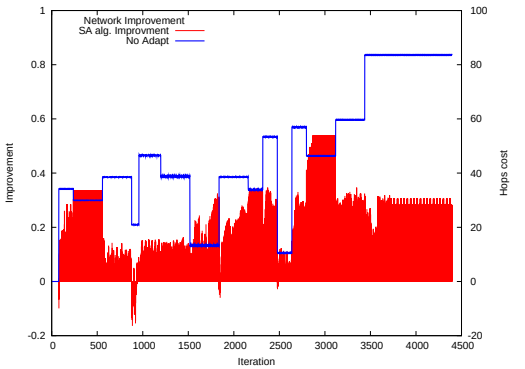
(b) Network path cost C_{net} for the Step-Adaptive Algorithm, random node placement.



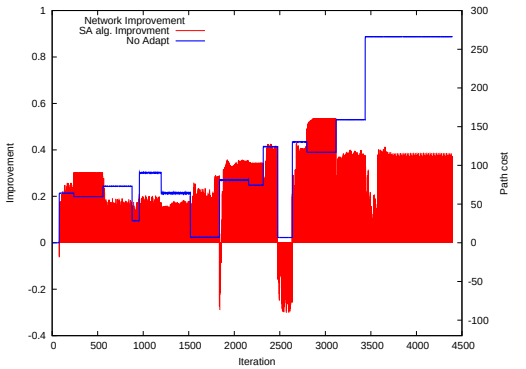
(c) Network hops cost H_{net} for the Step-Adaptive Algorithm, Multi-Best Node placement.



(d) Network path cost C_{net} for the Step-Adaptive Algorithm, Multi-Best Node placement.



(e) Improvment of network hops cost H_{net} by the Step-Adaptive Algorithm.



(f) Improvment of network path cost C_{net} by the Step-Adaptive Algorithm.

Figure C.7: Comparison of network parameters for the Step-Adaptive Algorith and placement application to the network.

Bibliography

A. Mello, L. Copello, F. G. M. and Calazans, N. (2004), 'Evaluation of routing algorithms on mesh based nocs'.

Alt (2007), *Stratix-II GX - Megafunction User Guide*.

Atm (1998), *AT40K series cache logic (Mode 4) configuration*.

Atm (1999*a*), *AT6000 series configuration*.

Atm (1999*b*), *AT6000(LV) series*.

Atm (2001*a*), *AT94K series cache logic (Mode 4) configuration*.

Atm (2001*b*), *AT94K series configuration*.

Atm (2002*a*), *AT40K series*.

Atm (2002*b*), *AT40K series configuration*.

Atm (2002*c*), *AT94K series FPSLIC*.

Banerjee, S., Bozorgzadeh, E. and Dutt, N. (2005), Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration, *in* 'Design Automation Conference, 2005. Proceedings. 42nd', pp. 335 – 340.

Bartosinski, R., Daněk, M., Honzík, P. and Matoušek, R. (2005*a*), Dynamic reconfiguration in FPGA-based SoC designs, *in* G. Takách, A. Hlawiczka and J. Sziraj, eds, 'Proceedings of the 8th IEEE Workshop on Designs and Diagnostics of Electronic Circuits nad Systems', University of West Hungary, Sopron, pp. 129–136.

Bartosinski, R., Daněk, M., Honzík, P. and Matoušek, R. (2005*b*), Dynamic reconfiguration in FPGA-based SoC designs. Abstract, *in* H. Schmidt and S. Wilton, eds, 'FPGA 2005 - ACM/SIGDA Thirteenth ACM International Symposium on Field-Programmable Gate Arrays', ACM, Monterey, p. 274.

Carvalho, E., Calazans, N., Brião, E. and Moraes, F. (2004), Padreh: a framework for the design and implementation of dynamically and partially reconfigurable systems, *in* 'SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design', ACM Press, New York, NY, USA, pp. 10–15.

Casas, J., Moreno, J., Madrenas, J. and Cabestany, J. (2007), A novel hardware architecture for self-adaptive systems, pp. 592 –599.

Chang, W. and Yubai, L. (2006), Network on chip: Key communication technology in mp-soc, *in* 'ITS Telecommunications Proceedings, 2006 6th International Conference on', pp. 1159 –1164.

Daněk, M., Honzík, P., Kadlec, J., Matoušek, R. and Pohl, Z. (2004), Reconfigurable system-on-a-programmable-chip platform, *in* Z. Peng, M. Fischerová and E. Gramatová, eds, 'Proceedings of the 7th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems', Institute of Informatics SAS, Bratislava, pp. 21–28.

Daněk, M., Honzík, P., Kadlec, J., Matoušek, R., Pohl, Z. and Heřmánek, A. (2005), Gin - notetaker for blind people: An example of using dynamic reconfiguration of FPGA, *in* 'Proceedings of the 1st HiPEAC Workshop on Advanced Computer Architecture and Compilation for Embedded Systems', Academia Press, Ghent, Belgium, pp. 15–18.

Danek, M., Kadlec, J., Bartosinski, R. and Kohout, L. (2008), Increasing the level of abstraction in fpga-based designs, pp. 5 –10.

DARPA-ACIP (n.d.), 'http://www.darpa.mil/ipto/programs/acip/index.htm'.

DARPA-HPCS (n.d.), 'http://www.darpa.mil/ipto/programs/hpcs/'.

DeHon, A. and Wawrzynek, J. (1999), Reconfigurable computing: what, why, and implications for design automation, *in* 'Design Automation Conference, 1999. Proceedings. 36th', pp. 610 –615.

EU-Runes (n.d.), 'http://www.ist-runes.org/'.

Foroutan, S., Thonnart, Y., Hersemeule, R. and Jerraya, A. (2010), An analytical method for evaluating network-on-chip performance, *in* 'Design, Automation Test in Europe Conference Exhibition (DATE), 2010', pp. 1629 –1632.

Giefers, H. and Platzner, M. (2010), A triple hybrid interconnect for many-cores: Reconfigurable mesh, noc and barrier, pp. 223 –238.

Gindin, R., Cidon, I. and Keidar, I. (n.d.), 'Architecture and routing in noc based fpgas'.

Guz, Z., Walter, I., Bolotin, E., Cidon, I. Ginosar, R. and Kolodny, A. (2007), 'Network delays and link capacities in application-specificwormhole nocs', *VLSI Design* **2007**(90941), 15.

Handa, M. and Vemuri, R. (2004), An efficient algorithm for finding empty space for online fpga placement, *in* 'Design Automation Conference, 2004. Proceedings. 41st', pp. 960 – 965.

Honzík, P. (2004*a*), AVR core supported dynamic reconfiguration, *in* L. Husník and L. Lhotská, eds, 'POSTER 2004. Proceedings of the 8th International Student Conference on Electrical Engineering', ČVUT FEL, Praha, pp. 1–5.

Honzík, P. (2004*b*), Communication library for AVR microcontrollers, Technical Report 2110, ÚTIA AV ČR, Praha.

Honzík, P. (2005), Analysis and implementation of dynamic reconfiguration for FPGAs (in Czech), *in* 'PAD 2005. Workshop for doctoral students', ČVUT FEL, Praha, pp. 1–6.

Honzík, P. and Kafka, L. (2005), Front end tools for a dynamic reconfiguration, *in* L. Husník and L. Lhotská, eds, 'POSTER 2005. Proceedings of the 8th International Student Conference on Electrical Engineering', ČVUT FEL, Praha, pp. 1–4.

Horta, E. L., Lockwood, J. W., Taylor, D. E. and Parlour, D. (2002), Dynamic hardware plugins in an FPGA with partial run-time reconfiguration, *in* 'DAC '02: Proceedings of the 39th conference on Design automation', ACM Press, New York, NY, USA, pp. 343–348.

Huang, C.-H. and Hsiung, P.-A. (2009), 'Hardware resource virtualization for dynamically partially reconfigurable systems', *Embedded Systems Letters, IEEE* **1**(1), 19 –23.

Ito, H., Oguri, K., Nagami, K., Konishi, R. and Shiozawa, T. (1998), The plastic cell architecture for dynamic reconfigurable computing, *in* 'Rapid System Prototyping, 1998. Proceedings. 1998 Ninth International Workshop on', pp. 39 –44.

Kadlec, J., Daněk, M. and Honzík, P. (2004), Reconfigurable 24-bit floating-point coprocessor Demo, Technical Report 2116, ÚTIA AV ČR, Praha.

Kafka, L. (2008), Analysis of applicability of partial runtime reconfiguration in fault emulator in xilinx fpgas, pp. 1 –4.

Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R. and Shippy, D. (2005), 'Introduction to the cell multiprocessor', *IBM Journal of Research and Development* **49**(4.5), 589 –604.

Krikke, J. (2005), 'T-engine: Japan's ubiquitous computing architecture is ready for prime time', *IEEE Pervasive Computing* **4**, 4–9.

Kuhnle, M., Hubner, M. nad Becker, J., Coppola, A., Pieralisi, L., Locatelli, R., Maruccia, G., DeMarco, T., Campi, F., Deledda, A., Mucci, C. and Ries, F. (2008), 'An interconnect strategy for a heterogeneous, reconfigurable soc', *Design Test of Computers, IEEE* **25**(5), 442 –451.

Liu, H. and Wong, D. F. (1999), Circuit partitioning for dynamically reconfigurable fpgas, *in* 'In International ACM/SIGDA Symposium on Field Programmable Gate Arrays', pp. 187–194.

Maruyama, T. and Hoshino, T. (1999), A reconfigurable architecture for high speed computation by pipeline processing, *in* 'FPL 99: Field Programmable Logic and Applications: 9th International Workshop', ACM Press, New York, NY, USA, pp. 514 – 519.

MIT-Oxygen (n.d.), 'http://oxygen.csail.mit.edu/'.

Ni, L. and McKinley, P. (1993), 'A survey of wormhole routing techniques in direct networks', *Computer* **26**(2), 62 –76.

Palesi, M., Kumar, S. and Catania, V. (2009), 'Bandwidth-aware routing algorithms for networks-on-chip platforms', *Computers Digital Techniques, IET* **3**(5), 413 –429.

Prophet, G. (2004), 'Reconfigurable systems shape up for diverse application tasks', *EDN Europe* pp. 27 – 34.

R.Matoušek (2003), 'Reconfigurable designs in FPGAs', *Postgraduate Study Report DC-PSR-2003-10* pp. 1–43.

Robertson, I. and Irvine, J. (2004), 'A design flow for partially reconfigurable hardware', *Trans. on Embedded Computing Sys.* **3**(2), 257–283.

Salminen, E., Kulmala, A. and Hamalainen, T. (2007), On network-on-chip comparison, *in* 'Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on', pp. 503 –510.

Sander, I. (2004), Network-on-chip topology, Technical Report 2B1447, KTH ICT Royal Institute of Technology, Sweden.

Soto, V., Moreno, J., Madrenas, J. and Cabestany, J. (2009), Implementation of a dynamic fault-tolerance scaling technique on a self-adaptive hardware architecture, pp. 445 – 450.

Straka, M., Kastil, J. and Kotasek, Z. (2010), Modern fault tolerant architectures based on partial dynamic reconfiguration in fpgas, pp. 173 –176.

Strunk, J., Volkmer, T., Rehm, W. and Schick, H. (2009), An on chip network inside a fpga for run-time reconfigurable low latency grid communication, pp. 539 –546.

Suzuki, M., Hasegawa, Y., Yamada, Y., Kaneko, N., Deguchi, K., Amano, H., Anjo, K., Motomura, M., Wakabayashi, K., Toi, T. and Awashima, T. (2004), Stream applications on the dynamically reconfigurable processor, *in* 'Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on', pp. 137 – 144.

T-Engine (n.d.), 'http://www.t-engine.org/'.

Kogel, T., Leupers, R. and Meyr, H. (2006), *Integrated SysteLevel Modeling of Networon-Chip enabled MultProcessor Platforms*, Springer. ISBN 1-4020-4825-4.

Sutton, S. R. and Barto, G. A. (1998), *Reinforcement Learning: An Introduction* , MIT Press. ISBN 0262193981.

Verkest, D. (2003), 'Machine chameleon [handheld devices]', *Spectrum, IEEE* **40**(12), 41 – 46.

Wigley, G. and Kearney, D. (2001), The first real operating system for reconfigurable computers, *in* 'ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture', IEEE Computer Society, Washington, DC, USA, pp. 130–137.

Wu, G.-M., Lin, J.-M. and Chang, Y.-W. (1998), 'The future of reconfigurable systems', *5th Canadian Conference on Field Programmable Devices* .

Wu, G.-M., Lin, J.-M. and Chang, Y.-W. (2002), 'Performance-driven placement for dynamically reconfigurable FPGAs', *ACM Trans. Des. Autom. Electron. Syst.* **7**(4), 628–642.

Xil (1997), *XC6200 field programmable gate arrays.*

Xil (2000), *Virtex 2.5 V field programmable gate arrays.*

Xil (2001), *Spartan-II 2.5V field programmable gate arrays.*

Xil (2003), *Virtex-II Pro platform FPGA user guide.*

Xil (2010*a*), *Platform Flash In-System Programmable Configuration PROMs.*

Xil (2010*b*), *Virtex-5 FPGA - User Guide.*

Xil (2010*c*), *Virtex-6 FPGA Configuration - User Guide.*

Xil (2010*d*), *Virtex-6 FPGA Data Sheet - DC and Switching Characteristics.*

Zou, C., Xia, C. and Zhao, G. (2009), Numerical parallel processing based on gpu with cuda architecture, pp. 93 –96.

List of Author's Publications

Papers in Proceedings

Bartosinski, R., Daněk, M., Honzík, P. and Kadlec, J. (2007), Modelling self-adaptive networked entities in matlab/simulink, *in* 'International Conference Technical Computing Prague', Humusoft CZ, Prague.

Bartosinski, R., Daněk, M., Honzík, P. and Matoušek, R. (2005*a*), Dynamic reconfiguration in FPGA-based SoC designs, *in* G. Takách, A. Hlawiczka and J. Sziraj, eds, 'Proceedings of the 8th IEEE Workshop on Designs and Diagnostics of Electronic Circuits nad Systems', University of West Hungary, Sopron, pp. 129–136.

Bartosinski, R., Daněk, M., Honzík, P. and Matoušek, R. (2005*b*), Dynamic reconfiguration in FPGA-based SoC designs. Abstract, *in* H. Schmidt and S. Wilton, eds, 'FPGA 2005 - ACM/SIGDA Thirteenth ACM International Symposium on Field-Programmable Gate Arrays', ACM, Monterey, p. 274.

Daněk, M., Honzík, P., Kadlec, J., Matoušek, R. and Pohl, Z. (2004), Reconfigurable system-on-a-programmable-chip platform, *in* Z. Peng, M. Fischerová and E. Gramatová, eds, 'Proceedings of the 7th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems', Institute of Informatics SAS, Bratislava, pp. 21–28.

Daněk, M., Honzík, P., Kadlec, J., Matoušek, R., Pohl, Z. and Heřmánek, A. (2005*a*), Dynamic reconfiguration in FPGA-based SoC designs, *in* 'Proceedings of the 1st HiPEAC Workshop on Advanced Computer Architecture and Compilation for Embedded Systems', Academia Press, Ghent, Belgium, pp. 35 –38.

Daněk, M., Honzík, P., Kadlec, J., Matoušek, R., Pohl, Z. and Heřmánek, A. (2005), Gin - notetaker for blind people: An example of using dynamic reconfiguration of FPGA, *in* 'Proceedings of the 1st HiPEAC Workshop on Advanced Computer Architecture and Compilation for Embedded Systems', Academia Press, Ghent, Belgium, pp. 15–18.

Danek, M., Philippe, J.-M., Honzik, P., Gamrat, C. and Bartosinski, R. (2008), Self-adaptive networked entities for building pervasive computing architectures, *in* 'ICES '08: Proceedings of the 8th international conference on Evolvable Systems: From Biology to Hardware', Springer-Verlag, Berlin, Heidelberg, pp. 94–105.

Kloub, J., Honzik, P. and Danek, M. (2010), Reconfigurable hardware objects for image processing on fpgas, *in* 'Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on', pp. 121 –122.

Research Reports

Honzík, P. (2004*b*), Communication library for AVR microcontrollers, Technical Report 2110, ÚTIA AV ČR, Praha.

Honzík, P. (2004*c*), Getting started with AVG-GCC, Technical Report 2115, ÚTIA AV ČR, Praha.

Kadlec, J., Daněk, M. and Honzík, P. (2004*a*), Reconfigurable 24-bit floating-point coprocessor demo, Technical Report 2116, ÚTIA AV ČR, Praha.

Kadlec, J., Daněk, M. and Honzík, P. (2004*b*), Reconfigurable scrolling demo, Technical Report 2117, ÚTIA AV ČR, Praha.

Matoušek, R. and Honzík, P. (2004), SDIO interface for the FPSLIC, Technical Report 2118, ÚTIA AV ČR, Praha.

Journals

Bartosinski, R., Daněk, M., Honzík, P., Matoušek, R. and Pohl, Z. (2005), 'Reconfigurable System-on-a-chip', *The Syndicated* **5**(2), 3.

Daněk, M., Honzík, P., Kadlec, J., Matoušek, R. and Pohl, Z. (2005), 'Reconfigurable System on a Programmable Chip Platform', *Atmel Journal* **4**(Spring), 9 –12.

Honzík, P. (2005*b*), 'Programming AVR in circuits (in Czech)', *A Radio. Praktická elektronika* **10**(4), 20.

Kadlec, J., Daněk, M., Honzík, P., Matoušek, R. and Pohl, Z. (2006), 'Platforma s částečnou dynamickou rekonfigurací FPGA (in Czech)', *Automa* **2**(2), 2.

Others Workshops and Posters

Honzík, P. (2004*a*), AVR core supported dynamic reconfiguration, *in* L. Husník and L. Lhotská, eds, 'POSTER 2004. Proceedings of the 8th International Student Conference on Electrical Engineering', ČVUT FEL, Praha, pp. 1–5.

Honzík, P. (2005*a*), Analysis and implementation of dynamic reconfiguration for FPGAs (in Czech), *in* 'PAD 2005. Workshop for doctoral students', ČVUT FEL, Praha, pp. 1–6.

Honzík, P. (2006), 'Analysis and implementation of dynamic reconfiguration for fpgas', [presentation], CAK 2006, CTU Prague.

Honzík, P., Kadlec, J. and Matoušek, R. (2005), SD card file system for atmel FPSLIC, *in* 'WORKSHOP 2005. Proceedings of the 8th International Student Conference on Electrical Engineering', ČVUT FEL, Praha, pp. 1–2.

Honzík, P. and Kafka, L. (2005), Front end tools for a dynamic reconfiguration, *in* L. Husník and L. Lhotská, eds, 'POSTER 2005. Proceedings of the 8th International Student Conference on Electrical Engineering', ČVUT FEL, Praha, pp. 1–4.

Funding and Projects

This appendix presents the list of project that the author participated in during his research and study at the Department of Signal Processing at Institute of Information Theory and Automation of the Czech Academy of Sciences in the Czech Republic and at the Department of Control Engineering at Faculty of Electrical Engineering of the Czech Technical University in Prague. All these projects were significant contributions to this work, and helped the author experience wider international and domestic research.

The majority of the projects are funded by the EU commission and supported the author's cooperation with foreign institutions. It allowed the author to research on the top of technology in his research area. The cooperation with researches from abroad brought the author experience with the technology and methods used in others research laboratories and institutions.

The commercial project with the FPGA chip supplier Atmel Corporation in the area of placement and routing for partial dynamic reconfiguration gave the author detailed information about the problems with the FPGA technology and the method for influencing the design during mapping and placement of internal resources.

Reconf 2

The aim of the RECONF 2 project was to develop a design environment suitable for an efficient use of the dynamically reconfigurable FPGA. This new type of FPGA made it possible to design innovative low cost architectures. That opened new application opportunities, such as implementation of adaptive computing systems.

The main targeted application domains were real time image processing, signal processing. These applications were included in most embedded systems found in aeronautic,

automation, multimedia, industrial process control.

Project ID	IST-2001-34016
Funder	
Members	6
Country	5
Web page	www.reconf.org
Kick off	1 March 2002
Final	31 December 2004

ATMEL Place & Route

The aim of the cooperation with the Atmel Corporation was to develop tools for mapping and placement logic blocks in field programmable gate arrays supplied by the Atmel Corporation. The tool reads generated logic netlist in the FGD format and generate bitstream files for configuring the programmable gate array. The main advantage of the tool is full support for partial dynamic reconfiguration. A partial result of the cooperation were suggestions for a new architecture of a field programmable gate array and its structure to make full use of all features of partial dynamic reconfiguration.

Funder	Atmel Corporation
Members	2
Country	2
Web page	www.atmel.com
Kick off	1 January 2006
Final	31 August 2006

Æther

ATHER was an IST-FET project funded under the 6th Framework Programme (FP6). Selected under the fourth call in the Advanced Computing Architecture (ACA) initiative of the Future and Emerging Technologies (FET) programme, ATHER's main objective was to study novel self-adaptive computing technologies for future embedded and pervasive applications.

Project ID	FP6-IST-027611
Funder	IST-FET
Members	14
Country	9
Web page	www.aether-ist.org
Kick off	1 January 2006
Final	30 June 2009

Scalopes

The project focuses on cross-domain technology and tool developments for multi-core architectures. These developments are driven by and proven for 4 different application domains: communication infrastructure, surveillance systems, smart mobile terminals and stationary video & entertainment systems. Focus in the technology developments are on application & programming models, composability, dependability, reliability, predictable system design, resource management and tools supporting these new developments. As much as possible generic cross-domain tools and architectures are worked upon, but also application-specific extensions are covered. The project focuses on enhancing as much as possible the generic aspects by means of identifying how future cross-domain reference platforms will be made.

Project ID	ARTEMIS-2008-100029
	MŠMT ČR 7H09005
Funder	Artemis JU and MŠMT ČR
Members	36
Country	11
Web page	www.scalopes.eu
Kick off	1 January 2009
Final	31 March 2011

C-A-K and C-A-K 2

Center for Applied Cybernetics (the Center) takes advantage of established research background and teams, and provides concerted action of leading research groups and hi-tech

companies in the country. It offers to its young researchers an opportunity for creative growths and top-class working conditions for research and development in a perspective field. The Government investment will pay off in progress in the field, increase of qualified workforce and directly through the financial growth of the participating industrial companies.

Our department deals with the following problems:

- Distributed control systems
- Operating systems for Real-Time control
- Internet programming
- Industrial automation and Fieldbuses
- Rapid prototyping

Project ID	LN00B096 & 1M0567
Funder	Ministry of Education Youth and Sports
Members	19
Web page	www.c-a-k.cz
Kick off	1 January 2000
Final	31 December 2011

Vita

Petr Honzík was born in Příbram, Czech Republic. He received a Socrates scholarship in 2002 at the Faculty of Science at the University of Lisbon, Portugal. He received the Ing. degree in 2003 from the University of West Bohemia in Pilsen. From September 2003 he is a PhD. student at the Department of Control Engineering at the Faculty of Electrical Engineering of the Czech Technical University in Prague. From October 2003 he is a member of the Department of Signal Processing at Institute of Information Theory and Automation of the Czech Academy of Sciences. Petr participated during his research on European Union projects RECONF2 IST-2001-34016, AETHER FP6-IST-027611 and SCALOPES Artemis JU 100029/MŠMT ČR 7H09005. Currently he is participating in the project C-A-K 2 1M0567.

Petr works as a researcher with specialization on high performance programmable hardware with dynamic reconfiguration. He is interested in programmable integrated circuits design and design of the System-on-Chip based on the FPGA devices. He is also interested in progressive processor and controller architectures and microcontroller programming based on INTEL51 and AVR architectures.

Address:

Department of Signal Processing,
Institute of Information Theory and Automation,
Pod Vodárenskou věží 4,
Praha 8, 182 08,
Czech Republic
E-mail: peters@utia.cas.cz