THE METHODOLOGY OF SYNTHESIS OF DYNAMICALLY RECONFIGURABLE COMPUTING SYSTEMS WITH TEMPORAL PARTITIONING OF HOMOGENEOUS RESOURCES

by

Valeri Kirischian

B.A.Sc. University of Toronto 2004

M.A.Sc. Ryerson University 2005

A dissertation

presented to Ryerson University

in partial fulfillment of the

requirements for the degreee of

Doctor of Philosophy

in the program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2010

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signed:_____ Date: May 5, 2010

Valeri Kirischian

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signed _____ Date: May 5, 2010

Valeri Kirischian

Acknowledgements

I would like to express my sincere gratitude to Dr. Vadim Geurkov Associate Professor of Electrical and Computer Engineering, Ryerson University. He has been my supervisor since the beginning of my graduate studies at Ryerson University. Dr. Vadim Geurkov provided me with many helpful suggestions, important advice and constant encouragement during the course of this work. My keen appreciation goes to Pil Woo Chun, Jamin Islam and Sergiy Zhelnakov for their valuable assistance in the field.

I would like to acknowledge the financial support from the following organizations: the National Science and Engineering Research Council (NSERC), the Ontario Centres of Excellence (CITO), Material and Manufacturing Ontario (MMO), MDA Space Missions, CMC Microsystems, Unique Broadband Systems (UBS), Xilinx Corporation and the Department of Electrical and Computer Engineering at Ryerson University for their financial support of this work. I also want to thank Jim Koch, Jason Naughton, and Daniel Giannitelli for technical support during my studies at Ryerson University.

I would like to express special thanks to my wife Irina and my son Ivan, who helped me to concentrate on completing this dissertation and supported me during the course of this work. Finally, my special appreciation goes to my parents who encouraged and supported me during my studies in graduate school.

Abstract

The Methodology of Synthesis of Dynamically Reconfigurable Computing Systems with Temporal Partitioning of Homogeneous Resources

Valeri Kirischian

Doctor of Philosophy

Electrical and Computer Engineering

Ryerson University 2010

The main motivation factors for the proposed research were the increase of cost-efficiency of FPGA based systems and the simplification of the design process. The first factor is optimization of design in multi-parametric constraint space. The second factor is the design of reconfigurable systems based on higher level of abstraction in a form of macro-functions rather than conventional HDL primitives. Main goal of this work was to create a methodology for automated cost-effective design synthesis of FPGA systems by utilizing temporal partitioning concept. Temporal partitioning provides powerful mechanism that allows to design cost-effective multi-parametrically optimized architectures. Another feature of these architectures is the ability for run-time self-restoration from hardware faults. As the result of the proposed research this methodology was created and successfully verified on the first prototype of Multi-mode Adaptive Reconfigurable System (MARS) with embedded Temporal Partitioning Mechanism (TPM). A special CAD software system was developed for automated application

programming, automated task segmentation, and further high-level synthesis of segment specific processors (SSPs). Several novel methodologies were proposed, developed, and verified including: a methodology for creation of macro-operators (MOs) and associated set of optimized virtual hardware components (VHCs); an automated task segmentation methodology and synthesis of segment specific processors from the VHCs; methodology for integration of fault tolerance mechanisms with the self-restoration capability. The latter mechanism made possible the mitigation of transient and permanent hardware faults in run-time. The proof-of-concept component of this research consists of implementation of the above methodologies and mechanisms in the special software CAD system and verification on the experimental setup based on the prototype of system with TPM (MARS platform). As the result, all the developed methodologies and architectural solutions were tested and their effectiveness was demonstrated.

Nomenclature

ACG Architecture Configuration Graph

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

ASP Application Specific Processor

BISR Built-In-Self-Recovery

CAD Computer Aided Design

CF Compact Flash

CLB Configuration Logic Block

CPLD Complex Programmable Logic Device

CPR Cost Performace Ratio

CRC Cyclic Redundancy Check

DAC Digital to Analog Converter

DDR Double Data Rate

DLP Data Level Parallelism

DRC Design Rule Check

DRCS Dynamically Reconfigurable Computing Systems

DSP Digital Signal Processing

EO Elementary Operator

ERSL Embedded Reconfigurable Systems Lab

FPGA Field Programmable Gate Array

FPOA Field Programmable Object Array

FPP Fast Passive Parallel

GUI Graphical User Interface

HDD Hard Disk Drive

HDL Hardware Description Language

ICAP Integrated Configuration Access Port

ILP Instruction Level Parallelism

IOB Input Output Block

IP Intellectual Property

LUT Look-up Table

LVDS Low Voltage Differential Signal

MAC Multiply Accumulate

MARS Multi-stream Adaptive Reconfigurable System

MIMD Multiple Instruction Multiple Data

MO Macro Operators

OTS Off-The-Shelf

PCB Printed Circuit Board

PLA Programmable Logic Array

PLD Programming Logic Device

PROM Programable Read Only Memory

RCS Reconfigurable Computing System

RF Register Files

ROCR Riverside On-Chip Router

RTR Run-Time-Reconfigurable

SEDR Single Event Dielectric Rupture

SEE Single Event Effects

SEL Single Event Latch-up

SET Single Event Transient

SEU Single Event Upset

SG Sequencing Graph

SIMD Single Instruction Multiple Data

SPC Segment Partitioning Creator

SPM Segment Processing Module

SPR Spatial Partitioning of Resources

SRAM Static Random Access Memory

SSP Segment Specific Processor

TPM Temporal Partitioning Mechanism

TPR Temporal Partitioning of Resources

USB Universal Serial Bus

VHC Virtual Hardware Component

VLIW Very Large Instruction Word

VME Versa Modular Eurocard

VPR Versatile Place and Route

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In the last decade some dramatic changes have occurred in the field of computer technology and even computing paradigms. A major change was decreasing of size of the transistor from $0.25\mu m$ in 1997 to $45nm$ in 2007. Decreasing of size of the transistor allowed for a larger number of transistors to be placed on to a single die. Hence, FPGAs increased from 70M transistors in 1997 to 2500M transistors in 2008. Another effect of transistor dimension reduction is the ability to increase clock frequency and therefore, to increase the performance of CMOS-technology based digital circuits. These factors in turn affected computing platforms based on these technologies. Several aspects of computer technologies that were effected by above changes, such as:

a) Architectural advancements (e.g. SoC)

b) Miniaturization and portability of systems that were previously large standalone processing systems, as in the case with handheld and embedded processing devices.

c) Application workload by requiring higher systems performance such as multi-task multi-mode data stream processing.

1.1.1 Application Aspect

Looking from the application workload point of view, the stream processing tasks became most performance demanding in various sectors of the market. Increase in processing speed, as well as, cost-effectiveness became the major focus of developments in areas of multi-media, advanced robotics, flexible manufacturing, automotive, aerospace, and many others. The areas associated with the real-time video/image processing, broadcasting, massive data-stream execution for modeling and complex process simulation became the main applications for high-performance computing systems. These advancements cultivated the emergence of new markets in high-performance computing such as: gaming, movie industry, and computational biology. At the same time, it should be recognized that traditional approach of increase in processing performance has met the physical barrier of the operation frequency. This limit required parallelization of computation process and therefore the implementation of many new architectures, as well as, concepts of task adaptive and reconfigurable computing paradigms.

1.1.2 Miniaturization and Embedded Systems Aspect

One of the major tendencies in evolution of computing systems is migration from large stand-alone units towards the embedded systems. In many cases the embedded systems are also of a small form factor and low power. This occurred in most areas of computer applications from manufacturing lines to small hand-held consumer devices. Nowadays it is almost impossible to find any complex system or product that does not contain one or several embedded platforms. However, the embedded implementation of computing system has several very important differences compared to the previous concept of stand-alone computers. These differences are as follows:

1. Strict constraints on area, power consumption, life-time period and many other performance parameters.

2. Application specific vs. general purpose orientation.

3. Shorter time-to-market and low-cost of volume manufacturing.

4. Orientation towards computation intensive applications: HDTV sets, digital camcorders, cell-phones, routers, DVB (Digital Video Broadcasting) systems, machine vision/surveillance/security systems, etc.

All of the above differences motivated the designers of embedded systems to utilize high-performance RISC embedded microcontrollers, application specific accelerators based on DSP processors, and ASICs. However, due to the rapid changes of industrial standards and requirements, decrease in time-to-market requirements, and reduction in cost of reconfigurable logic devices, companies started to move towards utilization of the FPGA devices for embedded platforms. Typically this is true for the products with small or medium volumes of production due to relatively high cost of FPGA devices compared to ASICs. In addition, recent advancements in CMOS technology, and FPGA micro-architecture advancements allowed more cost-effective application of reconfigurable systems.

1.1.3 Technological Aspect

In recent years the progress in process technologies has dramatically impacted the Field Programmable Gate Array (FPGA) development. It resulted in substantial increase of FPGA's number of system gates per device, as well as an increase in the speed of operation. Previously, a simple homogeneous FPGA's on-chip structure moved towards a complex heterogeneous organization of on-chip architecture by inclusion of embedded hardware components, such as: BlockRAM modules, multipliers, DSP elements, digital clock managers, and PowerPC cores. The above advances allowed to implement a complex stream processing systems on the FPGA based systems. A typical use of FPGA device is still a replacement of an ASIC: an FPGA is configured at the start-up time and its configuration never changes during the system operation. At the same time, the cost of such custom stream processors is relatively high. Reasons

for that is high cost of the R&D stage, as well as, high cost of the large FPGA devices (e.g. ~$16,000 USD for one Xilinx Virtex 5 XCV5LX330 FPGA device). Recently some FPGA manufacturers have started to provide rapid configuration interfaces, as well as a capability for partial configuration of their FPGAs. Due to these features it became possible to change the functionality of an FPGA during the operation by the run-time reconfiguration from one bitstream to another. This allowed the device to have a capability of run-time reconfiguration of any part of on-chip architecture without suspension of the rest of the FPGA device. The above novel options made possible multiple re-use of the same logic resources of the FPGA for different parts of application algorithm in different periods of time. Therefore, these options allow to increase the cost-effectiveness of computing platform based on such type of FPGA devices.

1.2 Motivation

All the above aspects of recent computing technology motivated several directions of research. One of the directions is R&D works in the area of run-time-reconfigurable (RTR) computing systems. This area of research considers FPGA based systems that allow dynamic adaptation of the computing architecture to specifics of an algorithm(s) and a data structure of an application (task). On the other hand, the system adaptation can be initiated not only from the external sources but also from internal ones, as well. That means, that the hardware faults (on-chip or on system level), along with a change in system level conditions (e.g. battery level, overheating, etc.) can trigger/request the architecture adaptation. In RTR computing systems it is assumed that all of the above changes could be compensated (mitigated) by the reconfiguration of system or on-chip architecture. If reconfiguration can be automated, the computing platform would become self-adaptable to the workload, and self-recoverable from the hardware faults. At the same time the development process of such systems should be

shortened and such system has to be cost-effective. This may be possible by utilization of virtual hardware resources design approach similar to software object oriented design concept. The virtual hardware components (VHCs) design methodology allows rapid composition of application specific processors (ASPs) from smaller pre-built components. However, in this case certain libraries of the VHCs should be provided, as well as, the associated CAD tools for automated synthesis of ASP architectures. All the above would require hardware support with system level architecture of RTR reconfigurable computing systems (RCS).

The dynamic composition of an ASP from VHCs can be performed in space (spatial partitioning of resources) and in time domain (temporal partitioning). Since the partitioning of computing resources is a function of task algorithm segmentation and multi-parametric constraints, certain research work should address development of methodology of resource optimization and resources partitioning in the multi-parametric design space.

Another important requirement of the high-performance embedded system that lately became critical is the fault tolerance. The temporal partitioning presents a unique capability for the fault tolerance to be performed at a run-time. Hence, research work in this area should closely consider this aspect too.

All the above reasons are major motivation elements for the presented research work, and therefore, dictate the associated objectives for research.

1.3 Objectives

The objective of this research can be summarized as follows:

1. Development of methods and procedures for automated high-level synthesis of virtual application specific processors based on the pre-compiled virtual hardware components (IP-cores). This includes the exploration of multi-parametric design space; the decision

making procedures for selection of optimal processing architectures; scheduling and binding of logic and routing resources in FPGA devices.

2. Research and development of partitioning mechanism and associated procedures for the temporal distribution of on-chip computational and communication resources between task segments. Development of the methodology for synthesis of the multi-parametric optimized task segment specific processor (SSP) architecture.

3. Development of methodology of run-time reconfiguration (RTR) of stream processing data path and investigation of the cost-effectiveness of RTR-architectures.

4. Development of methodology of run-time self-restoration of stream processing ASPs in the cases of transient and temporal hardware faults.

5. Investigation of the system level organization of multi-stream parallel processing architectures and development of a prototype FPGA platform that can incorporate temporal partitioning and self-restoration mechanisms.

1.4 Contributions

The main contribution to the presented research work is the novel methodology for creation of new class of run-time reconfigurable systems. These systems are based on dynamically reconfigurable macro-processors with temporal partitioning (TPM) of computing resources. The methodology allows to develop the architecture of the above RCS and design the set of virtual hardware components optimized in multi-parametric design space to the set of macro-operators. Overall contributions consist of the theoretical and proof-of-concept components.

Theoretical components of research contributions are:

1) A novel methodology for automated conversion of a macro-operator presented in a form of sequencing graph to a set of optimized virtual hardware components. The developed

methodology allows a quick selection of close-to-optimum variant of VHC for a given macro-operator and a set of parametric constraints. The proposed methodology was presented in the paper "Multi-parametric optimization of the modular computer architecture" and published in the International Journal for Technology Policy and Management (IJTPM) in 2006 [51].

2) New methodology for automated task/algorithm segmentation with high-level synthesis of Segment Specific Processors (SSPs). Methodology of the SSP synthesis involved optimization in multi-parametric design space. Developed methodology provided the framework for design of a CAD tool that would perform SSP synthesis and optimization according to hardware and performance constrains. The proposed methodology was presented in the paper: "Macro-programmable Reconfigurable stream processor for Collaborative Manufacturing Systems", published in December 2008 in the Journal of Intelligent manufacturing (JIM) [54].

3) Novel methodology for run-time hardware fault mitigation in partially reconfigurable FPGA devices. This methodology allowed for development of the self-restoration procedures for mitigation of both transient hardware faults and permanent hardware faults with and without functional degradation. The advantage of the proposed approach is that only functional diagnostic procedures can be involved in this process. The time and resources required for fault location procedures can be eliminated. That provides the ability for run-time mitigation of both types of hardware faults. The proposed methodology was presented in the paper: Multi-level Radiation Protection of Partially Reconfigurable Field Programmable Gate Array Devices" and published in 2006 in the Journal of Spacecrafts and Rockets (JSR) [50].

4) The procedure for the evaluation of cost-effectiveness of FPGA based systems using TPM. This procedure presents the analytical models and conditions to select the best suitable FPGA prior to design of RCS with TPM. This procedure was presented in conferences and published in conference proceedings: "Cost Effective Reconfigurable Architecture for Stream Processing Applications" in proceedings of 21st Canadian Conference on Electrical

and Computer Engineering [52] and "Reconfigurable Macro-Processor Cost-Efficient Platform for Rapid Prototyping" in proceedings of 17-th International Conference FAIM 2007 [53].

5) Novel architecture of RCS for system support of the proposed TPM based on the run-time reconfigurable FPGA devices. This architecture included all necessary components to provide the framework for temporal partitioning of on-chip FPGA resources for synthesized SSP cores. The architecture organization was presented in several conferences and published in conference proceedings and journal publications [54, 56, 52, 53, 55, 58].

The proof-of-concept part of contribution consists of the following:

1) Implementation of RCS architecture and TPM on the prototype of Multi-mode Adaptive Reconfigurable System (MARS) platform. The engineering design of embedded TPM was successfully completed on the MARS platform and prototype was manufactured and tested. For verification of TPM performance special SSP cores have been developed and implemented. These cores were associated with high-frame rate stereo-vision stream-processing algorithms. The MARS platform was demonstrated on several conferences and exhibitions: Discovery 2006, CMC Symposium 2006, SVAR 2007, SVAR 2008 with the first place awarded for best design demo.

2) Implementation of methodology for automated task algorithm segmentation and associated SSP synthesis in the special CAD system and further verification of its performance.

3) Creation of an experimental setup, test of firmware components (multi-core SSPs) and further collection of experimental data regarding different performance characteristics. Analysis of the above results and proof-of-concept of the TPM approach for the considered area of application.

Overall, the results of the research have been published in 3 journal and 9 conference publications and partially used in 3 R&D projects funded by Ontario Centres of Excellence

(OCE), Centre of Information and Communication Technologies (CITO), Materials and manufacturing of Ontario (MMO), MDA Space Missions, UBS Ltd. The research has been also supported by NSERC, CMC Microsystems, and Xilinx Corporation.

1.5 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 introduces processing architectures and continues on to classification of the reconfigurable computing systems with the focus on the FPGA devices that support temporal and spatial partitioning. Chapter 3 gives an overview of the different reconfigurable architectures and describes the various approaches that are taken for the implementation of the temporal mechanism (TPM) in reconfigurable computing systems (RCS). Proposal of the general approach of architecture for the TPM in RCS is also given. Chapter 4 discusses the important topic of fault tolerance of RCS that are based on FPGA devices. It gives an overview of existing solutions for the single event effects mitigation and provides several proposals for mitigation methods on temporal partitioning systems. The novel algorithm for the mitigation the both transient and permanent hardware faults is proposed and described in details. Chapter 5 introduces the notion of processing task segmentation and virtual hardware components. This chapter focuses on the cost-effectiveness aspect of the system design and provides methodology for evaluating cost-effectiveness of the design. A cost-effective solution allows to achieve the desired performance with the minimal possible system cost. Chapter 6 explains the essence of how macro-operators are created from the elementary operations. It describes novel methodology for conversion of a macro-operator algorithm to a set of virtual hardware components. It also describes the decision making mechanism to optimize a VHC associated with a given macro-operator in the multi-parametric design space. It shows that the optimization process can be done in a relatively short time since a small set of VHC variants is evaluated. Chapter 7 presents the methodology for creation of a sequencing graph that describes a processing task from available macro-operators. Further, the

methodology of automatic segmentation and selection of corresponding VHCs is explained. The algorithms and procedures for the synthesis of segment specific processors (SSP) are discussed in detail. This leads into the Chapter 8, which describes the CAD software that was implemented on the basis of developed algorithms from previous chapters. The implementation of (MARS) is described in detail from the hardware aspect of the research. Chapter 9 describes the setup of experiments that were based on RCS with TPM. The verification procedure and results are provided, along with the proof of the proposed methodologies. Also shown are: timing, power, resource utilization, compilation results, and the analysis of those results with the resulting overall recommendations for the design of cost-effective RCS with TPM. Chapter 10 summarizes the thesis, lists contributions, and suggests future work that can be continued from this research.

Chapter 2

Architectures of Reconfigurable Computing Systems (RCS)

2.1 Correspondence Between Workload Specifics and RCS Architecture

It is well known that there is a certain correspondence between the workload components (e.g. classes of tasks, their algorithms and data structure), and the architecture of a computing system. These components have to be optimally adapted for this workload execution. Therefore, in the section below the specifics of the workload and associated computing architectures will be considered.

2.1.1 Workload Specification and Definition.

First, the definitions of workload, data structure and mode of operation have to be given since in different contexts the workload can be interpreted in various ways.

Definition: *The workload of a system is a task or a set of tasks that computing system has to process.*

Definition: *Task is an information object that consists of an algorithm and a data structure.*

Definition: *Data Structure is a formal representation of the data elements (operands) and their dependencies.*

Task can be algorithmically intensive or computationally intensive. Where algorithmically intensive task performs multiple logical and arithmetic operations on small set of data, and computationally intensive task preforms repetitive computation on large sets/streams of data at a high speed.

Definition: *Computation intensive workload is a data-stream dominated workload.*

Definition: *Mode of operation of a task is a combination of one of the possible variants of the task algorithm and one of the possible data structures.*

Therefore, if any changes appear in the algorithm and/or data structure of a task, the mode of operation should be considered as different. Both an algorithm intensive task and a computation intensive task with associated workload may be Uni-modal or Multi-modal, as shown in Figure 2.3. Multi-modal workload can also contain several different types of tasks to be processed by the system.

Definition: *Multi-modal task is a task which consists of two or more modes where only one of the modes of operation can be active at a time.*

As an example, we can consider a video processing algorithm which performs edge detection. There are a couple of algorithms that can be employed for this purpose. Sobel algorithm uses a 3×3 matrix sliding window were Robert Cross edge detector uses a 2×2 [27, 38]. These algorithms have different performance parameters and similarly require different amount of logic resources. Hence, even though a task requirement is to perform edge detection, the actual algorithm of the task can be different. The processing operation of the edge detection task is the same, but mode of operation is different based on the type of algorithm used. Similarly, there could be several modes of the Sobel algorithm implementation.

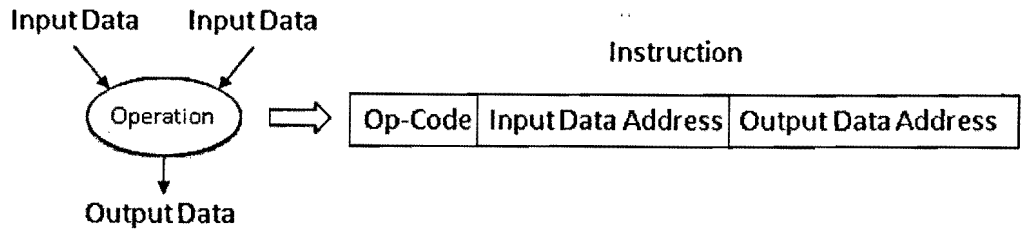


Figure 2.1: Correspondence Between Elementary Operations and Instructions

Some variations perform operation at different speeds, hence requiring more/less resources. This can be achieved by the parallel processing and/or processing of the sliding window(s) operation. Processing is performed proportionally to the number of the processing algorithms in parallel. These variations of Sobel algorithm would also constitute as a mode of operation. First, however, we have to talk about how these tasks are actually computed on a processing system. For that the purpose evolution and classification of computing architectures are explained in the next section.

2.1.2 General Evolution of Computing Architectures

Presentation of an algorithm is typically done in a form of a sequence of elementary operations (arithmetic & logic) where each operation is associated with some scalar data. This approach resulted in Von-Newman architecture which operates with a set of arithmetic or logic operations presented as an instruction set, as shown in Figure 2.1.

This type of architecture presented great advantages, such as: relatively low hardware cost; simplification of programming; high flexibility in implementation of algorithms; limitless complexity of implementation. At the same time, this architecture has disadvantages which are mainly reflected in the performance. Due to the sequential nature of the algorithm execution, scalar data representation, the number of operations per instruction, and the speed

of algorithm processing is significantly reduced. Over the years attempts for computational acceleration have been made through the exploitation of data level (structure) parallelism (DLP) and algorithm (instruction level) parallelism (ILP).

Algorithm parallelism involves execution of multiple independent instructions at the same time. This approach was implemented in the Very Large Instruction Word (VLIW) and superscalar architecture processors. VLIW processors improved processing speed by executing different sub-steps of instructions in a pipeline. VLIW [21] approach has separate processing units that can issue several instructions for execution simultaneously. VLIW has been also considered as ILP. This approach allowed a dramatic simplification of control unit in VLIW processor and thus, reduced its cost. Also, in some cases to utilize the CPU resources the instructions were executed in a different order than they appeared initially in the user program. VLIW processor executes instructions in parallel. Execution is based on a schedule determined at the time of program compilation. The effectiveness of such architecture depends mostly on the advancement of compilers and is identified by how well relevant user code matched to the machine code. The main area where VLIW processors are cost-effective is DSP applications [120]. Superscalar CPU architecture similar to VLIW executes more than one instruction during a clock cycle by issuing several instruction to the idle processing units on the CPU. In contrast to the VLIW architecture, superscalar architecture uses special CPU hardware to dynamically check for data dependencies of issued instructions at a run-time. In addition, specific hardware blocks have to be integrated to perform dynamic instruction rearrangement and branch prediction in order to load the appropriate instructions and to avoid data and control hazards. This significantly increases the complexity of the CPU hardware.

There is another approach to improve CPU performance similar to ILP. This approach involves data structure parallelism, which operates on execution of multiple independent data elements at the same time. This approach was implemented in Vector and Array processor architectures. This vector processor approach was also classified as Single Instruction Multiple

Data (SIMD) [26]. As the name states, the architecture had one instruction and multiple data that was processed in parallel. This contrasts the scalar processors which handle only one element at a time using multiple instructions. Such architecture is specifically useful in the cases of image rendering and, therefore, is integrated in many conventional processors beginning with x86 MMX series that supported enhanced graphics. Personal computer processors nowadays have many SIMD processors each executing short data vector instructions. A collection of many SIMD operating in parallel are combined into Multiple Instruction Multiple Data (MIMD) [42] processor architectures. The advantage in using SIMD is mainly to increase processing power of the system if the data is available in parallel as a vector. The rendering of video in video processors is a good example. However, there are disadvantages to this approach as well, such as packing and unpacking data from SIMD registers. Also, there is a poor support of compilers to implement more complex tasks [62]. When processor is required to perform operations on much shorter data vectors or if there is high inter-dependency in data the performance of the processor drops significantly. In many cases this in turn requires a programmer to try to adapt the program, and align the data, so that the SIMD capability is exploited. This becomes a very tedious and complicated task and it is not suited for many applications that do not operate with large arrays/vectors of data. The classification of computing architecture is presented in Figure 2.2.

All of these architectures provide acceleration from one to two orders of magnitude in comparison to the first mentioned architecture. However, they also a higher hardware cost. The reason for the increased cost is the complexity of the processor design, as well as much larger silicon area. The silicon area was reduced due to the advancement of process technologies and diminishing transistor cost. Another aspect is the complication of programming, since a programmer has to be closely familiar with the processor architecture in order to achieve relative improvement in performance. This is somewhat mitigated with the development of highly advanced compilers and operating systems.

However, the sequential nature of algorithm processing is still the major factor which limits acceleration of processor performance. For much higher performance and even more parallel execution of algorithm high customization is needed. High degree of customization can be achieved by implementing the task/algorithm in fixed architecture such as ASIC. This is described for the next section in the case of stream processors.

2.1.3 ASIC and FPGA Static Stream Processors

Stream processor is a processor that performs the same type of algorithm operation on constantly incoming homogeneous type/format of data. Such processing operation is typically suited for applications requiring high performance where processing algorithm (mode of operation) does not change. Stream processors are mostly used in video, audio, and gaming applications where vast amount of data has to be processed in parallel at a high speed. The processing is of the same mode of operation and does not change throughout the operation of the system. Stream processor typically is implemented in ASIC form, where critical processing operation is paralleled and pipelined to increase overall performance. In this scenario both parallel execution of algorithm and parallel data structure are exploited since they are built into the hardware. The obvious downside of the ASIC implementation is its inability to support various modes of operation. In order to support several modes of operation, they have to be included in the ASIC design. Another option is to use reconfigurable device such as FPGA. A static configuration of a stream processor is uploaded to an FPGA device and it operates for the rest of active time. Typically, the advantage of using FPGA is the capability of future upgrades in the scenario of changing mode of operation. In case when mode of stream processing has to be modified, it involves off-line bitstream recompilation and subsequent off-line upload to the FPGA system.

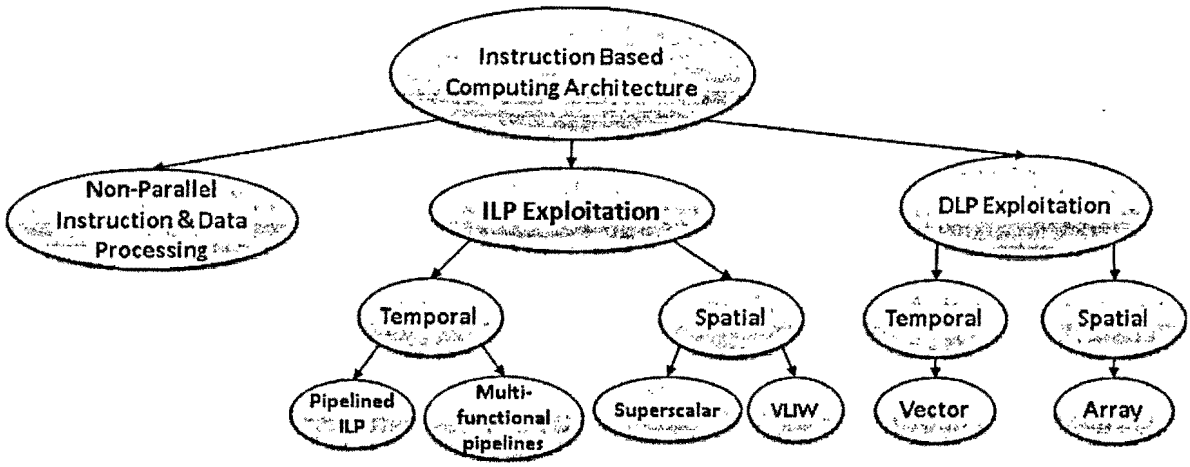


Figure 2.2: Classification of Computing Architectures

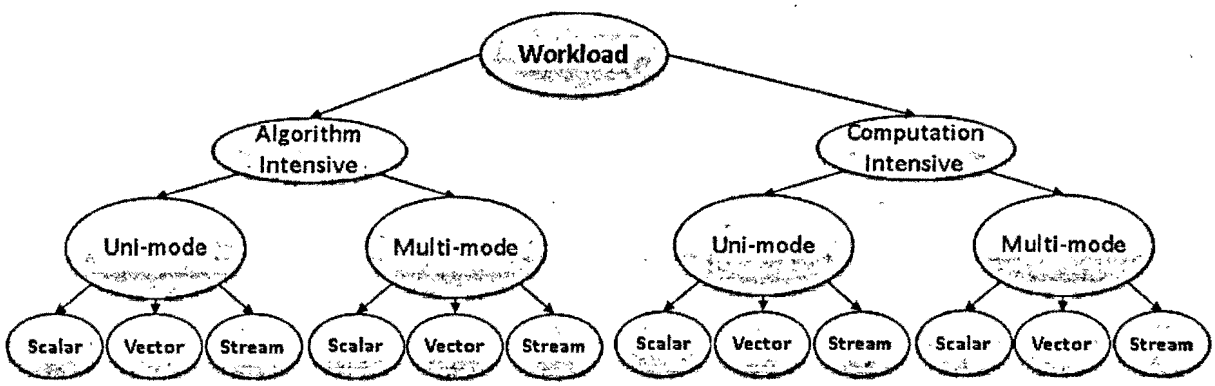


Figure 2.3: Workload Classification

This naturally leads to the notion of an adaptive stream processor based on a reconfigurable computing system (RCS). Next section introduces the RCS and classifies different types of reconfigurable systems.

2.2 Introduction: Concept and Benefits of RCS

The first reconfigurable computing systems were proposed in the 1960's [109]. However, in practical implementations they have emerged only in past two decades. Initially RCS started from the very basic systems that were capable of switching between the available resources [35]. Already at that time RCS showed the great benefits of workload adaptive systems. In reality, any variation in workload (e.g. changing in algorithm(s) or data structure) may cause

degradation of cost-performance parameters of a computing system. Therefore, there would always be some mechanism for workload/computing architecture adaptation. In conventional instruction based computers the only way of the above adaptation is the optimization of the program/data structure on certain platform architecture (e.g. processor architecture, memory structure and bus(es) organization). The concept of RCS is completely opposite to the above approach and optimizes computing circuits/memory and communication resources to the workload algorithm(s) and data structure(s). Conceptually, this approach allows reaching much higher performance and cost-performance characteristics for a computing platform, compared to the platforms with fixed architecture.

There are many different paths that were taken by the researchers around the world and number of classes of RCS developed over the years [33]. These devices are classified in several categories, and this chapter will address the main ones. This chapter will also describe different types of RCS and trends of their development.

2.3 Definition and Classification of RCS

An architecture of the computing system can be described in three main parts: Components - (C); Links - (L) between components; procedures - (P) associated with components and their links. In all computing systems some of the components of architecture are variable and some are fixed. Basing on this classification it is possible to describe any type of computing system according to flexibility of the above architectural parts.

Microprocessor architecture consists of Components that are fixed - $(\overline{C})$, and links between the hardware components $(\overline{L})$ that are fixed as well, however, the procedures associated with the components are variable - $(\widetilde{P})$. Therefore microprocessor architecture-(A) can be described as a: $A = \{\overline{C},\overline{L},\widetilde{P}\}$. If we have a system that has all of the components same as the microprocessor system, but the links are not-fixed $(\widetilde{L})$, we can describe its as a course grain

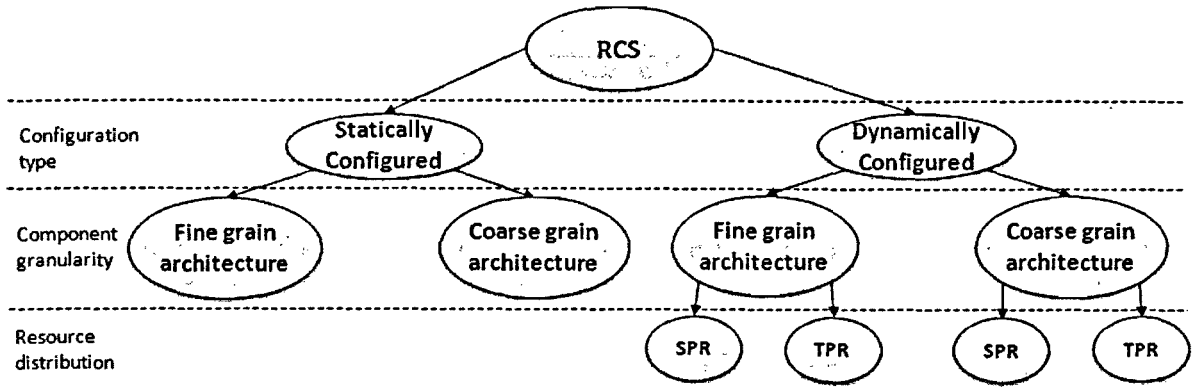


Figure 2.4: RCS Architecture Classification

configurability system $A = \{\overline{C}, \widetilde{L}, \widetilde{P}\}$. This type of system has fixed processing blocks, however links between them can be altered. This provides the flexibility of path interconnection for processing units, and increases the productivity of the system. This class of reconfigurable architecture is described in detail in Section 2.5.3.

A system where all components are variable thus can be fully reconfigurable, for example, PLD and FPGA devices: $A = \{\widetilde{C}, \widetilde{L}, \widetilde{P}\}$. It allows full customization of the computing platform. This approach allows to tailor the full structure of the system to the algorithm and data structure that needs to be processed and, therefore, it allows to achieve the highest possible processing acceleration. Section 2.5.2 describes these sort of devices in general.

Reconfigurable computing systems can be divided in several types based on the classification of the system and its function of operation as shown in Figure 2.4. First of all, reconfigurable systems are classified into statically and dynamically reconfigurable systems. The distinction between these two types [46, 1, 99] is discussed in the next section. Reconfigurable systems could be based on the fine grain or coarse grain architecture, which impacts the flexibility and thus performance of the system. Distinction between fine grained and coarse grained architectures is explained further in the chapter.

At the same time, dynamically reconfigurable systems for both fine-grain and [46, 1, 99] coarse-grain types can utilize Spatial Partitioning of Resources (SPR), as well as, Temporal Partitioning of Resources (TPR). SPR and TPR are be described later in the chapter.

2.3.1 Statically and Dynamically Reconfigurable RCS

As mentioned previously, one type of reconfigurable systems is a statically configurable system. It is configured at the boot-up stage or at a hard reset of the system. The initial configuration stays on the system until the next boot-up or restart. Such are the most common systems that are used in the industry [75, 7]. The main advantage of such systems is the capability of future or remote upgrades, as well as, the ease of these upgrades. Start-up time for these systems is relatively small (e.g. in the range from tens of milliseconds to unit seconds) [59]. Which is acceptable for most of the applications. Static configurable systems are used to address two main objectives: improving the performance and optimizing the utilization of resource components [94] for one specific application with fixed algorithm and data structure. Therefore, statically configured systems represent an ASIC type processor where the system is tuned to preform only one specific task (e.g. MPEG-4 encoder). This leads into the efficient utilization of resources, such as logic resources, as well as, power consumption. This system can be tailored to use all available resources, and to operate at maximum clock speed [67]. On the other hand, such systems are not adaptable to dynamic change in application modes of operation and/or peripheral interfaces. That is why, similarly to ASIC implementation, all circuitry associated with all possible tasks/modes of operation must be included into a single FPGA design.

Another type of reconfigurable system is a dynamically reconfigurable system. Dynamically Reconfigurable Computing Systems (DRCS) involve a processor that can be reconfigured [3] with a new configuration stream during the system's operation. The new configuration stream changes its architecture configuration at a run-time and adapts the system to a new application/mode/set of parameters as shown in Figure 2.5.

Consider an example situation where initially RCS was operating with a parallel input of two data sources, and two processing units which produced two parallel results. After some time a new requirement was introduced where a third data source to be processed was added,

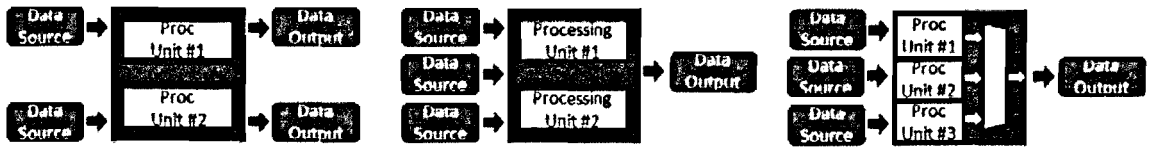


Figure 2.5: RCS Run-Time Resource Adaptation

as shown in the middle of the Figure 2.5. At the same time, a new restriction was set that only one output could be driven from the system. Therefore, a third processing unit was added along with a selector of these outputs, as shown in the right most depiction of Figure 2.5. Such adaptations are possible in both fine grained and coarse grained architectures, however, there are distinct differences between these adaptations which are explained in the next section.

2.3.2 Fine Grain and Coarse Grain Architectures of RCS

Reconfigurable computing systems can have the fine grain and course grain architecture organization. In the coarse grain architectures a reconfigurable system/device consists of large components which have fixed architecture. These components are interconnected between each other, and the connections can be reconfigured. It is possible to change the functionality of these components by reprogramming. However, the actual hardware architecture of these components is fixed. Such architectures can be found in many references (e.g. Goldstein *et al* [35]). Currently these systems did not get to the mass market use due to several unresolved issues. The main issue of such system is associated with limited flexibility of component architecture. In a case of new requirements or if the device/system needs to be used in a different application, the system utilizing course grain architecture might not be flexible enough to fully optimize the component functionality to the new requirements [77]. Coarse grain architecture does present an advantage in some cases, such as when all the modules are known, and specifications do not drastically change. In this case modules provide maximum performance and operate faster than fully reconfigurable devices with fine grain architecture [77].

Fine grain architecture systems operate on a different notion, which involves very small and simple operational blocks that are organized in a homogeneous manner. This homogeneous field of logic elements can be configured to perform primitive ALU operations. Throughout the years there have been different variations of fine grain architectures [93]. Fine grain levels varied from the configuration of individual transistor up to large processing hardware cores. The advantage of a fine grain architecture is the maximum possible flexibility that is offered by these types of systems [11], and vice versa. However, very fine grain architectures present a problem of increased routing resources, configuration bitstream resources, and associated configuration SRAM requirement. Another aspect that became quite significant in the system development in the past several years is the FPGA design place & route time. In fine grain architecture more routing resources have to be considered, and therefore, it takes even longer time in design compilation. In initial development several architectures have shown the optimal balance in granularity and became popular choice for fine grain device manufacturers, such as 4-input Look-up-Tables (4-LUT) to 6 input Look-up-Tables (6-LUT) configuration block architecture. As the IC process technology evolved over the years it became possible to have a large number of fine grain configurable blocks on reconfigurable devices which provide the users with capability to implement complex processing systems. These systems became comparable to ASICs in terms of performance and are more and more preferred by companies due to the shorter time-to-market, as well as, virtually no re-spin time and associated NRE costs. However, there are several additional trade-offs that have to be considered nowadays when dealing with the fine grain architectures [104]. The simpler the fine grain block is, the more complex the routing for configuration and interconnection becomes. Thus, there is more hardware overhead associated with links and configuration memory volume.

The compromise can be found in Hybrid architectures. Hybrid architectures include fine grain elements and application specific coarse grain homogeneous elements (e.g. in Xilinx

Virtex 5, Virtex 6 FPGAs) [118, 64, 121]. This approach has currently become the most popular choice for reconfigurable devices. The next section presents the concept of resource partitioning on the fine grain reconfigurable systems in details.

2.3.3 Resource Partitioning Schemes in RCS

The effectiveness of RCS depends on the effectiveness of architecture-to-workload optimization. In other words, performance parameters of RCS are a function of distribution of architectural resources on workload segments. The resources can be partitioned in spatial and/or temporal domains. Thus, RCS can be classified as follows:

Definition: *Spatial partitioning of resources assumes distribution of workload segments on computing resources "in space"- different sections of the reconfigurable components and links. In other words an RCS with spatially partitioned resources at any time dedicate different sectors of its architecture to certain algorithm segments.*

The architecture can be dynamically modified by reconfiguring one of the sections with the new algorithm segment.

Definition: *Temporal partitioning of resources is a division of tasks into smaller sub-tasks or segments that are reusing same resources of the RCS device in the different time periods (time slots).*

These two classifications have been in the works for the past decade and provided capabilities of adaptable hardware. These capabilities allow to have a run-time adaptive hardware with relatively low power consumption, smaller size, and more. General operation of these approaches are shown in the Figure 2.6.

As shown in the Figure 2.6, spatial partitioning of resources reuse sections of reconfigurable resources of the RCS to load new segment(s) of the algorithm/task. This achieves an adaptability of the RCS to a new algorithm/task. If system has to execute several algorithms/tasks, it can replace them by each other when needed. In the example shown in Figure

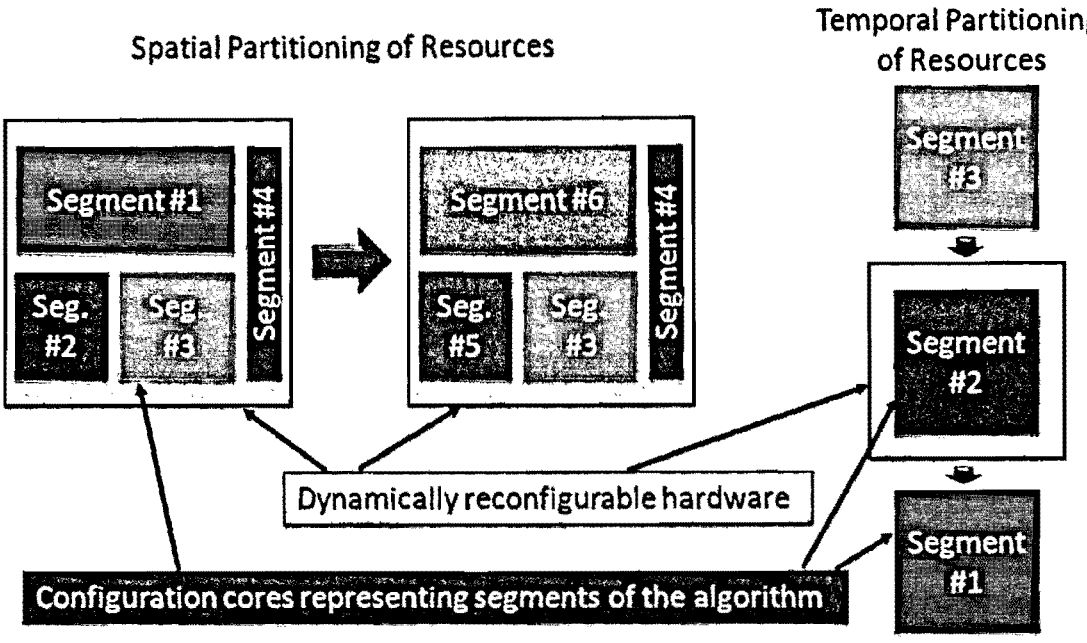


Figure 2.6: Spatial and Temporal Partitioning of Resources

2.6 segments #1 and #2 are replaced after completion of their execution with the segments #6 and #5 respectively. For example, a video processing algorithm that was using edge detection as part of its processing (e.g.. Segment #1) can be changed to perform colour intensity calculation instead, based on the request of a user or particular environment parameters.

For temporal partitioning of resources, segments can occupy the whole area of reconfigurable resources in the RCS, and then the segments are reloaded in specific sequence. Each segment performs a part of the algorithm/task within a required time slot. For example, a large JPEG2000 encoding algorithm was subdivided on Tier 1 encoding and Tier 2 encoding into two different segments, and these segments can be processed sequentially one after the other requiring smaller FPGA.

A workload can consist of algorithmic and computationally intensive tasks, therefore, depending on the type of algorithm at hand, a fine-grained or a coarse-grained architecture would be preferable. If the workload contains both algorithmically intensive segments and computationally intensive segments then the hybrid RCS architecture would be the most appropriate solution.

Definition: *Hybrid RCS architecture consists of both coarse grained and fine grained components and can be flexible enough for the computationally intensive tasks that deal with streams of data, as well as, capable of processing sequential algorithm intensive tasks.*

Most of the modern high-end RCS devices such as Xilinx and Altera FPGAs contain the fine-grain components [22, 118] such as CLBs and interconnection switches, along with coarse grain components, such as PowerPC cores, embedded memory, and others. The presence of both fine-grain and coarse-grain components allows to maximize the performance of the target application, as well as to minimize the time of implementation. The Task/Algorithm elements that are highly parallel and can be pipelined are designed in the fine-grain reconfigurable area. This customization gives the maximum performance for the parallelized units. At the same time, parts of the algorithm that require sequential processing and that are dependent on the result of inter-processing are realized on coarse-grained components. Such coarse-grained components are PowerPCs that are general purpose processors implemented in a hardware core. Consequently, they perform at the maximum available speed. The advantage of the hard processors like PowerPC in comparison to the soft-processors, such as: Nios II, MicroBlaze, and LatticeMicro32, is that PowerPC type processors performance is much higher since it does not incur routing delays through the switch network. It should be noted that only the higher end FPGAs are equipped with hard core microprocessors. In addition, the presence of these microprocessors significantly impacts the cost of the device. By including the hard core processors the valuable configuration logic area is occupied and, therefore, the amount of configurable logic is much lower when PowerPC core is included. Comparison between the XC5VLX330 and XC5FX200 is shown in Figure 2.7. These FPGAs are the largest in Xilinx Virtex 5 LX and FX families, and it can be seen that the logic amount is significantly affected in FX family where two hardware cores of PowerPCs are placed. This is due to the fact that hard core processors require a significant silicon area that instead could be devoted for logic.

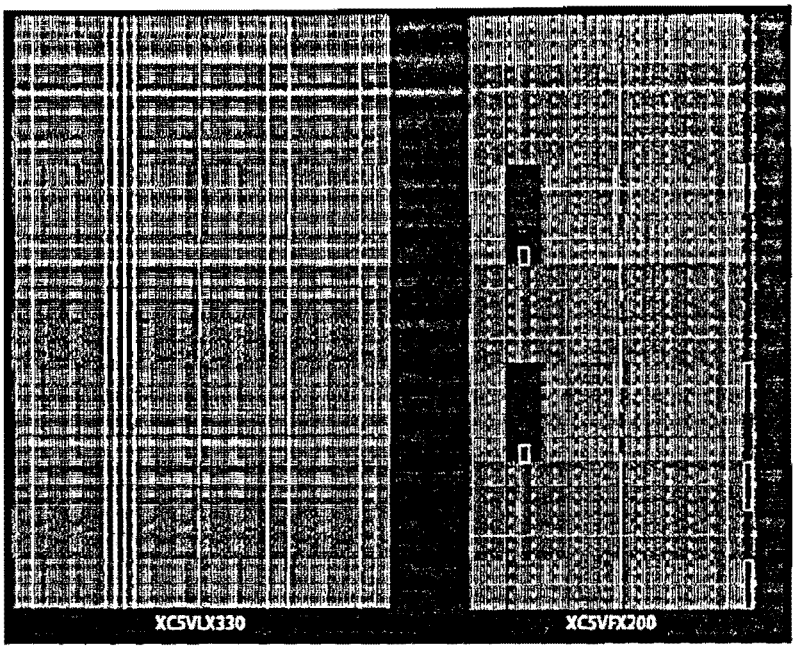


Figure 2.7: Largest Virtex 5 LX and FX devices Floor Plan Comparison

2.3.3.1 TPM to SPM comparison

As it was described in the previous sections, there are two approaches for the partitioning of resources: temporal and spatial. This section briefly describes the pros and cons of the spatial partitioning approach. The spatial partitioning focuses on the reuse of the FPGA fabric by replacing a module with a different one or with a set of modules that are performing a different type of operation, and therefore, reuse the physical resources. For this type of system architecture a much larger FPGA is required since it involves more units than in the temporal partitioning. The advantage of the spatial partitioning is that if a task requires resources that fit into the FPGA it allows to run the whole data processing in parallel, and achieve a higher speed of processing. In addition, if a task assumes multi-modal operation, the system can change a mode by reconfiguring a part of resources accordingly.

However, there are disadvantages to the spatial partitioning approach. If a processing task requires more physical resources than the largest FPGA can provide, the spatial partitioning approach might not be cost-effective. If the processing task does not fit into the selected FPGA the task either has to be divided into the additional FPGAs, or a part of FPGA will have to be

reloaded, essentially converting the system into a semi-temporal partitioning system. This fact would also affect the design complexity of the system since a partially configurable design will have to be implemented along with the temporal partitioning. A rapid temporal partitioning reconfiguration module would have to be added as well. This aspect reduces the advantages of the spatial partitioning approach and makes its applications tremendously limited. As was discussed before, the spatial partitioning approach also uses more power, as well as reduces the overall cost-performance of the system due to the use of a much larger device. As shown in the experiments and results section of Chapter 9, the use of a small to a mid-sized FPGA is more preferable in terms of the power and performance parameters. This is especially true for the cases where a stream processing task requires a run-time mode change. The proposed TPM performs this change seamlessly compared to the spatial partitioning approach, which requires suspending processing operation of a section of FPGA for a relatively long period of time.

On the other hand, if the performance requirements are very strict and assume reaching the possible maximum speed of data execution, then there is no other choice except the spatial partitioning of the FPGA resources on task segments.

2.4 Cyclically Reconfigurable Architecture with Macro-Block Processing Cores

As the process technology shrunken from 220nm to 40nm in the past 10-15 years, a dramatic increase in number of logic cells from just a 1700 (in Xilinx Virtex-I in 1998) to 758000 (in Latest Xilinx Virtex-6 in 2010) occurred as a result ($\sim$ 445 times increase). Yet, the computational power of conventional computers has not evolved at such rate. This, unfortunately, presents a problem which is intensifying with every new larger family of reconfigurable devices. Since place and route is a NP-complete problem, larger FPGA takes even longer time

to compile. When initially the compilation required only a couple of minutes, nowadays the compilation time for large and complex designs takes tens of hours. For this, a new approach has to be devised to mitigate this growing problem. This approach is going to be introduced in this section.

In addition to the compilation latency there are other issues that arise associated with the design complexity and signal timing. As devices increase in size, the complexity of the design is increased tremendously, and even more experienced designers than before are required. The FPGA manufacturers are trying to solve this problem by creating CAD tools that allow modular designs and capability of design planning and budgeting. However, such approaches still do not solve the overall problem, and overall design complexity is very high. Due to the large ICs, the signal latencies inside the FPGA fabric increased as well, which creates a problem for synchronization of different modules. A different approach in system design is the utilization of the temporal partitioning to provide a simplification of the design, better timing results, and in the end a cost-effective solution.

Initially, the temporal partitioning approach was introduced to resolve the limitation of available resources on a FPGA device. The same approach can be used to resolve timing and complexity issues. Typically, temporal partitioning is used to divide the algorithm in parts, which requires complex development, and in some cases it even increases the complexity of the whole system. At the same time, if taking a different path instead of the division of the algorithm/task, it can be composed from smaller sub-tasks.

2.4.1 Temporal Partitioning as a Macro Operator Approach

It is possible to use pre-defined modules to assemble the processing algorithm. This approach is similar to how one would construct a processing task/algorithm using MATLAB tool or an object oriented programming language. The advantage of such approach is that a macro operator is an already pre-designed module, with particular parameters which can be chosen by the

user. This simplifies and accelerates the system design tremendously, similarly as an object oriented language simplifies and accelerates the development of the large scale applications. Moreover, just as the object oriented programming changed the way software development is conducted, the same is possible in case of Macro Operator use. This approach also presents a cost-effective solution: the developer does not need to have any extensive knowledge of embedded system design, and "assembles" the processing task/algorithm in the higher level of abstraction. The idea behind the proposed research is that a CAD tool performs all of the operations of optimal Macro Operator selection, division and compilation. In addition, by performing resource binding, the CAD tool is capable of utilizing deep pipelining of the function specific data-paths. Since stream applications typically have large frames of structured data, such pipelining greatly improves the performance, while keeping the same frequency of operation. To optimize the cost-effectiveness of the whole design, the CAD tool performs the selection of an optimal reconfigurable device. The CAD tool accommodates the performance restrictions imposed by the user, and takes into an account the average cost of the peripheral components when making the suggestion of the reconfigurable device.

The approach and its advantages mentioned above present a motivation for this research work. Hence, the objective of the research consists of developing three main parts: a methodology for Macro Operator creation and task/algorithm segmentation; methodology of hardware design architecture with support of temporal partitioning; a CAD software that implements the segmentation methodology and creates a user interface for creation of cost-effective applications utilizing temporal partitioning methodology.

The following chapters present an overview of other reconfigurable systems and approaches that were developed by other research groups. The methodologies of task/algorithm segmentation, hardware development, and CAD software development are presented, as well. The methodology of task/algorithm segmentation covers the aspects of how segmentation has to be performed for the reconfigurable hardware systems that process tasks using temporal

partitioning mechanism. The methodology of creation of Macro Operators and their particular instances is developed as well. On the hardware side of the research, a methodology of designing hardware architecture for temporal partitioning systems will be presented with comparison to the existing approaches and systems. This methodology is closely linked to the task/algorithm segmentation aspect since hardware can not be designed efficiently without taking into consideration the processing task/algorithm. The CAD software implements the above mentioned methodologies. The design of the CAD software is described, since it is a vital interface component between the actual user and the final hardware. All of the above methodologies are presented in the next several chapters with the references of dependency links between these methodologies.

2.5 Definition and Classification of Programmable Logic Devices

Over the past two decades there has been an increasing demand for reprogrammable and reconfigurable computing devices. Makimoto's wave [70], a prediction done in early nineties, indicated that the most of the computing technologies would include field programmability. Various range of reconfigurable devices are systematically closing the gap between the ASICs that are oriented for high performance of one dedicated application, and the microprocessors with high programming flexibility. Industrial companies are starting to use reconfigurable logic devices in their system designs instead of ASICs due to several factors. For example, the time-to-market factor is one of the most important things in the competitive market. In addition, for smaller companies a reconfigurable device is the only solution in many cases as the manufacturing cost of an ASIC device can be simply not financially feasible. Customization is one of the key aspects that also makes the reconfigurable devices much more attractive. In many instances customer might require a system with some interface customization or

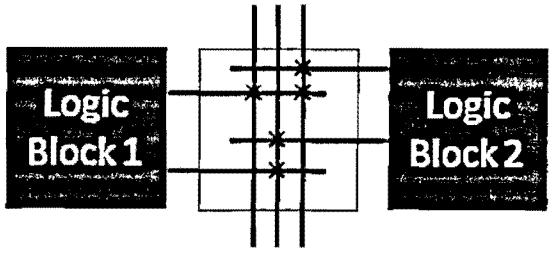


Figure 2.8: FPGA Switch Blocks Interconnections

other types of modifications for his system. When system is implemented in the ASIC, customization involves either additional peripheral hardware or the inclusion of various popular interfaces into the ASIC.

Over past the two decades programmable logic devices went through many stages of development. Several different types of reconfigurable architectures were developed and yet others are still in development. Lately manufacturers mostly concentrated on hybrid architectures of processing and architectures of communication buses. In this section different types of reconfigurable devices will be covered, with the focus on those that are directly applicable to this research.

2.5.1 The Concept of Programmable Homogeneous Logic Resources

The concept behind the programmable homogeneous logic devices is quite simple, and involves elementary configurable blocks. The whole idea behind the homogeneous architectures is the existence of identical structure blocks, which can be programmed to act as particular logic element. These blocks are organized in a homogeneous manner and can be configured to simple logic elements (e.g. AND, OR gates) as mentioned in Section 2.3.2. Reconfigurable devices are formed as field of simple configurable logic blocks that are interconnected between each other by configurable buses and switch blocks of interconnection routing. As shown on the Figure 2.8, the interconnection occurs by turning on the switches that are located between the outputs of these elementary logic blocks and interconnection routing.

At this point there are several possibilities that have been selected for the interconnecting switches, however, this will be discussed in the later sections. By interconnecting these elements together a very flexible architecture can be created. However, with the benefit of flexibility comes a disadvantage. This disadvantage is associated with the speed of operation of a system. For example a 16×16 multiplier implemented as a hardware core would have much higher performance characteristics than the one programmed in the homogeneous structure. This is mostly related to time for signal travel from one logic element to another.It takes much longer to traverse through the routing and all the switches than through the hardware core form which has direct connections. For this reason there was a shift in recent years from a purely homogeneous structure to a combination structure of hybrid architectures. These architectures involve homogeneous structures combined with some fixed hardware core blocks such as multipliers, memory modules, microprocessors and more. Over the years two major types of devices have emerged from the research and development and dominate the reconfigurable device market: Complex Programmable Logic Devices (CPLD) and Field Programmable Gate Arrays (FPGAs). These devices are described in detail in the next sections.

2.5.2 Fine Grain PLDs – CPLD and FPGA Devices

Fine grain CPLD and FPGA devices with homogeneous structure were the first ones to appear in the mid-80's from Xilinx, Actel, and Quicklogic companies. Initially, they evolved from the Programmable Logic Arrays (PLA), and nowadays include much larger configurable logic modules. This allowed them to become not just a replacement for the non-standard peripheral logic ICs, but to be used as a main processing component in the system design. There are considerable differences between CPLDs and FPGAs. FPGAs are much larger devices that are organized in the Configuration Logic Blocks (CLB) based on Look-up-Tables, whereas CPLDs are organized in Macro Blocks based on the actual logic elements. Due to their lower cost and lower power consumption CPLDs are used mostly as simple controllers/drivers that

have vast amount of I/Os, and they can operate on high speeds (e.g. 500MHz). Commonly, CPLDs do not require more than one voltage supply. They also support various I/O standards for the ease of chip-to-chip interfacing, thus making them an attractive solution for mobile and power sensitive devices. Typically CPLD devices are used for interfacing or drivers with low power consumption and low cost, and widely used in industry for that specific purpose in all types of electronic devices.

As mentioned before, FPGAs are much larger devices organized in the homogeneous manner and composed of CLBs, interconnecting routing, switch blocks and Input Output Blocks (IOB). Lately FPGAs started to include the integrated hardware cores, such as: Block RAM, Multipliers, DSP modules, and even PowerPC processors. Homogeneous structure of these new FPGAs gives a designer a flexibility of creating architecture of their custom processor, at the same time allowing to use the integrated devices, such as hard core 18×18 multipliers, which deliver a result within one clock cycle. This in turn avoids bottlenecks by reducing the processing time of such complex operations as multiplication. Similarly, embedding the blocks of $1K\times16$ Block RAMs all over the FPGA's fabric provides the distributed memory that can be used by the neighboring logic elements. This creates a local storage eliminating the need for external temporary memory. Access time is also faster to these blocks than the distributed FPGA memory since it is located in localized blocks. This in turn increases the processing speed of the overall system. Embedding PowerPC cores allows a designer to offload sequential processing that often exists in the embedded systems and not implement it in the FPGA's logic block. This is a better solution in some cases because the PowerPC is implemented in the hardware core form and performs faster than the solution implemented using CLBs. In the latest Virtex 4 and Virtex 5 families there are up to two PowerPC cores [118] that can be used and run in parallel connected to the CLBs.

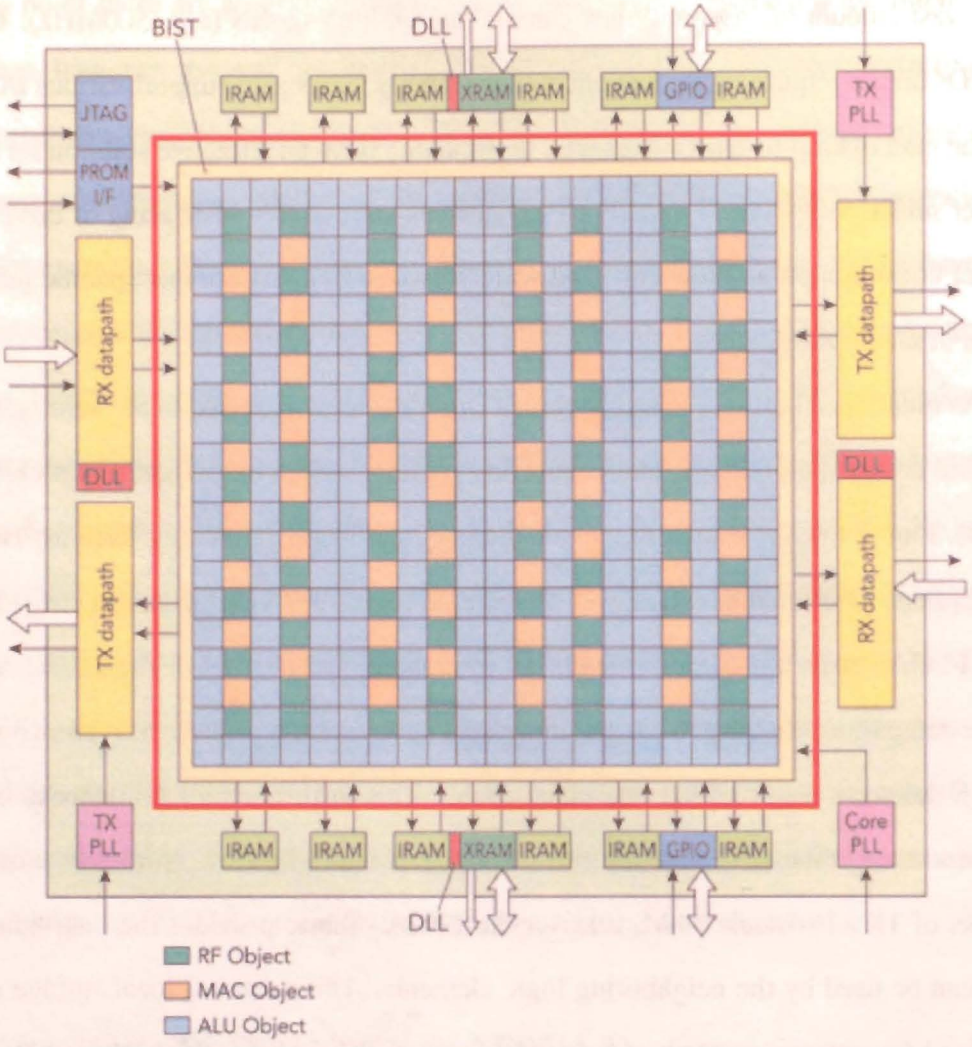


Figure 2.9: Field Programmable Object Array (FPOA) Architecture Diagram

2.5.3 Coarse Grain PLDs – Field Programmable Object Arrays

In very recent years some coarse grain devices have appeared in the marketplace, such as Field Programmable Object Arrays (FPOA). They were created to fill in the gap between the ASICs and FPGAs. FPOAs are not user-programmable at the gate level, they are programmed at the object level. Object types include several arithmetic logic units (ALUs), register files (RFs), and multiply/accumulate units (MACs), each of which is programmable. The object types are shown in the Figure 2.9. IC implemented modules are located in the devices on the interconnection grid like in modern FPGA devices. For the efficient processing and interfacing

the memory and I/O modules are arranged around the device perimeter. The modules are overlaid with high speed routing that can be configured by the user. Due to its benefits this approach is attractive to the space and aviation industry, and one of the reasons is radiation hardened. Since large blocks are implemented in hardware cores, they require less SRAM to hold the configuration data. The SRAM is susceptible to Single Event Upsets (SEUs) which are caused by the cosmic radiation. Therefore, the less configuration SRAM there is, the less is the probability of SEU and hence the occurrence of an error or a functional interrupt. At the same time, in the case of space applications, the flexibility is needed, as well. Therefore, FPOA fits applications where SEU is one of the major factors and where some flexibility is required. The move from ASICs to FPOAs was initiated by the fact that companies which used ASIC technology for implementation of their systems were forced into constantly making modifications of their algorithms. When the development of the ASIC was halfway done, a new change in the algorithm would be introduced which would cause the restart of the whole process and obviously increase the R&D costs. The interconnection combination of highly optimized ASICs produce a much more effective and modifiable solution. There is a trade-off in the performance of FPOA devices, but in the instances where a change of algorithm does not impact the architecture and is supported by the existing FPOA objects it is feasible to use such devices. Unfortunately, in typical applications, designers need much wider flexibility, and therefore FPOA suits only a very limited range of applications. In order to widen the range, the FPOAs have to include either more objects, or these objects have to be generic in their design. Both of these solutions compromise either the cost or performance of the end system and, therefore, are less attractive for the most of system designers.

2.5.4 Run-time and Partially Reconfigurable PLDs

The era of partially configurable devices started a while ago with the release of the first Xilinx XC2064 family of FPGAs in 1985. This FPGA allowed to addressing and reconfiguration of

a single cell. It offered tremendous advantage because the device was capable of operation even during the reconfiguration of its blocks. However, this family of devices had one major flaw preventing it to be successful in the market. Besides the fact that XC2064 family did not have a large amount of configuration blocks, it also lacked any type of firmware protection as a consequence of configuration flexibility. Anyone could read the contents of the device back to PC and decode the whole architecture and this would compromise the security of the IP-cores. Obviously with such problems, the industry could not use this device in their products. This device was mainly used in the research field where this flaw was not relevant. Later, Xilinx Virtex (e.g. Virtex E, Virtex II, Virtex II Pro, Virtex 4, Virtex 5, and Virtex 6) families also employed the capability for partial reconfiguration. Unfortunately, there is no software support for design of dynamic partially reconfigurable architectures, so they are limited to research labs.

As shown by many researches [102, 110, 60] dynamic reconfiguration is an extremely powerful capability, since it allows the hardware to evolve based on the basis of the surrounding environment. This capability also permits a creation of fault tolerant systems, which can recover from the hardware faults by aligning the IP-cores in such a way as to avoid the damaged regions. The design of systems that employ the partial configuration mechanism is quite complex and requires highly experienced systems engineers, as well as a very good software support. Due to that fact companies were not demanding the support for partial reconfiguration, and temporal partitioning has not been the first priority of partially configurable device manufacturers. Instead, device manufacturers were mostly concentrated on perfecting the current devices with the modular design capabilities and the speed of operation with only minimal support of the capabilities for partial configuration. Therefore, the most of the software tools were improved in the area of modular design, such as recent Xilinx PlanAhead.

2.6 Summary

This chapter covered the notion of workload for computing systems and its classification. The focus of this thesis work is on the computationally intensive multi-mode stream processing architectures. This chapter provided literature overview of the processing architectures with their classification. This led into the introduction of the concept of reconfigurable computing systems. Classification of different reconfigurable architectures was given, as well as the distinction of fine and coarse grain architectures. Particular specifics of CPLD and FPGA devices and their latest features and their benefits for various applications were described. This overview continued the introduction of temporal and spatial partitioning concepts, since the work in this thesis is focused on the temporal partitioning systems. The Notion of Macro-operator processing was introduced and explained. Further chapters will provide the in-depth explanation of the Macro-Operator (MO) creation, as well as hardware architecture requirements for the support of this methodology.

Chapter 3

Architectural Organization of Temporal Partitioning Mechanism

3.1 Introduction

In recent years there were several developments made in the field of temporal partitioning of FPGA resources. A large portion of the research [24, 28, 46, 85] describes the analysis of task segmentation. However, not many works focused on the architecture organization of temporal partitioning mechanisms and hardware support. The architecture organization for temporal partitioning mechanism involves three main components: configuration controller, configuration memory, and configuration memory manager.

This chapter is organized as follows: Section 3.2 covers different approaches in the field of architectural organization for temporal partitioning systems. It covers the aspects of the architecture organization and approaches that already tackled that issue. The rest of the chapter provides an analysis of the main sources of the timing overhead, a qualitative approach in estimation of required resources and the trade-offs in their selection. This chapter gives the timing measurements per device and proposes an organization of the temporal partitioning mechanism of FPGA resources (TPM). The architectural solutions that minimize or eliminate

the sources of timing overhead are discussed. Secondly, an architecture that is required for support of temporal partitioning is proposed as a possible solution for minimization of the hardware overhead. A trade-off scheme for estimation of performance and cost of the system is discussed. Temporal partitioning gives a user a capability of having cost-effective solution, where performance trade-off and cost is balanced.

3.2 Literature Review

Temporal partitioning techniques appeared in the past decade with the appearance of reconfigurable devices that supported run-time reconfiguration. As defined in Chapter 2, temporal partitioning assumes a division of tasks into segments to be executed on the computing system that are configured one after the other. A processing system, where processor architecture reconfigures during the task-execution, is called a run-time-reconfigurable computing system (RTR CS). In the past decade many research groups worked on the architectural and task partitioning solutions [107, 48, 32, 43, 29, 10] and several approaches were proposed and developed. These solutions can be classified as: i) approaches focused on the optimization of task/algorithm segmentation; ii) approaches focused on the architectural support of Temporal Partitioning Mechanism (TPM). In this section the most interesting architectures that have been developed are described along with their pros and cons.

Some research works concentrated on tools that can optimize task partitioning for applications such as Multimedia (e.g. [25, 12, 95]), DSP and Digital Communication (e.g. [5]). Others [25, 101, 108, 45] explored the theories behind the tasks segmentation, as well as their use in different types of applications. Few researchers implemented the experimental systems that used the temporal partitioning for the computation [41, 35], but utilized coarse grain architectures. Consequently, they had much lower algorithm-to-architecture tuning capabilities. Several works discussed specific architectural details of FPGA based platforms exploiting temporal partitioning, although emphasis was placed on software segmentation of

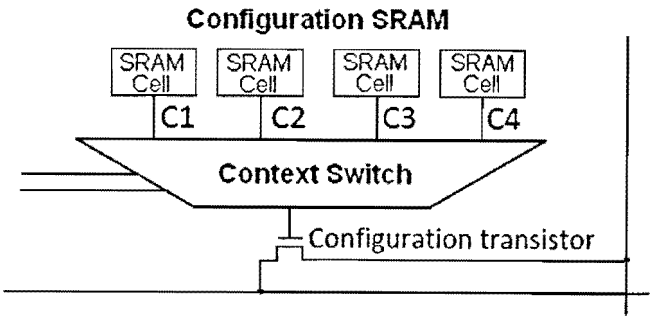


Figure 3.1: Diagram of Multi-Context FPGA Configuration Cell 3.1 [40]

tasks for later scheduling and processing [76]. Recently, there was a publication discussing the technique of reducing the configuration time of the platform, however it did not analyze the overall cost-effectiveness of the temporal partitioning. Overview of the architectures of existing systems is organized in three subsections: Context switch FPGA architectures; and partially reconfigurable FPGA architectures utilizing on-chip soft-core (e.g. MicroBlaze) controller; Partially reconfigurable FPGAs with external custom configuration controllers.

3.2.1 Context Switch FPGA Architectures

One of the directions that was pursued by many researchers is context switching (e.g. [37, 79]). It involves several configuration SRAM cells and ability to switch between the configurations by selecting the corresponding configuration bit. There were several different approaches taken in the area of context switching. However, all of them require an additional configuration SRAM cells and an associated multiplexer, as shown in Figure 3.1 [40].

The additional memory essentially increases the size of the FPGA die. It doubles the address and data buses, but requires highly parallel connectivity between the configuration SRAM cells that can to switch simultaneously from one configuration to the other. Highly parallel connectivity in turn increases the complexity of routing and adds the memory control circuit mechanism which tremendously increases the cost of the FPGA. Also context switching FPGA is the limited number of possible context switches by particular FPGA device. The advantage of the context switch FPGA is a very short transition time from one configuration

to another. Switch occurs within one or two clock cycles [97]. One more significant problem that arises from the implementation of additional configuration memory plane on an FPGA device is the increased power consumption [68, 18]. This is due to the fact that multiple memory cells are drawing static power but are not providing any functionality at that moment. There were several different approaches to mitigate the problem of power consumption and the area increase by using Floating-Gate-MOS Functional Pass-Gate [40], as well as Decoder-Based Multi-context interconnect structure [68]. However, these solutions are still limited to only few possible contexts and are appropriate only for applications with limited configuration variations. The multi-context FPGAs that support 2-4 configurations are suitable for applications that need to switch between existing configurations and do not involve the uploading of configurations during the operation. In addition, these types of FPGAs are best suited for small size applications, because larger multi-context FPGAs are significantly more expensive [111]. Therefore, context switching is not cost-efficient in the solutions that are used for industrial applications and where cost is critical. The external configuration controller of the multi-context FPGA is not discussed in detail by [97, 68, 18]. However, it is assumed to be a conventional configuration controller used in most of the FPGA devices. On the other hand, the internal configuration controller can be considered as an array of multiplexers that perform a selection of the configuration IP-core from the available SRAM cells. Configuration memory management also was not addressed in the [97, 68, 18] papers regarding the multi-context FPGAs, and therefore, cannot be evaluated in detail in this overview.

Hence the above analysis brings us to the realization that cost-effective solution can be found by utilization of readily available FPGA devices. These devices are relatively low in cost due to their mass production. In this case there are two main approaches used by the researchers in temporal partitioning system. They involve using an off-the-shelf FPGA and an external or internal configuration controller. The following sections talk about the schemes

typically employed by researchers utilizing temporal partitioning technique, and analyze the pros and cons of these schemes.

3.2.2 Partially Reconfigurable FPGAs Architectures that Utilize an Internal Soft-core Controller.

When considering using the conventional FPGAs in the temporal partitioning approach we assume that the processing module is fully reconfigurable. In most real-time video-processing applications such as: multimedia, machine vision, automotive, security, tele-medicine and others, at least three main operations are required. These operations are: capturing of the raw video-frames, the video-stream processing based on a particular algorithm, and outputting the results in some manner that is custom to the application (e.g. coordinates, intensity, converted/transformed image as a real-time video output, etc.).

The researchers have approached the architecture of temporal partitioning platforms from different directions. One of the proposed architectures considers using a Xilinx MicroBlaze soft-core processor [37]. MicroBlaze microprocessor acts as a configuration controller for the system. Figure 3.2 shows how the system's architecture is designed. An FPGA device is initially loaded with the configuration bitstream that includes architecture of MicroBlaze microprocessor. The initial configuration might contain no processing modules or I/O IP-cores in the Dynamic Reconfiguration Area. The consecutive processing IP-cores would be loaded into the specific dynamic reconfiguration area. The MicroBlaze performs that actual operation of loading the IP-cores from the configuration storage memory to the designated area. In this case configuration storage memory is an IP-core library which typically is a non-volatile memory. The configuration memory is external to the reconfigurable device and can be realized in a number of possible architectures, that are discussed and analyzed later in the chapter. To load the bitstream MicroBlaze uses the Integrated Configuration Access Port (ICAP) [119]. It is important to mention that MicroBlaze microprocessor and all connection

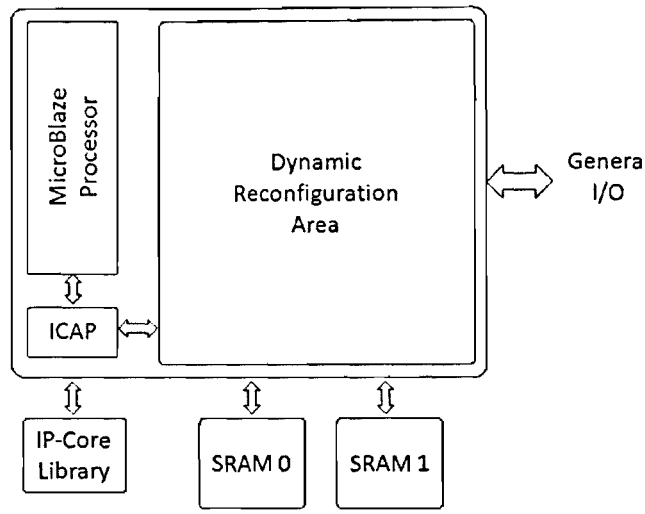


Figure 3.2: Temporal Partitioning System Utilizing MicroBlaze Controller

structure are fixed throughout the operation of the device. At the same time IP-cores are required to be designed in a form of partial bitstreams avoiding the logic area of MicroBlaze microprocessor. External interface between the peripheral devices and the IP-cores is done over bus macros and can operate without involving of MicroBlaze microprocessor.

The advantage of using a MicroBlaze microprocessor is the ease of the system implementation since it is included in the Xilinx EDK/ISE CAD tool [114]. In addition, a MicroBlaze microprocessor has pre-built interfaces from the EDK package to peripheral devices such as memory controllers, UART and Ethernet, CAN [78]. Therefore, integrating these common interfaces into a MicroBlaze soft-core processor does not involve significant engineering time investment. No special external hardware is needed for supporting the reconfiguration of the FPGA except an external FLASH which stores the IP-core library. It is easier to incorporate the interfaces in the design than to design and integrate them from scratch. Hence, a MicroBlaze processor can also act as a configuration memory manager which can perform the communication to the outside world through the available interfaces. This implementation of the memory manager is capable of updating IP-cores, modifying configuration schedule, and much more. In one of the papers [7], a MicroBlaze microprocessor is used in the CAN interface in attempt to replace a significant number of the static microprocessors present in the

conventional car. So, instead of running all of the microprocessors at the same time even when they are not used, a MicroBlaze microprocessor loads the required processor on demand. The dynamic reconfiguration area shown in Figure 3.2 is subdivided further into several processors, so and not one but four or more processors can be loaded on demand. Based on the current conditions in the system, the requests are prioritized and lower priority processor request are put in a queue [7].

However, the solution from [7] has disadvantages as well. First, the main drawback is the FPGA fabric requirement. Because MicroBlaze is a generic soft-core processor implemented in FPGA logic resources, it utilizes a significant amount of the valuable logic space of the FPGA. In turn, by considering the cost of the required FPGA fabric for implementation soft-controller translates into much higher overall costs than hard-core external processor if to be used as a reconfiguration controller. At the same time, the creation of the IP-cores is much more sophisticated, since they have to be designed as partial configuration bitstreams. Thus, this will be possible only for latest Virtex Family FPGAs, since only these FPGAs support partial configuration. Secondly, the soft-core MicroBlaze microprocessor is a sequential processing element and operates on a much slower speed than a hard-core version of a compatible microprocessor, therefore it cannot accommodate a rapid IP-core reconfiguration. MicroBlaze can operate at the top speed of 200MHz and requires three cycles per instruction, so the memory accesses can take up to 7 cycles [98]. Therefore, the actual speed of operation is less than 30MHz which is 3.3 times slower than the maximum speed of an ICAP operation. This introduces an additional delay since it cannot reconfigure the partial IP-core modules at the maximum speed supported by ICAP. At last, the area taken by the MicroBlaze microprocessor, Bus Macros and other servicing logic counts up to 10-20% of the FPGA and cannot be utilized [46] as shown in Figure 3.3. Consequently the cost of the system is increased since a larger FPGA has to be used.

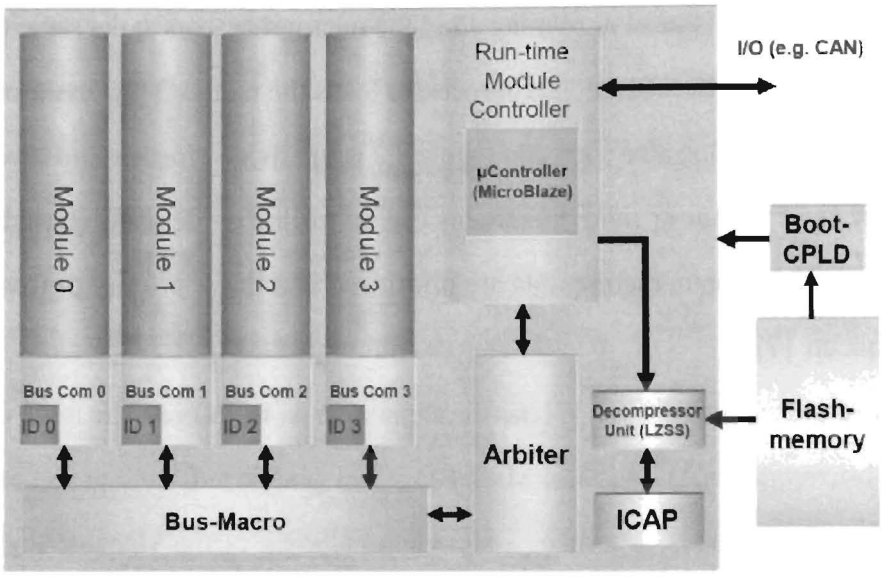


Figure 3.3: Bus Macros Based Temporal Partitioning Architecture [46]

A slightly different path was taken by research groups which implemented custom soft-core reconfiguration controller instead of using the MicroBlaze microprocessor, which is explained in the next section.

3.2.3 Partially Reconfigurable FPGAs with Custom Configuration Controller

Often a custom configuration controller is designed for the temporal partitioning architecture. As in the Section 3.2.2, the custom configuration controller is loaded at the start-up stage of operation and remains for the rest of the operation. A section of FPGA device is dedicated as a dynamically reconfigurable area. This dynamically reconfigurable area is used for partial IP-cores that are loaded into FPGA throughout the system's operation. This approach was used by several research groups [1, 81, 99] and described in their publications. However, these papers considered partially reconfigurable FPGA from Xilinx Inc. and used other methods of loading the partial configuration bitstreams. These methods use internal configuration controller ICAP [99]. In some cases the reconfiguration architecture includes external loaders based on CPLD

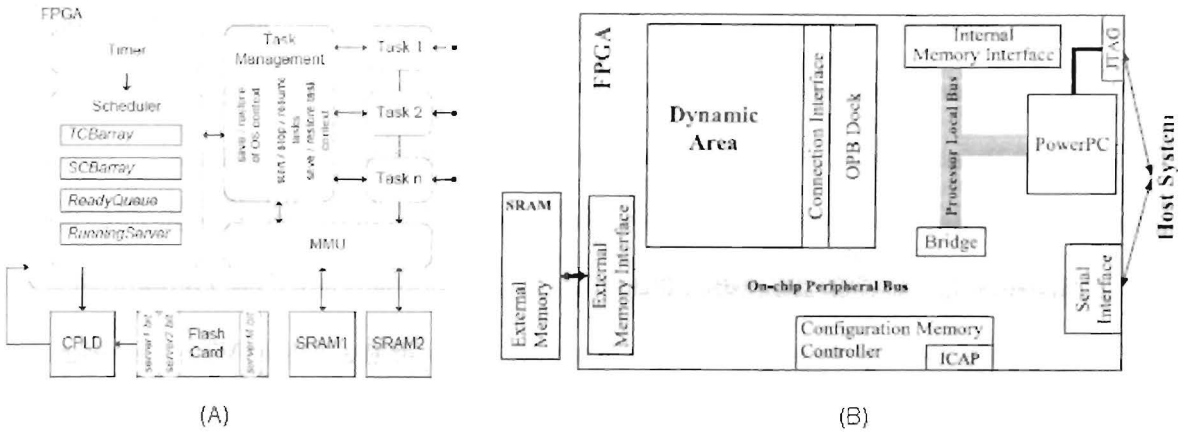


Figure 3.4: Examples of Temporal Partitioning Architectures with Custom Embedded Controller [1, 81, 99]

and FLASH memory combination [1]. A custom controller performs the operations such as scheduling and timing of reconfiguration, task management, and communication with the external memory [1]. These configurations are shown in the Figure 3.4.

In the Figure 3.4 one can see that similarly to the organization described in the previous section, a custom *Configuration Memory Controller* is implemented in the reconfigurable fabric. The dynamic reconfigurable area is also used for the IP-cores. Similarly, as in the previous section, these IP-cores are partial bitstreams and are specifically designed with constraints for a particular FPGA device. The configuration memory is assumed to be an external non-volatile memory, but, it is not discussed in detail in these works [1, 81, 99]. The memory management is performed by the same custom controller, and for this the design of interfaces is needed. In the Figure 3.4 a configuration controller is implemented externally in a form of a CPLD device. The configuration memory for all of the IP-core bitstreams is stored on FLASH memory and is interfaced to the CPLD only. The CPLD provides the bitstream to the FPGA based on the requests from the *Scheduler*. The configuration memory manager in this architecture is implemented as a custom controller, and performs all of the operations needed for IP-core management. In this scenario the IP-core replacement is done by the CPLD device. Another option to update the IP-cores is to replace the FLASH card. In general, such

physical intervention is not desirable since typically embedded processing system is not easily accessible after deployed in the field.

For both of these architectures temporal data memory usually is considered as an external SRAM. Depending on the application the single or dual bank configuration is used for the storage of intermediate and final results of TPM processing. The controllers communicate with the integrated PowerPC cores that are embedded in some of the Xilinx Virtex FPGAs [99]. However, as with the MicroBlaze soft-core processor, the custom soft-core controllers occupy FPGA's logic resources. Most importantly, they are again limited to the FPGA devices that support partial configuration. Due to that fact, the design of the IP-cores is more complex and limited in implementation t. In order to overcome the limitations and create a generic support for all types of FPGA devices the IP-core configuration memory manager and configuration controller should be located outside of the FPGA.

3.3 Architectural Organization of Temporal Partitioning Mechanism

As mentioned in the introduction and literature review sections, all of the major components such as: i) configuration memory manager, ii) configuration controller, iii) configuration memory have to be analyzed and implemented for the TPM architecture. It is preferable that the configuration controller along with the configuration memory manager is implemented as an external device since both are relatively cheaper than an on-chip FPGA resources (e.g ~ $30). This allows for the implementation of the TPM architecture on any off-the-shelf (OTS) reconfigurable device, without being tied to a specific family or manufacturer of reconfigurable devices. In this section every component of the TPM architecture is discussed along with quantitative analysis and possible architecture options.

Table 3.1: Memory Types Chart and their Characteristics

Memory Type	Bandwidth	Read Latency	Capacity (MB)	Control Complexity
Hard Disk Drive	3×10^9 Bits/sec	~10 ms	1.2×10^{13} Bits	High
NOR FLASH	8×10^8 Bits/sec	20 ns	51.2×10^7 Bits	Low
NAND FLASH	3.6×10^8 Bits/sec	1.5 ms	51.2×10^{10} Bits	Medium
DRAM	12.8×10^9 Bits/sec	20 ns	1.2×10^{13} Bits	High
SRAM	6.4×10^9 Bits/sec	7 ns	72×10^6 Bits	Low

3.3.1 IP-core library - Configuration Memory Architectures and Performance Analysis

The configuration memory in temporal partitioning architecture essentially represents the IP-core library which contains vast amounts of different variations of IP-cores. This library should have capabilities of immediate on-the-fly access to the stored IP-cores in order to deliver them to the target reconfigurable device. This library should also be updatable and therefore it should be on an erasable/rewritable type memory. When considering the configuration memory interfaces one should assume that the configuration cores can be stored on any type of memory. Memory could be volatile, like DRAM and SRAM or non-volatile, such as hard drive, FLASH memory, or even a network device. The difference between the memories lies not only in the type of memory and its maximum speed, but also in the initial access latency, its maximum capacity, and control complexity. Differences can be found even within the same type of memories, for example, NAND and NOR FLASH memories. Based on similar technologies these memories differ in that the NAND FLASH can have large capacity, whereas NOR FLASH has very fast access time. The choice of the memory mainly depends on the system bandwidth requirements of configuration interface. The chart of different memory solutions is given in Table 3.1.

Every memory type in the above table has some aspects that are desirable for various temporal partitioning architectures. As seen in the table, there are several parameters that are important for the target architecture. The hard disk drive (HDD) has the largest capacity, as

well as relatively high bandwidth, however, the initial latency and complexity of interfacing are the main problematic issues. Typically, a hard drive requires a microprocessor or a specialized controller to utilize the full speed of the interface, therefore the implementation of it in an embedded system is considered of a high complexity. DRAM is also of a large capacity, but, similarly, requires a relatively complicated controller, and in addition, it is a volatile memory. SRAM, on the other hand, is very simple in control, but it is also a volatile memory that has a limited capacity. NOR FLASH memory is a non-volatile memory, with relatively simple control which is similar to that of the SRAM, and with very low latency. The main issue with NOR FLASH is its capacity which is currently limited to 512 MBits per single die. NAND FLASH has comparable bandwidth to the NOR FLASH, and an ever growing memory density, but it suffers from the initial readout latency.

From the above findings we can observe that each type of memory suites different type of requirements, that can depend on the environment for which the system architecture is designed. The purpose of a cost-effective design is to select a memory that is closely applicable to the architectural requirements. At the same time, the design complexity and the future migration to other designs has to be also considered as the part of the overall cost system evaluation. Hence, a if system is designed to be interfaced to a PC and the mode change can be done slowly, a hard drive IP-core storage system would be most suited. On the other hand, if the system has to be highly optimized in terms of power and speed with large storage capacity, a NAND FLASH with pre-fetch SRAM memory would be more effective.

When a memory is selected, the memory management has to be considered. The memory manager has to be present to perform loading and updating of IP-cores to a non-volatile memory. The memory manager is discussed in the following section.

3.3.2 Architecture of Configuration Memory Manager

The configuration memory manager performs the following functions: schedules the reconfigurations, updates IP-core in the IP-core library, communicates with the external devices (e.g. host computer, network server, etc..) and, in most of cases, initiates the reconfigurations.

The configuration memory manager can be implemented in various ways. For example: purely software running on a host PC, soft-core manager as described in the [1, 81, 99], as external microprocessor/microcontroller. Configuration memory manager can have different types of interfaces to the outside world. Interfaces can be as simple as a serial communication, or as complex as Ethernet or satellite interfaces with a connectivity to an IP-core library on a remote server. There is also a possibility of hardware-software co-design where some aspects of the configuration manager are implemented in hardware, such as interfacing, and some in software, such as the order of the IP-core prediction. The complexity of the actual configuration memory manager also varies with the system requirements. In some cases configuration manager needs to cycle only through few IP-cores and, therefore, can be implemented even on a small size logic device. When requirement is for a system that has to have various interfaces and performs complex memory management operations, a much more complex processing device has to be used. Moreover, the development of the firmware for that device is complicated, and, as a result, the overall cost increases. Hence, the configuration memory manager should closely reflect the overall system requirements.

Proposed configuration memory manager is flexible in supporting both temporal and spatial partitioning. The main difference of a partial bitstream from a full configuration bitstream is in size. The configuration procedures are virtually identical. The partial configuration does not require a reset of FPGA configuration memory, which, in most cases, is an assertion of a reset input. The header information is included in all of the bitstreams that are produced by CAD software, therefore configuration memory manager is greatly simplified. Since the configuration memory manager performs the actual update of the IP-core library it also can

perform the memory use optimization, by reorganizing the IP-cores to utilize as much memory as possible. Keeping the database of the IP-cores simplifies the user configuration side interface and makes it portable and independent of the specific user. This allows to change the schedule of the temporal or spatial reconfiguration without any knowledge of where exactly the particular IP-core is located in the storage memory and what size it is.

As was seen in the previously reviewed works [1, 81, 99, 46], the configuration memory manager was mostly present as a soft-core microprocessor on the FPGA fabric or was not discussed at all. The main reason for integrating the configuration manager inside the FPGA device was to avoid the use of an external controller and minimize the complexity of the configuration memory manager by utilizing pre-built soft-core processor. However, as was explained, the soft-core controller occupies valuable logic resources on the FPGA device, complicates the IP-core design, but is supported by FPGA vendors. Design of the configuration memory manager is also possible in other types of architectures, however as in the Section 3.3.1, the design closely depends on the overall system requirement. Several configuration memory manager architectures are given below in the order of increasing bandwidth and complexity below:

- Software GUI with JTAG configuration interface
- Low cost microcontroller with 1 or 8 bit configuration interface
- Combination of a microcontroller with a high speed IP-core loader controller based on a CPLD/FPGA/ASIC
- Microprocessor with a high speed IP-core loader interface and various complex communication interfaces (e.g. Ethernet)

It is important to mention that the configuration memory manager does not necessarily include the configuration controller. In some cases to increase the bandwidth of the configuration bus an additional driver is added to achieve the desired configuration speed.

3.3.3 Configuration Controller Architecture

The purpose of a configuration controller is to perform the loading of the configuration bitstream from the IP-core library to the target reconfigurable device. There are various configuration controller interfaces that can be employed and they can be as simple as a serial interface running at low interface (e.g. KHz range) frequency, and as complex as a highly parallel and high frequency (e.g. 32 bit running at 100MHz). The IP-core storage memory has several options as well: integrated into configuration controller (e.g. Xilinx PROM), external memory module (e.g. FLASH memory IC), or removable (e.g. CF card, HDD, network) device. The configuration controller typically has an interface to the configuration memory manager, which issues commands to the configuration controller. Typical commands for the configuration memory manager are: load IP-core from a particular slot/address, perform maintenance on a particular slot/address. Maintenance can include erasing a particular slot/address on the storage memory, as well as, write an incoming stream of data to a slot/address. The interfaces of the configuration controller to the target FPGA device vary in bandwidth. It's important to state that the performance of the external configuration controller is always designed to match the FPGA's maximum configuration bandwidth. Hence, the external configuration controller always operates with the maximum required performance and is in no way lower in performance than previously described soft-controllers [46, 1, 99]. Therefore, based on the system architecture requirements an appropriate selection has to be made. The next section talks about the comparison of configuration interfaces and about the actual steps of FPGA reconfiguration.

3.3.4 Configuration Interfaces Selection

As mentioned earlier, specifications of the desired system dictate the complexity of the design. Depending on these requirements a temporal partitioning mechanism can be designed with various configuration interfaces that have different configuration bandwidths. One of the

drawbacks of the temporal partitioning approach is the timing overhead that is introduced by the reconfiguration time of an FPGA device. For high performance stream processing systems such overhead becomes a significant problem and might result in a violation of the timing restrictions. Hence, one of the objectives of the research was to minimize the configuration timing overhead. In order to do that we have to analyze what is involved in the calculation of reconfiguration time.

When we are considering minimization of configuration time overhead we first have to describe general interfaces that are available for FPGA configuration. Most of the FPGA manufacturers support several common types of configuration interfaces that can be utilized by designers for particular applications. Based on the speed of these interfaces the calculation of configuration times can be performed.

Most common interface includes serial configuration over two line interface, where one line provides clock input and the other provides the data. Serial configuration is available in different flavors. The proprietary protocols such as Xilinx Master-Serial and JTAG TAP chain are common to most manufacturers. JTAG TAP chain operates at about 8MHz [34]. Proprietary serial configuration interfaces usually support much higher speed which in case of Xilinx Master-Serial is 25MHz. For fully embedded solutions FPGA manufacturers provide configuration PROMs that can support up to 8 different configuration bitstreams. Recently, configuration PROMs were embedded directly in the FPGA ICs, such as Lattice XP and Xilinx Spartan 3N FPGAs. This decreased the external component count, the cost and the size of the overall system. It also provided an additional security against reverse engineering of IP-cores. The speed of bitstream upload was not improved, however.

As FPGA devices increased in size so did the configuration bitstream, which in turn caused longer configuration latency. Serial configuration would require tens of seconds and thus was unfeasible for embedded products that needed to have reasonably fast start up times. Some manufacturers began to increase the configuration bus width. Most common ones were 8

bit width operating at 50MHz. In case of Xilinx FPGAs this interface is called SelectMAP and is usually interfaced with a microprocessor, or a combination of a microcontroller and CPLD. The speed of this interface was also increased to about 50Mhz to allow rapid power-to-operation start-up times and to cut the start-up times by the factor of 16. In the past couple of years, with the release of latest Xilinx Virtex 4, Virtex 5 and newly announced Virtex 6 FPGAs, this interface was developed even more and bus width was increased to 32 bits, while frequency was pushed to 100MHz of configuration clock. This fact benefited the approach of temporal partitioning tremendously, since it has greatly reduced the configuration time of FPGA, which, as mentioned before, is the main bottleneck in the implementation of temporal partitioning.

From the above described interfaces we can arrive at a general formula for calculation of bitstream upload time:

$$T_{config_{bitstream}} = \frac{S}{W \times F} = \frac{S}{BW_{config}} \tag{3.1}$$

where W is the bit width of the configuration bus, F is the frequency of the configuration clock that is used to clock-in configuration data which results in configuration bandwidth, BW_{config} is the bandwidth of the configuration interface. Assuming S is the size of a bitstream in bits, the time it takes to upload the actual bitstream information $T_{config_{bitstream}}$, is directly related to the configuration bandwidth.

Table 3.2 shows the comparison between time requirements for configuration of different interfaces for smallest FPGA in Virtex-4 Family XC4VLX15 which has bitstream size of 4,765,568 bits.

It is clear from Table 3.2 that the configuration interface plays a crucial role in the reduction of configuration time overhead. High clocking speed and highly parallel interface, such as SelectMAP32, gives 400 fold time reduction in comparison to a conventional JTAG

Table 3.2: Bitstream Configuration Timings for Various FPGA Interfaces.

Interface	JTAG at 8MHz	Serial at 25MHz	Parallel SelectMAP (8bit) at 50MHz	Parallel SelectMAP32 (32bit) at 100MHz
Time Required	595.69ms	190.62ms	11.91ms	1.49ms

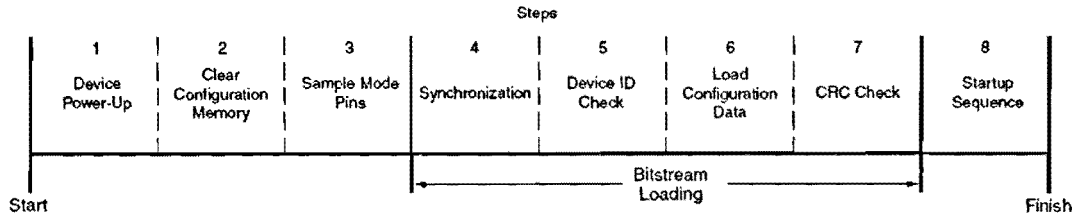


Figure 3.5: FPGA Re-configuration Stages [83]

interface. Therefore, when considering configuration interface in architectures that employ temporal partitioning, the largest configuration bus width, and the highest configuration speed is preferred to minimize the bitstream configuration overhead.

At the same time, it is important to note that T_{config} consists of several latencies and not just $T_{config_{bitstream}}$. Other time overheads have to be included in calculation of T_{config}. These overheads are explained in the next section.

3.3.5 Calculation of FPGA Reconfiguration Time

When calculation time involved in an FPGA device reconfiguration we have to consider the time for 8 latencies, as shown in the Figure 3.5.

The power up latency can be avoided since system is considered to be powered up already. Nonetheless, it has been included as first latency and counted as $T_{config_{power-up}}$ for the sake of completeness. The next two steps are associated with the reset of configuration memory and FPGA responding with being ready to accept a new bitstream. PROGRAM signal has to be asserted for at least $T_{config_{memreset}}$ (e.g. Xilinx FPGAs 300ns) in order to initiate the FPGA configuration memory reset [83]. Upon completion of the reset, the INIT signal will

be asserted to high with a typical time latency of $T_{config_{init\,latency}}$ (e.g. Xilinx FPGAs $300\mu s$) and the bitstream upload takes place.

Step 4 through step 7 take care of the bitstream configuration. The bitstream itself consists of several sections, such as: synchronization, device ID check, CRC check, and the actual configuration data upload. The procedure is fairly similar for all types of SRAM based FPGAs. $T_{config_{bitstream}}$ varies greatly depending on the size of an FPGA. As can be seen in example of Xilinx Virtex 4, the series different devices in the same family vary greatly in configuration data size, which reflects its logic size. Therefore, for a large device, such as XC4VLX200, the bitstream is about 11 times larger than for XC4VLX15. We have to also consider the fact that if a bitstream compression is utilized, then the size of a bitstream can be reduced. The smaller bitstream size in turn decreases the time of bitstream upload. Upon completion of the bitstream upload there is typically a delay of several clock cycles for FPGA start-up, but it is negligible compared to all of the other steps in bitstream configuration (e.g. Xilinx Virtex $8cc \times \frac{1sec}{100MHz} = 80ns$). In the Table 3.3 the reconfiguration times for Virtex 4 FPGA devices are listed along with the total configuration FPGA off-line time which in this thesis is given as:

$$T_{config} = T_{config_{power}} + T_{config_{memreset}} + T_{config_{init\,latency}} + T_{config_{bitstream}} \tag{3.2}$$

This formula includes all the time requirements described previously. As it can be seen from the Table 3.3, the reconfiguration times increase almost linearly with the size of the FPGA device. For the temporal partitioning it has to be decided which FPGA is most suitable based on how much configuration overhead can be tolerated. In the later section a selection scheme will be presented on how FPGAs are chosen for a particular temporal partitioning implementation.

Table 3.3: Reconfiguration Time for the Xilinx Virtex 4 FPGA Devices via Slave Serial SelectMAP32 Mode.

Devices	Bitstream Size in bits	Bitstream Configuration	Total FPGA configuration
XC4VLX15	4,765,568	1.489*ms*	1.789*ms*
XC4VLX25	7,819,904	2.444*ms*	2.744*ms*
XC4VLX40	12,259,712	3.831*ms*	4.131*ms*
XC4VLX60	17,717,632	5.537*ms*	5.837*ms*
XC4VLX80	23,291,008	7.278*ms*	7.578*ms*
XC4VLX100	30,711,680	9.597*ms*	9.897*ms*
XC4VLX160	40,347,008	12.608*ms*	12.908*ms*
XC4VLX200	51,367,808	16.052*ms*	16.352*ms*

3.3.6 Proposed Platform Architecture with Custom External Controller-Scheduler for Run-Time TPM

Based on the proposed methodologies/approaches of different parts of the TPM in previous sections a proposed methodology of overall architecture for temporal partitioning system is given in this section. This section explains the architecture of configuration memory manager, its configuration memory (IP-core library), and its configuration controller. First, however, system requirements have to be stated since architecture of the TPM depends them. For this work four main specifications are considered:

1. The fastest possible FPGA reconfiguration to minimize the downtime.
2. An interface to a PC for user interaction with the configuration manager
3. An ability of the IP-core library to be updated from a PC (e.g. Ethernet bridge, Bluetooth bridge, serial port, etc.)
4. An ability to issue reconfiguration commands from 3 sources:
   (a) FPGA device
   (b) GUI software
   (c) Physical input button

Based on these initial specification requirements the TPM architecture design was proposed for the target platform described in this work [58]. Overall architecture is shown in Figure 3.6.

The external configuration memory manager is designed by utilizing a microcontroller that interfaces with a IP-core library and configuration manager. Its main functions are: i) scheduling, ii) updating IP-cores, iii) communication with the instrumental PC over a USB protocol. The microcontroller is a Microchip PIC18F8410 with 8-bit Harvard RISC architecture. It was selected due to requirements of all of the above interfaces and its low cost. The microcontroller is connected to the configuration controller and FPGA with a parallel interface. The purpose of the interface to both the configuration controller and FPGA is to have the flexibility of initiating reconfiguration based on a request from FPGA and to pass command information to the configuration controller. The use of the microcontroller instead of much more powerful microprocessor is to simplify the design, to make it portable and implementable on a much smaller scale platform and on a smaller FPGA. Possible disadvantages are that for support of high speed Ethernet protocol requires much more powerful microprocessor.

The configuration memory was selected to be a non-volatile NOR FLASH type memory connected to the configuration controller only. This solution was selected because the NOR FLASH memory has high pin-count of four FLASH modules and hence provides rapid upload of a bitstream. To achieve the maximum bandwidth of reconfiguration it was needed to connect four modules in parallel. Even though capacity of NOR FLASH is significantly smaller than that of the NAND FLASH, NOR FLASH has the lowest initial latency for a non-volatile memory and has an SRAM-like control architecture. Due to these reasons and considering the scope of the work, NOR FLASH is a suitable type of memory for IP-core library.

The configuration controller delivers configuration bitstream at the maximum possible bandwidth allowed by the reconfigurable device. The high pin-count was needed to interface to four NOR FLASH modules mentioned previously. However, the memory operations

are not complex and, therefore, a CPLD device has enough logic resources for that purpose. Similarly, as with the configuration memory manager, it has a dual parallel bus connected to both FPGA device and configuration manager. Additionally, it has a 32-bit configuration bus to the FPGA device, on which the actual configuration data is delivered. The main advantage of the CPLD device is that it is a configurable logic device based on a non-volatile memory operating from a single power source and that it can operate at high frequencies. This allows for easily parallel implementation of configuration controller which can be highly portable across different architectures and scalable for higher speed.

The great advantage of such implementation is an ability of designing platforms with any type of FPGA device, independent of the partial configuration capability. More importantly, temporal partitioning system can be designed for any FPGA device, which is controlled and configured by the central scheduler. This also simplifies the generation of IP-cores since they are designed with full use of resources and are not limited by area constraints. Since there are no embedded controllers/schedulers on target FPGA, the design can be done without any specific considerations.

There are different approaches that can be employed in design of a system with temporal partitioning methodology using external configuration controller. Some can be a single-FPGA and some can be multi-FPGA solutions. Figure 3.6 shows proposed general organization of a system with single FPGA, which is similar to [46, 85, 107].

In this setup the configuration memory manager is connected to the configuration controller, which in turn is connected with the IP-core library. Configuration manager has interfaces to the outside world which gives it capability to upload or update IP-cores. At the same time, it is possible to change parameters of the configuration manager to adapt to different environment conditions in real-time, as will be described in Section 8.2. The configuration controller design has several variants of implementation and they are mentioned in Sections

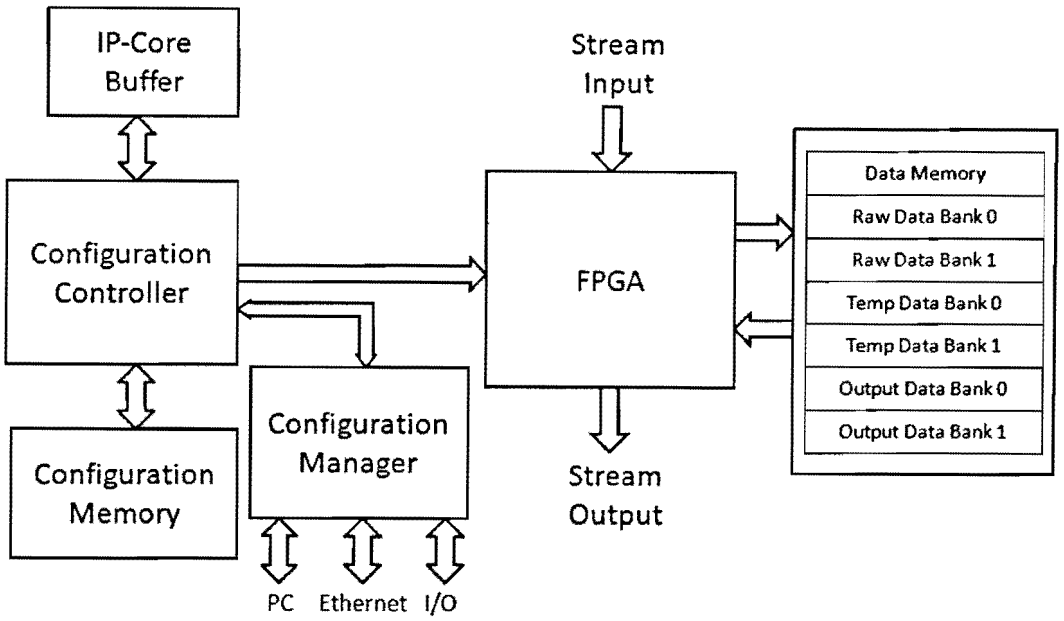


Figure 3.6: Architecture of Temporal Partitioning Platform with External CPLD Based Controller/Scheduler

3.3.7 and 3.3.7.1. As for the scenario of the system with a single FPGA device implementation, the IP-cores are required to be designed to include the peripheral device control (such as control of SRAM) as shown in the Figure 3.6. Even though this is an acceptable solution from the stand point of overall cost, and resource utilization, it requires external hardware resources which have to perform the service operations. These resources would have to be designed and integrated into the target architecture and would require custom hardware.

In the multi-FPGA setup which is shown in Figure 3.7, three FPGA units are used. This architecture is general for any types of FPGA devices. This architecture involves two processing FPGAs: one or many is/are dynamically reconfigured and one statically configured FPGA. The statically configured FPGA device manages global inputs/outputs, as well as acts as an interface to external memory modules. This is especially necessary for applications that require un-interrupted control, such as video output, or constant control output. This statically configured FPGA, as shown in Figure 3.7, also acts as a bridge between the processing FPGAs and memory modules. At the same time, requirements for this FPGA are minimal and

have to satisfy the bare minimum of interfacing. The other two FPGAs are run-time reconfigurable task Segment Processing Modules SPM1 and SPM2. The IP-cores associated with task segments are loaded there one after another according to the schedule by the configuration loader/scheduler. One of SPMs is configured with next IP-core while the second performs the processing of the current IP-core. This allows for elimination or reduction of reconfiguration overhead and is described in detail in Section 3.3.8. As in the previous scenario, microprocessor performs communication functions with other interfaces for IP-core update, schedule modification and other maintenance tasks. In a scenario where three FPGAs are used the pipelining technique for configuration/execution of IP-cores is employed. Here, reconfiguration of one FPGA occurs when the other FPGA is preforming data processing and vice versa. There are different approaches that were implemented by other researchers, such as RACE [88], Firefly, RENCO [94] and other architectures that are similar to this approach in using four to eight FPGAs. However, in the architectures approaches a separate PC was used to perform all of the scheduling and loading in run-time. This dramatically increases the cost of the system and may eliminate the advantage of cost-efficiency of TPM. Therefore, the only solution with one FPGA (non-pipelined) and three FPGA (pipelined) devices were considered in further research.

3.3.7 Configuration Controller Architecture with IP-core Pre-fetching

One of the design architectures of configuration controller could be simplified and made more cost-effective if the frequency of reconfigurations is lowered. Even if a bitstream has to be delivered to the target device at the maximum configuration bandwidth, with lower frequency all highly paralleled FLASH modules can be replaced with a single SRAM buffer. It is important to mention that over time the average bandwidth of IP-core to FPGA configuration should be same or lower than bandwidth FLASH memory. Otherwise, IP-core library may become a

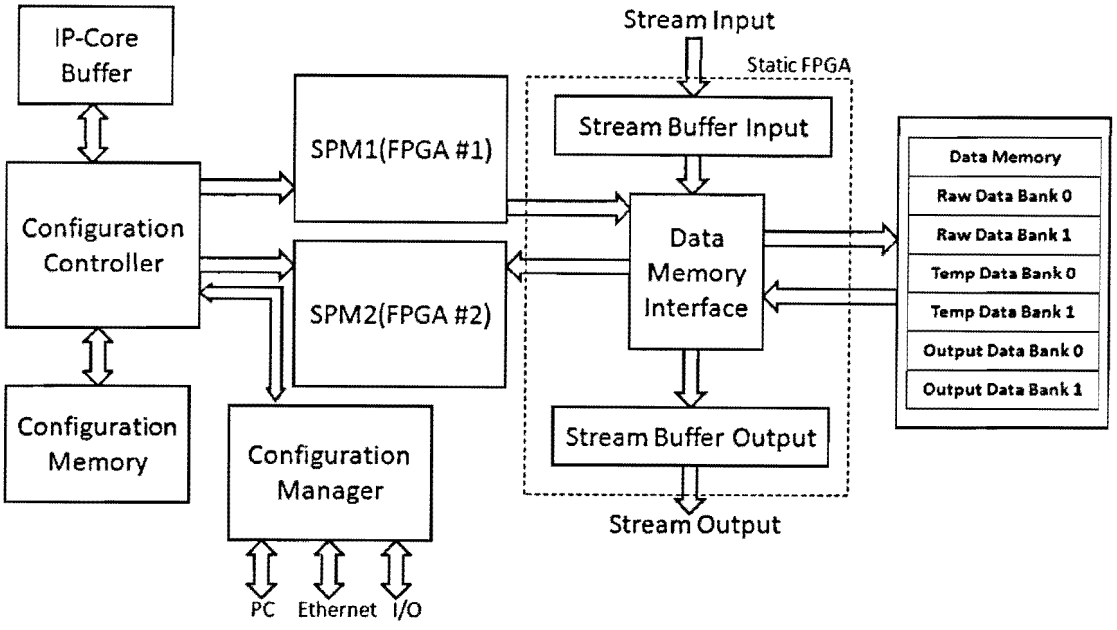


Figure 3.7: Architecture of Temporal Partitioning Platform with Pipelined Organization of Configuration/Execution Processes

bottleneck in the TPM design:

$$BW_{FLASH} \geq BW_{average\,config} \tag{3.3}$$

As in previous designs the configuration manager is realized as a microcontroller/microprocessor unit, which performs operation of IP-core management by uploading them from external sources such as Internet and others. In addition, it keeps the schedule of reloading IP-cores and performs the actual reconfiguration of the FPGA with the scheduled IP-core. This configuration controller architecture involves a FLASH memory for the storage of IP-cores [85]. It is well known that the readout speed of a FLASH memory module is limited to about $25ns$ per memory access. Due to that, configuration itself will result in extra overhead of latency. The current maximum configuration bus speed is 100MHz for Virtex 4, Virtex 5 and Virtex 6 devices. Considering that the SelectMAP32 configuration interface for the Xilinx FPGAs is currently 32 bits, it is required to deliver bitstream data at 3.2Gbit

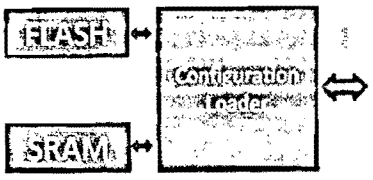


Figure 3.8: Architecture of SRAM and FLASH Configuration Loader

per second. As noted previously, a single FLASH chip would be able to deliver only at $\frac{1}{25\times10^{-9}} \times 16bit = 640$Mbit/sec maximum speed, as shown in the Table 3.1. This, however, is not a difficult task for an SRAM memory which easily surpasses 100MHz. SRAM memories are also available in 32 bit interface data buses. Therefore, a design which involves pre-fetch scheduling can effectively decrease the configuration latency on FPGA. With constantly revolving IP-cores it is possible to pre-load required IP-cores into the SRAM buffer. Based on the schedule of configuration configuration controller can deliver the bitstreams at the maximum speed, therefore minimizing the bottleneck of reconfiguration time on a FPGA. Example of the loader architecture arrangement is shown on Figure 3.8.

For partially configurable devices configuration time would be even less since partial bitstream size would be a fraction of a full bitstream. Several pre-fetching techniques have been developed to reduce the overall reconfiguration overhead by performing the scheduling reconfiguration from the SRAM buffer in run-time [91]. In an example of an application where a mid-sized FPGA is used which has a 4Mbit configuration bitstream, and utilizes a single IC 72Mbit SRAM, it is possible to keep 18 temporal IP-cores. These 18 IP-cores can be re-placed on individual basis in a case of operation mode change. Again, this operation is possible if the condition shown in equation 3.3 is satisfied. Overall cost calculation for this approach is given in equation 3.10, which can be used for cost comparison to parallel FLASH approach.

There is also some research [89] on the same topic that concentrates on the increase of the reconfiguration speed even further to minimize the configuration delay. The idea is to use multiple homogeneous sections of FPGA, each with its own configuration SRAM that can be accessed individually. Setup example of this approach is shown in Figure 3.9.

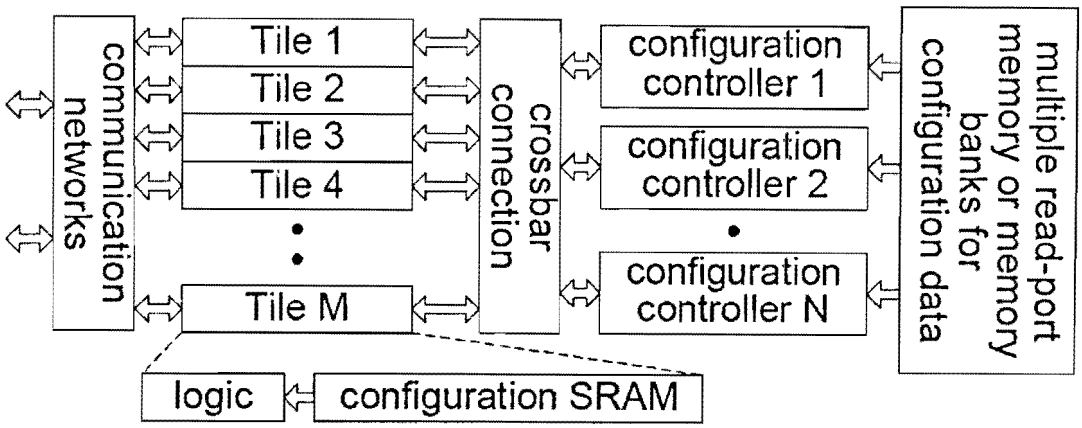


Figure 3.9: Organization of Parallel Tile Configuration [89]

This architecture provides parallel load of the configuration stream, which in turn speeds up the whole system operation. Experimental results showed that by increasing the number of SRAM modules and loading up to ten controllers in a single FPGA, it is possible to achieve a 40% improvement in configuration time. The disadvantage of this approach is that it requires a design of different architecture of FPGA and would increase the cost of FPGA device due to the complication of the configuration logic and SRAM memory.

3.3.7.1 Configuration Controller Architecture with Paralleled FLASH Memory Organization for IP-core Configuration

Another configuration controller architecture solution is to have a parallel FLASH organization. This architecture provides simplified loading mechanism, hardware requirements and implementation. The design also depends on the type of FLASH memory used, such as: NOR or NAND types and its data bus width. In order to design a parallel FLASH loader for any type of FPGA device a configuration bandwidth should be known. Configuration bandwidth can be calculated based on the maximum configuration clock frequency and configuration bus width. FLASH readout bandwidth is calculated in the same manner. Based on the configuration bandwidth of the FPGA's configuration bus and an average FLASH memory readout bandwidth, the number of FLASH ICs can be estimated. These FLASH modules would have to be linked

in parallel with common address/control lines. Equation that calculates the number of FLASH ICs is as follows:

$$N_{FLASH\,IC} = \left\lceil \frac{BW_{config\,bus}}{BW_{FLASH}} \right\rceil \tag{3.4}$$

Such configuration loader is obviously be much simpler in hardware requirements since it only has to buffer the FLASH data and output the configuration data in configuration bus width bits. A bridging device between the FLASH memory modules and the configuration bus could be based on a low cost/low power CPLD/FPGA device. In comparison, designing an IP-core SRAM buffer would require more complex loader, and, thus would result in a much higher cost of implementation. In later chapters a hardware example of such architecture design is given along with the cost analysis for this overhead.

3.3.8 Hardware Overhead Cost in Temporal Partitioning Architecture Implementation

Besides minimizing the configuration time overhead, the configuration hardware overhead cost should be minimized. Since configuration hardware overhead is directly related to the cost-performance ratio, it has to be addressed in order to maintain the effectiveness of the temporal partitioning approach. In this section, several approaches associated with the hardware overhead and their approximate costs, are considered. As mentioned in the previous section, to minimize configuration time overhead we have to use widest possible configuration bus operating at the maximum speed of available configuration clock. Therefore, as seen in previous section, with the current SelectMAP32 interface it is desirable to have a configuration loader that would have the interface and bandwidth close to the configurable device. In case of proposed implementation that would be a 32bit data bus operating at 100MHz totaling to $BW_{config} = 32 \times 100 \times 10^6 = 3.2 \times 10^9$ bit/sec. At this point there are several possible

architectural organization possibilities and associated costs. It is important to note, that configuration controller has to be designed with a cost-effective approach in mind. Hence, if system architecture requirements are lower than the maximum configuration bandwidth, then TPM architecture should reflect that requirement and preferably not exceed it, in order to achieve the optimal cost-effectiveness.

The overhead cost estimation and cost-efficiency can be found for architectures covered in Sections 3.3.6, 3.3.7, 3.3.7.1. As previously discussed, a TPM architecture requires these major components: configuration memory, configuration controller, and configuration memory manager. Each of these parts should have minimal in cost while providing adequate performance. This would decrease overall overhead cost and provide even better cost-performance ratio for the whole system. Therefore, overall TPM overhead cost is calculated by:

$$C_{TPM} = C_{configmemory} + C_{configmemorymanager} + C_{configcontroller} + C_{PCBarea} \tag{3.5}$$

Where C_{TPM} is the cost of TPM hardware, $C_{configmemory}$ is the cost of the configuration memory, $C_{configcontroller}$ is the cost of configuration controller, and $C_{PCBarea}$ is the cost of the additional PCB area that is needed for the TPM hardware components. Every component can have sub-components depending on the architecture. For an architecture with parallel FLASH arrangement every TPM component should be estimated. As per equation 3.4:

$$C_{configmemory} = C_{FLASHIC} \times N_{FLASHICs} = C_{FLASHIC} \times \frac{BW_{configbus}}{BW_{FLASH}} \tag{3.6}$$

In this case the cost of $C_{configcontroller}$ corresponds to the cost of inexpensive reconfigurable device that satisfies the pin-count need for the configuration memory and configuration interface. $C_{FLASHIC}$ is the cost of the FLASH memory IC and $N_{FLASHIC}$ is the number of the FLASH memories needed to achieve the maximum configuration bandwidth. Also configuration controller contains enough configurable logic to support configuration controller

implementation. Typical cost of such devices is around $10-$15 USD in single unit quantity from a wide range of distributors. The cost of configuration memory manager purely depends on the complexity of operation and the number of interfaces required for the configuration memory access from external sources. To provide feedback control memory manager has to have a sufficient I/O for interconnection between configuration controller and reconfigurable device. A typical implementation can be accommodated by a general purpose microcontroller which typically costs <$10 USD. In a much more involved implementation with specific pre-processing/compression implementations or a highly demanding processing task, a microprocessor can be used that costs up to $50 USD. The last overhead cost that is considered for TPM architecture is the overhead cost of additional PCB area needed by the TPM components. It is an important factor which should not be overlooked, due to significant costs of $\sim$ $0.4 USD cm^2 for a 6-8 layer from most PCB manufacturers. In some scenarios due to area PCB constraints and overhead costs an even more expensive memory packages is favored. Hence, a sample formula for parallel FLASH TPM architecture is:

$$C_{TPM} = C_{FLASH\,IC} \times \frac{BW_{config\,bus}}{BW_{FLASH}} + C_{config\,memory\,manager} + C_{config\,controller} + C_{PCB\,area} \quad (3.7)$$

Similarly to previous analysis, an architecture of pre-buffering has similar type of architectural, however, in this scenario C_{memory} is modified to include costs of different type of memories. In this architecture there are other parameters that have to be addressed as well:

$$C_{config\,memory} = C_{FLASH} + C_{SRAM} \quad (3.8)$$

In the pre-buffering configuration bitstream transfers need to be considered. First a bitstream has to be pre-loaded to the pre-fetch SRAM memory and only then it can be programmed into the FPGA. Both of these transfer times have to be considered in the calculation to identify the amount of needed FLASH modules, and the associated costs. As mentioned in

the previous example the bandwidth of FLASH memory has to satisfy an average bandwidth of the bitstream configuration with an addition of SRAM bandwidth as shown in the equation below:

$$C_{FLASH} = C_{FLASH\,IC} \times \left\lceil \frac{F \times S_{bitstream}}{BW_{FLASH}} + \frac{F \times S_{bitstream}}{BW_{config}} \right\rceil \tag{3.9}$$

Here, F is the frequency of IP-core reconfigurations and S is the size of IP-core bitstream. This equation can be used in estimation of feasibility of this architecture for particular application, assuming that the frequency of reconfiguration, as well as, the type of reconfigurable device is known.

The cost of SRAM C_{SRAM} depends on the IP-core bitstream size since it has to accommodate at least one uncompressed IP-core. Hence, overall calculation of configuration memory is:

$$C_{config\,memory} = C_{FLASH\,IC} \times \left\lceil \frac{F \times S_{bitstream}}{BW_{FLASH}} + \frac{F \times S_{bitstream}}{BW_{config}} \right\rceil + C_{SRAM} \tag{3.10}$$

Based on the Equation 3.10, the cost of configuration memory can be estimated. The estimate can be used to access if this architecture would be feasible for a particular application. For example, if the frequency of reconfigurations is 10 reconfigurations per second and the size of IP-core bitstream is about 20Mbits, then using previous parameters for configuration bandwidth, as well as memories bandwidth, we get:

$$C_{config\,memory} = \$15 \times \left\lceil \frac{10 \times 20 \times 10^6}{3200 \times 10^6} + \frac{10 \times 20 \times 10^6}{800 \times 10^6} \right\rceil + \$60 = \$90$$

Component costs were obtained from various distributors (e.g. Digikey, Avnet, Mouser) at the time of writing this chapter.

Besides the fact that from 8 FLASH ICs we have decreased to only 2 FLASH ICs, additional savings also come from the C_{PCB} due to smaller area requirement.

This approach shows that if TPM system requires a minimum downtime, while having average configuration bandwidth lower than maximum system BW_{config}, it is possible to exploit the pre-fetching technique to achieve lower cost of the TPM architecture with additional PCB area reduction. The evaluation of effectiveness of this approach is given by the Equation 3.10 which uses specification parameters of the system and available components.

3.4 Summary

This chapter introduced a hardware architectural organization of RCS with temporal partitioning of computing resources. A thorough literature review of different architectures that were designed for this purpose was presented. This chapter discussed different approaches such as context switching which allows a rapid transition from one configuration to another by SRAM cell multiplexing. It was shown that context switching architecture is limited to a few configurations and is not flexible if the number of IP-cores exceeds the amount of context cells present in a multi-context FPGA. Other reviewed architectures involved a configuration controller. The main differences were in the location and type of the configuration controller. Architectures involved an implementation of the configuration controller within the FPGA fabric in a form of a generic soft-microprocessor or a customized controller. Other researcher works involved an implementation of the controller on an external microprocessor. As the result of this analysis, a novel approach for architecture organization of temporal partitioning mechanism (TPM) was proposed. The proposed approach was described with the specification of architecture and the explanation of why specific type of non-volatile memory has to be used. Finally, the proposed pipelined and non-pipelined system architectures were described with the methodology of configuration controller design. This architecture was published in conference publications [58], [54], and journal publications [51]. Next chapter describes the essential need and suitability of the temporal partitioning approach in fault tolerance designs. It introduces various fault tolerance techniques used in the industry and shows how temporal

partitioning can greatly improve the survivability of devices in hostile environments and at the same time decrease the design costs of the system.

Chapter 4

Architecture to Hardware Faults Adaptation (Self-restoration)

4.1 Introduction

One of the major problems with FPGA based systems is their sensitivity to various radiation effects. This is especially true for SRAM based FPGA devices. During the last decade that was critical only for aerospace applications, nowadays, when $45nm$ CMOS technology became the basis for FPGA production, it is important for many terrestrial applications as well. Effects of the cosmic radiation on the electronics equipment is called Single Event Effect. There are three main types of Single Event Effects (SEEs): Single Event Upset (SEU), Single Event Transient (SET), and Single Event Dielectric Rupture (SEDR) [92]. SEU is the most frequent effect and in most cases recoverable. SEU affects configuration memory cell that keeps the value of a LUT in CLB or the state of interconnection switch transistor. Therefore, the affected circuit may change its functionality. SEU occurs when SRAM cell is struck by a charged particle, it then charges up and flips the value of its cell. SET could happen when a voltage spike occurs, making the circuit produce incorrect result. SET is not as critical as SEU and SEDR since the operation is just briefly altered and the device is not operational

only temporally. The worst case scenario is the SEDR fault, where, after being hit by an ion particle, a dielectric destruction occurs. When dielectric, which isolates two charged conducting surfaces, is destroyed the planes are coupled. This is a hardware fault which can't be repaired and thus poses the biggest threat to the space-borne electronic systems. This type of fault can also occur because of thermal fatigue, oxide breakdown and electron-migration [28]. Due to these effects, the systems that were designed for aerospace industry often employ full or partial triple redundancy. In addition, FPGAs that are used for aerospace applications are radiation hardened to withstand a hit of a charged particle. However, such solutions significantly increase the cost, power parameters, and in some instances prevent the use of FPGA based systems.

All of these SEEs pose a serious problem for use of FPGAs in space applications and there are several approaches that have been taken to mitigate them. In following sections the approaches are described, along with, their pros and cons. In addition, a methodology of fault mitigation is proposed that uses of temporal partitioning. This chapter will show how temporal partitioning can mitigate hardware faults with and without performance degradation.

4.2 Methods of SEE Mitigation

As it was mentioned above the space-borne systems are susceptible to SEU, SET, and SEDR hardware faults due to the charged particle bombardment (e.g. cosmic radiation, solar wind, etc.). several researchers [122, 86] developed methods to mitigate transient faults. Mitigation methods are divided on methods to mitigate transient, and permanent faults. Some mitigation techniques allow recovery without functional degradation and some with functional degradation. Some mitigation techniques approach this problem from completely different angle by performing the restoration by just-in-time compilation, or by component relocation [69, 2, 9]. Some of these approaches also perform recovery on very fine-grain level, and some on coarse

grain. All of these approaches are valid, however, their applicability strongly depends on the system specification.

4.2.1 Mitigation of Transient Faults Using a Scrubbing Technique

As gate features shrink in size, the SEU faults are becoming more and more of a problem for scaling transistors in ICs. As a gate of transistors decreases in size it becomes much easier to flip the value of an SRAM cell (e.g. to charge up or to discharge the gate by a hit of a charged particle). In addition, since there is an increased amount of logic/memory cells, the probability that one of these cells will be hit by a charged particle is increased as well. Thus, combined issues of decreasing size of transistors and increasing density of FPGAs make the SEU a serious problem for FPGA based systems.

SEU does not physically damage the FPGA's die, so it is possible to repair this fault. There are several techniques of repairing faults and they depend on the type of FPGA device at hand. A very common approach currently used by the industry for SEU mitigation in FPGAs is the scrubbing technique [24, 92]. Essentially scrubbing, is a periodic update of configuration memory of the FPGA to keep the configuration SRAM cells in the required states. There are several variations of how the scrubbing technique can be implemented. First of all, based on the research from Rockett *et al. [92]* the reconfiguration of the FPGA has to occur at the rate of ten times per fault occurrence. This assures that FPGA would potentially have a downtime of not more than 10% of the probability of fault occurrence [14]. The advantage of using the scrubbing technique is that it allows to repair the SEU faults without the time consuming diagnostics, simply by reconfiguring the FPGA. When a non-partially configurable FPGA is used, to mitigate the SEU fault it is needed to suspend device's operation and to reconfigure the device with the same configuration. Reconfiguration would set the transistor affected by SEU to the proper value. The advantage is that system can be restored without any physical intervention such as IC replacement. The disadvantage, however, is that the entire device is

suspended from operation during the whole time of reconfiguration. For large FPGA devices the reconfiguration time is measured in tens of milliseconds. In the case of triple redundancy (TMR), where a processing unit is triplicated, full reconfiguration would not pose a problem. The reason for that is that while one FPGA is suspended and the reconfiguration occurs, the other two FPGAs are operating normally and whole repair of the SEU fault occurs seamlessly. In the case of partially reconfigurable devices it would be possible to repair the SEU fault by reconfiguring only the affected area and keep most of the device in operation, and without interruption. Scrubbing would occur in cyclic operation by going through all of the sections of FPGA. The only suspended regions of FPGA are going to be ones under reconfiguration and operation would take only few hundreds of microseconds. It is important to mention that SEU occurs only in reconfigurable devices where configuration memory is based on the SRAM [44, 8], and that full reconfiguration of the FPGA resets all of the internal registers/BlockRAM memory which results in loss of data currently processed. It has to be noted that, since 80-90% of configuration SRAM and area on the FPGA device is used up by the routing resources, most of the SEU and SEDR faults effect the routing resources of the FPGA [36, 39].

Another scrubbing technique is to use an external special processor [24, 92] that runs a special SEU mitigating engine, as shown in Figure 4.1.

The scrubbing technique, unfortunately, is not capable of recovering system from the SEDR faults. When a SEDR fault occurs the affected SRAM cell will still be stuck-at-1 or stuck-at-0. Another problem associated with the scrubbing technique, is that it is not always possible to recover the BlockRAM contents if partial configuration is used [73] and full system reconfiguration would be needed. Therefore, even though scrubbing is an effective technique for recovering from SEU faults, it is not always sufficient for space-borne systems and, therefore, this technique has to be expanded to accommodate the downfalls listed above.

Another solution offered by Xilinx is to implement radiation hardened Virtex II Q Pro, Virtex 4Q Pro-V FPGAs, and latest SiRF FPGA that are built in a technological process that

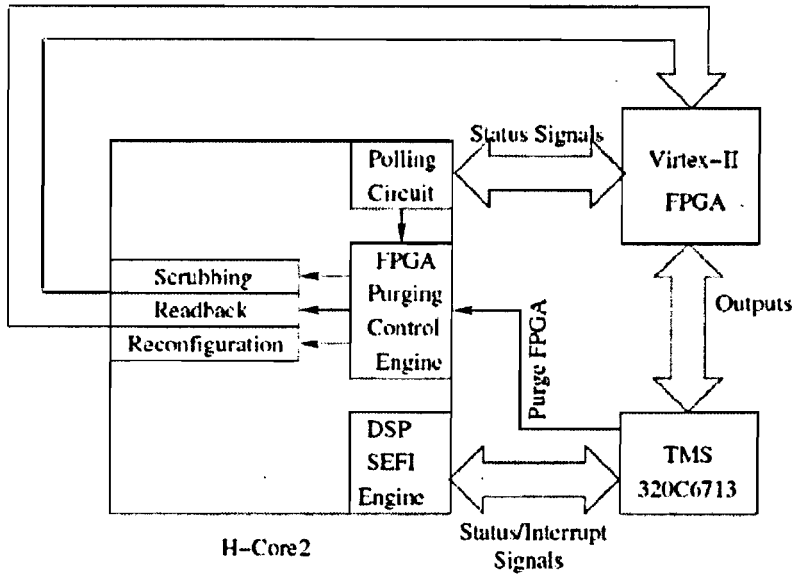


Figure 4.1: Platform Architecture for SEE Mitigation [24, 92]

can withstand charged particle hits. Radiation hardened FPGAs guarantee SEE latch-up Immunity to LET > 100 $MeV/mg-cm^2$ [115]. This solution provides an immunity to a single event latch-up (SEL) for satellites located on low earth orbits (e.g. 300-500km altitude) at the same time provides an alternative to fixed architectures FPGAs (e.g. Actel anti-fuse) [65].

4.2.2 Restoration From Permanent Faults

Permanent faults can occur due to the thermal fatigue, electron migration, manufacturing defect, as well as, SEDR and cause a permanent latch-up, bridging or a permanent open circuit [106, 104]. As mentioned in the Section 4.2.1, if SEDR type fault occurs, scrubbing of the FPGA configuration SRAM is not sufficient since it is a physical hardware damage which can not be repaired by recharging SRAM cell gate. Hence, lots of research has addressed the mitigation of permanent faults, and several methods for different reconfigurable architectures are presented in this section.

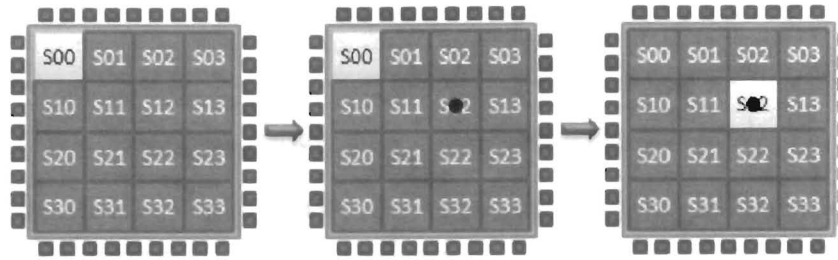


Figure 4.2: Recovering from SEDR Fault on Partially Re-configurable Device

4.2.2.1 Restoration From Permanent Faults Without Functional Degradation

SEDR faults damage the device to the point where it can't be repaired by the means of scrubbing. Therefore, in many current systems a redundancy is designed into the system. Often, a double or triple redundancy is used, however, that significantly impacts the power consumption, the weight, and ultimately cost of the system. This presents a problem for space-borne systems, as well as for any embedded, hand-held, and autonomous systems, because both power and weight parameters are highly restricted for these applications. However, FPGA systems can be configured to avoid the damaged regions of the configurable space. It is possible to design and synthesize IP-cores that perform exactly the same function/algorithm while occupying different areas of FPGA. By dividing the FPGA into tiles and then creating IP-cores that are avoiding one tile at a time, the end result is a set of IP-cores that can be selected to avoid any sector in FPGA. If a fault does occur, and the tile that contains the SEDR is identified, FPGA is reconfigured with IP-core that performs same operation, and while avoiding the damaged sector [20]. Figure 4.2 shows an example of how FPGA is adapted to avoid the sector with a SEDR fault without affecting the operation of the device.

Initially FPGA is loaded with the default IP-core during the temporal partitioning. Any IP-core can be selected for processing, as shown in Figure 4.2. When a fault occurs, a diagnostic method identifies the faulty sector and marks as a damaged tile in the loader memory. Based on this information all of the consequent reconfigurations of the FPGA would use IP-cores that avoid that tile. As for example in Figure 4.2, it shows that initially an IP-core with an

unused *S00* tile is loaded. When a SEDR occurs in *S12* and system identifies that fault, the replacement is performed. An IP-core with the same functionality is reloaded onto FPGA. This implementation of IP-core avoids *S12* tile, and therefore, after reconfiguration SEDR fault is located in the unused area. Obviously, the question that arises from this technique is the size of tiles in the IP-cores. The smaller the granularity of a tile size, the smaller the "wasted" logic area on the FPGA. At the same time, small granularity increases the amount of IP-cores that have to be generated to accommodate all of the sectors. In addition, all of these IP-cores will have to be stored in the non-volatile IP-core library (e.g. FLASH memory). Hence, there is a trade off which has to be considered in the design of such systems. As in the case of scrubbing, the temporal data that was being processed by the FPGA is discarded and has to be recomputed, since reconfiguration process resets all of the registers/BlockRAMs. This is true for all of the restoration methods described in this chapter, however, typically it does not cause a significant problem because the stream processing effects single data frame or part thereof their of. In critical applications an IP-core can be designed to perform a context save before performing restoration, but this requires a more complex IP-core design.

The techniques used in diagnostic and identification of the SEDR faults are covered in the next sections. However, if system experiences a heavy bombardment of charged particles another restoration mechanism should be employed, which is discussed later in the chapter.

4.2.2.2 Restoration by Component Relocation in Spacial Partitioning RCS

If a system is based on a partially reconfigurable FPGA, the restoration can be done by relocation of components from the damaged area of the FPGA to an undamaged one. Initially the diagnostics of FPGA have to be performed to identify the area with a hardware fault, and then either scrubbing or partial scrubbing has to be applied. Also, as mentioned in the previous sections a set of partial IP-cores occupying different areas on FPGA fabric should exist to allow relocation of components. There are two ways to design the set of IP-cores.

The first approach involves a notion of "pluggable" component. Partial IP-cores are designed with the unified interface to an internal standardized bus architecture. This bus contains slots into which processing modules can be "plugged". If an error is detected in one of the processing modules this processing module is "relocated". Relocation is done by removal of the processing module from the current slot and uploading it into a vacant slot on FPGA [7, 96, 60, 105]. The operation of such relocation is same shown in Figure 4.2, except in spatially partitioning case, $S12$ is a pluggable module which gets moved to the empty slot of $S00$. The benefits of such approach are tremendous as the system becomes very flexible and virtually indestructible. Unfortunately, the support for spatially reconfigurable hardware and tools by FPGA vendors is very limited and currently is used only in the academic field [17, 90]. Due to that reason it is not a feasible approach for the most designers in the industrial setting at this time. The other possibility of component relocation is variation of the same IP-core.

4.2.2.3 Restoration from Permanent Faults with Functional Degradation

As mentioned earlier, a different recovery technique has to be used if multiple SEDRs are encountered. If multiple tiles are affected by SEDRs or if there is no IP-core with same performance parameters that avoids the affected areas, a functional degradation has to be employed. For this type of scenario several sectors have to be avoided, and therefore the algorithm cannot fit in the remaining area. In order to keep the system operating one of the solutions is to avoid a much larger section of FPGA. This, in effect, will decrease the performance of the system. In some algorithms it is not possible to keep the same functionality and decrease the performance. In those scenarios algorithm division will have to be performed. However, if it is possible to keep functionality by decreasing performance, then multiple SEDR faults could be mitigated. Overall operation of fault recovery with functional degradation is shown in Figure 4.3.

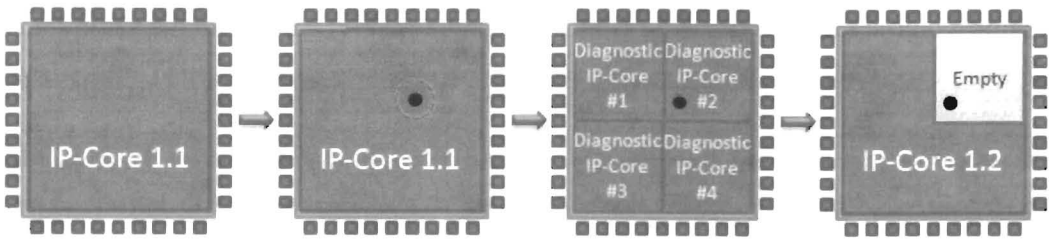


Figure 4.3: Fault Recovery from Permanent Fault with Functional Degradation

When temporal partitioning is used a test IP-core is loaded into the FPGA and test vectors are generated by each section. Based on the results, a faulty section is identified and an IP-core with same functionality but degraded performance is loaded into the FPGA device, as shown in the Figure 4.3. As shown, IP-Core 1.1 was replaced with an IP-Core 1.2 which implements similar algorithm with degraded performance and avoids the top right quadrant of the FPGA fabric. Granularity of the test IP-cores can be varied depending on the reconfigurable device, available variations of IP-cores, and types of test complexity. It would require creation and synthesis of the algorithm on reduced area, and therefore, possibly slower speed, and/or less paralleled computation. Example of that would be a 32×32 multiplier that is decreased in complexity and divided in several stages of 16×16 multiplication. The latency of processing is increased, and required area for this IP-core is decreased. New IP-core with a decreased area will be able to fit into the target FPGA device, and it will be operating at lower speed. The scheduling of the IP-cores will have to be adjusted, as well. Similarly to Section 4.2.2 a decision will have to be made regarding which area to avoid in order to generate a set of IP-cores that are degraded in performance and strategically placed in the FPGA device. This method allows the system to continue operation with lower performance even after a large portion of it is permanently damaged. As it was done for restoration without functional degradation, a set of IP-cores is synthesized that avoids several tiles of the FPGA. This library will have to be carefully developed since, as it can be predicted there is a very large number of combinations of how the IP-cores can be synthesized. FPGA devices will have to be strategically chosen for the design to achieve the best trade off between the encompassing of the possible SEDR faults

and non-volatile memory use. The choice between the IP-cores within the same set is done by an external configuration loader which will keep the parameters of damaged tiles. The choice of IP-cores and schedule of reconfiguration is based on these parameters. An additional benefit of this method is that area reduced IP-cores can be used for power reduction. Power reduction is achieved by utilizing only some of the FPGA resources and, therefore, using less static and dynamic power. Decrease in use of static power occurs because there are lower number of transistors that leak power when not in use.

To repair the SEDR it is needed to identify the region/sector where the fault has occurred. If the region is too big for an IP-core replacement without functional degradation then an IP-core with reduced functionality of performance is chosen.

4.2.2.4 Restoration by Component Routing Constraint Variation

Another approach of system restoration from permanent faults is IP-core variations. During the synthesis portion of the design, the generation of several IP-cores is performed. IP-cores are generated with components restricted to different placements on the FPGA. Since 70-90% of configuration SRAM is used for routing interconnect, hence large number of configuration SRAM cells are not used in a design. Therefore, there is probability that a design with different placement & routing of the same components will not occupy the damaged cells of the configuration SRAM. The number of variations is obviously restricted by the functionality of IP-core, the type of routing, and resource utilization. The final number of IP-core variations is specified by the user, and additional IP-core storage requirement will have to be considered by the designer. As in Section 4.2.2.3, component variation can be done by employing different algorithms that perform similar functionality. This approach is more complex than the restoration from SEDR with functionality degradation. It involves variation of different aspects of the IP-core generation. Mostly, it is highly customized and requires user's specification of timing

and placement constraints. This depends on the critical regions of the circuit that are also identified by the user. These critical regions are constrained in different section of FPGA and are synthesized in several IP-cores that could occupy whole FPGA, however its routing and placement can be varied. The downside of this approach is that it requires extensive re-compilation to obtain different variations. In addition, it will require a post reconfiguration testing to check if the different variation of the IP-Core managed to avoid the damaged transistor/trace [2].

4.2.2.5 Restoration by Just-in-Time FPGA Compilation

Another restoration scheme that was proposed by several researchers [69, 7, 9] is a just-in-time FPGA compilation. It operates on the notion of run-time recompilation of FPGA configuration. If a fault was detected and identified in some region of the FPGA, an error is reported to an external processor. This processor re-runs a routing algorithm that performs the placement/routing of the FPGA avoiding the damaged area. The damaged area could be a switch block or a logic block, or even an embedded hardware component, such as BlockRAM. At the completion of the Place&Route operation the damaged FPGA is reconfigured with the new IP-core. Riverside On-Chip Router (ROCR) was designed for simple FPGA configurations [69, 9]. Authors showed comparable results to Versatile Place and Route tool (VPR) in terms of timing and much smaller memory requirement for an FPGAs of size of 67×67. For larger FPGAs, that have much more complex structure and contain embedded hardware, such as RAM Blocks, multipliers, PowerPCs, this approach is not feasible, because of the time it takes to do the place and route of a complex FPGA circuit. Place and route is an NP-complete problem and requires a heuristic approach for finding an optimal solution. Currently, to perform place and route for a comparable Virtex 4 FPGA, a PC system equipped with 2 GBs of memory and dual CPU architecture operating at 3.4 GHz requires anywhere from 10 minutes to over 6 hours, depending on the constraints that are applied to the design. This is assuming that it would be an embedded microprocessor which would be used and operate much faster

than current desktop PCs to perform place and route. This approach is suitable for systems that can be taken offline for long periods of time to perform such a task. Therefore, this technique is not acceptable for systems that are required to operate in real-time and to have a minimal downtime and cost.

4.3 Methods for Fault Diagnostic and Fault Localization in SRAM Based FPGA Devices

Equally important is the research of the fault detection and fault localization in the SRAM based FPGAs. Previous section described various methods for fault are mitigated and briefly mentioned some detection techniques. This section focuses on generic diagnostic techniques and their applicability to the FPGA based systems.

4.3.1 SEU Diagnostic in Configuration SRAM

Since the SEU effects SRAM cells, one of the simplest detection methods that is provided by the FPGA manufacturing companies is the *Readback* operation. Xilinx FPGAs allow the readback procedure [15] while device is in operation. For such diagnostic, however, an external processing device is needed. This device may be a microprocessor, that can perform readback, verification, and full/partial configuration bitstream manipulation. Each IP-core should have an accompanying bitstream mask to be stored on an external FLASH memory for the purposes of comparison with the readback data. A scheduled readback may occur during the operation of the system. If discrepancy is found, microprocessor will conduct FPGA re-configuration procedure to mitigate the incurred fault. The re-configuration should be able to mitigate transient faults. However, in case of a permanent fault, another method should be utilized as was discussed in the previous sections. It should be noted that the overhead introduced by the diagnostic mechanism increases the cost and power consumption of the system.

It is important that readback operation is done in a specific manner. If the design is using an embedded BlockRAM of FPGA, the readback will acquire erroneous bitstream and will not correspond to the mask bitstream [4, 15]. Therefore, if it is done in a run-time, only configuration SRAM has to be considered for SEU repair. In addition, if repair operation is required, only configuration SRAM will be reconfigured. Otherwise the inter-processing data will be lost from the BlockRAM. Often, the scrubbing or reconfiguration is done 10-50 times more frequently than the predicted fault rate. On the other hand, with readback implementation, the scrubbing procedure can be initiated only in the case of real fault. The advantage of this approach is power savings, and lower overall system downtime in contrast to continuous scrubbing. However, the run-time full/partial reconfiguration is supported only by Xilinx Virtex FPGA family [84]. If system implementation has to utilize different types of FPGA devices that do not support readback, then a different approach should be used. The same is true of a system that does not support real-time readout without interruption of operation, in that case, system downtime is increased even more.

4.3.2 Off-line Diagnostics of Permanent Faults in Data-paths

A common practice is an offline diagnostics of faults [30, 106]. This type of diagnostic is usually performed by taking the system off-line and either physically interfacing it to a test platform or utilizing existing I/O port interfaces, such as JTAG. Over an available connection a series of test vectors is provided to the system and the result is then compared to the expected one. Data vectors are usually selected to cover the maximum amount of processing elements, and to identify which of the elements produced the fault. Depending on the data-path specifics the type of fault, can be: stuck-at-0, or stuck-at-1. In some instances faults are not easily detectable because they are occluded and it requires enormous amount of test vectors to identify every possible fault. Such testing requires substantial amount of time [74], as well as peripheral support to be able to operate while the main processing is disabled. Test vector storage

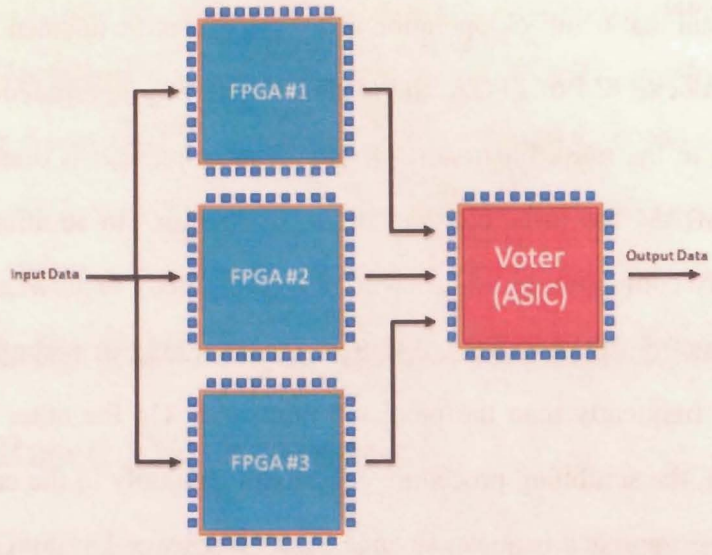


Figure 4.4: Typical TMR Organization

memory has to be integrated into the system, too. Finally it is typically the human operator that performs the testing, and for this reason such such approach is suited for a narrow field of reconfigurable systems.

4.3.3 On-line Diagnostics of Permanent Faults with TMR-approach

A common approach to diagnostic of faults is a Triple Modular Redundancy (TMR) [87]. This approach dramatically improves the reliability of the system, but triples the use of area, power consumption, cost, etc. There are different methods of design using the TMR; some of them are implemented on a single FPGA and some on multiple. The main idea of TMR approach is for system to have three identical processing units, results from which flow into the voter that compares the results.

As shown in the Figure 4.4, FPGA #1, #2, and #3 are provided with the same input and their results are transmitted to the voter which simply compares them and checks for discrepancies. If the result for one of the systems is not complying with the other two, a faulty system is immediately identified. Most of systems that used TMR, replicate the same processing unit in three different FPGAs and use an external voter comparing the produced results (e.g. Figure

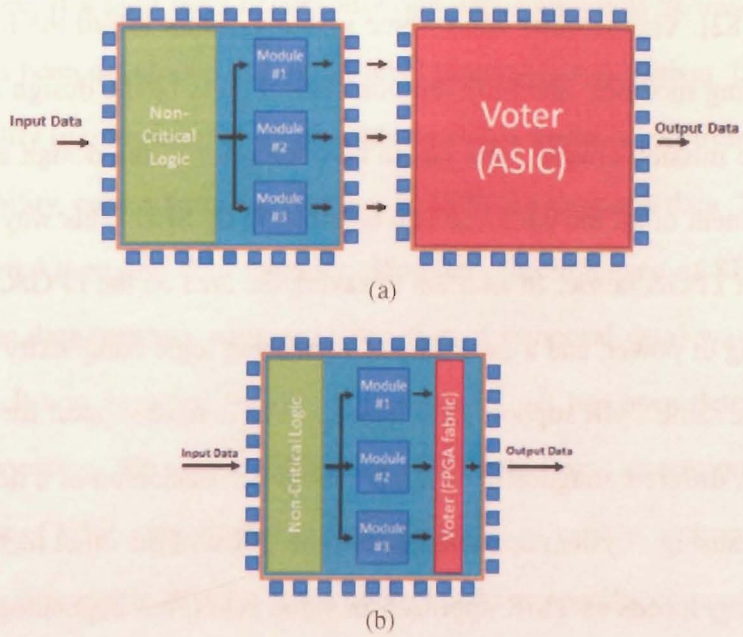


Figure 4.5: TMR Implementations on a Single FPGA: with (A) External Voter (B) Integrated Voter

4.4). There are several variations on this approach, such as replicating the function three times within one FPGA [87]. As before, the result is outputted to the external voter that performs comparisons. Such implementation is shown in Figure 4.5 (A).

Similarly to the previous approach, the voter can be integrated in the FPGA as part of logic, as shown in Figure 4.5 (B). The downside of the integrated voter approach is that if SEU occurs in the voter, the correct result will not be known. Although, as the implementation of voter is compact and occupies limited logic, hence, in some scenarios it may be beneficial for systems that are constrained with peripheral components. Some development tools already include an ability of automating the generation of TMR circuit of critical elements (e.g. Xilinx TMR tool) [87, 116].

Nonetheless, some applications can not accommodate the costs and hardware overhead associated with the TMR approach. For example, many of space-borne applications are very sensitive to the total mass and power consumption of the system. Therefore, tripling hardware resources for the processing system in some complex data-paths is not cost-effective for space

applications [82]. On the other hand, some systems cannot afford the full triple replication of the processing modules, and only mission critical parts of the design are triplicated. The analysis of the mission critical parts has to be done prior to the design and be based on the critical assessment of all the parts that can be effected by SEU. This way may save a significant amount of FPGA fabric. In addition to saving the area on the FPGA, there are two other benefits: saving of power, and a capability of increasing logic complexity by the saved space, with almost the same TMR support [87]. Not all FPGAs have support for partial reconfiguration, therefore, different diagnostic methods should be considered or a device has to be fully reconfigured causing a system operation downtime. Most of the other methods cannot deliver same processing speeds as TMR approach in some cases, but depending on the application needs, other methods are suitable for the systems with SEU recovery requirement [82].

4.4 The Method of Multi-level Mitigation of Transient and Permanent Hardware Faults in RCS with TPM

In this section novel method for self-restoration of RCS based on FPGAs with SRAM configuration memory is proposed and discussed in details. This method has been developed specifically for an RCS with embedded TPM and allows mitigation of:

1. Transient hardware faults (e.g. SEU, SET), discussed in Section 4.4.1
2. Permanent hardware faults (e.g. SEDR), discussed in Section 4.4.2

The method takes an advantage of the TPM nature, which assumes cyclic reconfiguration (with new IP-core) of the target FPGA device or an associated slot of FPGA device. This allows minimization of on-line diagnostic procedures, since the new IP-core updates the content of configuration memory of target FPGA device or its slot. The mitigation of SEU or other transient fault occurs within one cycle time (period of processing of one block/frame/packet

of data). Therefore, if a fault does occur, only one data-frame will be invalidated. The proposed method has been developed as a multi-level protection mechanism [50] to provide the maximum flexibility in mitigation of all possible hardware faults in run-time or "close-to-run-time". This flexibility, comes from the fact that in TPM the temporal data (between IP-cores) is stored on external temporal data memory. However, the influence of SEU or other radiation effects on the data-memory content (corruption of temporal data) was put out of scope of this research. It was assumed that if any hardware fault has been determined within i^{th} cycle of data-processing, the associated results of i^{th} data frame execution must be ignored entirely. For most of DSP, video/image processing tasks, as well as multimedia applications, this assumption is acceptable. But for some control tasks or specific computation tasks where each data-vector is considered valuable, the above approach may not be suitable. The following assumptions also were also considered in the proposed multi-level run-time protection mechanism [50]:

1. The probability of SEU and other transient faults is much higher than the probability of permanent faults.

2. A permanent hardware fault is a fault which cannot be mitigated by the scrubbing procedure. Therefore, if after a certain number of scrubbing procedures (e.g. re-configuration of the same IP-core and cycle of functional diagnostic) the fault still exist, then the it is assumed to be permanent.

3. A permanent fault can be mitigated by:

   (a) restoration without functional degradation

   (b) restoration with functional degradation of some performance parameters

All the above aspects of multi-level protection mechanism will be discussed in following sections, including the description of all stages of the proposed mitigation algorithm.

4.4.1 Mitigation of SEU and Other Transient Faults by IP-core Scrubbing and Functional Diagnostic Cycle

When we are talking about temporal partitioning system it is assumed that FPGA device is periodically reconfigured with the next processing sub-task (IP-core). This is a significant benefit for system that has to be fault tolerant. On every cycle of reconfiguration, whole FPGA is updated with a new IP-core. IP-core automatically mitigates all of the transient SEUs, this in turn decreases the frequency of fault diagnostics that has to be performed. At the same time, a permanent fault diagnostic IP-Core can be inserted between any sub-tasks, and perform testing of FPGA device. The test IP-Core could have different granularity as shown in Figures 4.3 and 4.2, depending on the application requirements and upon the completion of the test it provides the user/system with the fault results. The overall operation of the TPM with fault tolerance mechanism is shown in Figure 4.6.

Initially, IP-core$_i$ is loaded into the FPGA and initial test is run on the IP-core$_i$ to identify if the IP-core$_i$ is operating correctly. Considering that all the tests are passed, the processing of data frame is followed by loading of IP-core$_{i+1}$. If, however, a fault is identified, the scrubbing (reconfiguration with the same IP-core$_i$) procedure is initiated and the fault counter is incremented. If the fault persists for several reconfigurations, it is identified as a permanent fault and permanent fault mitigation procedure is required. As mentioned before, there are two possible procedures mitigation with diagnostic, and without diagnostic. The choice of the permanent mitigation method depends on:

1. Timing constraints for fault recovery
2. Granularity of IP-cores with degraded performance
3. Bitstream size of the FPGA, and consequently the time of reconfiguration

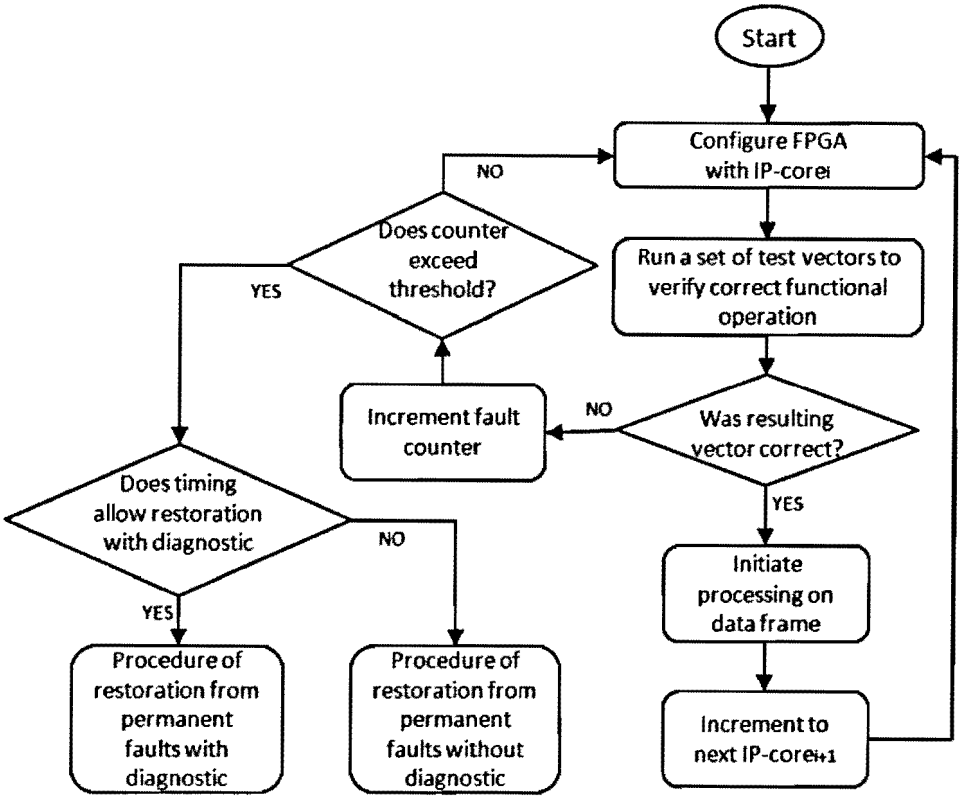


Figure 4.6: Operation of the TPM with Fault Tolerance Mechanism

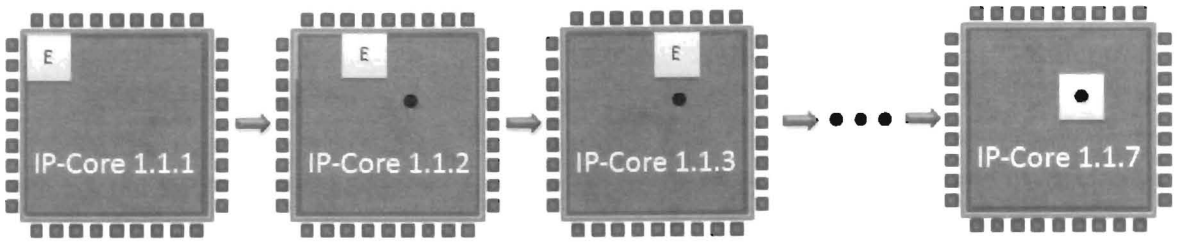


Figure 4.7: Sequential Reconfiguration Process for Permanent Fault Mitigation without Fault Location Procedure

Next sections cover both of these procedures in closer detail, and describe their flow of operation along with the fault tolerance mechanism shown in Figure 4.6.

4.4.2 Run-time Mitigation of Permanent Faults with/without Functional Degradation

Most of the fault mitigation methods mentioned fault location/diagnosis. This diagnosis typically involves the loading of a specific test IP-Cores that would perform testing of various sections of the reconfigurable device to localize the fault. In the proposed approach it is possible to resolve the SEDR fault by repeated reconfiguration without the need for diagnostic procedure.

In this approach, as in Section 4.2.2.1, it is assumed that there exist several IP-cores that avoid different sections of FPGA and perform same functionality while having same performance, as shown in Figure 4.7.

The idea is that these IP-cores are configured one by one onto the FPGA device. On the start up of each IP-core a self-check is performed by feeding test bench vectors to the IP-core to identify if this core operates properly and avoids the damaged region, similar to the approach in Section 4.2.2. In many cases such approach is beneficial due to the fact that exhaustive diagnostics can take much more time than several reconfigurations of the FPGA device [50].

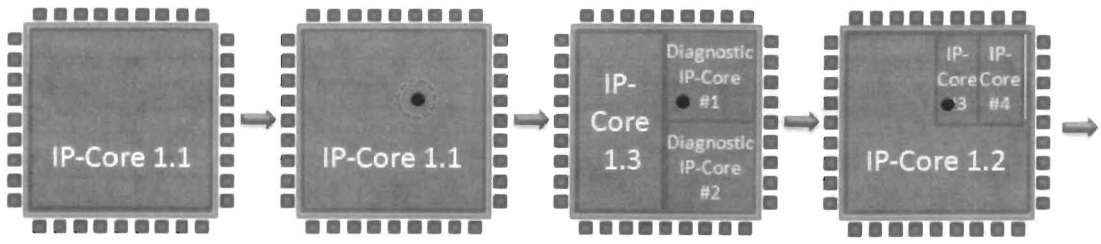


Figure 4.8: Permanent Fault Mitigation with Incremental Recovery to the Maximum Possible Performance

Another benefit of such approach is that it can combine all previously discussed methods and provide a rapid fault recovery. Initially, an IP-core with degraded performance is loaded into the FPGA and occupies half of the device. This way within a maximum of two configurations the system resumes operation, although with reduced performance. The second half of the device is occupied with a diagnostic core, which identifies the quarter where fault is found, as shown in Figure 4.8. In the next reconfiguration three quarters of the device are loaded with the processing core, and the remaining quarter is loaded with the diagnostic IP-Core that performs further testing. This operation is repeated until the smallest granularity is reached and operation performance is restored to the maximum possible operation level.

4.4.3 Complete Algorithm for Multi-level Protection Mechanism Embedded to the TPM

Based on the steps outlined in previous section, the flow chart for fault mitigation is produces, as shown in Figure 4.9. This flow chart in Figure 4.9 is a continuation of the flow chart shown in Figure 4.6. The fault mitigation with a diagnostic IP-core is a "recursive" operation where the algorithm selects first large portion of FPGA for diagnostic and with a rapid test identifies which section contains the fault. Following that, a smaller section of FPGA is selected for diagnostic while the rest of FPGA is occupied with the larger sized IP-core. As shown, the process is repeated until the smallest granularity is reached. This "divide and conquer" procedure allows to rapidly return the system to operation, and after several cycles to restore its

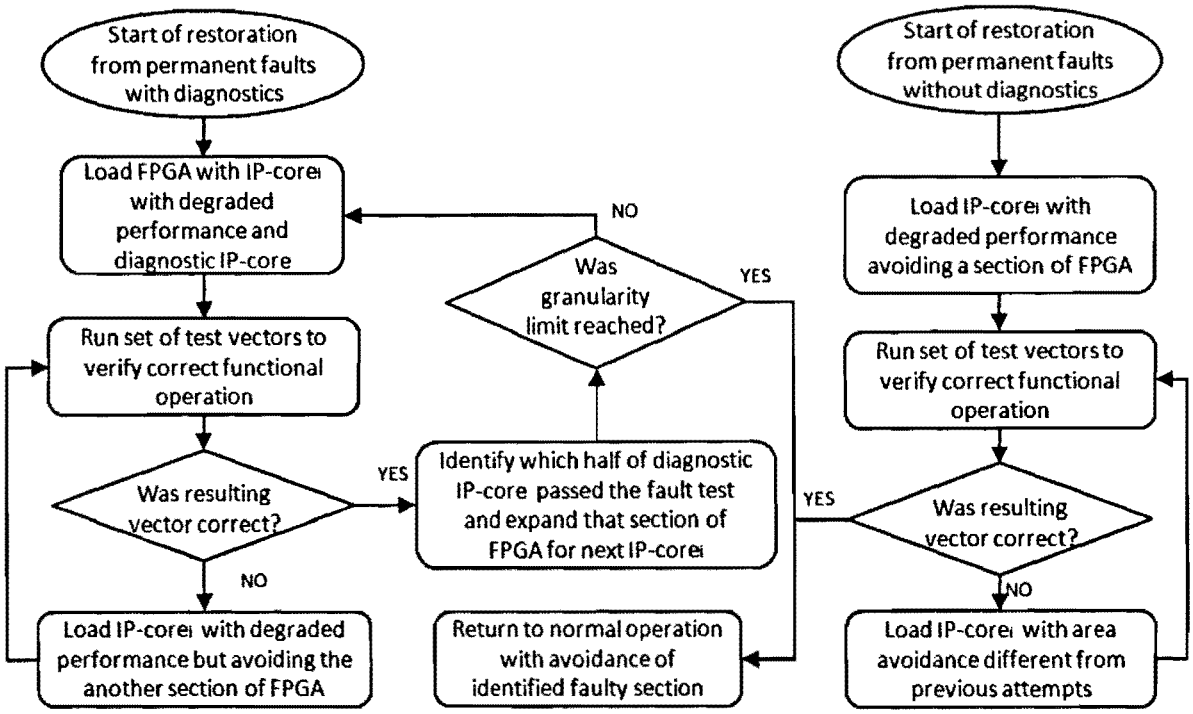


Figure 4.9: Permanent Fault Mitigation Flow Chart with/without Diagnostic Procedure

performance with the return to normal operation with some area avoidance. The other method is not recursive and involves reconfiguration with avoidance of different sections of the FPGA based on the theory described in the previous section. As soon as the correct operation of the IP-core is observed, the system returns to normal operation with avoidance of faulty section of FPGA.

There is no exact answer which approach is beneficial prior to knowing the nature of processing system. As discussed before, the main factors are the system timing and the complexity of diagnostic, as well as the granularity of the area avoidance blocks, and the availability of performance degraded IP-cores.

This section presented methodology of how the recovery mechanism has to be designed in order to be able to mitigate SEU and SEDR faults with or without functional degradation.

4.5 Cost-efficiency and Performance Comparison of the TMR Approach and Multi-level Mitigation of Transient Faults in TPM Systems

There is always a big debate on which fault tolerance methodology to pursue in initial stages of system design since there are several variations available. In this chapter different fault tolerance methodologies were discussed, and the current section compares them in terms of performance, application suitability, and outlines, their pros and cons. The comparison is done based on division of different types of systems, since it greatly effects the type of fault tolerance approach to be used.

4.5.1 Uninterrupted Mission Critical Systems

For some systems, such as space borne or medical life support systems, it is imperative not to have any type of interruption in operation. An interruption in service can directly affect the life of humans, be it on terrestrial applications, or on a space station. In those types of scenarios reducing cost and, in effect power is not an option and TMR approach is necessary. Moreover, as it was described in Section 4.3.3 a voter which is an ASIC or an anti-fuse FPGA, would also need a backup to make sure that the voter itself does not become the critical part. However, it should be noted that the critical systems that are necessary to be TMR-ed typically are not of a large size and, therefore, it is possible to use TMR approach in a partial architectural arrangement. The reason why the proposed methodology of fault mitigation is not fully suitable is because no incorrect result is allowed. This comparison is only applicable to the mitigation of the SEU and SET faults. The overall cost of the system would be calculated as follows:

$$C_{TMRsystem} = 3 \times C_{ProcFPGA} + 2 \times C_{voter} + C_{PCBarea} \tag{4.1}$$

Where $C_{TMRsystem}$ is the cost of the TMR system, $C_{ProcFPGA}$ is the cost of the FPGA involved in the data processing, C_{voter} is the cost of voter, and $C_{PCBarea}$ is the cost of the additional PCB area that is needed to support TMR approach.

On the other hand, if the TMR system incurs, a SEDR hardware fault in one of its processing FPGAs, the TMR approach becomes useless, and system's performance is downgraded to a single FPGA processing without the TMR, while continuing to consume power of the whole TMR system. The ideal solution in that case is to combine the proposed fault mitigation methodology and the TMR approach. Hence, if SEDR fault does occur in one of the FPGAs, the proposed methodology diagnostic technique can be employed to repair it, and this way the system can proceed with the TMR mode of operation. The overall cost of the system would be:

$$C_{TMRsystem} = 3 \times C_{ProcFPGA} + 2 \times C_{voter} + C_{PCB_{area}} + C_{TPM_{lowcost}} \quad (4.2)$$

Where $C_{TPM_{lowcost}}$ is the cost of the TPM support hardware used to add the capability of temporal partitioning to TMR system. The other variables are same as in the previous equation.

This cost can be considered as a cost of hardware, as well as power consumption and in both parameters the increase is not significant. The reason is that configuration manager is not required to support high reconfiguration speed and can be based on a low cost/power microcontroller that loads configuration in a serial manner. Hence, the cost of such system would increase on ~$10 (based on the unit prices from the major part distributors). In regards to power consumption the increase is less than 20-40mW in active operation.

4.5.2 Critical Systems with Non-Real-time Control

For the critical systems that can tolerate interruption in service TMR system becomes a much less cost efficient solution. The proposed methodology can rapidly repair a fault within several

milli-seconds, so the interruption of service is short. On the other hand, the cost & power savings are quite significant, when compared to $C_{TMRsystem}$:

$$Ratio_{TMR/TPM} = \frac{C_{TPMsystem}}{C_{TMRsystem}} = \frac{3 \times C_{ProcFPGA} + 2 \times C_{voter} + C_{PCB_{area}} + C_{TPM_{lowcost}}}{C_{ProcFPGA} + C_{PCB_{area}} + C_{TPM}} \approx 5 times \tag{4.3}$$

Since the main contributing cost factor in FPGA based systems is the actual cost of the FPGA, the overall cost mostly depends on the number of FPGAs present in the system. In TMR system there are three processing FPGAs, one main voter and one backup voter (typically present in some mission critical designs) totaling 4-5 FPGAs. Whereas in TPM approach there is one FPGA in the non-pipelined case and dual FPGA configuration in the pipe-lined case, which results in 2 to 5 times more cost effective solution for the critical systems with non-real-time control.

4.5.3 TMR and TPM Approach Comparison Summary

In conclusion to the comparison section, that there are advantages in use of the TMR approach in mission critical systems with real-time control which do not allow any sort of service interruption. However, the overall increase in cost for such systems varies from 2 to 5 times depending on the complexity of the system. This increase also affects the overall power consumption, since at high processing speed FPGA becomes the main consumer of power. In all other cases the TMR approach is not cost effective and TPM approach of fault mitigation is a much more preferable option.

4.6 Summary

In this chapter the main approaches and methods for mitigation of hardware faults in FPGA based systems were presented, along with the major causes of these faults. It was found that

the major focus of R&D works is placed on mitigation of Single Event Upsets in the FPGA devices with SRAM based configuration memory. Different approaches for SEU mitigation have been discussed in details. However, in literature there are not many publications regarding mitigation of permanent hardware faults that can be caused by many other radiation issues. Nevertheless, two approaches for mitigation of permanent hardware faults have been discussed in this chapter. It was shown that in general case, the location of the hardware fault has to be found using fault location procedures.

It was shown that mitigation of the permanent hardware fault could be done without time and resource consuming fault location procedures. Furthermore, it was demonstrated that by using only fault detection procedures and cyclic reconfiguration of target FPGA by the TPM it is possible to mitigate all transient hardware faults, as well as permanent faults in run-time. In a case of a transient fault, TPM cyclically reloads the same IP-core by the run-time scrubbing procedure. In a case of a permanent fault, TPM cyclically loads different IP-cores with the same functionality but having a different place-and-route topology to avoid the damaged area. The topology of each IP-core is designed to avoid certain areas of the FPGA (area avoidance concept). On the basis of the above concepts, the novel method for run-time mitigation of both: transient and permanent hardware faults in FPGA systems with SRAM configuration memory was proposed and developed. This is the first method that incorporates fault mitigation procedures which can hierarchically call each other according to response of FPGA until the moment when the system is fully recovered from the fault. This method has been put on the basis for the Built-In-Self-Recovery (BISR) procedure to be embedded to the developed TPM for partially reconfigurable FPGAs. The method was published in the Journal for Spacecrafts and Rockets [50].

It is necessary to mention, however, that R&D of fault detection and fault location algorithms and methods were out of the scope of this research. Only published methods for the

fault diagnostic and location have been considered for application. Next chapter will introduce the developed methodology of task segmentation, as well as, the notion of virtual components.

Chapter 5

Task Segmentation and Efficiency of the TPM

5.1 Introduction

When an algorithm has to be implemented on a system and it exceeds the size of the available FPGA device, there are two ways to resolve the situation. One is to choose an FPGA with larger amount of logic resources or use multiple FPGA devices. The second option is to process the algorithm/task by parts in different time slots. The latter, requires a methodology of algorithm/task division in appropriate segments. The first option mentioned above, is typically used in the industry. Though this is a straight forward solution, it does not always translate into cost-efficient result. The reason for that is an overall increase in cost of the processing platform. The cost increase is induced by an added FPGA device and all of its peripheral components, as well as, higher power consumption, extended dimensions, and weight of the system. During preliminary design of a multi-FPGA system, a highly parallel bus(es) are designed to communicate between two or more FPGA devices. These buses are fixed and, therefore, if requirements will change in the future, it would be more difficult, if not

impossible, to modify bus topology. The second option of task segmentation and is not yet widely used in industry since it is an emerging technology.

This chapter will discuss the approach of the task segmentation and the creation of Segment Specific Processor (SSP). This approach provides an ability to implement tasks, which require more resources than one FPGA can provide. By being able to reuse the hardware resources in time domain, it is potentially possible to decrease required FPGA resources. This approach allows to reduce weight, power consumption, as well as associated systems cost. Another benefit of the temporal partitioning is the capability to mitigate of hardware faults (e.g. SEU - Single Event Upsets) in run-time. Being embedded withing design of the system, fault mitigation further decreases system's cost. It also increases system's reliability, and as was shown in Chapter 4, does not require much of an additional hardware. Nonetheless there are certain trade-offs associated with this approach of system design:

1. First, a trade-off comes from the timing overhead which is required for transition of the FPGA system from one configuration to another. Timing overhead is associated with:

   (a) Temporal data reading/writing procedures

   (b) Switch from one IP-core to another

2. Second, there is a certain hardware overhead that is associated with the run-time reconfiguration mechanism, as was discussed in Chapter 3.

3. Last, there is a problem in optimization of algorithm segmentation, which has to be resolved and completed in a non-NP time.

These drawbacks have been analyzed and

assessed to maximize system's performance and cost-performance ratio. This chapter introduces a methodology for task segmentation from the temporal partitioning point of view. Also, this chapter provides the analysis and results for the cost-effectiveness of the segmentation approach.

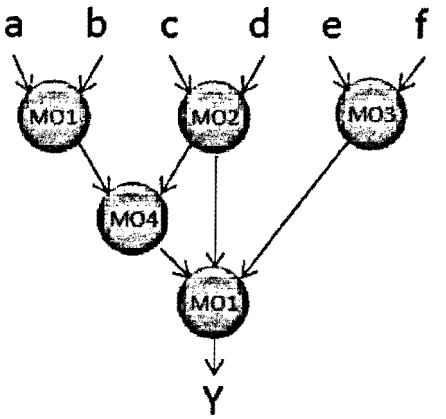


Figure 5.1: Sequencing Graph Example

5.2 High-Level Synthesis of Application Specific Processors

The development of the TPM methodology, which provides the most cost-effective data-stream executions on the RCS with temporal partitioning of FPGA resources, is a first aspect of the task segmentation. A discussion of algorithm representation techniques that are used already by different researchers is presented below.

Every algorithm/task can be described in various ways, such as: a text explanation, a pseudo code, a connected graph or a data flow graph (DFG). One of the common ways of describing an algorithm for execution on dedicated digital circuits is a connected sequencing graph, as shown in Figure 5.1.

Sequencing graphs (SG) are widely used to represent algorithms and data dependencies within the processing algorithm [48].

Definition: *A Sequencing Graph is a collection of Vertexes and Edges that represent a flow of data and operations of a stream processing algorithm in an acyclic manner.*

Vertexes represent a data processing operators (e.g. Add, Divide, Shift, etc...), and edges represent data dependencies [88]. Elementary processing operators can be combined into Macro Operators (MOs) that can include some proprietary processing elements, such as filters,

encoders, DSP operators, etc. Eventually, a flow of data reaches bottom of the graph, that is an output of the system.

5.3 The Concept of Task Segmentation

In some applications it is absolutely necessary to have the whole system built on one chip due to strict timing constraints and where the cost of the system is not a defining factor. However, there many applications that are not as time critical, on the other hand, are more cost sensitive and need to take into account many other parameters (e.g. power, reliability, dimensions, etc.) In those cases it may be more preferable to employ Temporal Partitioning Mechanism (TPM). The TPM approach usually allows to increase the cost-performance characteristics of the system.

In this context, the temporal partitioning is defined as:

Definition: *Temporal Partitioning is a division of a task algorithm into segments that are executed one after another in different time slots on an FPGA-based processing platform.*

Definition: *Run-Time-Reconfigurable (RTR) processor is a computing paradigm with architecture that can reconfigure a part of or a whole architecture during the task execution.*

The optimal segmentation of a task and the generation of series of configuration bitstreams requires a development of an associated methodology. Typically, the operation of task division is performed based on the reconfigurable device. A segment is populated with the task operations until the area constraint is reached, at which point the segment would be enclosed. All of the temporary results for that segment would be saved to an external memory. The external memory is required since the memory embedded in FPGA gets reconfigured along with the logic during reconfiguration. This procedure of segmentation would repeat until all parts of the task are segmented.

The proposed segmentation approach presented in this section is quite different. Instead of exactly forming hardware architecture, a task algorithm is initially assembled in a form of a

Sequencing Graph (SG) using predefined operators that describe the operation of the task and its constrains. The predefined operators are called "Macro Operators" (MOs) because they represent an overall description of an operation that is performed on single data frame. At the same time MOs, do not specify exact implementation of the operation.

Definition: *Macro Operator (MO) is a macro-function, which consists of a set of elementary functions/operators and can be represented in a form of a sequential graph of predefined interconnection of Elementary Operators (EO).*

The analogy of the MO in the already existing systems can be an algorithm function included in MATLAB. A function receives a specific input and produces an output in a particular format. User is not concerned with the implementation of the function and concentrates only on invoking functions in sequence. The sequence performs the operation of data processing, by passing results from one function an other. Similarly, in object oriented programming a class can perform a particular function which can include a series of elementary operations. The class can have many implementations of that function. Consider sorting an array, example where a class *Sort* has several different implementations, of the sort function *Bubble Sort.* The soft function can be invoked on an array and produce the index of the item in question. The user in such operation simply performs the invocation of the *sort* procedure and is not concerned with the actual implementation of the search function. Moreover, *Sort* class implements one of the algorithms using elementary operation, such as: move, add, subtract. Similarly, MOs are assembled from the elementary operators that form an algorithmic implementation of the MO.

Every MO represents a particular macro-function which can have many variations of its implementation in hardware. Variation of implementations are possible due to parallelism found in some operations. These implementations can be represented in a form of MO-optimized processing units called Virtual Hardware Components (VHC). Each VHC varies by operation and resource parameters, as well as performance. Contradictory parameters are

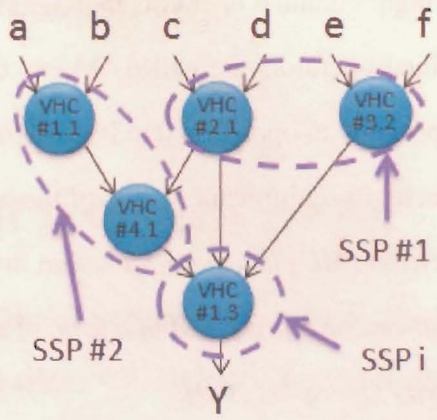


Figure 5.2: Task Algorithm Represented by VHCs Corresponding to MOs

selected to create a wide range of variants that encapsulate limiting conditions of these parameters.

Definition: *VHC is an Application Specific Virtual Processor (ASVP) designed to implement a particular MO with a specific set of constraints.*

Since same sub-set of MOs can be combined into segments that compose the algorithm SG, associated VHCs can be combined into Segment Specific Processors (SSP) as shown in Figure 5.2.

Definition: *SSP is a processor that is optimized for a particular reconfigurable device based on the set of given restriction such as: time, logic area, data transferred between segments and more.*

In turn, these SSPs are synthesized into bitstreams and are loaded in sequence on to the target FPGA. Intermediate results that are produced by the SSPs are passed between the reconfigurations by the means of storage in SRAM or SDRAM.

Generation of these SSPs presents a challenge in the design of temporal partitioning, because it has to be done in an automated manner. Each of VHCs has several particular properties such as: processing cycles, required logic, power consumption, and special embedded

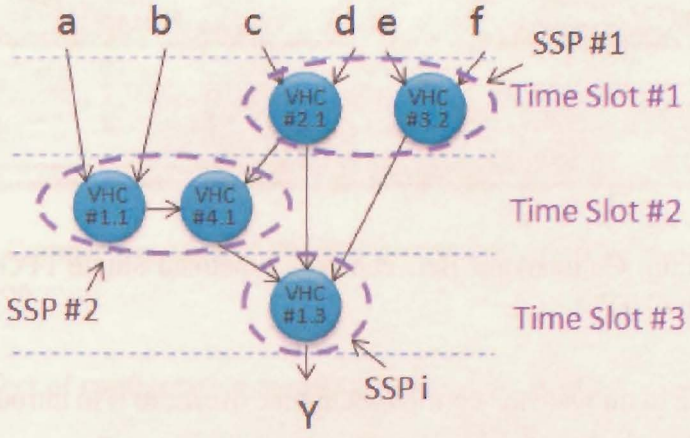


Figure 5.3: Segmented and Scheduled Sequencing Graph

devices (e.g. BlockRAM, Multipliers), etc. Combination and inclusion of VHCs into the segments, as well as, proper evaluation of resource utilization is the key to increase of the overall performance, which, in turn, increases the cost-performance ratio (CPR) of the whole system.

As mentioned previously, the temporal partitioning of a task algorithm and corresponding SSPs allow to reuse the same hardware resources. SSP_i should be scheduled for respective time slot according to the task SG, as shown in Figure 5.3.

This, however, introduces a reconfiguration time overhead that has to be accounted for in the task processing application. The details of reconfiguration delays where explained in Chapter 3. Effect of this overhead is to be considered in the actual task execution. In next sections cost-effectiveness of the TPM architecture is examined for different scenarios Additionally, the optimal applications of the TPM architectures are introduced.

5.3.1 Cost-Effectiveness of TPM

As described in the previous section, the configuration time overhead may be a bottleneck in data processing with TPM. Therefore, the mitigation of this overhead is the major goal in platform architecture design. The analysis of possible architectural solutions is presented below.

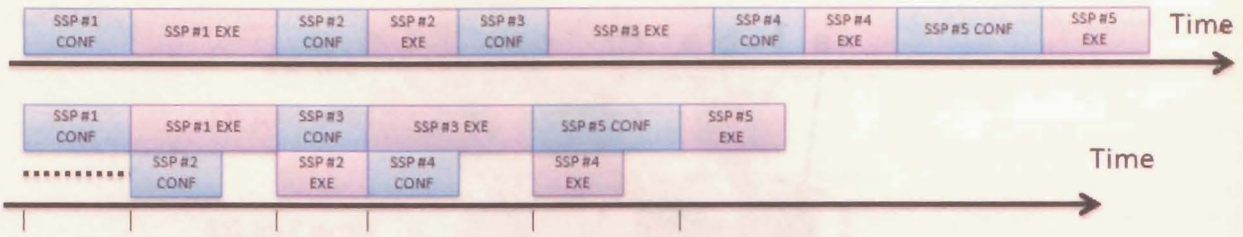


Figure 5.4: Timing Comparison Between Non-pipelined single FPGA and Pipelined dual FPGA Operation of TPM

One approach to mitigate the configuration time overhead is to introduce a "pipelined" dual slot or dual FPGA architecture by placing FPGAs in parallel and switching between them as a discussed in Section 3.3.6. In an ideal scenario it is possible to eliminate the reconfiguration overhead by "fetching" the next SSP_{i+1} configuration bitstream while executing SSP_i. This is possible in case where the processing time T_{exe} of SSP_i is larger or equal to the configuration time T_{config} of SSP_{i+1}.

$$T_{config}(SSP_{i+1}) \leq T_{exe}(SSP_i) \tag{5.1}$$

If $T_{config}(SSP_i)$ doesn't meet the timing requirements and takes longer than processing time of the previous segment, then a fraction of the configuration overhead is still going to be present in the system. A comparison between non-pipeline and pipelined operation is shown in Figure 5.4 Figure 5.4 shows that in time period *T3,* while SSP2 is processing data frame, a configuration overhead of SSP3 is fully hidden, since $T_{config}(SSP3) = T_{exe}(SSP2)$. Similar scenario occurs in the *T2* and *T4* time slots. In *T5,* however, we see that SSP4 finished execution before SSP5 completed configuration operation and in this case $T_{config}(SSP5) > T_{exe}(SSP4)$. Ideally, such scenarios should be avoided to maintain the maximum speedup.

The ideal scenario of pipe-lined execution where $T_{config}(SSP_{i+1}) = T_{exe}(SSP_i)$ is shown in Figure 5.5. Depending on the speedup ratio, as well as the ratio of increased cost of the system, cost effectiveness of the pipelined TPM solution has to be evaluated and compared to the non-pipelined architecture. Techniques, such as bitstream compression, can be employed

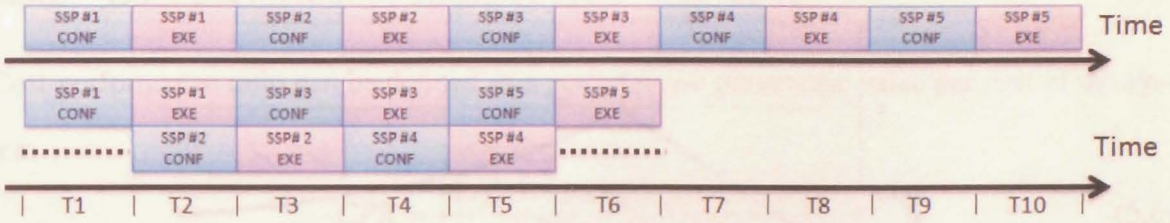


Figure 5.5: Timing Comparison of Ideal Pipelined Implementation of TPM to a Non-pipelined TPM Implementation

to minimize the effect of configuration overhead, and it is possible to achieve close to double speedup with such approach, as shown in the following derivation:

$$T_{config\,overhead} = T_{config}(SSP_{i+1}) - T_{exe}(SSP_i) \tag{5.2}$$

$$if\ T_{config}(SSP_{i+1}) \leq T_{exe}(SSP_i)\ then \tag{5.3}$$

$$T_{config\,overhead} = 0$$

$$Speedup_{np-TPM/pl-TPM} = \frac{T_{TPM}^{np}}{T_{TPM}^{pl}} = \frac{\sum_{i=1}^{n}\left(T_{exe}(SPI_i) + T_{config}(SPI_{i+1})\right)}{\sum_{i=1}^{n}\max\left\{T_{exe}(SPI_i), T_{config}(SPI_{i+1})\right\}}$$

In ideal case where both time periods are equal:

$$When\ T_{exe}(SSP_i) = T_{config}(SSP_{i+1}) = T_{timeslot} \tag{5.4}$$

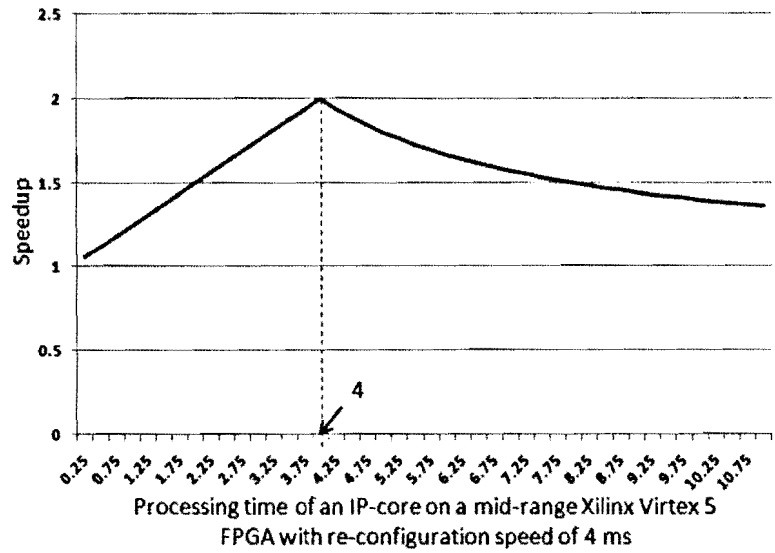


Figure 5.6: Non-Pipelined TPM vs. Pipelined TPM Speed-up

$$Speedup_{TPM^{np}/TPM^{pl}} = \frac{\sum_{i=1}^{n}(T_{timeslot} + T_{timeslot})}{\sum_{i=1}^{n}\max(T_{timeslot}, T_{timeslot})} = \frac{n \times (T_{timeslot} + T_{timeslot})}{n \times \max(T_{timeslot}, T_{timeslot})} = \frac{T_{timeslot} + T_{timeslot}}{T_{timeslot}} = 2 \quad (5.5)$$

As expected, the maximum speedup is achieved when configuration time is equal to the processing time. In all other cases the speedup is less than double for the case of a dual-slot architecture [52]. Figure 5.6 shows the speedup vs. the percentage of configuration overhead.

In Figure 5.6 Y axis represents the speedup of pipelined architecture versus non-pipelined is generated based on equation 5.5. X axis represent the variable execution time $T_{exe}(SSP_i)$ that varies from $250\mu s$ to $10.75ms$. Considering that configuration time T_{config} is constant and equal to $4ms$, it can be seen, that the speedup factor of 2 is reached when $T_{exe}(SSP_i) = 4ms$. At all the other times there is less than 2x speedup either due to configuration overhead, or execution overhead. At the same time, the cost of the system which increases with the added hardware. This introduces the notion of the Cost Performance Ratio (CPR) which is explained in the next section.

5.3.2 Cost-Performance Ratio of RCS with TPM

Cost-performance ratio can be defined as a performance parametric value per cost of the system

$$CPR = \frac{Performance\,parameter\,value}{Cost\,of\,system} \tag{5.6}$$

In general, performance parameter value can be measured as, cycle time, latency of the response (response time), dynamic power consumption, reliability, and other factors that are critical to a particular system specifications. Cost, on the other hand, is most often associated with the cost of system production and associated development costs. This section will compare CPRs of with different architectures, such as: statically configurable, dynamically reconfigurable systems using non-pipelined TPM, and dynamically reconfigurable systems using pipelined TPM.

5.3.2.1 CPR for Single Statically Configurable FPGAs

Most of the processing systems that are designed nowadays are implemented using statically configurable FPGAs. In these systems whole data processing circuits are loaded at the start-up and are not modified throughout the operation of the system. The cost of production of the statically configurable system in a general case can be estimated by:

$$C_{static\,system} = C_{processing\,unit} + C_{peripheral\,components} + C_{pcb} \tag{5.7}$$

Where $C_{processing\,unit}$ is the cost of the FPGA, $C_{peripheral\,components}$ is the cost of peripheral components needed for the platform, and C_{pcb} is the cost of a printed circuit board and its assembling, debugging, and packaging operations. The cost of FPGA should include costs of all required processing elements and IP-cores needed for the application operation. Due to that fact this FPGA is needed to be substantially large. Therefore, large FPGAs would be

Table 5.1: Lists of Xilinx Virtex 4 and Virtex 5 FPGA Costs

Virtex 4 FPGA Device	Cost (USD)	Cost / 1K 4-LUTs
XC4VLX15	$238	$16
XC4VLX25	$330	$13
XC4VLX40	$570	$14
XC4VLX60	$1100	$18
XC4VLX80	$1357	$17
XC4VLX100	$2605	$26
XC4VLX160	$5625	$35
XC4VLX200	$7563	$37

Virtex 5 FPGA Device	Cost (USD)	Cost / 1K 4-LUTs
XC5VLX30	$250	$8
XC5VLX50	$452	$9
XC5VLX85	$1066	$13
XC5VLX110	$1512	$14
XC5VLX155	$2171	$14
XC5VLX220	$3661	$17
XC5VLX330	$8731	$26

Figure 5.7: Video-stream Processing Task Processing Block Diagram

non-proportionally higher in cost than smaller sized models, as shown in Table 5.1. Non-proportional increase of cost effects the CPR of the system as it is shown further in cost-performance comparison.

To illustrate the concept of cost-performance increase consider a video-stream processing system that runs 5 sub-tasks. These sub-tasks depend on each others input, as shown in Figure 5.7. It is also known that each sub-task requires $2ms$ for each processing data, to produce temporal/partial result. All of these processing sub-tasks fit on FPGA device XC4VLX200 which is the largest in the Virtex 4 family. From the initial conditions the frame processing

performance of a statically configured stream processing platform can be calculated:

$$PR_{static\,system} = \frac{1sec}{\sum_{i=1}^{n} T_{exe(i)}} = \frac{1\,sec}{\sum_{i=1}^{5} 2ms} = 100fps$$

where $PR_{static\,system}$ is processing performance of the static system. From this calculation the performance $PR_{static\,system}$ of the statically configured platform is $100fps$. At the same time, the cost of a platform based on the statically configured XC4VLX200 is:

$$C_{ss} = C_{pu} + C_{pc} + C_{pcb} = \$7563 + \$200 + \$100 = \$7863$$

Costs of the FPGA and peripheral components are taken from the average list costs from major parts distributors and PCB manufacturers. By knowing $PR_{static\,system}$ and the overall cost C_{ss} the approximate CPR of such system can be estimated:

$$CPR_{ss} = \frac{PR_{ss}}{C_{ss}} = \frac{100fps}{\$7863} = 0.0127frames/\$$$

For this example the peak performance is not required, but the, designer must take into consideration constraints of other performance parameters. For instance, if for the above example video-processing system constraint is $30fps$, then frame execution period cannot exceed $33ms$. This may allow utilization of a much cheaper FPGA when TPM is employed.

5.3.2.2 CPR for Non-pipelined and Pipelined architectures utilizing TPM

In previous case of statically configured FPGA, the frame processing time was equal to $10ms$, considering all 5 stages. Therefore, if TPM is used, it may be possible to reuse smaller FPGA. When the whole task is subdivided into several sub-tasks and all of them executed one after another within the restricted time the overall system cost can be decreased. Equation 5.8 shows the calculation to be done to verify if particular TPM architecture is acceptable for the application. This is a proposed general equation for calculation of any stream processing

system that is implemented in TPM architecture.

$$\sum_{i=1}^{n} \left(T_{config}(SSP_i) + T_{exe}(SSP_i)\right) \leq T_{cycle} \tag{5.8}$$

Equation 5.8 performs summation of configuration and processing times for every sub-task. This sum has to be less or equal to the T_{cycle} restriction. T_{cycle} restriction is time period between the incoming data frames. For this example the data frames are image frames arriving at $30fps \Rightarrow 33ms$ period from a CMOS sensor. This equation is targeted for a non-pipelined implementation of TPM architecture, since configuration and execution times are always added together. In addition, it is assumed that the task can be split into 5 parts and executed in sequence on FPGA device. This results in 5 times smaller logic resources of FPGA that could be utilized. In the above example (see Table 5.1), instead of XC4VLX200 (cost: \$7563) it may be possible to use XC4VLX40 (cost: \$571). XC4VLX40 also has T_{conf} reconfiguration time of $4.13ms$ which is much smaller than that of XC4VLX200. Based on these values the non-pipelined TPM architecture can be evaluated to see if it is sufficient for the processing task:

$$\sum_{i=1}^{5} (2ms + 4.13ms) \quad = 5 \times 6.13ms = 30.65ms < 33ms$$

As seen from the calculation, the timing requirement for this frame processing is successfully met. At the same time, the CPR calculation shows:

$$CPR_{TPM}^{np} = \frac{PR_{TPM}^{np}}{C_{TPM}^{np}} = \frac{\sum_{i=1}^{n} \left(T_{config}(SSP_{i+1}) + T_{exe}(SSP_i)\right)}{C_{processing\,unit} + C_{peripheral\,components} + C_{TPM} + C_{PCB}} = \frac{\frac{1}{30.65ms}}{\$570 + \$200 + \$100 + \$120} = 0.033 frames/\$ \tag{5.9}$$

Thus, from this simplified example one can see that CPR^{np}_{TPM} is about 3 times higher than the CPR_{SS}, where the overall cost drops from \$7863 to just \$990, which is about 8 times less.

The reconfiguration components that deliver bitstreams after every sub-task is computed should be included in the cost of a TPM system. The cost of the hardware or RCS with TPM is calculated based on the Equation 3.7 from Chapter 3.

A pipelined architecture can be analyzed in the same manner on non-pipelined TPM architecture. In calculation of the system hardware overhead, the four main costs associated with it as it was done for C^{np}_{TPM} back in Equation 3.7. For the pipelined case, the cost of the FPGA device C_{pu} and cost of peripheral components for that device $C_{pu\,pc}$ are added. For generalized expression of more then one processing device the number of processing units is parametrized by N_{pu}. Since every FPGA has associated peripheral components, they also have to be accounted for in cost calculation and denoted as $C_{pu\,pc}$. The overhead equation is shown below:

$$C^{pl}_{TPM} = C_{peripheral\,components} + C_{TPM} + C_{PCB} + N_{processing\,units} \times (C_{processing\,unit} + C_{pu\,periph\,comp}) \tag{5.10}$$

For the pipelined case, in contrast to non-pipelined the time calculation is considered to be maximum value out of $T_{config}(SSP_{i+1})$ and $T_{exe}(SSP_i)$. Therefore, by keeping all the conditions from the previous non-pipelined example:

$$T_{cycle} \geq \sum_{i=1}^{n} \max(T_{config}(SSP_{i+1}), T_{exe}(SSP_i)) = \sum_{i=1}^{5} \max(4.1ms, 2ms) = \sum_{i=1}^{5} 4.1ms = 20.5ms \tag{5.11}$$

As shown, pipelining performance of the system is increased:

$$PR^{pl}_{TPM} = \frac{Unit\,Time}{\sum_{i=1}^{5} \max(T_{config}(SSP_{i+1}), T_{exe}(SSP_i)} = \frac{1sec}{20.5ms} = 48fps \qquad (5.12)$$

If we calculate CPR based on these values we will observe that CPR is increased to:

$$CPR^{pl}_{TPM} = \frac{PR^{pl}_{TPM}}{C_{pl}} = \qquad (5.13)$$

$$\frac{\frac{1sec}{\sum_{i=1}^{n} MAX(T_{config}(SSP_{i+1}), T_{exe}(SSP_i))}}{C_{peripheral\,components} + C_{TPM} + C_{PCB} + N_{processing\,units} \times (C_{processing\,unit} + C_{processing\,unit\,peripheral\,components})} =$$

$$= \frac{\frac{1sec}{\sum_{i=1}^{5} 4.1ms}}{\$150 + \$100 + \$100 + 2 \times (\$570 + \$50)} = 0.03fps/\$$$

where N_{pu} is number of processing units used on the platform. There are 2 FPGAs in this example. Obtained CPR result is the same as in the previous non-pipelined TPM. This is not surprising, since our requirements are $\sim$1.5 times lower than performance obtained with pipelined solution.

The, requirement of the processing system is $30fps$ allows for further increase in divisions of the algorithm to sub-tasks and, therefore for a, potential selection of a smaller device. If processing algorithm is divided into 8 segments (provided algorithm is divisible onto 8 segments) an FPGA device of even smaller size can be selected, such as XC4VLX25. Again, to satisfy the T_{cycle} restriction the calculation has to be redone. In this case though, the configuration time for the FPGA is $2.7ms$, according to the Table 3.3. T_{cycle} restriction is still satisfied, since time required for reconfiguration is 2.744ms and, therefore, for 8 divisions $T_{cycle} \geq \sum_{i=1}^{8} 2.744 = 22ms$. On the other hand, CPR is increased, since the cost of a smaller FPGA device is lower

$$CPR^{pl}_{TPM} = \frac{PR^{pl}_{TPM}}{C^{pl}_{TPM}} = \qquad (5.14)$$

$$\frac{\frac{1sec}{\sum_{i=1}^{n}\max(T_{config}(SSP_{i+1}),T_{exe}(SSP_i))}}{C_{pc}+C_{TPM}+C_{PCB}+N_{pu}\times(C_{pu}+C_{pu\,pc})} =$$

$$\frac{\frac{1sec}{\sum_{i=1}^{8}2.744ms}}{\$150+\$100+\$100+2\times(\$330+\$50)} = 0.041 fps/\$$$

This result is a $0.011 fps/\$$ improvement from the 5 sub-task division scenario. In addition, the above result shows that CPR is increased as $T_{config} \approx T_{exe}$ is getting closer to configuration time.

When comparing different architectures of TPM organizations between each other, CPR comparison formula should be used, since it is the true indicator of solution efficiency. For 8 and 5 task divisions CPR ratio gives:

$$Ratio_{CPR} = \frac{CPR_{8div}}{CPR_{5div}} = \frac{0.041}{0.03} = 1.36 \tag{5.15}$$

If task divisions can be balanced even more to achieve $T_{config}(SSP_{i+1}) \approx T_{exe}(SSP_i)$, a higher CPR can be achieved.

These results might give an impression that increasing division of algorithm indefinitely always increases CPR of the system, but it is not true, and there is a limit of division which is optimal for a system. One of the reasons why CPR does not increase indefinitely is the actual process of division. There are always constraints and some of them do not permit the division of a task. A more apparent factor is the increase of N_{pu}s. From the above equation a balanced implementation produces a CPR increase. There is strong dependence of the percentage cost of FPGA device to the cost of the whole system. Increasing the number of FPGAs devices on RCS does not give a linear increase in CPR. When designing a processing system, where cost-effectiveness is important, the system has to be evaluated based on the above equation, and not solely by the performance increase.

As seen from the example, employing TPM can result in significant cost savings as CPR of the system is increased. The reason for the increase in CPR is due to the non-linear relationship

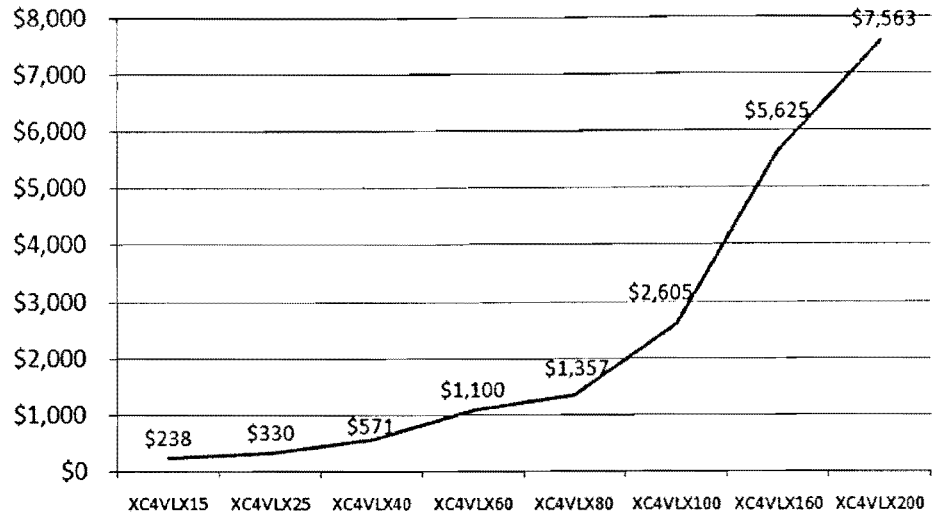


Figure 5.8: Xilinx Virtex 4 FPGA Device Costs in Relation to Logic Resources

of size of FPGA and its price. Table 5.1 contains the costs of latest Virtex 4, and Virtex 5 FPGA families, as well as their costs per 1K logic cells. Figure 5.8 depicts the non-linear relationship of increasing logic and cost. The key advantage is to use the reconfigurable devices in the range where cost and size of the device increasing linearly, up to XC4VLX80 FPGA.

From the Figure 5.8 it can be seen that Virtex 4 family has exponentially higher costs for their largest devices in comparison to their mid-range ones such as XC4VLX40 and XCVLX80. As with the Xilinx Virtex 5 devices, as well as Altera Stratix III family, and this trend is true for most of FPGA vendors and their FPGA families. Graphs representing device-cost relation are shown in Figure 5.9 and Figure 5.10.

This is not too surprising, since yield on larger sized dies is much smaller than on smaller sized dies. The overall cost of manufacturing is, therefore, disproportionally higher [72]. Yield is calculated as a function of an average number of defects (D) per unit area (A).

$$Y = f(A, D) \tag{5.16}$$

Since defects are uniformly distributed across an IC die, the increase in the area of IC will be directly proportional to the probability of defects that the IC can receive.

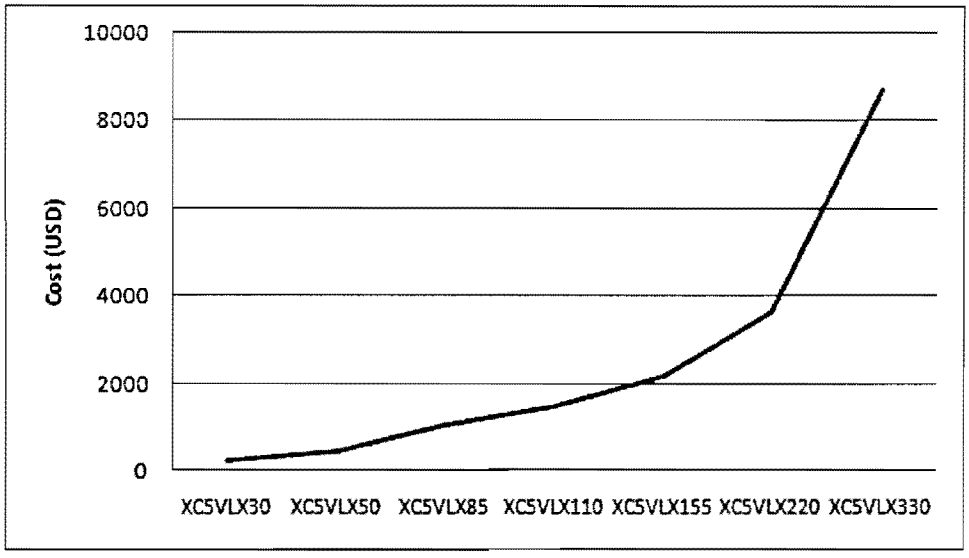


Figure 5.9: Xilinx Virtex 5 FPGA Device Costs in Relation to Logic Resources

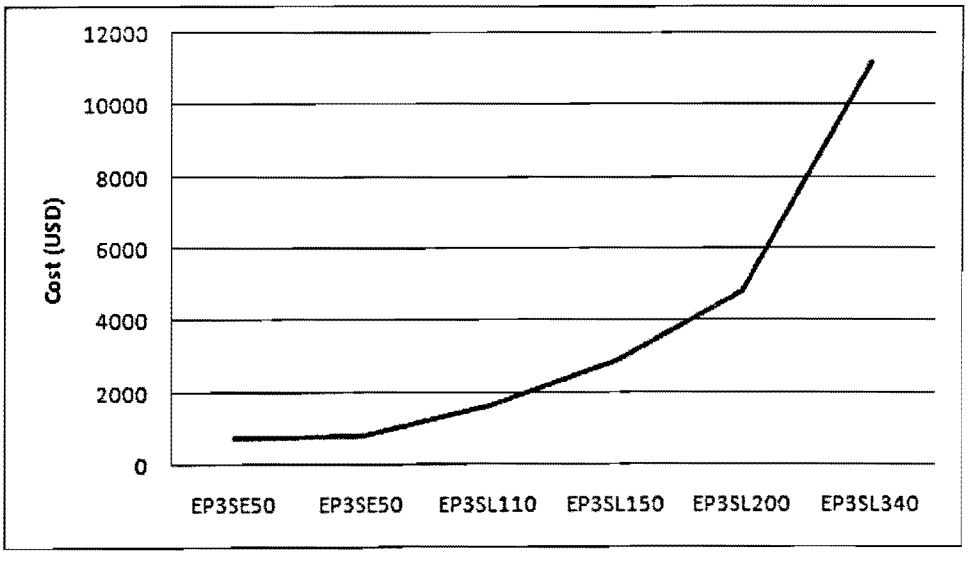


Figure 5.10: Altera Stratix III FPGA Device Costs in Relation to Logic Resources

From the graph a range of devices that are in near linear range can be identified and by utilizing them instead of larger FPGAs, the CPR can be increased.

Selection of the FPGA device for TPM approach depends on several constraints. If those restrictions are not met, a larger device has to be selected, and TPM approach is not suitable for that particular system implementation. First, the size of the device put a restriction on the number of divisions of the algorithm

$$\frac{DS_{full}}{N_{div}} = \lceil DS_{part} \rceil \quad T_{conf} = \frac{DS_{part}}{BW_{conf}} \tag{5.17}$$

$$DS_{part} = \max\{DS_{div_i}\}$$

Where DS_{full} is the size of a reconfigurable device needed for implementation of all sub-tasks of the processing application on one device, N_{div} is the number of balanced divisions of the task, and DS_{part} is the size of the closest fitting FPGA device to be used for the TPM architecture. When device is selected it has to satisfy Equation 5.8. If restriction is not met, a number of divisions has to be adjusted. A larger/smaller device has to be selected accordingly and again re-evaluated by Equation 5.8. This selection can be automated with a CAD software that would perform the balancing operation and provide user with several solutions from which the most suitable one can be selected.

Additional benefit that is obtained by implementing the design on a smaller size FPGA is an ability to do module design by parts, and thus not to worry about overall timing. This fact is currently a growing problem on large FPGAs for developers, since a combination of separately designed modules leads to timing problems [100]. These timing issues further complicate the design and makes the final system even more costly. Another important reason why TPM approach is beneficial for implementation of embedded systems is a smaller latency and, therefore a higher operation speed. Utilizing smaller sized FPGA makes internal routing

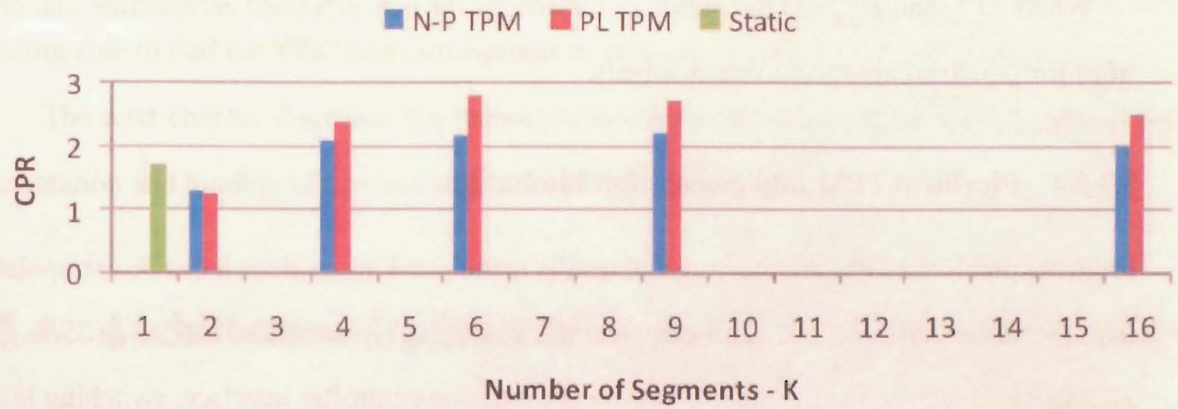


Figure 5.11: Cost Performance Ratio vs. Task Segmentation Granularity

shorter, which allows signals to travel faster from pad to pad and makes the critical delay shorter overall.

5.3.2.3 Optimal Number of Partitions

When analyzing system design using temporal partitioning approach there is a factor of division of algorithm into segments, which has an optimal region where it is most cost-effective. When too many divisions of an algorithm are introduced, a reconfiguration overhead would cause processing stall even in pipelined form. Configuration overhead involves reading previously processed temporal data and writing newly processed data. Figure 5.11 shows that with more than 8 reconfigurations, the CPR of the system drops.

In order to guarantee that the processing of an algorithm would be completed within the T_{cycle} time maximum number of divisions has to be calculated by following formula for non-pipelined TPM approach:

$$D_{div}^{np} = \left\lfloor \frac{T_{cycle}}{T_{config} + T_{exe}} \right\rfloor \tag{5.18}$$

Similarly, for pipelined TPM approach:

$$D_{div}^{pl} = \left\lfloor \frac{T_{cycle}}{\max(T_{config}, T_{exe})} \right\rfloor \tag{5.19}$$

Where D_{div}^{np} and D_{div}^{pl} are the number of divisions for non-pipelined architecture and divisions for pipelined architecture respectively.

5.3.2.4 Pipelined TPM Implementation Limitations

From the pipelined TPM architecture and results in the previous sections it can be extrapolated that system can have $N_{pu} > 2$. This can potentially mitigate issues related to $T_{config}(SSP_{i+1}) > T_{exe}(SSP_i)$. However, the overhead costs of configuration controller interface, switching logic, PCB area, PCB routing complexity, and actual FPGAs would become much more dominant, and make CPR much lower. At the same time, there will have to be a re-design and an increase in bandwidth for configuration controller, since multiple FPGAs will have to be configured at the same time. The automated scheduling and timing of the configuration bitstreams, which is handled by the configuration manager, would be greatly increased in complexity. Similarly, the costs associated with software/firmware development are tremendously increased due to the configuration manager complexity. All these factors would effect CPR of the system and CPR would decrease. Therefore, in this research only $N_{pu} = 2$ was considered.

5.3.3 Automated Partitioning

Traditionally, the task of algorithm partitioning and creation of IP-cores was done manually by the designer. However, this requires a lot of effort, as well as highly experienced engineers. Designer usually performs the division of an algorithm into segments, as well as the generation of reconfigurable cores from those segments. The transfer of temporary data between the cores has to be analyzed and accounted for as well. Such a task was doable when FPGA devices were small in logic size [88], such as an early XC4000 Family. Current Virtex 4, Virtex 5, and Virtex 6 FPGAs are too large for manual implementation and CAD tool with automated partitioning is necessary. An automated partitioning capability would provide a much faster segmentation of an algorithm in close to optimal fashion. Automated partitioning tool would

bind the segments based on two conditions: meeting the FPGA device size requirements and being able to find the VHC that corresponds to that requirement.

The next chapter discusses the technique developed for close to optimal algorithm segmentation and binding of resources to meet the system constraints.

5.4 Architecture Optimization for ASP Based on Configurable Modules

Processing platforms can perform different types of processing. Considering media applications such as Image, Video, and other digital signal processing where there is a constant stream of data, a stream processing. Stream processing is: processing of a constantly incoming data using the same algorithm/procedure, where procedure, as well as the format of data doesn't change throughout the operation of the system. Stream processing allows for faster and more efficient execution by sacrificing the flexibility of algorithm modification. Stream processing is well suited for applications that share three main characteristics: data locality, data parallelism, and computational intensity. In addition, if the processed data is used once or twice and then discarded or saved in storage, it is possible to process the constantly incoming data in a sort of a pipeline. After a pipeline frame fill-up latency, the system produces a result with a period $MAX(T_{MO}(i))$. Where $MAX(T_{MO}(i))$ is the maximum delay of a largest MO component of the IP-core. This provides tremendous speedup for any type of processing, especially if many pipeline stages are present. At the same time, if it is possible to acquire input data in more parallel fashion, where more than one data is available, parallel processing can be done and several stream processing algorithms working at the same time. This increases the speedup of the processing system even more. The exact factor of performance increase would roughly depend on processing algorithm multiples. Following chapters will show the optimal uses of proposed research and how it compares in cost-efficiency to the non-TPM

approach. It will also address limitations of the proposed approach and ways of overcoming these limitations.

5.5 Summary

This chapter presented the concept of task segmentation and the process of tasks execution on a reconfigurable system. The main focus of the chapter was to convey the idea of balancing the requirement for the system to achieve most cost-efficient design solution. These requirements also impact other aspects of design, such as power consumption and speed of operation. Notion of cost-effectiveness of RCS with TPM was introduces with evaluation of system's cost performance ratio while taking in account cost of the systems components and overall performance. It was shown that hardware architecture a components should be closely related to the system requirements to achieve cost-effective solution. The methodology for evaluation of cost-effectiveness of RCS with TPM, described in this chapter, was presented on the Canadian Conference of Electrical and Computer Engineering [52]. This research was used in further methodologies and segment specific processor synthesis. Next chapter will describe the methodology behind creation of the high level Macro Operators (MO) and corresponding Virtual Hardware Components (VHCs).

Chapter 6

Methodology for High-Level Synthesis and Optimization of VHCs

6.1 Introduction

In the previous chapter a notion of TPM architecture was introduced. General overview of the TPM approach and the concepts of MOs, VHCs, and SSPs, was presented. Chapter 5 also described the limitations of the TPM architectures and the conditions where this approach is most applicable. An idea of cost-effectiveness was introduced, that is used to match the architectural design to the system requirements and minimizing the overall cost of the system. The core of the this research is synthesis of the optimal SSPs from VHCs.

This problem consists of two parts:

1. High level synthesis and optimization of VHCs in a multi-parametric design space.
2. Compilation (combining) of an SSP from a set of optimized VHCs.

This chapter focuses on a methodology of high level synthesis of VHCs according to MOs, the methodologies of SSP compilation is discussed in the next chapter.

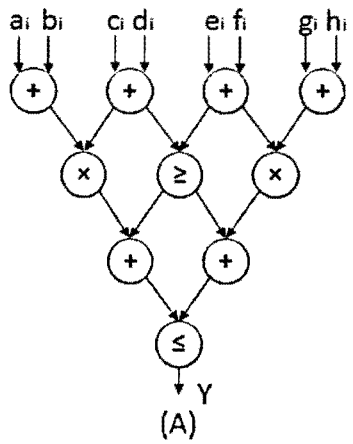
The problem is divided into two parts: given a particular algorithm, how to create a set of VHC that correspond to an MO, and how to find VHCs so that all of the system requirements are met (e.g. timing, area). Decision making mechanism is similar for both parts, however, selection process is different. The reason is because VHC synthesis is based on elementary operators and the SSP compilation uses macro-operators (MOs) to perform architecture selection process. The analogy can be made with the software development, where one uses elementary programming functions to create various complex functions. In turn, the complex functions are used to implement an application that performs a particular task.

Therefore, a methodology for VHC synthesis from the elementary operators has to be developed first. The main focus of this chapter is description of methodology for MO analysis and generation of the associated set of VHCs according to the set of parametric constraints.

6.2 Correspondence Between MO and VHC

As was discussed in the previous chapter, an MO, is a mathematical representation of a function used within a task. A VHC is an implemented instance of an MO. In this section the correspondence between MO structure and architecture of VHCs will be discussed with influence of parametric constraints in mind.

First of all, for each MO_i,$i = 1,2,\ldots n$, it is possible to find a set of $VHC_{i,j}$ $j = 1,2,\ldots k$, where each of $VHC_{i,j}$ is optimized for the set of parametric constraints P_s, $s = 1,2,\ldots p$. For example for the MO_i -"Matrix Multiplication", the set of parameters (e.g. P_1—"execution time"-T_{exe}; P_2—"logic resources"-R_{log}; P_3—"Power consumption"-W, etc...) may be assigned. Therefore, for certain requirements and constraints of the above parameters different $VHC_{i,j}$



Operation	Latency
×	10 ns
+	2 ns
≤	4 ns

(B)

Figure 6.1: Sequencing Graph (SG) for the MO Represented by Equation 6.2

architectures can be synthesized:

$$\begin{aligned} &For\ MO_i\ and\ \left\{T_{exe} < T_{exe_{lim1}};\ \min\left\{R_{log}\right\};\ W_{in} < W_{in_1}\right\}_1 \rightarrow VHC_{i,1} \\ &For\ MO_i\ and\ \left\{T_{exe} < T_{exe_{lim2}};\ \min\left\{R_{log}\right\};\ W_{in} < W_{in_2}\right\}_2 \rightarrow VHC_{i,2} \\ &\qquad\qquad \ldots \\ &For\ MO_i\ and\ \left\{T_{exe} < T_{exe_{lim\,j}};\ \min\left\{R_{log}\right\};\ W_{in} < W_{in_j}\right\}_j \rightarrow VHC_{i,j} \end{aligned} \tag{6.1}$$

The above derivation shows that for a single MO_i it is possible to have j number of VHCs. Each $VHC_{i,j}$ has a variation by one or more parametric constraints. Each parameter can have different limitations that can not be exceeded (e.g. $T_{exe_{lim\,j}}$, W_{in_j}) for each version of VHC. Performance parameters contradict to each other. For example, an amount of logic used for implementation of the algorithm is inversely proportional to the processing time. To illustrate this fact, consider an example from Figure 6.1. Figure 6.1 (A), represents the Sequencing Graph (SG) implementation of the MO, which is based on the Equation 6.2.

$$\begin{aligned} Y = \max\{&[(a+b)\times(c+d)+\max\{(c+d),(e+f)\}], \\ &[\min\{(c+d),(e+f)\}+(e+f)\times(g+h)]\} \end{aligned} \tag{6.2}$$

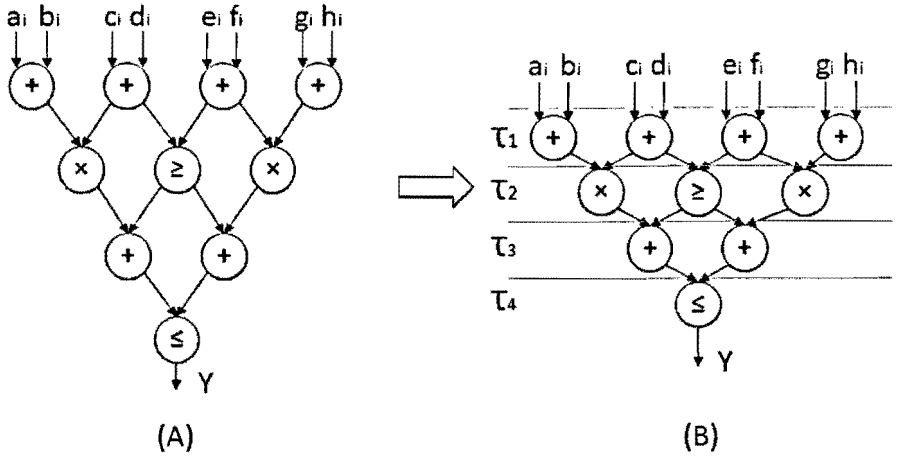


Figure 6.2: Stage Divided Sequencing Graph for the MO Execution

As it can be seen from the Figure 6.2 (A) the MO algorithm consists of 10 elementary operations: six additions, two multiplications, and two comparisons. Based on the availability of the associated operators (e.g. adders, multipliers, comparators, etc.) operator's use can be achieved by binding and scheduling procedures [79]. Variations of VHC implementation can be generated from combinations of resource binding and scheduling. If all of these resources are available at the same time, the processing latency for a single set of data $(a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i)$ will be minimal. However, in case of the stream processing applications the cycle time is reduced to close to the execution time of the slowest elementary operator (e.g. multiplier). Considering an example above, the latency of the algorithm can be calculated. Since the nodes of the SG depend on each other, the MO is divided in four stages, as shown in Figure 6.2 (B).

The formula for latency calculation has to consider all of the stages of the MO. A general formula is given by:

$$T_{latency} = \sum_{i=1}^{m} \tau_i \; i = 1,2,\ldots,t \tag{6.3}$$

$$\tau_i = \max\{\forall t_{exe} \in \tau_i\} \tag{6.4}$$

Where m is the number of stages in the SG and τ_i is the latency of the particular stage. The latency of each τ_i is found by determining the operation with the maximum execution time - t_{exe}, since it becomes the bottleneck of the processing stage. Therefore, the total latency of the MO from Figure 6.1 (B) is equal to:

$$T_{latency} = \tau_1 + \tau_2 + \tau_3 + \tau_4 = 2ns + 10ns + 2ns + 4ns = 18ns$$

If a large set of data (e.g. a_i, b_i, c_i, d_i, e, $k = 10^6$ sets) is computed sequentially in a non-pipelined fashion where $T_{latency} = T_{exe}^{np}$, it would take:

$$T_{exe} = k \times T_{exe}^{np} = 18 \times 10^6 = 18ms$$

If, however, the algorithm is implemented in a fully pipelined data-path, the cycle time per data set ($a_i, b_i, ..., g_i, h_i$ for each $i = 1, 2, ..., k$) over a large amount of data may be decreased. It should be noted that for fully pipelined implementation there is a much higher requirement for logic resources, as illustrated later. Equation for cycle time calculation is:

$$T_{cycle}^{pl} = \max(\forall \tau_i \in MO) \tag{6.5}$$

Where cycle time of fully pipelined implementation is equal to the largest latency of the stage in a pipeline. Equation 6.6 reflects the pipeline speedup for the large sets of data on the pipelined data-path.

$$T_{exe} = T_{latency} + T_{cycle}^{pl} \times (k-1) \tag{6.6}$$

where k is the number of data sets and T_{exe}^{pl} (of fully pipelined data-path) shows the execution time per one data set. Below is the calculation of the execution time using Equation 6.6, when a fully pipelined data-path is utilized for the SG in Figure 6.2 (B). Here, $k = 10^6$ data sets:

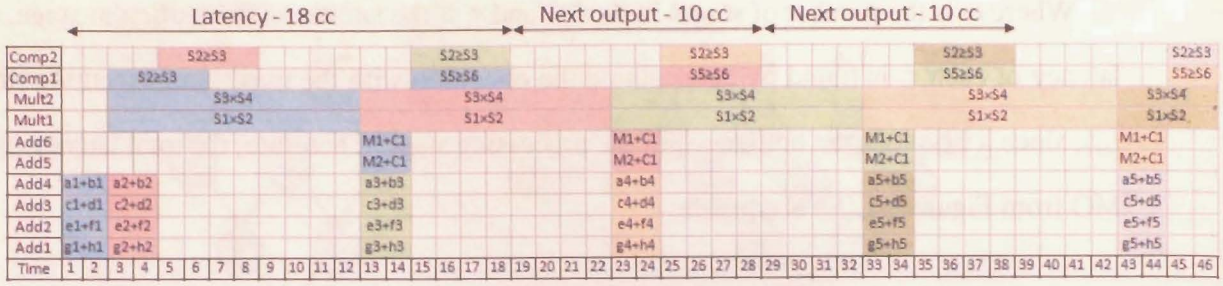


Figure 6.3: Pipelined Implementation of SG from Figure 6.2

$$T_{exe} = T_{latency} + T_{exe}^{pl} \times (k-1) = 18ns + (10^6 - 1) \times 10ns = 1000008ns \approx 10ms$$

As can be seen from the result, it almost cuts in half the total execution time T_{exe} when fully pipelined architecture used on large sets of data, however, it does come with an additional hardware cost. Figure 6.3 shows the detailed resource utilization based on the schedule of pipelined data-path implementation. This schedule shows that the initial pipeline fill-up latency and subsequent pipeline outputs correspond to the equation calculations.

The previous case was an extreme one where such amount of resources was available that permitted to create a fully pipelined implementation. In other cases, only limited amount of resources is available, so binding and scheduling come into effect. The following two cases illustrate the methodology of operation.

In first scenario let us consider that only one adder (A), one multiplier (M), and one comparator (C) are available for operation at any given time. Based on the dependency of the sequential graph in Figure 6.1, we can bind and schedule SG as shown in Figure 6.4. Since only one instance of resource (e.g. A, M, and C) is available at a time, elementary arithmetic/logic operations (e.g. additions, multiplication, comparison) are bounded by the resource represented by the dotted line. As it is seen in Figure 6.4, every bounded resource is scheduled only once per processing stage. This results in seven stages execution process. Latency for

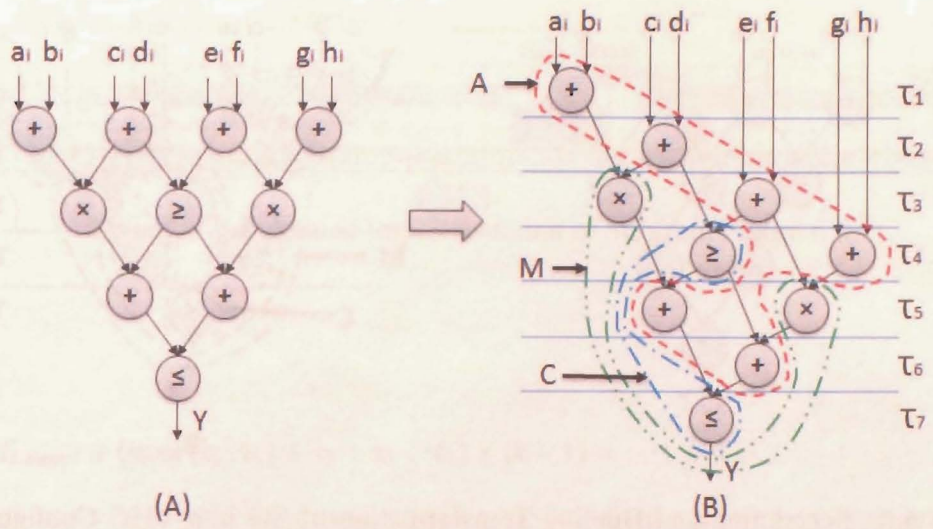


Figure 6.4: Scheduling and Binding Transformation of SG into VHC Configuration Including Single: Adder, Multiplier, and Comparator

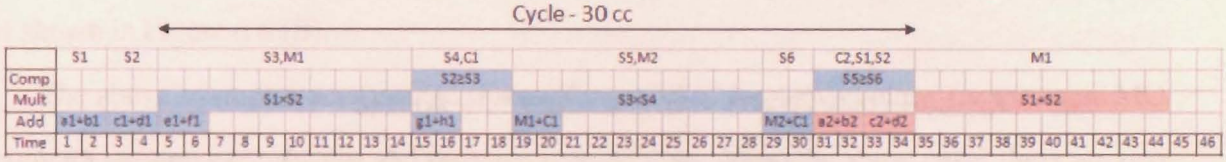


Figure 6.5: Pipelined Implementation of SG from Figure 6.4

this implementation of the algorithm is calculated by Equation 6.3, same way as the sum of latencies for all the stages was computed. The overall latency is equal to:

$$T_{latency} = 2ns + 2ns + 10ns + 4ns + 10ns + 2ns + 4ns = 34ns$$

The cycle time is less than latency as before. However, it is much higher than in previous implementation due to limitation in resources. Figure 6.4 illustrates the schedule of resources after the binding operation.

The schedule latency is now equal to $34ns$, and cycle time is $2ns$ less, since stages τ_1 with τ_7 can be overlapped. Hence, equation for calculation of 10^6 data sets, considering T_{cycle} time

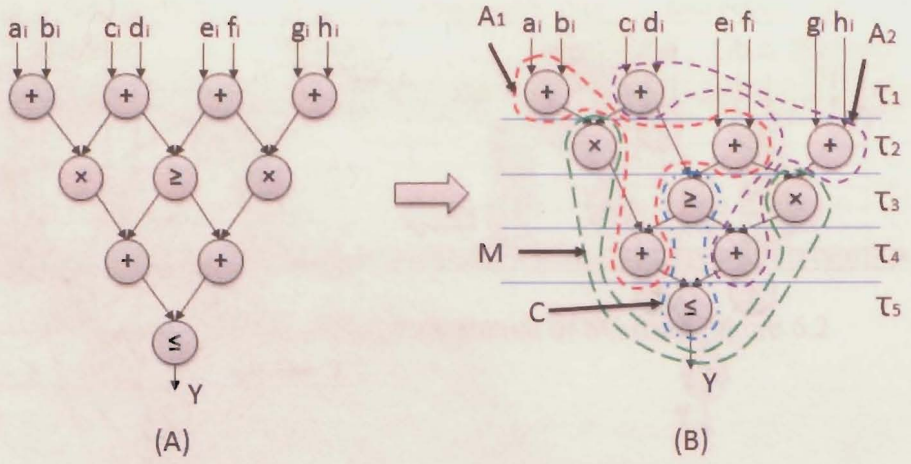


Figure 6.6: Scheduling and Binding Transformation of SG into VHC Configuration using Double Adder, Single Multiplier, and Single Comparator

of overlapped τ_1 and τ_7, is:

$$T_{exe} = T_{latency} + (\max\{\tau_1, \tau_7\} + \tau_2 + \tau_3 + \tau_4 + \tau_5 + \tau_6) \times (k-1) =$$

$$34 + (\max\{2,4\} + 2 + 10 + 4 + 10 + 2) \times \left(10^6 - 1\right) = 32000002ns \simeq 32ms$$

In some applications (e.g. real-time high speed image processing) this execution time may not be acceptable, and an increase in resources may be required.

Now consider a case where two adders are available. There are one multiplier and one comparator as in previous case. A different binding and scheduling results in this case.A1 and A2 represent the two adders in Figure 6.6. Each of these adders bind three "add" operators. Since these adders operate in parallel, they can be scheduled in the same time slot (e.g. $\tau_1, \tau_2, \ldots \tau_n$). This allows to decrease the overall number of time slots stages to 5. The overall latency is

$$T_{latency} = 2ns + 10ns + 10ns + 2ns + 4ns = 28ns$$

The data throughput, as in previous case, is slightly lower than the latency, since τ_1 and τ_5 can be overlapped in cycles (see Figure 6.6). In this case:

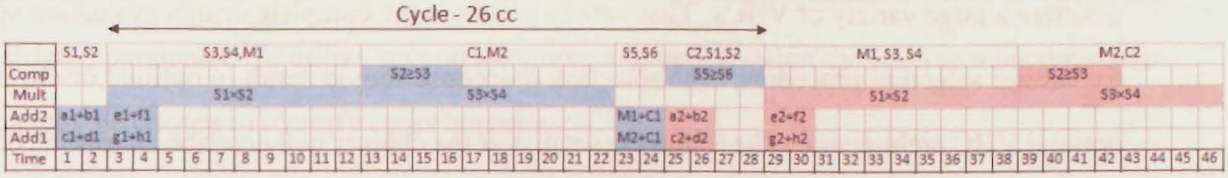


Figure 6.7: Pipelined Implementation of SG from Figure 6.6

$$T_{exe} = T_{latency} + (\max(\tau_1, \tau_5) + \tau_2 + \tau_3 + \tau_4) \times (k-1) =$$

$$28 + (\max(2,4) + 10 + 10 + 2) \times \left(10^6 - 1\right) = 26000002ns \simeq 26ms$$

The schedule showing the resource utilization, latency and cycle time of the scheduled SG is shown in Figure 6.6 (B).

The above examples demonstrate three different VHC implementations for the same MO but with varying performance and resource requirements. From the above cases there are two possible extremes can be identified as:

a) non-pipelined data-path, with a single resource for each type of operation. This is the slowest and cheapest design solution.

b) Fully pipelined data-path, with individual resources available for every operation. It is the fastest and the most expensive solution for VHC implementation.

Between these extremes there are VHCs with different latencies and cycle times, as shown in Figure 6.6. VHCs presented in the above cases and as well as others, create a design space of VHCs, associated with the MO. Each VHC implementation is an element (point) in this design space.

In most real life cases the number of possible variants of VHC implementation (design space size) is very extensive (e.g. from 10^3 to10^{10} variants). It is obvious, that with these numbers the process of high-level synthesis of VHCs and sets of parametric constraints must be automated. By automating the process of resource binding and scheduling it is possible to

generate a large variety of VHCs. That variety produces the complete design evaluation space for further selection procedure. The selection procedure should result in optimal or close-to-optimal VHC architecture. The following section of the chapter will describe all aspects of the novel methodology which allows rapid selection of near-optimal variant of VHC architecture for a given SG of the MO and a set of parametric constraints.

6.3 The Problem of VHC Synthesis and Optimization

The synthesis of optimum set of VHCs from elementary operators, corresponding to particular MO, is essentially an optimization problem. Such problem is NP-complete, especially for cases with multi-parametric optimization. There can be hundreds of nodes in an SG for a given MO and an enormous set of various combinations of resource binding and scheduling. Thus, it may require exponentially growing amount of calculations to perform the exhaustive search of all the variants of VHCs in the design space. These calculations have to estimate the processing time, power consumption, area of logic resources, and other performance parameters.

Selection of the optimal set of VHCs for corresponding MOs may be done using different approaches, as presented in [29, 47, 103]. One of these approaches is to use the Pareto point set, where every VHC variant has a Pareto point [79]. Pareto points, therefore, can be considered as trade-off points in the system design space.

In the past several years various heuristics were proposed by researchers, however, they mostly considered small sets of MOs. In cases of large sets of MOs, the calculations may become unmanageable due to exponential increase in number of the calculations. In addition, most of researchers have not consider the multi-parametric restrictions [47] [103]. If only one restricted parameter is considered, the optimization is done only in relation to that parameter. This, would not be feasible in real designs where many parameters typically have to be considered, such as: latency, area, power consumption. The, conventional methods do not

have capabilities and flexibility to optimize an algorithm implementation with more than one restricted parameter. To utilize several parameters in optimization the inverse related parameters may be chosen to identify cross boundaries.

A "Spacewalker" method was proposed by [103] for design space exploration. It makes use of he information from the previous point of the design space to minimize the search area. However, this creates a problem of having local extremes as a solution in final selection on the design space. Selection of local extremes eliminates all other branches and can lead to missing the global minimum or maximum of the parameter in question. In this case a genetic algorithm search would be more applicable, that search as more than just neighboring nodes [61].

In other papers [47] an automated selection of VLIW architecture was performed by decomposing system architecture to sub-systems. It is possible to reduce the number of variants to be estimated by the decomposition of hierarchical design. However, complex models associated with VLIW architecture specifics make method very computationally intensive and difficult in implementation for other types of architectural synthesis and optimization.

In general, the effectiveness of any method based on heuristics always depends on quality of these heuristics and their orientation for a certain application. In the case, where MO specific data-path circuits have to be designed, the formal method will have non-NP complexity. This method must provide the maximum possible reduction of numbers of VHC variants to be evaluated. At the same time this method in should result the near-optimum solution and avoid the local extremes of the performance parameters in the design space. Therefore, a different approach based on the design space arrangement has been selected by [49]. This approach proposes a partial arrangement of the design evaluation space as a methodology to minimize the number of variants needed for selection of the close-to-global optimum. The method assumes representation of the design space in a form of a decision tree called Architecture Configuration Graph - ACG. An example of such ACG is shown in Figure 6.8.

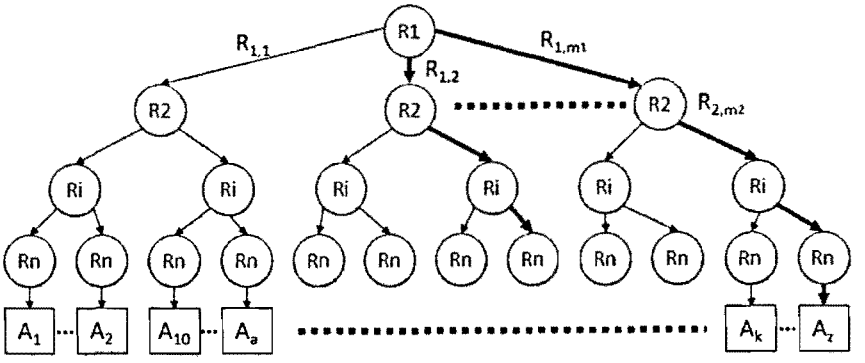


Figure 6.8: Design Space Arrangement

Each node on ACG resents a resource type. Resources in this example can be various arithmetic/logic operators (e.g. $R1$-"Adder", $R2$-"Multiplier, etc.). Each resource can have several variations of implementation. Variants of resource implementation are represented by the edges associated with certain nodes. For example, $R_{1,1}$ is "Adder" available in one instance. Whereas $R_{1,m1}$ is the "adder" available in 6 units of adders. Therefore, all possible configuration of the VHC_i for certain MO_i can be represented by the ACG terminals: $A_{1,},A_{2,}...,A_Z$. For example, the A_1 configuration represents the path from the root ($R_{1.1}$) to the R_n (e.g. $R_{2.1}$). In this case A_1 means that there are minimum resources available for the data-path: one adder, one multiplier and one comparator. This variant is similar to the example discussed in Section 6.2, Figure 6.6 (B). On the other hand, the architecture configuration A_z represents the VHC variant with the maximum possible resources available for the data-path: 6 adders, 2 multipliers and 2 comparators. This is the previously considered example of fully pipelined data-path, shown in Figure 6.2 (B).

All variants $(A_1,A_2,...A_z)$ of a resource can be arranged in ascending order by one of the performance parameter and/or by descending order for the contradictory parameter [51]. Such parameter pairs could be execution time and data rate, power consumption or dissipation, etc. An ideally monotonic arrangement of performance parameters allows to find the global extreme located in case of ascending order at the very right terminal/leaf the graph (e.g. A_Z) or vice versa. Nonetheless, the above methodology, though being able to reduce the number

of variants to be evaluated for finding close-to-optimal architecture, has one, but important limitation: it is mono-parametric. In other words, it allows finding the optimal variant for one performance parameter satisfying the constraint of another contradictory parameter (e.g. variant with highest data execution rate and certain logic resources available). Real design process usually coincides more than two parameters. Therefore, an extension of this method is needed for the design process that requires with multi-parametric design optimization. Thus, the goal of the further research is to develop a methodology for selection of a close-to-optimal variant of VHC in a multi-parametric design space while evaluating the minimum number of possible variants.

6.4 Methodology of VHC Synthesis and Optimization

An appropriate method for high-level synthesis and optimization of VHC architecture should provide:

1. A solution for general multi-stage, algorithm specific data-paths.
2. Minimization of number of variants to be evaluated for finding the optimal configuration of VHC.
3. Close-to-global extreme solutions (avoiding local extremes) for each performance parameter.
4. Pareto-optimal variant of VHC configuration in multi parametric design space.

The next section describes all aspects of the developed novel methodology of selection of close-to-optimal variant of VHC in the multi-parametric design space.

6.4.1 Multi-parametric Design Space Decomposition

The first step for the VHC synthesis procedure is determination of performance parameters and their constrains. These parameters are used to create the design sub-spaces that are used to select optimal variants (Pareto-points) for each of the parameters. Let performance parameters be denoted as P_s, where $s = 1,2...p$, and assume that there are constrains for each $P_s \rightarrow P_{s_{lim}}$. It is possible to find many pairs of inversely proportional performance parameters (e.g. processing latency and amount of logic resources). For example, if a designer needs to reduce the processing time latency, then the amount of hardware/logic has to be increased. The decrease in processing time may also be inversely proportional to power consumption. These types of parameters can be chosen for selection of the design space represented by ACG. At the same time, these parameters have corresponding constrains, $P_{S_{lim}}$. Design constraints limit the design space and, consequently, decrease size of the ACG. However, with the addition of third parameter (e.g. hardware area), the architecture design space becomes three-dimensional and complicates the problem of selection. The third parameter can be inversely related to the speed of computation, and close to linearly proportional to the power consumption. Hence, design space selection problem becomes a double two-parameter design sub-space selection problem. Where it has Power vs. Time and Area vs. Time, instead of three parameter design space selection problem. It is less computationally intensive, to solve two two-dimensional problems instead of one three dimensional. It is also known that $n^3 \gg n^2 + n^2$ for $n > 2$. Thus, selection of the inversely related parameters should be done by the designer. Designer also has to determine the initial restriction specification of parametric constrains, $P_{s_{lim}}$ for $s = 0,1,...,p$. Later these constrains will be used in narrowing design space by pruning of ACG.

After the number of restriction parameters is selected according to all of the constraints, a reduction of the ACG can be performed. As it has been shown in [49], ACG can be pruned by arranging the ACG in a descending/acceding manner and applying m-airy search procedure. The Figure 6.9 illustrates this process.

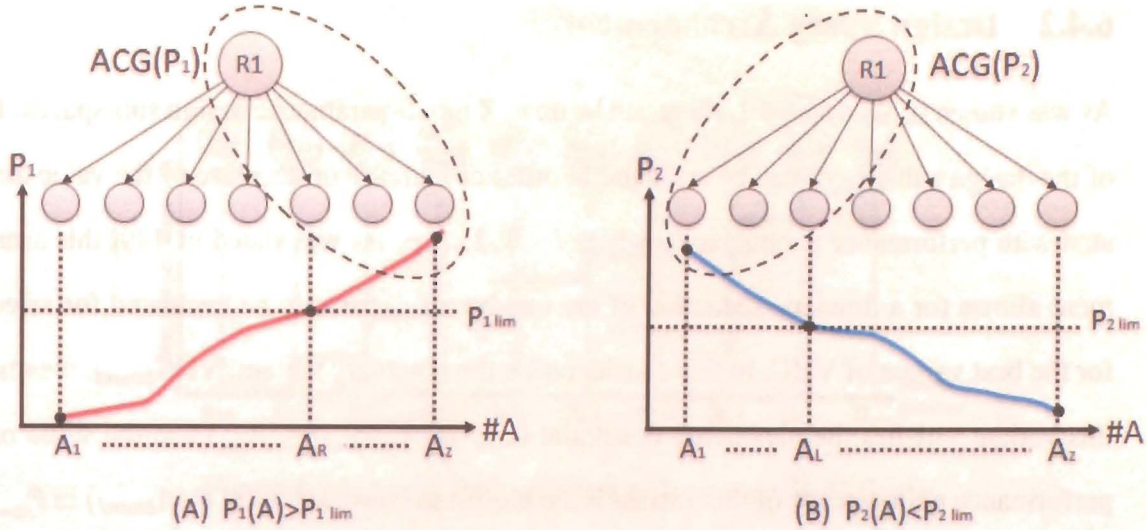


Figure 6.9: ACG Pruning by Pair of Parametric Constraints

Figure 6.9 (A) shows the design space represented by ACG(P_1) arranged in ascending order by the performance parameter P_1 (e.g. "Power Consumption"). The bottom part of this figure presents the diagram of rising value of the parameter P_1 according to the number of architectural variants $A_1, A_2, ..., A_z$. As it can be seen from this figure, the parametric constraints P_{1lim} allow to cut from further consideration some part of architectural variants (from A_R to A_z). Similarly, Figure 6.9 (B) demonstrates how the constraint for the performance parameter P_2 (e.g. "execution time") can cut out (prune the ACG) another part of architectural variants (from A_1 to A_L). Continuing this process for all other parameters P_s, where $s = 1, 2, ..., p$, result in a set of architectural variants which satisfy all parametric constraints. After that the optimization procedure for finding the optimal variant of VHC architecture can be applied. Therefore, the first step of the proposed methodology is to get mono-dimensional design space arranged for each performance parameter. Thus, the procedure for rapid arrangement of ACG for each parameter needs to be developed.

The arrangement of ACG graph includes several steps and is described in detail in the next section.

6.4.2 Design Space Arrangement

As was shown in Section 6.4.1, there can be up to Z mono-parametric design sub-spaces. Each of the design sub-spaces can be arranged in order of increase or decrease of the value associated with performance parameter-P_i, where $i = 1,2,...,p$. As was stated in [49] this arrangement allows for a dramatic reduction of the number of variants to be evaluated for selection for the best variant of VHC. In this consideration the boundary variant, VHC_{bound}, means that this variant satisfies the parametric constraint (e.g. $P_s(A_{boarder}) \leq P_{s_{lim}}$) and the value of the performance parameter P_s of this variant is the closest to constraint (e.g. $P_s(A_{board}) \simeq P_{s_{lim}}$). In other words the next variant of VHC on the design space represented by ACG does not satisfy the constraint (e.g. $P_s(A_{board+1}) > P_{s_{lim}}$). In the example considered in Section 6.4.1, Figure 6.9 variants A_{r-1} and A_{l+1} are border variants. Therefore, the set of border variants A^s_{board} for $s = 1,2,...,p$ represents the trade-off points in the Pareto-optimal design evaluation space [79].

Arrangement procedure for the ACG is divided into two sub-procedures: local arrangement of the resource variants, and hierarchical arrangement of all resources. First we will talk about the local arrangement of resources included in ACG is discussed.

6.4.2.1 Selection of a Set of Resources for an MO and their Local Arrangement

The first step in arranging the resources is to group all of the common resources/elementary operations and create sorted sub-trees in ACG. The sub-ACG trees are arranged by the number of allowed resources, going from minimum on the left of the graph to maximum on the right. The number of resources (e.g. adders, multipliers, etc.) can be listed ascending order. As can be seen in Figure 6.1 that there are: six addition operations, two multiplications, and two comparisons. Therefore, locally arranged sub-ACG trees are arranged in ascending order as shown in Figure 6.17.

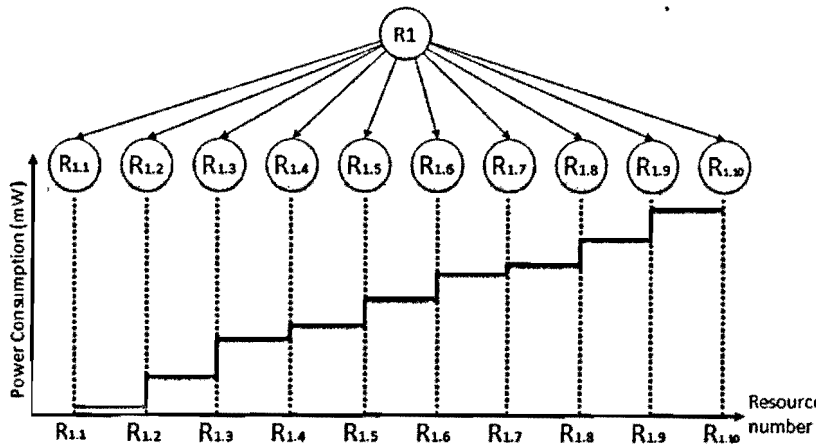


Figure 6.10: Local Arrangement of Resource-R_i Variants

Now it is possible to create an ACG on the basis of the above sub-trees using any possible combination of resource variants. Each combination in turn produces a particular variant of the VHC with the corresponding performance parameters. In VHC variants, where parallelism is utilized and the maximum amount of resources is used, the timing parameter would result in higher processing speed. On the other hand, if the minimum amount of resources is used, the area parameter will be minimized, but, timing will significantly increase since several stages of calculation will have to be implemented. So, a single resource can have different variations of implementation with different performance parameters caused by its internal structure.

As an example, a higher bit-width adder (e.g. 16 bit adder vs. 8 bit adder) would require more power/logic/area than a lower bit-width adder.

It is always possible to arrange resources so that $P_s(R_{i,j}) < P_i(R_{i,j+1})$, where $i = 1..n$ is a resource number and $j = 1..m_i$, is an index of resource variant implementation, as shown in Figure 6.10.

After arranging sub-ACGs, they have to be combined in a hierarchical tree representing the whole ACG. ACG represents the set of all possible VHC variant (design evaluation space) and should be arranged according to P_s, $s = 1, 2, ..., p$.

6.4.2.2 Mono-parametric Partial Arrangement of ACGs

consider a case when the value of the cost function associated with performance parameter P_s, where $s = 1,2,...,p$, increases proportionally to the number of the architectural variant $A_1,A_2,...,A_z$:

1. The value of the cost function reaches the global minimum at A_1: $P_s(A_1) = \min\{P_s(A_k)\}$ where $k = 1,2,...,p$.

2. The value of the cost function reaches the global maximum at A_z: $P_s(A_z) = \max\{P_s(A_k)\}$ where $k = 1,2,...,p$.

3. There are one or more variants of A_k^{extr} where:

$$P_s(A_1) < P_s\left(A_{k-1}^{extr}\right) < P_s\left(A_k^{extr}\right)$$
$$\&$$
$$P_s\left(A_k^{extr}\right) > P_s\left(A_{k+1}^{extr}\right) \tag{6.7}$$
$$\&$$
$$P_s\left(A_k^{extr}\right) < P_s(A_z)$$

This case shows that a local maximum that can occur in resource arrangement as illustrated in Figure 6.11.

Similarly, in a case when the cost function decreases proportionally to number of variants $A_1,...,A_z$ and:

1. The global maximum is at A_1.

2. $P_s(A_k)$ reaches the global minimum at A_z:

$$P_s(A_z) = \min\{P_s(A_k)\}, s = 1,2,...,p, k = 1,2,...,z$$

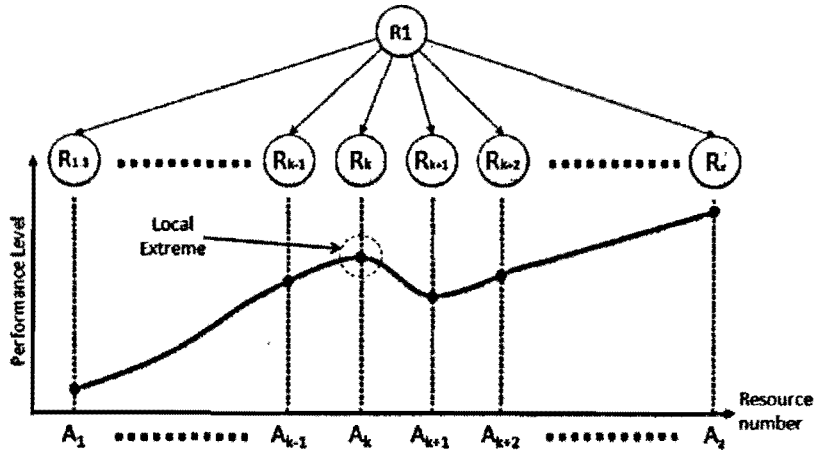


Figure 6.11: Ascending Mono-Parametric Partial Arrangement

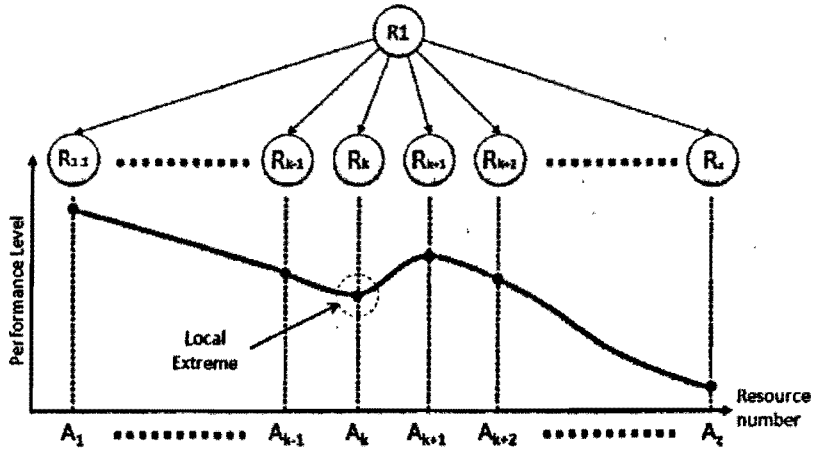


Figure 6.12: Descending Mono-Parametric Partial Arrangement

3. There may be local extremes of the cost-function $P_s(A_k), k = 1,2,...,z$ at one or more variants A_k^{extr} where:

$$P_s(A_1) > P_s\left(A_{k-1}^{extr}\right) > P_s\left(A_k^{extr}\right) > P_s(A_z) \tag{6.8}$$

$$\&$$

$$P_s\left(A_{k+1}^{extr}\right) > P_s\left(A_k^{extr}\right)$$

This case of local minimum is shown in Figure 6.12.

In both of the above cases the ACG was partially arranged according to the value of parameter P_s, $s = 1,2,...,p$. In the real design practice, evaluating the design space by partially arranged ACG, is the most realistic scenario.

Therefore, the assumption taken for this methodology is that in real design practice the optimal variant of VHC architecture should satisfy all parametric restrictions (e.g. power consumption is less than the specified limit) and one of the parameters should reach a value close to global optimum (e.g. highest data processing rate).

In most cases it is acceptable if the value of the parameter to be optimized is close to optimal but not reaching it. However, it is not acceptable if the selected as optimal variant of architecture will provide the local extreme of the parameter to be optimized.

The approach described in [79] has been used for performing ACG partial arrangement by one parameter. This approach states that the most monotonic arrangement of ACG terminals can be reached by the following procedures:

1. Local arrangement of each sub-tree (bush) of ACG associated with corresponded resource: $R_1,...,R_p$. This procedure has been discussed already in Section 6.4.2.1

2. Hierarchical arrangement of the sub-trees on the leafs of the ACG. In [79] the criterion proposed:

$$K(R_i) = \frac{|P_{s_{max}}(R_i) - P_{s_{min}}(R_i)|}{m_i - 1} \tag{6.9}$$

where m_i is the number of variants for a particular R_i resource. Criterion calculation has to be done at the resource when it is placed on a root of the ACG graph and the hierarchical placement of other resources has to be adjusted accordingly. In the Equation 6.9 $P_{s_{max}}(R_i)$ is the global maximum of R_s, which is reached in A_1 variants of architecture (in the case of descending order) or in A_z variants (in case the of ascending order). The $P_{s_{min}}(R_i)$ parameter is the value of P_s reached in a so-called critical variant of architecture. This variant assumes

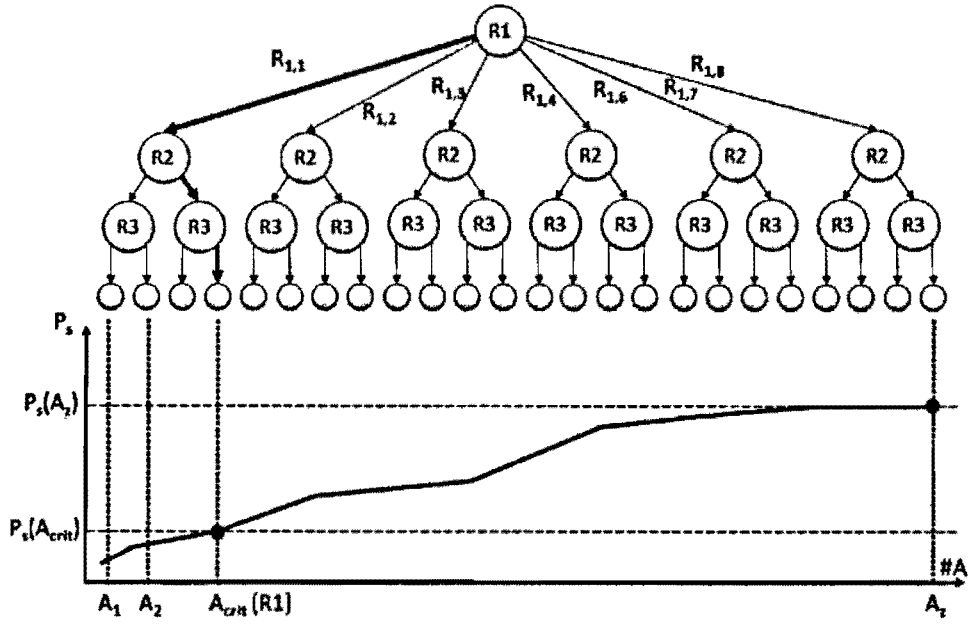


Figure 6.13: Monotonic Ascending of Parametric Value Corresponding to VHC Variants

a utilization of a minimum of R_i resources (e.g. one adder unit for R_1) and maximum resources of all other resources [57, 51]. An example of the critical variant for the R_1 (adder), represented is the ACG arranged in ascending order by P_s is shown in Figure 6.13.

For example, as shown in this figure, the criterion for hierarchical arrangement of the resource R_1 is equal to:

$$K(R_i) = \frac{|P_{s_{max}}(A_z) - P_{s_{min}}(A_{crit}(R_1))|}{8-1} \tag{6.10}$$

where $m_1 = 8$, because there are 8 branches in the hierarchical level of R_2. As one can see, the criterion $K(R_i)$ is the average gradient of the value of the P_s from one variant of R_1 implementation to another. Therefore, in a general case:

$$K_s(R_i) = \frac{\sum_{j=1}^{m_i-1} |P_s(A(R_{i,j+1}) - P_s(A(R_{i,j})))|}{m_i - 1}$$

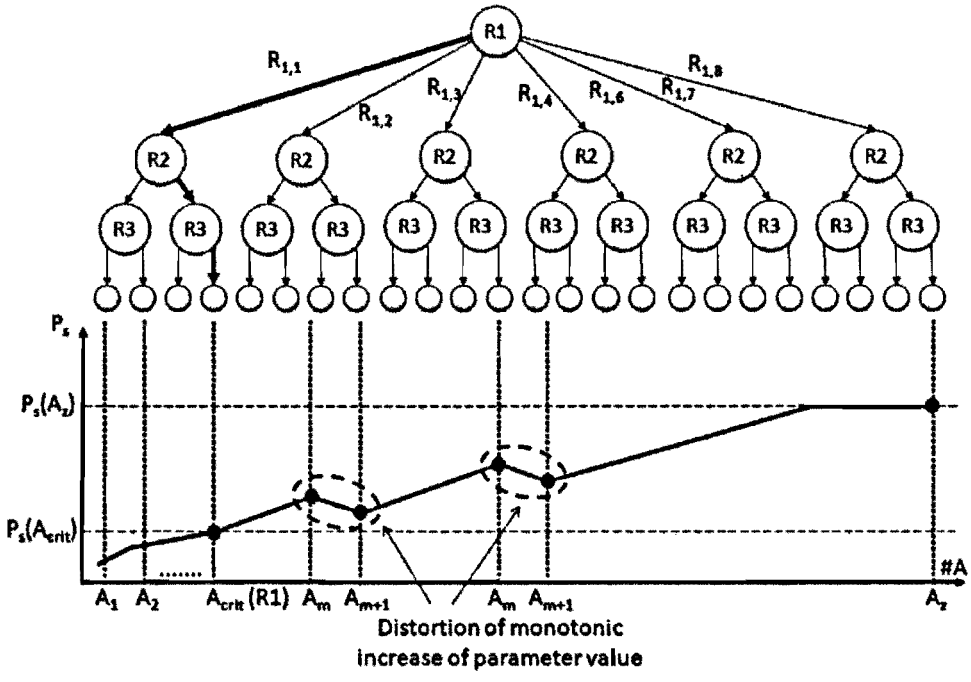


Figure 6.14: Monotonic Increase of the Value of Parameter-P_s with Several Local Extremes

for $i = 1,2,...,n$, $j = 1,2,...,m_i$ and $s = 1,2,...,p$. In other words, the criterion $K_s(R_i)$ shows the influence of variation of R_i resources on the performance parameters P_s.

Thus, the resource with a higher criterion value should be located on the higher level of ACG than the resources with a lower criterion value of the same parameter. If $K_s(R_i) > K_s(R_r)$, where $i,r = 1,2,...,n$ and $i \neq r$, then the resource R_i should be located on a higher level on the ACG than resource R_r.

As stated in [49] and [16] if the above arrangement procedures is performed by traversing through variants A_1 to A_Z from left to right, P_s would increase or decrease most monotonically. The most monotonic increase or decrease of the value of P_s means that divisions of P_s in local extremes are minimal compared with other orders of hierarchical ACG arrangement. The left most and right most terminals of ACG represent global extremes. A general picture of the behavior of the most monotonic increase of P_s on the optimally hierarchically arranged ACG is shown in Figure 6.15.

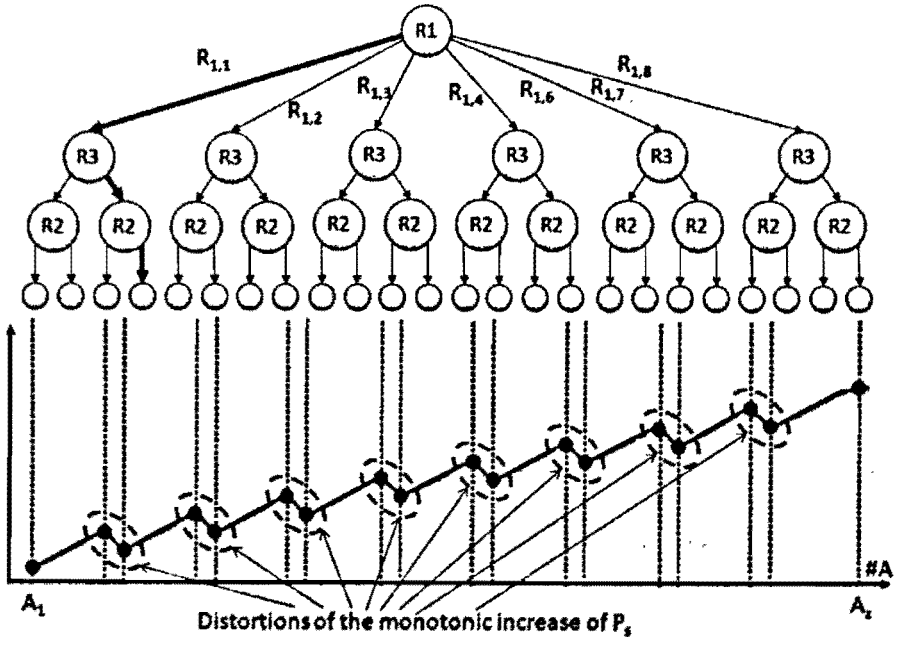


Figure 6.15: Periodic Distortions on Monotonic Behavior of the Parameter-P_s

In contrast, the behavior of P_s on the ACG with non optimal hierarchical arrangement is shown in Figure 6.15.

To sum up, the steps for creation of a partially arranged ACG for any performance parameter P_s, $s = 1,2,...,p$, are:

1) Identify and evaluate the performance for variant A_1, with minimum resource requirement, and A_z, with maximum resource requirement. This is a min-max analysis and will require validation of two architectural variants.

2) Conduct a hierarchical arrangement of resource R_i, $i = 1,2,...,n$. This procedure will require a validation of n - architectural (critical) variants $A_{critical}(R_i)$, each of the resources $P_{j_{min}}(R_i)$ for $i = 1,2,...,n$. Thus, for each performance parameter the $(2+n)$ variants of architecture should be evaluated. In every resource evaluation case lowest performance branch is selected, while largest performance branches are selected for all of the remaining resources. This result is subtracted from the overall max result and divided by total number of branches for the resource in question, as shown in equation 6.10. Therefore, the total amount of variant

evaluations for all design space is equal to $p \times (2+n)$, where p is the number of performance parameters of P_s, $s = 1,2,...,p$.

In the example of the MO discussed in Section 6.2, Figure 6.1, three resources: $R1-$Adder, $R2-$ Multiplier, and $R3-$Comparator, are considered.

Assuming that the selection of the optimum variant of VHC has to be done considering 4 performance parameters, only $4 \times (2+3) = 24$ variants of VHC architecture configurations need to be evaluated. Even for very large design spaces associated with complex MO algorithms the number of variants to be evaluated for a near optimal arrangement of ACG (P_s) is still not that large. Assume that an MO algorithm requires 16 types of resources and each resource can be implemented in 8 possible variants. Also, there are 4 parametric constraints. In this case, the exhaustive search, needed for fully monotonic arrangement of the design space, will require:

$$N_{variant_{exhastive}} = 4 \times 16^8 \approx 4.5 \times 10^{15}\ variants$$

In contrast, the partial arrangement of ACG by each performance parameter will require $(2+16) = 18$ variants of VHC architecture, a total of and $4 \times 18 = 72$ variants for all performance constraints. Obviously, 72 variants of VHC can be evaluated very fast.

6.4.3 VHC Architecture Selection on Partially Arranged ACG

When the procedure of decomposition of ACG to ACGs(P_s) is complete, and partial arrangement of each ACG(P_s) is done, a selection of prioritized parameters has to be performed based on the priority of P_s, where $s = 1,2...,p$. Parameters with higher priority will be considered in selection before the ones with lower priority in descending order.

6.4.3.1 Identifying the Set of Variants

To find an architectural variant which satisfies the P_s^{lim} for every parameter P_s, $s = 1,2,...,p$ a search procedure should be conducted on the partially arranged ACG(P_s). Search procedure

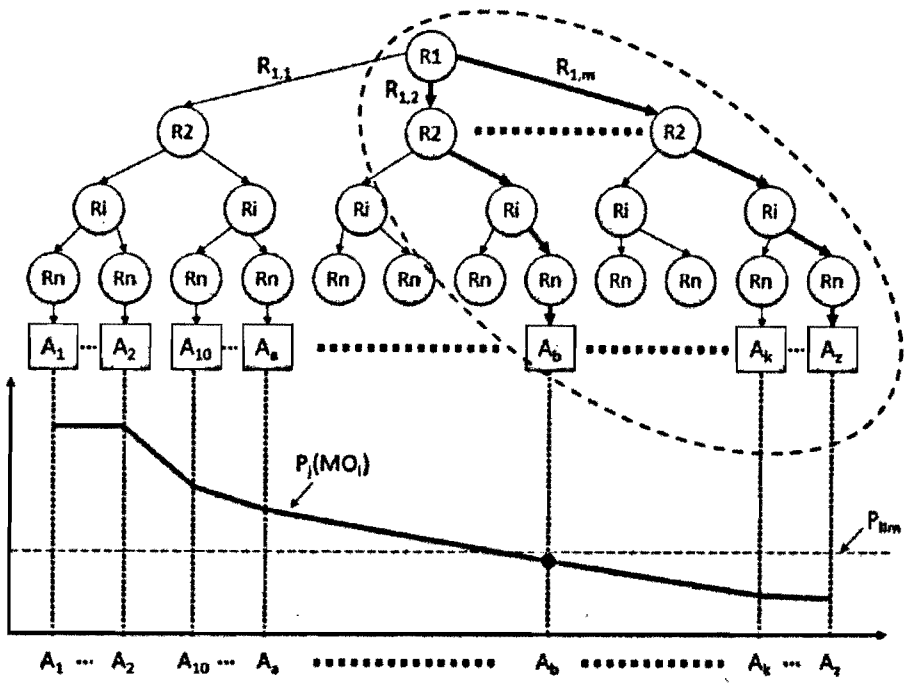


Figure 6.16: Determination of the Sub-set of the Architecture Variants that Satisfy Specification Constraint for One Performance Parameter $P_s(A) \leq P_s^{lim}$.

identifies the boundary variants on the ACG(P_s) graph. Thus, the set of variants is identified such that $P_s^{lim} > P_s(R_j)$, $(\forall j \exists R)$ or $P_s^{lim} < P_s(R_j)$, $(\forall j \exists R)$ depending on the requirements.

Definition: *Border variant of architecture - A_b* is an *architecture variant for which the performance parameter value $P_s(A_b)$ is close to the requested limit - $P_s^{lim} > P_s(A_b)$ or $P_s^{lim} < P_s(A_b)$.*

Identification of the border variant - A_b on the arranged ACG(P_s) allows further pruning of ACG(P_s) as shown in Figure 6.16. In Figure 6.16, all the variants to the right hand side of the A_b, are an accepted set of design space, and A_b is the first variant that is accepted as a one satisfying P_s^{lim} restrictions. The border variant A_b for the parametric constraint P_s - A_b can be found on the arranged ACG(P_s) using binary type search procedure. In each step of this procedure the set of variants is divided in 2 parts side half and the value of performance parameter for the middle variant is calculated. This procedure is described in detail in later section.

6.5 Determination of the Pareto-set of Architectural Variants

When architectural variants are considered for the implementation of temporal partitioning system, a traditional approach is to create an ACG selection tree similar to the one in Figure 6.13. Typically, after creation of this tree it is pruned by eliminating variants that violate the user defined parameters, as shown in the Figure 6.16. This is a valid approach, and in many cases it provides a visual representation of the ACG tree and the region of allowed variants. It becomes much more complicated when more than two performance parameters are used. This approach also has a problem with using a large number or resources each having many variations of implementation. As mentioned before, the number of variants can become exponentially large:

$$N_{VHC} = \prod_{i=1}^{n} m_i \tag{6.11}$$

where N_{VHC} is the number of VHCs (leafs) generated from the ACG, and m is the number of variants for a particular resource for all i resources present in the design where $i = 1,2,...,n$. Hence, the previous example with 16 resources each having only 8 variants, results in 2.8×10^{14} variants which is an enormous number of calculations for any type of memory. In addition, in order to optimize optimize for each parameter, an arrangement of restriction parameters has to be done based on the criterion of the resource. Therefore, the number of possible variants grows to:

$$N_{VHC_{total}} = \sum_{j=1}^{p} N_{VHC}, j = 1,2,...,p \tag{6.12}$$

At the end, to generate such selection trees, and to store them, an enormous processing power and amount of memory are required..

Resources should be arranged in a descending order based on the criterion value, as was done before. However, in the propose approach there is no need for the creation of full ACG tree. In this section an algorithm and example are presented which explain the methodology behind the run-time resource selection.

This algorithm operates on a very limited memory and performs an order of magnitude less processing operations to identify the variants. First step is to identify all of the resources that are used in the implementation of the MO. Designer also needs to identify restriction parameters and their initial values. Later on, these parameters are varied to create several variants of VHCs corresponding to the same MO. Each resource has several implementations and each implementation has different performance parameters. These implementations of resources have to be arranged in ascending/descending order for each restriction parameter, as shown in Figure 6.17. In this methodology the sub-trees are used only for look-up and therefore there is no need to arrange them hierarchically in fully formed ACG tree. It is important to note, that sorting has to start with a parameter of a highest priority. Since all of the other parameters on average would be monotonically increasing or decreasing, the rest of them will be sorted in ascending/descending manner, as shown in Figure 6.17.

At the same time, the rate of increase of performance parameters is different for different types of resources. Hence, as mentioned in the previous section, a criterion value has to be calculated for each parameter of every resource. This is done in order to obtain a monotonic arrangement of the ACG without having any local extremes. It is needed to find the criterion of each resource for every single restriction parameter, as was shown in Figure 6.15. When criterion value is calculated for every resource, an evaluation of border variants can commence, with subsequent creation of a VHC set. Border variants are identified based on the initial restriction for the parameter that was provided by the user or by the system specification/limitation. The main advantage of this procedure is that instead of exhaustively generating all of the possible combination of resources and then performing search for a border variant,

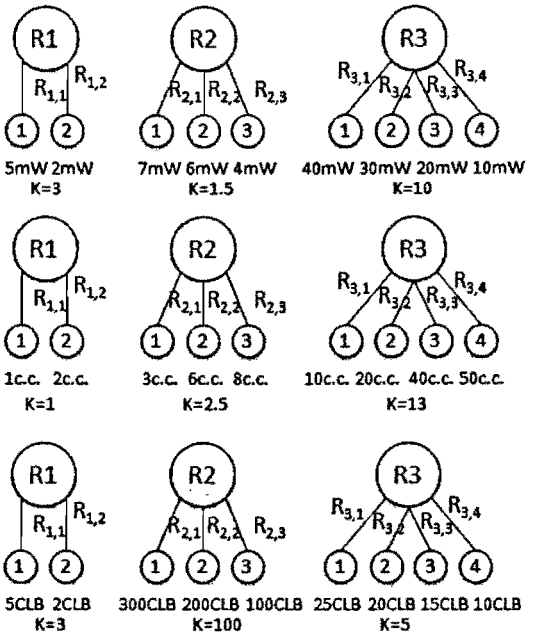


Figure 6.17: Resources Sorted According to Different Performance Parameters , where K is criterion value

only a few selected calculations have to be performed. Hence, instead of N_{VHC} operations, as by Equation 6.11, at most:

$$N_{VHC_{boarder}} = \sum_{i=1}^{n} \log_2 m_i \tag{6.13}$$

operations need to be performed to find the border variant. Where m is the number of variations for resource i. The algorithm for border variant search is shown in the Figure 6.18. The complexity of this algorithm is $O(nlog_2 n)$.

An example of the actual border variant search is shown in Figure 6.19. In this example, there are three resources, each having different variations of parameters, as was shown in Figure 6.17. The border variant that is being searched for is based on the execution time parameter with a limit of maximum 50 clock cycles. As described in the algorithm, a criterion was calculated for all of these parameters and it was found that R_3 has the highest criterion and R_1 has the lowest, hence the hierarchical arrangement in Figure 6.19. Figure 6.19 shows the graphical representation of the the full ACG tree if it would be generated and then exhaustively

Figure 6.18: border VHC Variant Search Algorithm

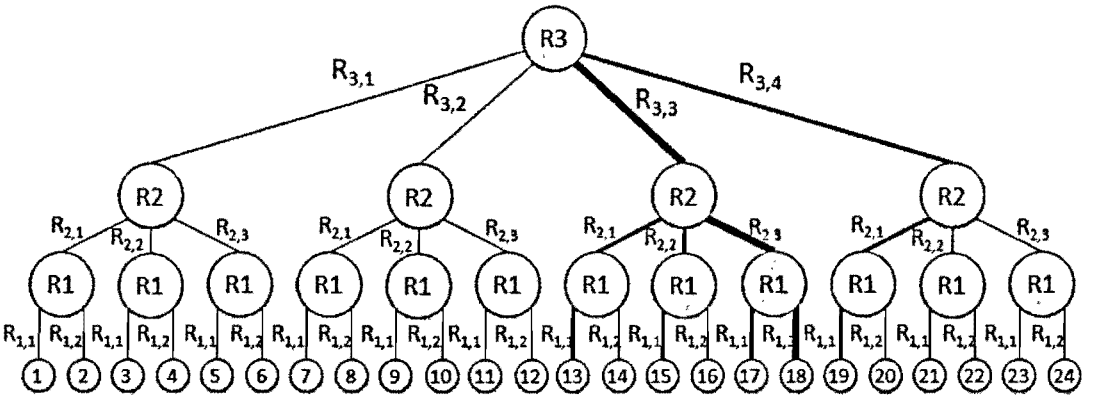
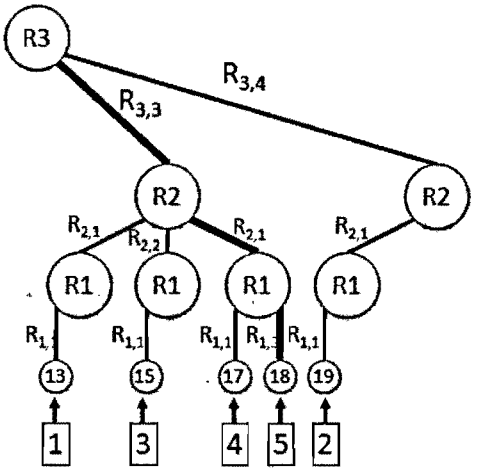


Figure 6.19: Example of ACG with Selected border Variant of VHC



#	Result	Path	Fixed	<,>,=
1	44cc	R3,3→R2,1→R1,1	-	>
2	54cc	R3,4→R2,1→R1,1	-	<
3	47cc	R3,3→R2,2→R1,1	R3,3	>
4	46cc	R3,3→R2,3→R1,1	R3,3;R2,3	>
5	50cc	R3,3→R2,3→R1,2	R3,3;R2,3	=

Figure 6.20: VHC Variants Considered by the Search Algorithm on ACG

searched. In this example, bold edges show the search paths that algorithm took, and double bold edges show the path of border variant 18. As can be seen, only 5 attempts out of 24 variants were needed to find the actual border variant was found, and a new rule of $(\leq) R_{3,3} \rightarrow R_{2,3} \rightarrow R_{1,2}$ was added to the list of rules. This rule indicates that path of branches to the right of this branch are going invalid for variant selection. The usage of these rules will be demonstrated later in the chapter.

Figure 6.20 shows the sequence of algorithms operation and the branches that were selected before reaching the border variant. As it can be seen from the Figure 6.20, since the border variant in question was to be maximized or, in other words, to be located as right as possible, all of the non-fixed branches from the left were initially in the resource. Such case

Table 6.1: Effectiveness of Proposed Search Algorithm in Comparison to Exhaustive Search

# Resources	Total Possible Variants	Boundary Searches	Additions	Comparisons	Operations for selections	Total Operations
1	8.0E+00	4	4	8	3	15
2	6.4E+01	7	14	14	4	32
3	5.1E+02	10	30	20	5	55
4	4.1E+03	13	52	26	6	84
5	3.3E+04	16	80	32	7	119
6	2.6E+05	19	114	38	8	160
7	2.1E+06	22	154	44	9	207
8	1.7E+07	25	200	50	10	260
9	1.3E+08	28	252	56	11	319
10	1.1E+09	31	310	62	12	384
11	8.6E+09	34	374	68	13	455
12	6.9E+10	37	444	74	14	532
13	5.5E+11	40	520	80	15	615
14	4.4E+12	43	602	86	16	704
15	3.5E+13	46	690	92	17	799
16	2.8E+14	49	784	98	18	900

is shown in #1 and #2 searches, where branches of resource R_3 were searched. However, in the search #3 the resource branch $R_{3,3}$ became fixed, and R_2 was the resource in question, hence only most left variant R_1 was selected. Figure 6.20 shows that only 5 variants had to be calculated in contrast to 24 calculations of full ACG in Figure 6.19. As mentioned before, the effectiveness of this technique dramatically increases as the number of resources increases. Table 6.1 shows the number of calculations needed for the exhaustive ACG tree generation compared of the boundary search algorithm. In this table, resources increase linearly from 1 to 16 while every resource has 8 implementations.

As can be seen, even for 16 resources with 8 implementations each, the total number of variants grows to 2×10^{14}. By implementing the boundary search only 49 searches are needed to find the boundary variant, and create a restriction rule. When all of the border variants were identified, as shown in Figures A.2 and A.4, of the Appendix set of border variants "rules" were recorded, to be used in the next step of VHC set generation.

6.5.1 Semantic Filtration of Architectural Variants for VHC generation

Semantic filtering is based on logic comparison of the generated variants to the border variants "rules" that were determined for each parameters constraint. Semantic filtering uses border variant path, as well as a range (e.g. $(\geq)$,$(\leq)$) to identify if the branch path of variant in question violates the "rules". It is also important to note that in many cases semantic filtering allows to identify if a variant passes or violates the rule without traversing through the whole path. Semantic filtering is performed in the sequence of the rule, where $(\rightarrow)$ indicates sequence order of the resource variants. For example, if rule is given by $\{(\leq) R_{2,2} \rightarrow R_{3,3} \rightarrow R_{3,3}\}$, the first branch that to be compared is $R_{2,2}$ and variants under test have to be less or equal to the branch path, hence the $(\leq)$ sign. In a case where a variant under test contains a branch that exceeds $R_{2,2}$ (e.g. $R_{2,3}$), then the rest of branches are not checked and the variant is discarded. If a variant under test contains a branch that is less than $R_{2,2}$ (e.g. $R_{2,1}$), then the variant is automatically excepted without checking the rest of the branches. In the scenario where a branch is equal to the one in the rule (e.g. $R_{2,2}$) then next branch in sequence is tested with the same method, which in the above example is $R_{3,3}$.

To obtain variations of VHCs corresponding to the same MO, one of the restriction parameters has to be modified and the border variant has to be found again for this parameter. Following this procedure, a semantic test has to be performed, as described above. If none of the "rules" are violated, then this variant is accepted and added to VHC list. Suppose, that the rules obtained from example in the previous section are: $1 : \{(\geq) R_{2,2} \rightarrow R_{3,2} \rightarrow R_{1,1}\}$, $2 : \{(\geq) R_{3,2} \rightarrow R_{1,1} \rightarrow R_{2,3}\}$, $3 : \{(\leq) R_{3,3} \rightarrow R_{2,3} \rightarrow R_{1,2}\}$, with the initial restrictions of 225 CLBs, $40mW$, and 50 c.c. The operation of VHC selection procedure can now be illustrated. If we want to find weather border variant (2) for $40mW$ power consumption is valid for all of the parameters we have to check it with both remaining "rules". Checking with the first rule reveals that since variant (2) has a branch of $R_{2,3}$, it is automatically excepted by rule #1 because it is larger than $R_{2,2}$. No further comparisons have to be made, since if the top

Table 6.2: Test for "rule" #1

Condition	Branch under test	Result
$R_{2,2} \geq$	$R_{2,3}$	Pass
$R_{3,2} \geq$	-	-
$R_{1,1} \geq$	-	-

Table 6.3: Test for "rule" #3

Condition	Branch under test	Result
$R_{3,3} \leq$	$R_{3,2}$	Pass
$R_{2,3} \leq$	-	-
$R_{1,2} \leq$	-	-

of the hierarchy is satisfied, then the rest of the branches are satisfied. When checking with rule #3, conditions are also satisfied, since $R_{3,2}$ is smaller than $R_{3,3}$. Hence, the variant corresponding to the branch $R_{3,2} \rightarrow R_{1,1} \rightarrow R_{2,3}$ is accepted to the VHC list. The last step before adding the VHC to the set of chosen VHCs is to calculate the rest of performance parameters that correspond to the selected branch. In branch scenario these parameters are 29c.c. and 125CLB.

To illustrate the failure effect we can try to check if the border variant #1 can be used as one of the VHCs in the VHC set. Following the procedure shown in the above tables, we test variant #1 under "rule" 2 : $\{(\geq) R_{3,2} \rightarrow R_{1,1} \rightarrow R_{2,3}\}$ and the result is a failure of the variant at the last branch test, and the in exclusion of this variant from the final VHC set. At this point, the restriction for a parameter is modified by a predefined step, and the procedure is repeated. In turn, this allows to obtain a broad range of VHCs that have variations by each of the restriction parameters. The result is a set of VHC variants for a particular MO where for each

Table 6.4: Test for "rule" #2

Condition	Branch under test	Result
$R_{3,2} \leq$	$R_{3,2}$	Check Next
$R_{1,1} \leq$	$R_{1,1}$	Check Next
$R_{2,3} \leq$	$R_{2,2}$	Fail

parameter there is a maximum and minimum value, as well as intermediate variants. Depending on the scenario this allows to have optimization by any of performance parameters. This is especially important in a temporal partitioning architecture where area/power/processing speed is restricted and strongly depends on a reconfigurable device used in a target system.

When the border variants are determined for all parametric constrains, then the design space can be reduced by exclusion of variants restricted by all parametric constraints. The result is the Pareto-optimal set of variants of architectures that can be used for VHC selection, which is described in the next chapter.

At the same time, resulting Pareto-point set cannot be pruned to the point of being empty. If all leaves are removed, the system would not have any variants to choose from. If such case does occur user will have to either:

1. Select a larger device, therefore, increasing amount of logic per device.

2. Ease the design parametric constrains.

Easing constrains though is a much more problematic solution since it is usually linked to modification of other sub-systems or, in many cases, is not possible at all due to the specification parameters. This is especially true if the overall system is real-time critical and does not have an option of longer delays, or excessive power use. If Pareto-set with many variants is requested, then the most optimal, with highest value of performance parameter, should be selected. Selection should be made based on the performance parameter with highest priority. Best performance does not necessarily constitute the fastest operation. Best performance can also be the lowest power consumption or the smallest area requirement.

Similarly, this approach is applicable in power sensitive systems where power restriction might change and a different VHC would be needed. These generated VHCs associated to a particular MO are used in SSP generation in later chapters.

6.6 Summary

The focus of this chapter was on description of novel methodology developed for creating macro-operator (MO), and subsequent generation of Virtual Hardware Components that correspond to the given MO. This chapter presented a methodology of resource binding and described how binding effects the overall scheduling of resources. Methodology of VHC synthesis was presented, together with the procedures of VHC variant generation. It was shown how the generated VHCs have to be arranged in order to provide an efficient method of selecting the optimal variant. Methodology for creation, arrangement and pruning of Architecture Configuration Graph (ACG) was described in detail. For that purpose the non-exhaustive border variant selection algorithm was developed and implemented. The extension of this the methodology for selection of an optimal variant of VHC in multi-parametric design space is presented. It was shown that the proposed methodology provides the means of finding an optimal variant of VHC for a particular MO by evaluating minimal variants and, therefore, can be performed in minimum time. Methodologies described in this chapter were published in journal [51] and conference publications [57], [58].

Chapter 7

Methodology of Automated Assembly of Optimal VHCs into SSPs

7.1 Introduction

As described in the Chapter 6, an application task is presented in a form of SG which is assembled from MOs. MOs, in turn, are assembled from elementary operators (EOs). Each MO is associated with the a set of VHCs, which can process data according to the MO algorithm with different performance parameters. This chapter presents a novel methodology of task segmentation, and selection of optimal VHCs for further synthesis of Segment Specific Processors (SSP). It covers all of the steps required of creation of temporally processed application, and give some examples for the proposed approach. The task segmentation methodology includes: automated level dependency arrangement, scheduling, and resource binding. The implementation of binding based on multi-parametric restrictions is also covered.

This chapter considers the works that have been tackling the issue of algorithm segmentation, as well as, the advantages and pitfalls of those approaches. In addition, it compares the proposed methodologies to the existing ones. The proposed approach was implemented in a form of CAD software. Resulting segmented algorithms were executed on the reconfigurable

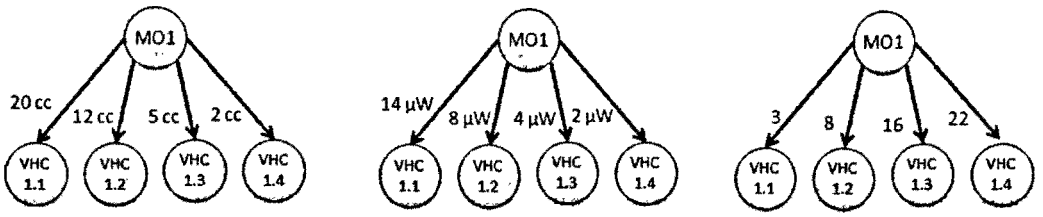
hardware platform that was developed in Embedded Reconfigurable Systems Lab (ERSL) at Ryerson University. Hardware and software implementations are described in detail in the next two chapters.

7.2 Methodology of Segmentation of an Application Sequencing Graph

In order to take advantage of the temporal partitioning approach, an application task has to be processed in segments on reconfigurable platform. As shown in Chapter 5, for each task segment an associated Segment Specific Processor (SSP) should be synthesized. Creation of the optimized set of SSPs requires a proper SG segmentation methodology.

As was described in Chapter 6, MOs are created along with the sets of corresponding VHCs. With these MOs an application task algorithm can be formed, as shown in the Figure 7.2. The concept of temporal partitioning assumes that the outcome is a set of configuration bitstreams corresponding to the synthesized SSPs. These bitstreams are loaded in sequence one after the other onto the target FPGA based on a schedule. The actual combination of the MOs into segments requires an optimal (cost-effective) segmentation of the application's SG to be carried out automatically.

The optimization of resources has to include the optimization of memory transfers between the segments. For every segment the configuration system will have a time overhead of saving the temporal data to an external SRAM memory. The temporal data readback operation has to be performed at the point of the start-up of the next SSP core. If a large amount of temporal data needs to be transferred between segments, the time overhead increases and impacts the overall performance. As was explained in Chapter 6, T_{config} consists of T_{read}, T_{write}, and $T_{config_{bistream}}$. Increasing the T_{read} and T_{write} parameters increases the T_{config}, and as a result, the CPR of this approach is reduced. Therefore, one of goals is decrease of the temporal



(a) Processing Latency (b) Power Consumption (c) Area Requirements

Figure 7.1: Correspondence Between MO_1 and associated VHCs

data transfer overhead. For that, the methodology of actual algorithm segmentation should be discussed first.

7.2.1 Division of an Algorithm into Segments

The most common approach for division of algorithms into segments is based on area restriction [76, 88, 29, 13]. The novelty of the proposed approach is consideration of multiple parametric constraints (performance parameters). This is much more realistic and practical approach. However, the implementation of this approach is much more complicated.

Every macro operator (MO) has several associated VHCs. Each of these VHCs have various performance parameters to satisfy different parametric constraints. VHCs are added one after another into the SSP assembly until one of the restrictions for the segment is violated. Different combination of VHCs can be selected for SSP in order to fit into an FPGA device. If none of the VHCs associated with an MO satisfy the restriction, the MO is moved to the next segment and/or the user is notified that this MO cannot be inserted into the current segment. In a scenario where all VHCs associated with MO are larger than given FPGA, a larger FPGA device has to be used for implementation of this SSP.

Since every MO is associated with several VHCs, selection of an appropriate VHC for SSP based on several parameters should be considered. Such parameters could be: timing, latency, power consumption, as shown in Figure 7.1.

Figure 7.1(A) shows four versions of VHC corresponding to a single MO, arranged by delay times. The delay times vary from 2 to 20 clock cycles. Figure 7.1(B)(C) similarly show, power consumption, and area/logic parameters associated with VHCs. Other parameters can be considered as well, if necessary. Multi-parametric constraints can be applied in optimal SSP selection similarly to the optimal VHC generation [54].

7.2.2 Algorithm Segmentation, Binding, and SSP Generation

Synthesis of a set of SSP cores and their schedules involves several steps:

1. Assignment of dependency levels for all of the MOs in the algorithm.
2. Creation of sub-Architecture Configuration Graphs (ACGs) by selecting MOs.
3. Selection of the optimal VHC variants from sub-ACGs and assembly of SSP sets, based on given constraints.
4. Selection of a particular SSP set for final bitstream generation based on the parameter priority.

The above steps are discussed in the following subsections and are implemented in the CAD tool software. CAD tool is described in Section 8.2 of Chapter 8. This section will concentrate on methodology of algorithm segmentation and SSP generation.

7.2.2.1 Automated Dependency Level Assignment Algorithm/Level Division

First operation that is required for synthesis of the optimal SSP set is a proper level division based on ASAP scheduling algorithm.

Definition: *Dependency Level Division is the MO execution arrangement where MOs of the same level have exactly the same start time.*

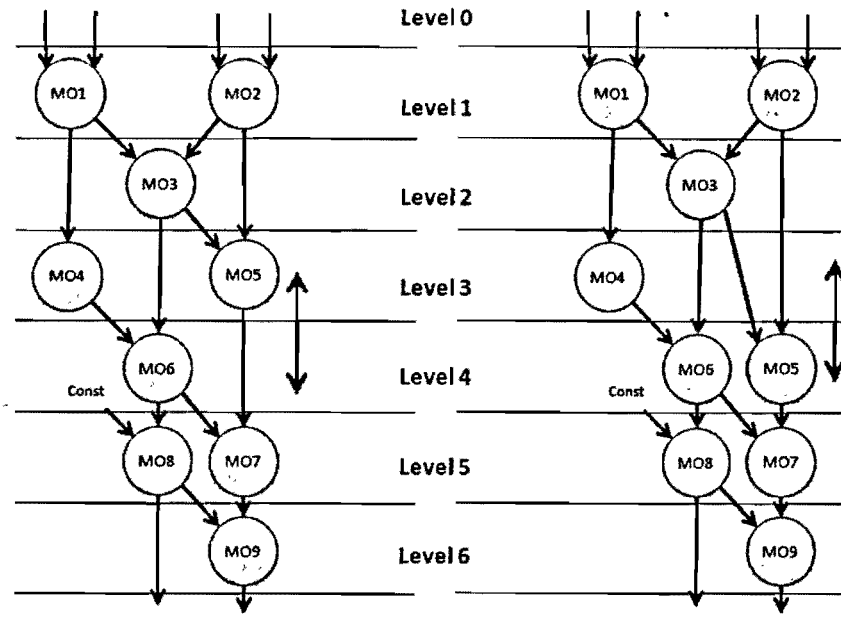


Figure 7.2: Level Dependency Division

Level division has to be done by means of precedence-relation. In Figure 7.2 the segments MO1 and MO2 are not interdependent and thus can be placed on the same level.

In a case of incorrect level assignment a dependence would be formed and an MO would be waiting for the input from the previous MO. This would create a deadlock. Such example is shown by MO9, where it receives input from MO7 and MO8. If placed on the same level MO9 would be waiting for the input from MO8, and MO7. Therefore, MO9 would not be able to proceed with processing until result is received. On the other hand, if dependent MOs are included into the same segment, the overall memory transaction overhead (read and write) could be reduced. For purpose of providing balanced level assignment the special algorithm was proposed and developed.

7.2.3 ASAP Level Assignment

In ASAP level assignment [79] MOs are assigned a level, as soon as all of the predecessor parent nodes are processed. In this section the assignment of levels to MOs is discussed.

The first step in level division algorithm is identification of primary inputs to the system. The goal is to identify the MOs of the task processing algorithm that are first to acquire the inputs to the system. The MOs that have only primary inputs are called primary MOs. Since primary MOs have no inputs from any other MOs they are dependent only on the primary inputs. Therefore, they are assigned to the first level. To schedule the rest of MOs similar approach be used, as is shown in Figure 7.3(B) where output of the primary MOs is an input to the child MO. However, as it can be seen in Figure 7.3(C) being an immediate descendant of a primary or parent MOs does not guarantee being on the consecutive level from the parent MO. This is due to the fact that since an MO can have multiple inputs from different parent MOs, it might have a dependency on an MO from lower level in the hierarchy. To archive a proper level dependency arrangement an extra step has to be added to the automatic level arrangement algorithm. This step involves marking the output edges with the same level as the source MO. This way by checking the incoming edges it is possible to immediately identify the levels of the parent MO. If one of the incoming edges is unmarked it indicates that MO in question is at least two levels lower than any of the incoming edges. Having at least one unassigned incoming edge gives uncertainty of which level has to be assigned. At this point level assignment for this particular MO has to be postponed until all of the incoming edges are assigned a level.

Non-primary MOs are assigned levels based on the maximum level of the incoming edges plus one, considering that all of the incoming edges are marked with some level. This guarantees that child node will always be at least one level lower than any of its parent nodes. By traversing through the SG this procedure would eventually mark all of the nodes and edges with their appropriate dependency levels. The algorithm in action is shown in Figure 7.3, where progression of the level assignment is illustrated.

In Figure 7.3(A) that first primary inputs are assigned level 0 and in Figure 7.3(B) nodes and their output edges are assigned the number of maximum input edge level plus one. Figure

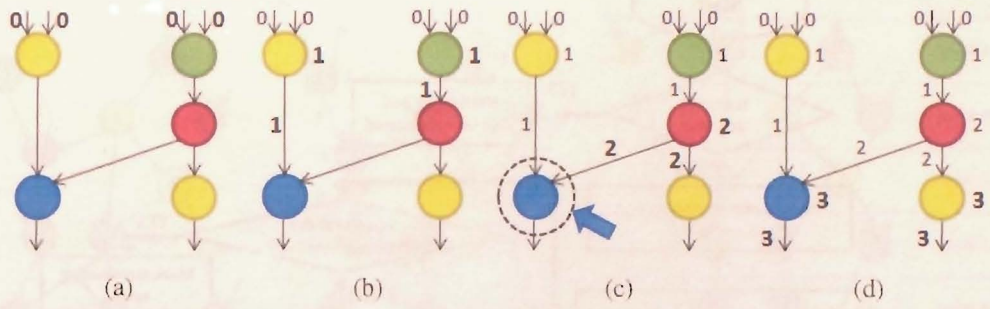


Figure 7.3: Illustration of Level Assignment Algorithm Operation

7.3(C) shows the dotted circle around the node that does not have all of its input edges marked with a level, hence, it is not assigned at this point. As it is seen, red coloured parent node is not assigned a particular level and that is why it is uncertain which level should be assigned to the circled node. In the last step all of the nodes and edges are assigned to the appropriate levels as shown in Figure 7.3(D).

Level assignment algorithm's flow chart is shown in Figure 7.4.

The algorithm can be summarized in 4 steps:

1. Global inputs/edges are assigned level 0. $Level(E_{in}(i)) = 0$
2. MOs are assigned the level equal to maximum level value of all the incoming edges plus one. $Level(Node(i)) = MAX(Level(E_{in}(i)) + 1$
3. Edges outgoing from an MO node are assigned the level number of the node $Level(E_{out}(i)) = Level(MO(i))$
4. Any of the terminating nodes are marked accordingly, so they will not be considered in the next iteration of the algorithm.

At the completion of the algorithm based on the flow shown in the Figure 7.4 all of the MOs and edges are assigned a particular level. The overall complexity of this algorithm is $O(nlog_2n)$.

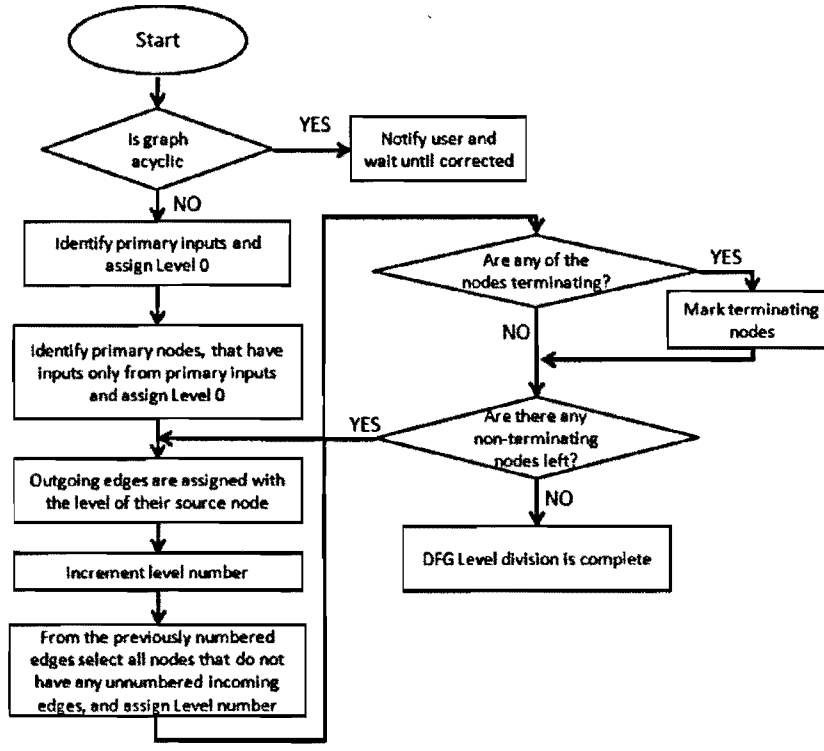


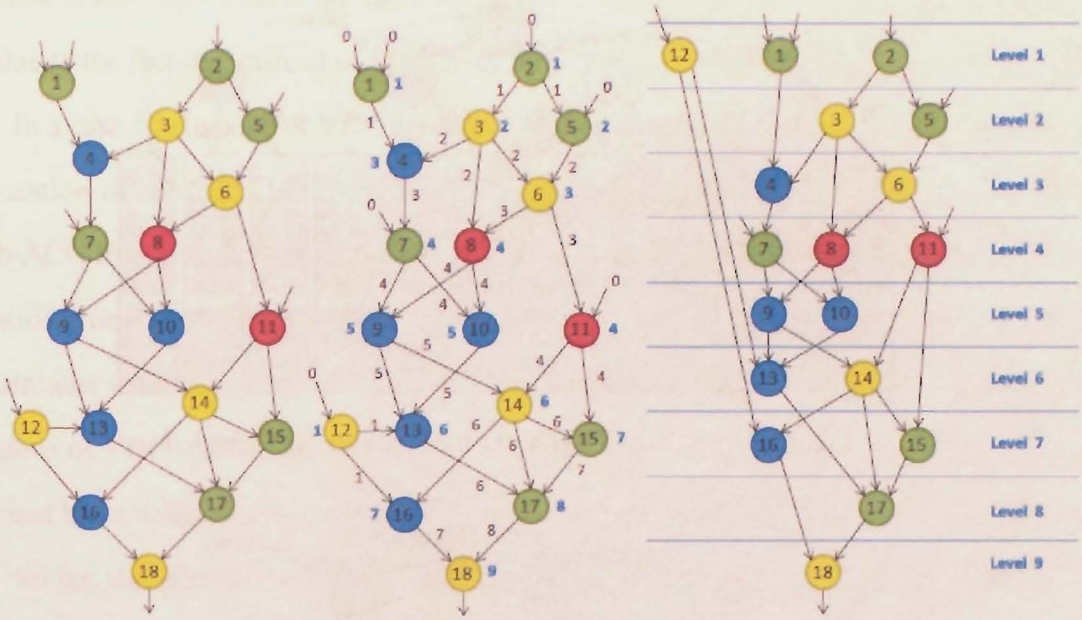
Figure 7.4: MO Level Assignment Algorithm Flow Chart

The second step of VHC selection and grouping of VHCs is carried out after level assignment.

7.3 VHC Selection and Grouping Methodology

In this section a methodology of decision making in task sequencing graph segmentation is discussed. Numerous cases are considered when performing segmentation of a sequencing graph of connected MOs and selection of appropriate VHCs.

Each particular MO was assigned with a level dependency number (e.g. Figure 7.5 (B)) and re-arranged into a level dependent SG (e.g. Figure 7.5 (C)). At this point all of the MOs are arranged so that parent MOs are located on the level above the children MOs. The Selection process starts from the top level (e.g. Level 1) where MOs receive only primary inputs from the system, as shown in Figure 7.5 (C). Segmentation algorithm begins by adding MOs on the



(a) Original SG of the Task (b) Marked SG (c) Level Arranged SG

Figure 7.5: Task SG Representation by Level Arranged MOs

first level of the arranged graph to a sub-ACG tree for the purpose of optimal VHC selection, as shown in Figure 7.6. Similar to the approach described in Chapter 6 it is possible to find an optimal selection of VHC variant for the associated MOs by forming a sub-ACG selection tree. Each MO is associated with the set of possible VHCs. This set of $VHC_{i,j}$ is formed according to specific performance parameters, as shown in Chapter 6.

By performing simple border variant search, the limits of acceptable VHCs are identified, based on the restriction parameters of the system. For example, as shown in Figure 7.6, a limit of 20CLBs was imposed by the system specification, and therefore, the branches exceeding that limit are marked with a dotted line. Subsequent MO_{i+1} is added to every branch of the parent MO_i, where every branch corresponds to the available versions of VHCs for that MO_{i+1}. Branch carries the information of performance parameters which are used for calculation of the restriction parameter violation. If there are VHCs that satisfy restriction parameters an additional MO can be added to the sub-ACG tree. Upon addition of a new MO_{i+1} which was selected from the same hierarchy level of the level arranged graph, a re-calculation of the

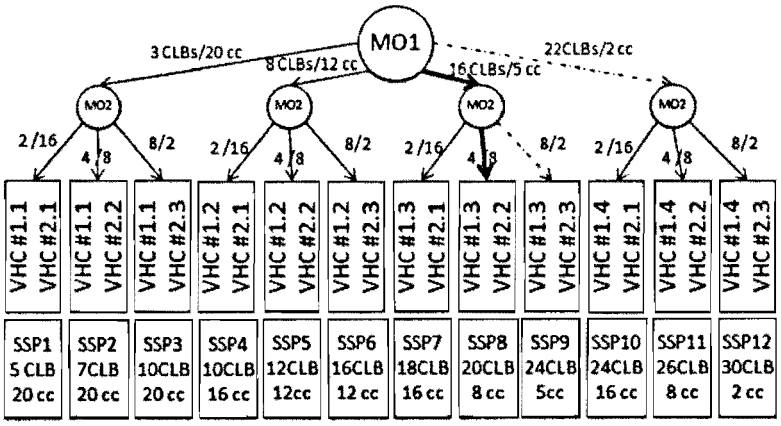


Figure 7.6: Sub-ACG Tree with MO1 and MO2 Added

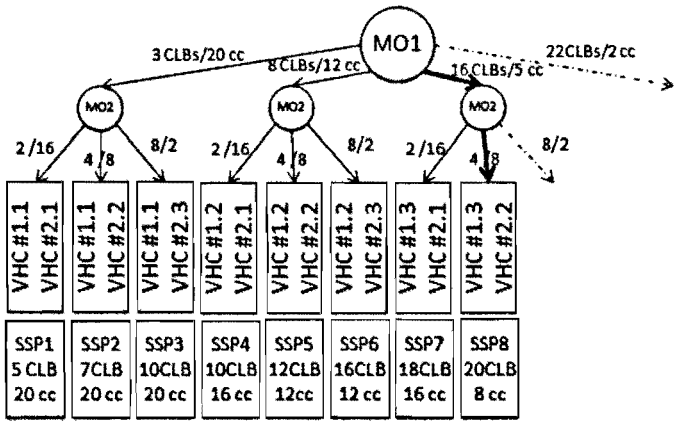


Figure 7.7: Sub-ACG Tree with Excluded Restriction Violated Branches

criterion K has to be performed. Calculation of criterion was described earlier in the Chapter 6. In order to avoid local extrema MOs have to be arranged, so that MOs with the higher criterion would be on a higher hierarchical level in a sub-ACG tree. Branches that did not meet the restriction parameters in the previous step are cut out (pruned) from the expansion, as shown in Figure 7.7.

By excluding the above VHCs the sub-ACG tree is reduced to only a few brunches, and this speeds up the consecutive sub-ACG border searches. Following the addition of a new MO_{i+2} a new border search is performed and the sub-ACG tree is reduced further. If at least one combination of VHC variants was identified, then a new MO_{i+3} may be selected, and the procedure of re-creation and limitation of sub-ACG tree as described above is repeated. border

variant search algorithm is the same as the one described in the Chapter 6, Section 6.5. This is due to the fact that instead of EOs (as in VHC selection) there are VHCs that form the SSP.

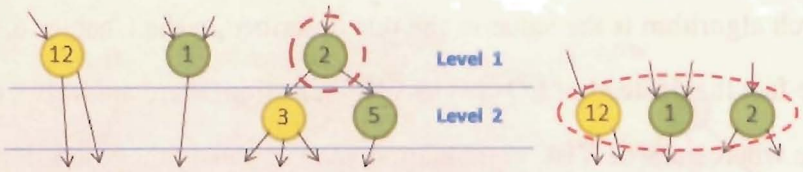
In a case where none of VHC combination variants satisfy the restriction parameters, configuration of sub-ACG tree is rolled back to the previous successful arrangement. When a sub-ACG tree is completed and a set of VHC variant configurations is created, these configurations form the SSP set [54]. These SSPs can now be synthesized into SSP configuration bitstreams with associated parameters. Such example is shown in Figure 7.7, where combinations of VHCs satisfy the restriction parameters and span variants #1 to #8. The right most variant has smallest latency and the left most one occupies the least amount of space/logic.

So far, the selection of MOs to be added to a sub-ACG was done without mention of the methodology behind it. The following section presents the methodology of MO selection in the process of creation of SSPs.

7.4 Methodology of "Next" MO Selection for SSP

Creating sets of SSPs by the method outlined in the previous section is done in several steps. When adding a consecutive MO to the sub-ACG certain selection rules have to be followed to achieve optimal results. First and the most straight forward way of adding MOs is to select them from the same dependency level. The simplest solution is to group all of the MOs from same level to create a set of n SSPs corresponding to n levels. However, such approach is not possible because either all MOs from same level do not satisfy all the restrictions or a lot of logic resources are be un-utilized. Below, with the help of several scenarios, the methodology of MO selection is described.

In Case 1, depicted in Figure 7.8, selection starts from addition to sub-tree of all of the MOs on the same dependency level. Priority, however, is given to the MOs that have dependent nodes on the next level. This is done to minimize the probability of dependent nodes being moved to the next SSP, which in turn would stall the processing of the dependent nodes. From



(a) Initial Selection (b) Final SSP Selection

Figure 7.8: Case 1: Priority of Selection of Nodes with Dependencies on the Next Level

Figure 7.9: Case 2: Partial Level Inclusion in a SSP

the example of the algorithm in Figure 7.5 such nodes are: 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 15, 16, and 17.

If not all of the MOs from the same level fit into the SSP_i, so those that do not are placed in the consecutive SSP_{i+1}. For the Case 1, as well as all of the other cases, the total logic area is equal to the sum of logic areas occupied by each VHC in the SSPs.

$$SSP_{i_{latency}} = \max\left\{VHC_{j_{latency}}\right\} \ (j = 1,...,k; \forall VHC_j \exists SSP_i) \tag{7.1}$$

$$SSP_{i_{area}} = \sum_{j=1}^{k} VHC_{j_{area}}; \ (\forall VHC_j \exists SSP_i) \tag{7.2}$$

Processing latency in this case is calculated by adding up the maximum latencies from all VHCs present in SSP_i. Since in Case 1 none of the VHCs depend on each other, the processing is done in parallel, and the latency is not aggregated. Time required for processing of the input depends on the VHC that requires the longest propagation delay.

In Case 2 demonstrates partial level inclusion of VHCs. It is similar to the previous case with the exception of a possibility of re-iteration of sub-ACGs creation. Figure 7.9 shows a situation where there is only partial inclusion of the level in one SSP.

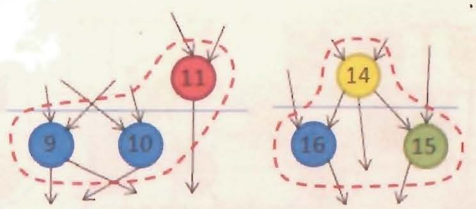


Figure 7.10: SSP Composition from VHCs located on Consequent Levels of SG: a) without Dependency in Case 3; b) with Full Dependency in Case 4.

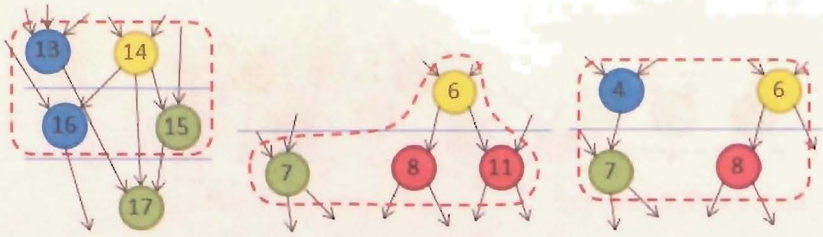
The re-iteration can potentially identify a different combination of MOs that could provide more optimal result. However, a threshold of reiterations has to be set, in order to avoid exhaustive searches and slowly down the overall segmentation process.

When MOs from one dependency level have been exhausted, and none of the restrictions have been reached, Cases 2, 3 or 4 are assumed, as shown in Figures 7.10 (A) and 7.10 (B).

First choice of selection from the consecutive dependency level L_{i+1} is a selection of MOs that are not depended on the MO from L_i, assuming that these MOs are included in the same SSP as shown in Case 3. Same as for Case 1, the latency is calculated by taking the maximum latency out of all of the MOs latencies in that SSP instead of summation of latencies in case of presence of dependencies. Case 3, however, do not occur as frequently as Case 4 where there is dependency on at least one of the MOs from the next level. For this case processing latency has to be calculated by summing the latencies of dependent MOs:

$$SSP_{i_{latency}} = \max\left\{VHC_{i_{latency}} + VHC_{j_{latency}}\right\} \ (i = 1,...,l;\ j = 1,...,k;\ \forall VHC_i \& \forall VHC_j \exists SSP_i) \tag{7.3}$$

where R_i and R_j are inter-depended MOs located on the consecutive levels. If there are multiple dependencies, as shown by Case 4, the maximum latency of all of these dependencies should be considered in the latency calculation. If the timing restriction is satisfied for Case 4 and logic/area is still available, then another MO can be added to the sub-ACG tree. This MO could also be from the L_{i+1} level, that is not related to the MOs in Case 4, as shown in



(a) Case 5 (b) Case 6 (c) Case 7

Figure 7.11: SSP Composition from VHCs located on Consequent Levels of SG with partial dependency: Case 5, Case 6, and Case 7

Case 5, depicted in Figure 7.11 (A). This is also very similar to Case 5, where MO_2-Node#14 on level L_i provides input to MO_3-Node#15 and MO_1-Node#16 on level L_{i+1}. At the same time, MO_1-Node#13 is independent from all other MOs in this particular SSP and should be considered separately for timing constraint. Therefore, timing calculation for this scenario is:

$$T_{SSP_k} = MAX\left\{T_{R_o}, MAX\left\{T_{R_i} + T_{R_j}\right\}\right\} \tag{7.4}$$

where R_o is a resource on level L_i or L_{i+1} which is not related to R_i and R_j. An expanded version of Case 6 is where even more MOs are added from L_{i+1}. In Case 7 new MO creates second dependency between the L_i and L_{i+1}: MO_2-Node#6$\leftrightarrow MO_4$-Node#8 and MO_1-Node#4$\leftrightarrow MO_3$-Node#7. At the same time, these pairs of MOs are dependent on each other. Similar to the Case 6, the time restriction is compared to the maximum time of two sets of dependencies:

$$T_{SSP_n} = \max\left\{\forall Dependent\, MO_i\right\} \tag{7.5}$$

At last, a case where design should exclude specific area of the device from being used for fault tolerance applications, is considered. At this point every level of dependency graph is populated with an MO that contains particular area constraint. In the process of SSP creation every SSP is first populated with a "dummy" MO node. The rest of MOs are added after to

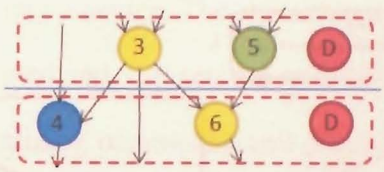


Figure 7.12: Case 8: Area Avoidance MO

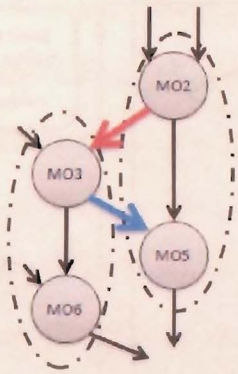


Figure 7.13: MO Deadlock Example

create an SSP, as shown in Case 8. "Dummy" MOs in every SSP contain a parameter which specifies the area that MO occupies, thus preventing other MOs from being placed in the fault section of the reconfigurable device.

7.4.1 Precaution Regarding Deadlock in MO Segmentation

An additional rule check has to be performed on consecutive segments which should be chosen in a way that they will not have bidirectional dependency on each other. If two segments are interdependent it creates a deadlock situation. Therefore, a segment should contain MOs from same or consecutive levels. Otherwise, the segments will be waiting for inputs from other segments indefinitely. An example of this is shown in Figure 7.13.

In this example, MO3 which belongs to SSP_{i+1} depends on the processed data from MO2 that belongs to SSP_i, as shown by the red arrow. At the same time, MO5 which belongs to SSP_i, and depends on processed data from MO3, shown by the blue arrow. Therefore, even

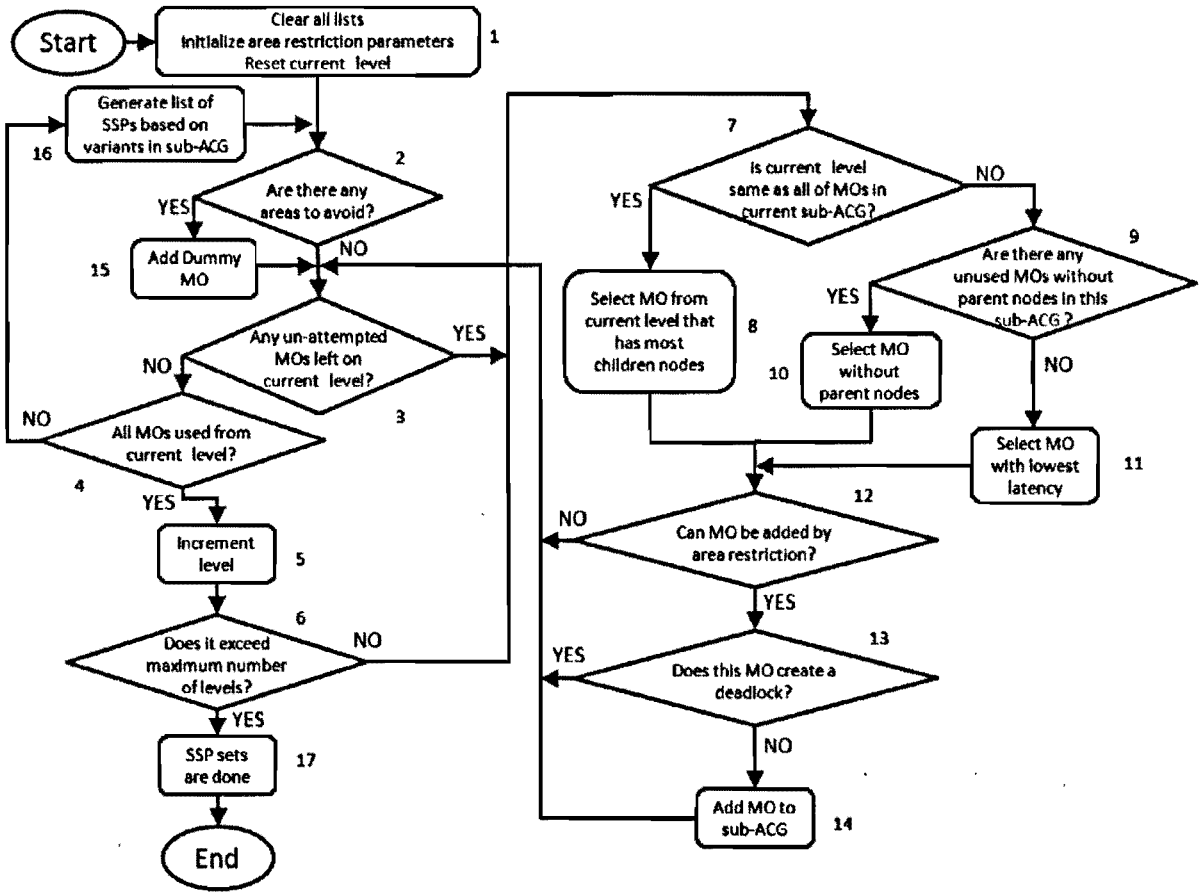


Figure 7.14: Flow Chart of the SSP set Generation Algorithm

though the right segment is loaded first, because it depends on the input from the left segment, it will not be able to produce result since it is waiting for the input from MO3.

7.4.2 SSP set Generation Algorithm

From the cases described in previous section an algorithm can be formed that performs generation of SSP sets. Its full flow-chart is presented in the Figure 7.14.

For algorithm to operate it requires a user to reset all parameters and initialize global area/logic constraints, as well as to specify if fault tolerance has to be built into the design. In addition, for all of the MOs that are used in the algorithm there must be at least one VHC

implementation that can fit the target device. The algorithm it is of iterative nature and operates until all of the MOs are segmented and SSP sets are generated.

Algorithm starts with initializing parameters and checking if fault tolerance is required, which is set by condition #2. If condition #2 is true, then a dummy MO is added which avoids the specified area. Condition #3 at that time checks if there are any MOs left that were not attempted to be fitted into the sub-ACG graph. If condition #3 is satisfied, the algorithm proceeds to the actual procedure of selecting most suited MOs for sub-ACG. In the steps 7 to 13 the actual selection is performed. If condition #3 is not satisfied, then a subsequent check #4 is made. It identifies if there are any MOs left on the current level that were not added to the sub-ACG. If there are some unused MOs remaining on the level, that means that they did not satisfy constraints and moving up to the next dependency level is not allowed. At this point SSP set is finalized in operation #16 and sub-ACG is cleared. When all of the MOs are used from same level a current_level can be incremented by operation #5 and selecting MOs can be continued, provided that there are MOs remaining.

Condition #7 identifies if the current level being explored is different from the ones that were added to sub-ACG previously. This condition indicates if selection proceeded to the next level and that there are possibilities of having dependencies between MOs in the sub-ACG tree. It should be avoided because latencies of dependent nodes have to be added together for timing calculation. Hence, condition #9 checks if there are MOs that can be selected that are not depended on the MOs that are present in the current sub-ACG tree. If there are no such MOs then lowest combined latency should be selected.

At a point when MO is selected, two additional checks #12 and #13 are done to make sure that the MO satisfies the area restriction, and does not create a deadlock. If both of these conditions are satisfied, then MO is added to the sub-ACG tree and operation is repeated by proceeding to condition #3.

On every addition of the MO to an sub-ACG graph, the MO is added as a single leaf of the tree. Following the MO addition, each of the VHCs are expanded as new leaves of the sub-ACG tree. New leaves contain area parameter summation of previous VHCs, as well as, other parameters (e.g. time, power, etc...). If these parameters violate initial restrictions they are discarded and not expanded in the further additions of MO. In the worst possible scenario of this algorithm every MO would be expanded with all of its possible variants and algorithm's complexity would therefore be $O(n^2)$. However, due to the nature of the algorithm, sizes of VHCs and limited number of variants that are available for each MO the overall number of calculations is not exhaustive.

Based on this algorithm a Windows application was created with a GUI interface that performs all of the above steps in order to create a set of SSPs and their schedules. This application is described in detail in the next chapter. Next section has an example to illustrate the operation of the algorithm. It illustrates creation of a segmented set of VHCs based on an initially given SG.

7.4.3 Example of Segment Specific Processor Synthesis

This section presents an example to show how the above algorithm creates a set of SSPs. Creation of a first set of SSPs is shown, as well as a the final result of algorithm segmentation based on the initial restriction parameters.

In this example a processing task is used that was mentioned previously in Figure 7.5 (A). For this particular example constraint of 20 CLBs, and, 20 clock cycles was assumed, where CLBs the are area of the device, and clock cycles represent maximum permitted latency. To illustrate the operation of the automatic VHC selection and SSP generation this example will traverse the algorithm through several stages.

The procedure for assembling VHCs into SSPs consists of the following:

1. ASAP level dependency algorithm is executed to assign an appropriate dependency level to each MO.

2. MOs are added to the sub-ACG graphs with their corresponding VHCs. A set of VHCs that satisfies user's constraints is selected and grouped into SSPs.

3. SSPs who's performance parameters are closest to the specified restrictions are selected for bitstream generation. A structure file is created with specification of the temporal data locations on the external SRAM memory.

4. Bitstreams along with the structure files are composed into a scheduled temporal partitioning system.

The level dependency algorithm was described in Section 7.2.3 and the level assignment is shown in Figure 7.5. All MOs are assigned a dependency level, and then re-arranged to form a level arranged task, as was done in Section 7.2.2.1. At this point the SG is ready for the segmentation. The VHCs that are available for the selection need to determine first. From the task which is described by the Figure 7.5four different MOs can be identified.

Table 7.1 shows the 4 MOs and associated set of VHCs for each MO with the performance parameters. These parameters are used in creating SSPs for this example.

The first step of segmentation is to create of the sub-ACG graph, as was mentioned in Section 7.3. To start the creation of sub-ACG, an MO has to be selected from the SG. Based on the methodology described in Section 7.4, MOs are selected from the top level, proceeding to the consecutive levels. Also, selection priority is given to the MOs that has dependent MOs on the next level. From the case 1, shown in Figure 7.8, MO_3-Node#2 is selected as the first one to be added to the ACG tree. This results in the three branches, since MO_3 has three versions of VHCs as shown in Table 7.1. The sub-ACG resulting after addition of the MO is shown in Figure 7.15.

Table 7.1: MOs and Corresponding VHC parameters

	MO_i	$VHC_{i,j}$	Logic requirement (CLBs)	Latency (clock cycles)
●	MO_1	$VHC_{1,1}$	22	2
	MO_1	$VHC_{1,2}$	16	5
	MO_1	$VHC_{1,3}$	8	12
	MO_1	$VHC_{1,4}$	3	20
●	MO_2	$VHC_{2,1}$	8	2
	MO_2	$VHC_{2,2}$	4	8
	MO_2	$VHC_{2,3}$	2	16
●	MO_3	$VHC_{3,1}$	12	4
	MO_3	$VHC_{3,2}$	10	6
	MO_3	$VHC_{3,3}$	6	12
●	MO_4	$VHC_{4,1}$	35	7
	MO_4	$VHC_{4,2}$	17	15

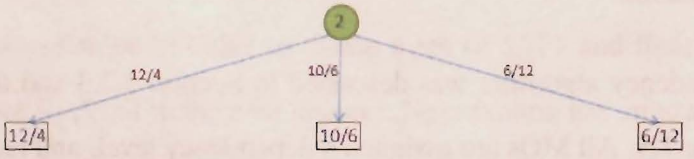


Figure 7.15: Sub-ACG After Addition of the First MO from the Task SG

Next MO to be added is either MO_3-Node#1 or MO_2-Node#12. MO_3-Node#1 is selected, since it has dependent nodes on level 3, where MO_2-Node#12 has dependent nodes on level 6. Since the criterion for both of these MOs is the same, the tree is simply expanded by adding MO_3-Node#1 to every single child of MO_3-Node#2. The resulting sub-ACG tree is shown in the Figure 7.16. As seen in the Figure 7.16, several children of the resulting sub-ACG are coloured gray. The reason for that is the violation of restriction parameters. As stated earlier in the chapter, violated leaves and branches are discarded, and thus decrease the amount of calculation when needed to add a new MO to the sub-ACG. Note that criterion did not need to be compared for these nodes, since they belong to the same MO. Because there are several leaves that do not violate restriction parameters, sub-ACG graph can be expanded further by addition of another MO. The last remaining MO on the Level 1 is MO_2-Node#12. Criterion for both MO_2 and MO_3 is the same, therefore, the sub-ACG tree does not have to be re-ordered.

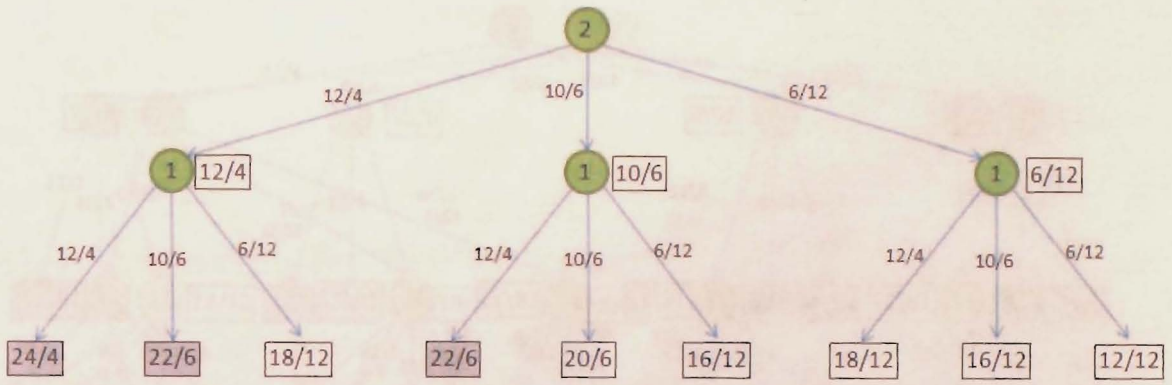


Figure 7.16: Sub-ACG Tree after Addition of the $MO3$-Node#1 from Task SG

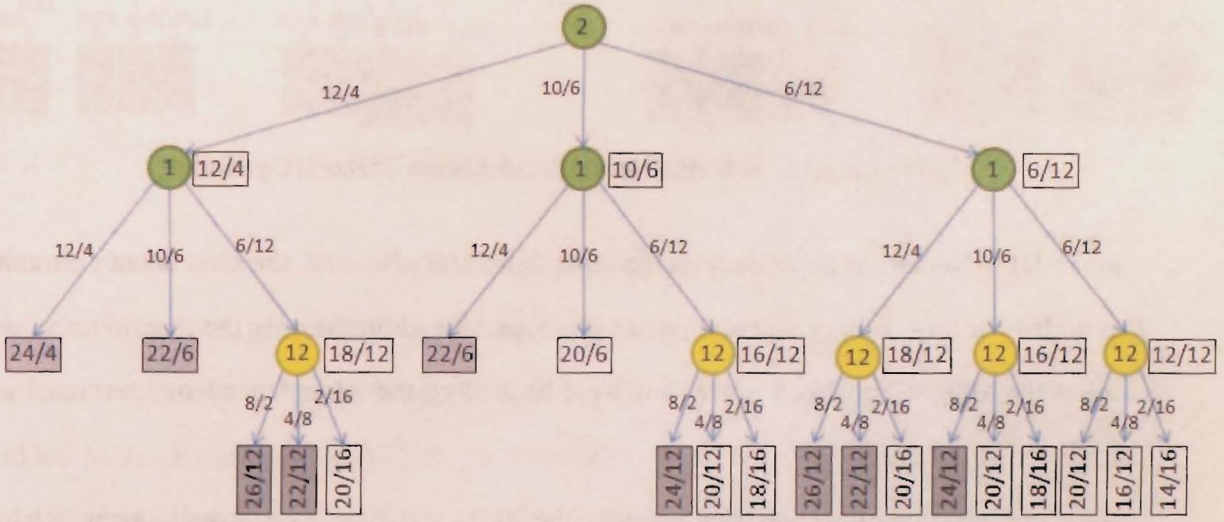


Figure 7.17: Sub-ACG Tree after Addition of the MO_2-Node#12

As in the previous step, MO_2-Node#12 is added to every non-violated child and the resulting in sub-ACG graph shown in Figure 7.17.

Similarly to the previous case, as with addition of MO_3-Node#1 there are some leaves that do not meet restriction parameters. These branches are removed and not expanded in the further MO additions. At this point all of the MOs from the Level 1 are added to the sub-ACG, however, there are leaves that can be expanded even further. Therefore, selection is moved to the next level. Since both MOs on Level 2 are equally depended on MO on Level 1 any of them can be picked at random. MO_3-Node#5 was selected and added to all of the non-violated leaves of sub-ACG tree it results in Figure 7.18. It is important to note that

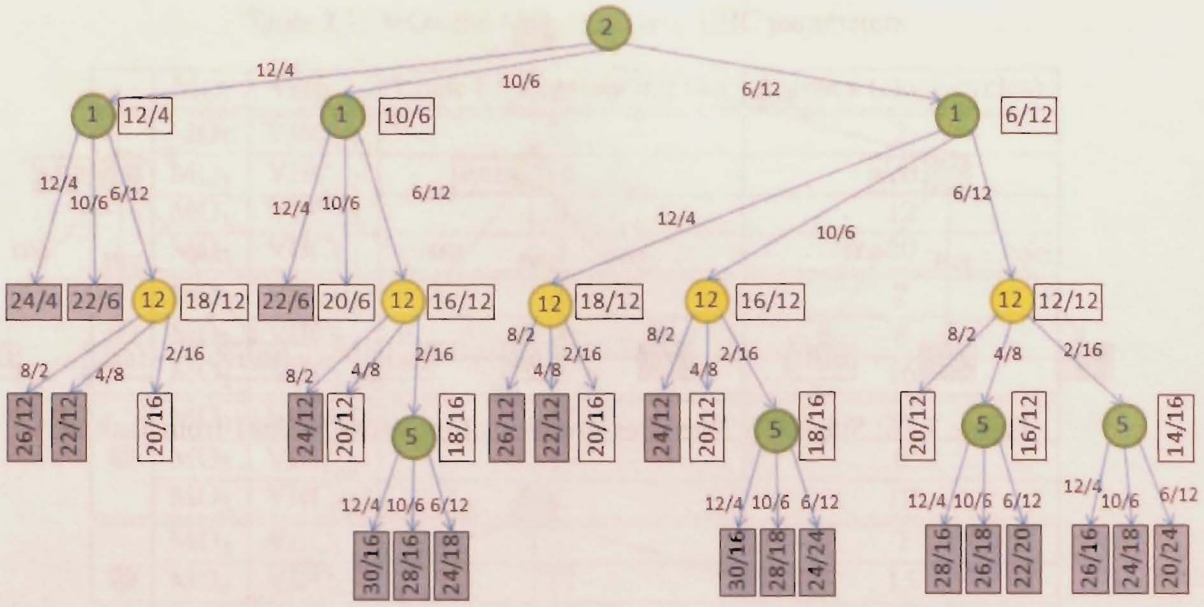


Figure 7.18: Sub-ACG tree after Addition of the MO_3-Node#5

since MO_3-Node#5 is dependent on the input from MO_3-Node#2, the time latency parameter is added for total latency calculation. In previous MO additions only the maximum value of all of the MOs where taken, since they were located on the same dependency level and were executed simultaneously.

After insertion of MO_3-Node#5 none of the leaves satisfy all of the conditions which leads to removal of MO_3-Node#5 from the sub-ACG graph. Assuming there are MOs remaining on that level, an attempt should be made to try to fit remaining MOs. In case of the task from Figure 7.5, it can be seen that MO_2-Node#3 is one that is remaining on the level 2. After addition of the MO_2-Node#3 to the sub-ACG graph it is found that there are two leaves that satisfy all of the restriction parameters. New arrangement of the sub-ACG graph is shown in the Figure 7.19.

Even though there are variants that can be expanded even more, in this scheme it is not allowed, since none of the unused MOs is present on the Level 2. Also, because there is at least one MO that did not fit to sub-ACG tree, the selection of the next MO from the consecutive level is not allowed either. Theoretically, it is possible to insert another MO from

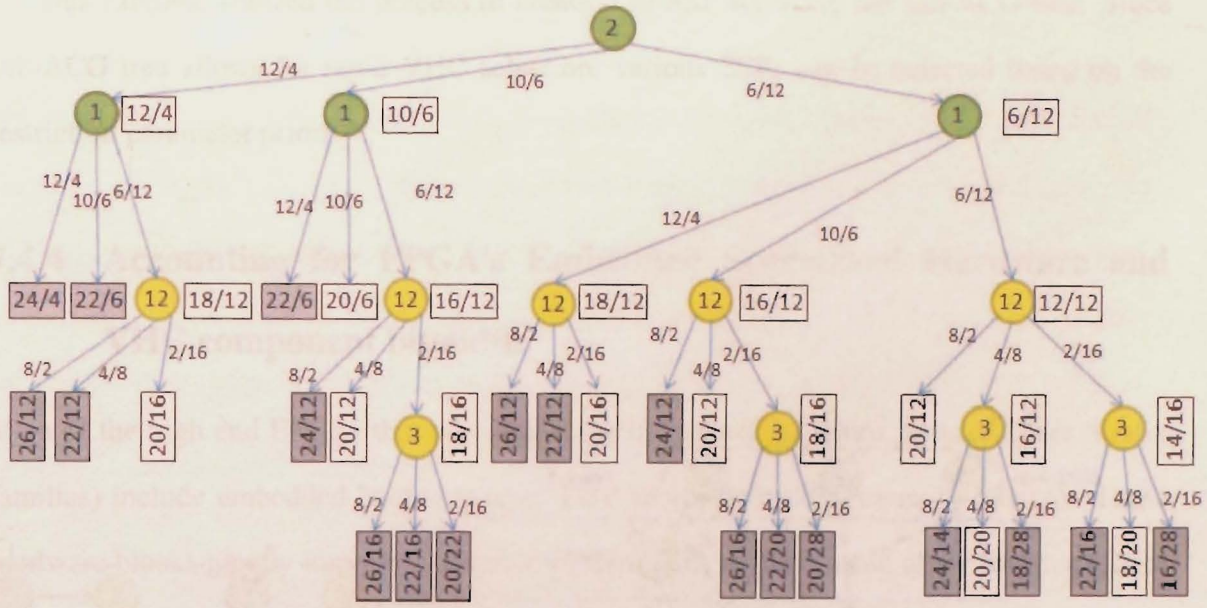


Figure 7.19: Sub-ACG tree after Addition of the MO_2-Node#3

the next level, however, this creates a large pool of variations and can lead to an exhaustive search. Therefore, proceeding to the next level, is not allowed until all of the MOs on the current level are used up. At this point there are two SSPs that can be selected from the four added MOs. By traversing through the branches of sub-ACG tree shown in Figure 7.19 it can be seen that $SSP_{1,1}$ is composed of $VHC_{3,3}$-Node#2, $VHC_{3,3}$-Node#1, $VHC_{2,2}$-Node#12, $VHC_{2,2}$-Node#3. The resulting *Area* is 20 CLBs and *Latency* is 20 CC. By the same procedure $SSP_{1,2}$ is composed of $VHC_{3,3}$-Node#2, $VHC_{3,3}$-Node#1, $VHC_{2,3}$-Node#12, $VHC_{2,2}$-Node#3, with resulting *Area* of 18 and *Latency* of 20. At this point sets of VHCs corresponding to MOs have been identified and selected for SSPs. Depending on the priority of the performance parameters, one version of the SSP can be selected for the final bitstream generation. In the case of this example it is the area parameter, and therefore, $SSP_{1,2}$ is the right most possible variant. Following same procedure of MO selection based on the methodology outlined in the Section 7.4 the rest of the SSPs are generated. The resulting segmentation of the Task into SSPs is shown in the Figure 7.20 (A). Figure 7.20 (B) shows the SSP re-arranged MOs of the Task.

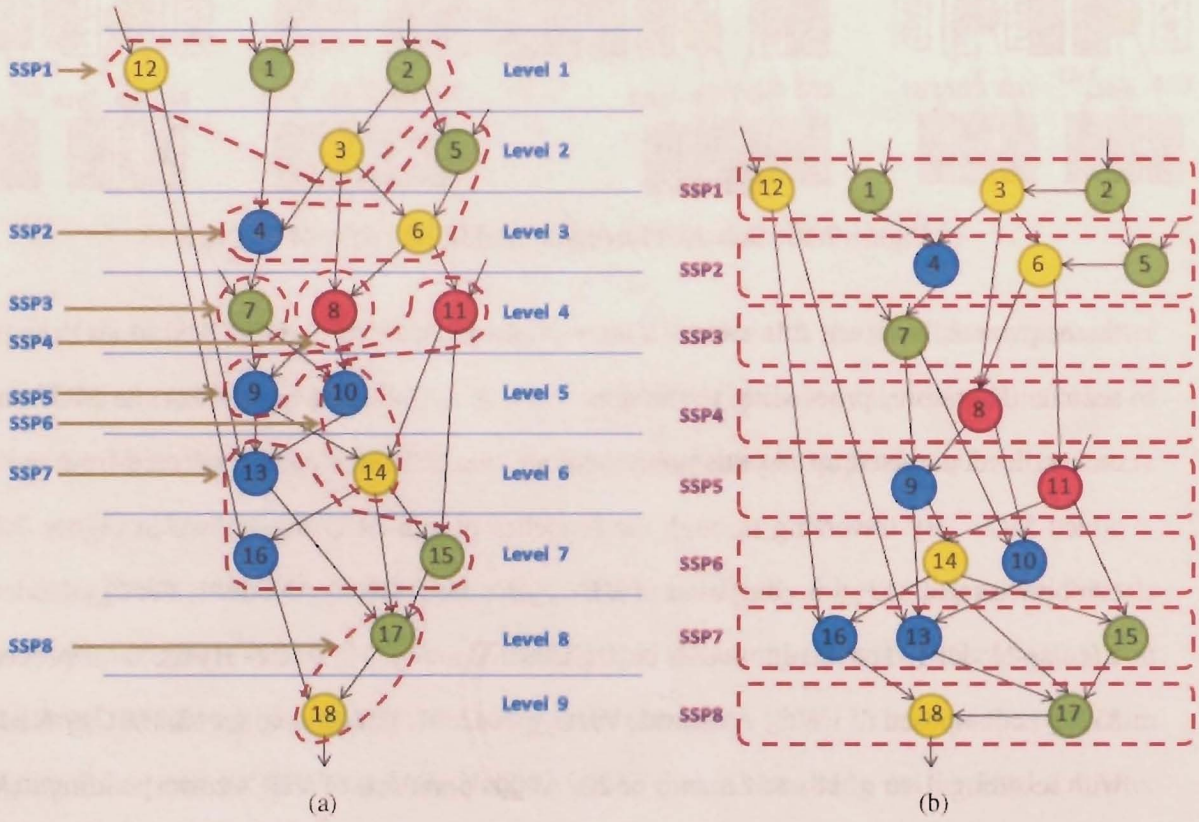


Figure 7.20: Final Segmented SG Implementation on the Set of SSPs

This example showed the process of creation of SSP set using the sub-ACG tree. Since sub-ACG tree allows for rapid VHC selection, various SSPs can be selected based on the restriction parameter priority.

7.4.4 Accounting for FPGA's Embedded Specialized Hardware and VHC component bitwidth

Most of the high end FPGAs that are used in the industry (e.g. Xilinx Virtex, Altera Stratix families) include embedded hardware (e.g. DSP slices, embedded memory blocks). These hardware blocks greatly improve utilization of the FPGA resources and allow for much more cost effective design. The CAD software can also benefit the design by utilizing embedded specialized hardware resources (e.g. DSP blocks, hardware multipliers). If a target device is specified prior to the implementation of an SG, then CAD software can estimate the amount of resources that can be mapped to the embedded hardware during place and route procedure. This would allow to utilize more of the FPGA resources in much more cost-efficient manner.

Another aspect of the SG segmentation that comes up is the datapath bitwidth of the algorithm/task and how it translates from the description in terms of MOs to implementation in VHCs. In any datapath of the algorithm each connection between the MOs has a particular bitwidth depending on the application. In the example shown in previous section the overall operation of the algorithm was demonstrated and avoided the use of embedded hardware. However, in the case of the VHC selection this aspect is considered. Following the same selection algorithm VHCs that do not satisfy the datapath width requirement are pruned and VHC variant search list is reduced. The specification of the datapath width is done by the parameter specification of the MO, similarly how the connectivity of the MO is specified. In turn, when VHCs are being generated one of the parameters which is used for generation of the VHCs is datapath bitwidth.

7.5 Summary

This chapter covers the proposed novel methodology for synthesis of a set of Segment Specific Processor (SSPs) optimized in multi-parametric design space for a given task. The task segmentation procedure is discussed in detail, as well as the actual SSP synthesis and optimization process. This process is described with all major steps which included identification of dependencies between macro-operators in the task algorithm, associated segmentation, and selection of near optimal set of VHCs for each segment specific processor. Depending on the parametric constraints, segmentation may vary resulting in different SSP compositions. This chapter also presented the algorithm for creating SSPs that was implemented in GUI application described in Chapter 8. For illustration of the proposed methodology a step by step example of a task segmentation was shown with full task segmentation arrangement which resulted in 8 SSPs. The next chapter will focus on the architecture design and implementation of the software, hardware, and firmware components of the temporal partitioning mechanism.

Chapter 8

Implementation of the methodology of SSP synthesis and execution

8.1 Introduction

The previous chapters described various methodologies for creation of the MOs, VHCs, and SSPs. The purpose of this thesis was not only to propose methodologies and architectures, but also to implement and test all of the described methodologies and architectures in a complete RCS which incorporates TPM. Implementation encapsulates several aspects including software, hardware, and firmware development. Implementation of CAD software is described in the next section, followed by the system implementation of the proposed architecture, and concluded with proposed reconfigurable device on-chip architectures. Therefore, the implementation part of thesis covers all levels of design of RCS with TPM from the top level (CAD) to the level of system and on-chip architectures.

8.2 Implementation of SSP Synthesis and Optimization Methodologies in the CAD System

As mentioned in the introduction, one of the aspects of the research was to design the CAD software that would implement the methodologies of VHC and SSP generation. Since the methodology of task segmentation requires many computational steps, as it was described in Chapters 6 and 7, operation of task segmentation and SSP generation cannot be done efficiently by the user alone. A CAD support is definitely needed for the user. Therefore, a Segment Partitioning Creator (SPC) CAD software was created in Visual Studio.NET environment. It provides the user with the GUI for creation (programming) of the task algorithm in a form of Sequencing Graph (SG). User is required to create an SG of the task from an existing set of MOs and interconnect them according to dependencies in task operation. Specification of system performance parameters such as: width, timing restrictions and other constraints should be specified by the user. CAD tool is then performs the proper selection of component instances that were chosen for processing the algorithm. Selection is performed based on the set of multi-parametric restrictions that were specified by the user.

The CAD tool automatically conducts level dependency division of the SG, based on the algorithm described in Chapter 7. After completion of the level division the CAD tool performs segmentation of the architecture by the algorithm. CAD tool allows optimization of one of the most critical parameters (e.g. area, latency, cycle time and power) as specified by the user, while keeping the other parameters within the restriction range. It can also perform bus width modification/replacement, if higher performance is needed, or if area has to be minimized. The restrictions are tied to the reconfigurable device specifics and can include area avoidance, power consumption and execution time limits, and more.

Though not in current implementation, CAD tool should be able to estimate and suggest a optimal device for a target design. It should also be able to suggest a range of optimal

reconfigurable devices based on different variations of parameters that can potentially change in the future.

When final selection of VHCs is done the CAD tool generates a broad spectrum of SSPs, where each SSP is responsible for a particular variant of conditions, such as speed of processing, area requirement, power consumption, latency, and other parameters. The user is also capable of selecting the number of variants that should be generated and the parameters to be optimized. Another factor that has to be considered, is the generation of time SSP bitstream set. Since each segment forms an SSP IP-core, a set of SSP cores need to be synthesized for each of the variations of the task algorithm. Therefore, if an algorithm has K segments, each having N_i variations, where $i = 1,2,...,k$, for the algorithm implementation there are $\sum_{i=1}^{K} N_i$ SSPs that have to be generated into the bitstreams. Adding fault tolerance capability to avoid different sections of FPGA increases, the number of generated as well. The main factor is the granularity of the sections G, which defines how many different combinations of FPGA sections have to be avoided. Therefore, if fault recovery is added as one of the restrictions, then number of generated SSP cores increases up to $N_{SSP} = G \times \sum_{i=1}^{K} N_i$. Due to that fact, it is crucial for the CAD tool to estimate and eliminate the variations that do not satisfy the given restrictions. For example, the compilation of a single SSP core requires $T_{comp}^{SSP} \simeq 300 \rightarrow 600$ seconds [63] for XCV1000 FPGA, and the total compilation time is given by:

$$T_{comp} = T_{comp}^{SSP} \times G \times \sum_{i=1}^{K} N_i \tag{8.1}$$

The total compilation time for SSP generation can be estimated with the following approximate data:

- Number of task segments and associated SSPs, $K = 8$;
- Number of possible variations of each SSP, $N_i = 4$;
- Granularity of sections of FPGA area, $G = 4$ (4 quadrants);

- Approximate time for compilation of one SSP to be accommodated in 1M system gates, FPGA $T_{comp}^{SSP} = 300s$ (according to [63] for the Xilinx Virtex XCV1000)

In this case :

$$T_{comp}^{Total} = 4 \times 8 \times 4 \times 300sec = 38400sec = 10.6hours$$

This would allow RCS to work in $N_i^K = 4^8 = 2^{16} = 65536$ possible modes, and restore the functionality for each of the modes, in case if one of 4 quadrants in the FPGA logic/routing area would get a permanent hardware fault. The transient faults can be mitigated by scrubbing procedures as described in Chapter 4. Therefore, the automated generation of the required set of SSPs for a task partitioned in 8 segments could be done within one business day.

The benefit in the use of the SSP approach capability of future modifications. If a particular MO, such as FIR filter, requires a modification, only the VHC for that particular MO would be modified. The only SSPs that need to be re-synthesized are the ones containing the modified VHCs. This in turn brings a dramatic reduction of the re-design time for any modifications, compared to the time for HDL reprogramming and re-synthesis for the whole design. Another advantage of the SSP set, as mentioned previously, is that it allows a run-time adaptation, since all of the SSP bitstreams are stored on the non-volatile FLASH memory.

To minimize the time for SSP generation, the CAD tool first performs estimation of variants. Actual generation of the set of SSPs occurs only in the case when estimated processing times of the algorithm match the restrictions imposed by the user and existing/selected hardware. Since placement and routing requires most amount of time in SSP generation, the estimator plays the crucial role in filtering out the unfeasible variants.

When the generation of SSPs is complete, the user is provided with a set of directories, each containing a set of SSPs in form of bitstreams. Along with a schedule of configurations and a global file that specifies which set of SSPs to use in a case of parameter changes. Such parameters can be power reduction, time requirement modification, restricted area, etc. The sets of SSPs are then stored on a non-volatile memory and used based on the schedule. Figure

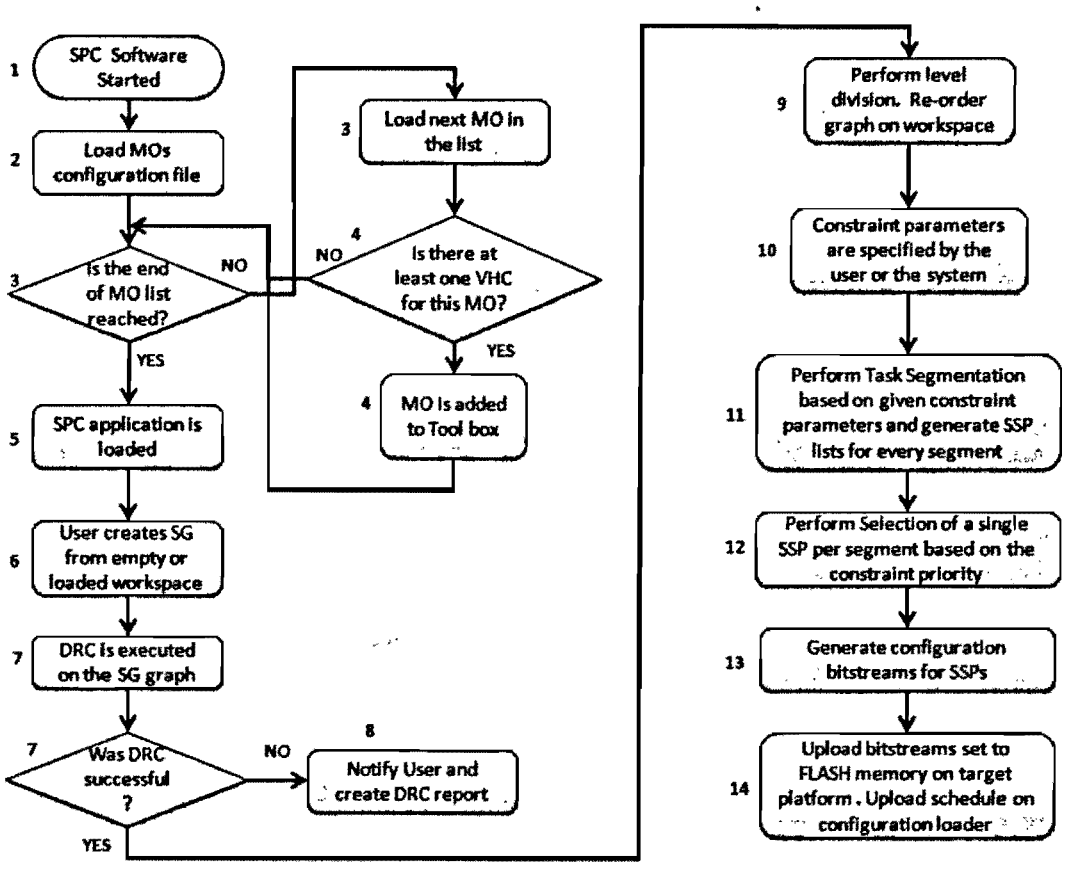


Figure 8.1: CAD Software operation flow chart

8.1 shows all steps of the CAD software operation. A number beside an operation refers to the operation's order in the schedule.

The sequence of software operation is as follows:

1. The SPC software is initiated by the user on a PC.

2. As it loads, the SPC software searches for the configuration the file *mos.mo*, which contains a library of existing MOs. This library specifies the name of an MO, its description and its associated icon. Configuration file also contains all the IDs and locations of all VHCs that implement each MO.

3. Configuration file is parsed and MOs are populated into the MO *Tool box*.

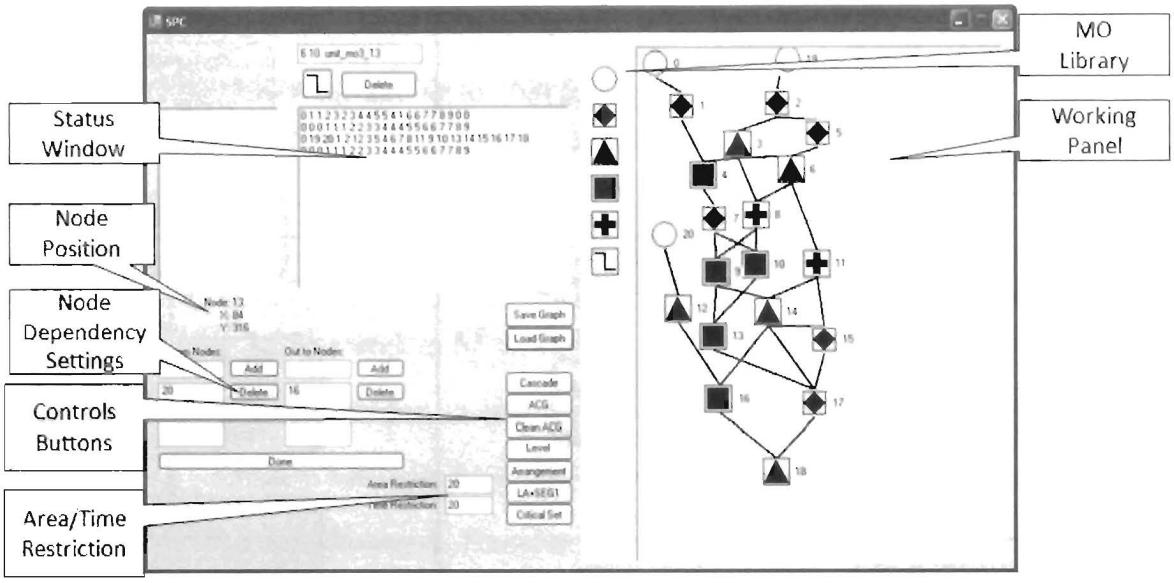


Figure 8.2: GUI Application

4. Every MO is checked if it has at least one VHC, and if it does not, it is removed from the *tool box*. At this stage, however, VHCs are not loaded to conserve the operating memory resources.

5. User is presented with the GUI that allows user to drag the MOs from the *Tool box* to the *Working panel*.

6. An SG is created by the user either from scratch on empty work space, or by loading a previously saved SG and modifying it in any way needed. MOs can be interconnected as they are added, or they can be connected after all MOs are added. At this stage SPC software loads the VHCs associated with the added MOs into an internal list. SPC application creates a tree-like linked list of *Node* objects, where every node has a reference to MO type and incoming and outgoing nodes.

7. When the SG of the task is completed, Design Rule Check (DRC) performed to ensure that no erroneous connections are present. SPC checks that all of the inputs are present to every MO requiring an input. Inputs could be: external input to the system; input from some other MO; input of a constant value. Also SPC checks for at least one output from the system.

8. If DRC fails then user is presented with an error report.

9. Application continues on performing *Level Division/Segmentation/SSP generation*, that can be run separately. The advantage of separate invocation is that user can modify the SG if *Level division* or *Segmentation* error is identified, or some other modification has to be done. *Level Division* of the SG is performed based on the method described in Section 7.2.3. After the execution of this stage every MO node is assigned with the specific level. SPC application also re-arranges the SG based on the levels giving the user much clearer view of how the MOs are organized in the dependency manner.

10. *Constraint Parameter Specification* is done at this step where user inputs the restriction parameters according to the procedures described in Chapter 7. It is also possible for system to automatically select restriction parameters based on the initial constraints of a reconfigurable device specified at the initialization of the project.

11. *Segmentation and SSP generation* is the key component in the SPC application. It implements the segmentation methodology described in Section 7.4. An MO is selected based on the methodology shown in Section 7.4, and added to a sub-ACG tree. The sub-ACG trees are represented with a heap data structure. The leafs of the sub-ACG node represent the VHC variants of the MO with the corresponding performance parameters, as was described in the Section 7.4.2. Every new added MO creates a new level of VHC combinations on the sub-ACG tree. At the same time, two temporary lists are created that hold the current and previous level of VHC node sets in the ascending order. The temporary list for *previous level* is used to recall the last successful level in the case that none of the leafs of the *current level* list satisfy the restriction. Latest level that satisfies the restriction contains the combination of VHC variants. These variants are stored as SSPs in the final SSP list. Each of the SSPs contains the information such as: specific VHC version, interconnection of VHCs, total area/power/latency, and VHC file location. When all of the VHCs are selected for the MOs presented in the SG, and the segmentation of these VHCs is completed user is presented with the list of possible SSPs

for each segment. There is a list of SSPs due to the fact that each segment can have several combinations of VHCs that satisfy the imposed constraints, as shown in Figure 7.19. The software segmentation operation is based on the flow chart shown in Figure 7.14.

12. *SSP Selection* for bitstream generation is done based on the parameter priority. This step can be done together with the previous one, or initiated later by the user. The option of later execution is provided for the user's additional flexibility. Before the bitstreams are generated a report with lists of SSPs is presented to the user and the user can modify restrictions or priorities to better fit the system's requirements. The *SSP selection* is initiated as the SPC application goes through the list of SSPs in each segment and selects the SSP that is closest to the constraint of highest priority. If two SSPs have the same parameter for a particular constraint, then they are compared by the constraint of the lower priority.

13. When SSPs *are* selected, each of them is compiled into a loadable configuration bitstream. This is done by invoking the design suite, such as Xilinx ISE 11, or Altera Quartus II. The design suite performs the synthesis, translation, mapping, and place and route of the combined VHC modules.

14. The result of the operation is a set of bitstreams with a configuration file of the schedule for reconfiguration. Bitstreams are then uploaded to the non-volatile memory on the target platform. Configuration file is uploaded to the Configuration Scheduler/Loader for later execution.

8.2.1 Area Avoidance Implementation

In case of mitigation of permanent hardware faults, an SSP has to be designed to avoid a specific area on a reconfigurable device, as it was described in Chapter 4 and Chapter 7. For this purpose a specialized "dummy" MO is inserted into every level of the level divided graph. When SSPs are assembled, each of them contains the VHC corresponding to the "dummy" MO. "Dummy" MO essentially represents a setting in the constraint file that directs Xilinx

ISE compiler to avoid the specified area during the place and route operation. In the CAD software an algorithm that performs optimal selection of VHC treats the area-avoidance "dummy" VHC as a automatic selection with highest priority. In this case CAD tool does not replace the VHC. As it can be estimated, number of VHC variants with an associated MO is equal to number of all possible area avoidance restrictions. As mentioned in previous chapters, FPGA can be divided into several tiles, therefore, VHC parameter contains the specification of the granularity of $n \times n$ tiles and which particular tile it is. It is important to mention that an SSP can contain several different areas to avoid. Thus, with an adequate fault detection mechanism, it is possible to have reconfigurable device to continue to function with more than one permanent hardware fault.

The first version of the above CAD software application provides the user with all of the features discussed in this section and is to be expanded further in the later sections.

8.3 System Level Architecture to Accommodate TPM Based on SSP Processing

Another major aspect of the research is the design and implementation of the hardware platform that would be able to execute the generated set of SSPs. This, however, involves a design of several different sub-systems each playing its own important role in the TPM operation. The intent of the platform design is to implement and test most of the research aspects of this work. Platform was to be able to process the tasks which can have multiple modes of operation and to have a capability for rapid adaptation to a different mode [53, 55]. Hence, the platform is called Multi-mode Adaptive Reconfigurable System (MARS). This section describes the platform architecture and the associated hardware design solutions.

8.3.1 Reconfigurable Field of Resources (RFR)

In the Section 3.3.6 two approaches of reconfigurable device architecture were described, which included single FPGA device and triple FPGA device configurations. In the triple configuration one FPGA performs the memory management and interfacing functions and the other two are used for SSP processing, and reconfiguration respectively in order to "hide" the reconfiguration overhead. In the MARS platform, different research concepts were implemented:

1. Temporal and partial reconfigurations.
2. Rapid system reconfiguration/adaptation.
3. Use of SSP library resources.
4. Processing streaming applications.
5. Self-reconfiguration for fault recovery.

A single partially configurable FPGA was selected for the implementation. FPGA selected had to have: rapid configuration interfaces, on-chip configuration controller and all the necessary computing resources for stream processing of task implementation. As was explained in Chapter 3, Xilinx Virtex 4 family was chosen with FF1148 ball grid array package. This was an ideal choice for the platform since five different types of devices in a Virtex 4 family could be used (from 4 million to 16 million system gates). Larger device also contained impressive amount of embedded hardware such as: 152064 configurable logic cells; 5 Mbits of integrated memory; 96 DSP slices; and 12 clock managers. This family of FPGA device has support for SelectMAP32, which currently provides the highest available configuration bus bandwidth (3.2 Gbit/sec). On the MARS platform XC4VLX160 and XC4VLX80 were used, and testing of the partial and temporal configuration methodologies was conducted.

8.3.2 SSP Configuration Mechanism on MARS Platform Design

Based on the research aspects described in Chapter 5, highly parallel configuration interface was used to perform rapid configuration. To verify the temporal partitioning in run-time adaptation, several configuration options were explored. To provide support for architectural experiments three configuration interfaces were included in the design: JTAG, Serial SelectMAP, and Parallel SelectMap32. These interfaces can be compared by their performance use in different architectural approaches. In order to accommodate the Parallel SelectMAP32 interface, a specialized loader had to be developed. IEEE1149.1 (JTAG) is the first configuration interface that is common to all [118, 66, 22] FPGA and ASIC manufacturers. It is used for the testing of internal modules, such as memory integrity, and other sub-systems. In an FPGA device it is used for communication with an on-board controller, that uploads a bitstream to the device. In addition, it is used for communication with soft-processors such as MicroBlaze, NiOS, ChipScope Pro [113]. In addition to JTAG, a proprietary serial configuration interface is present in most of the FPGAs. Historically, loading configuration over JTAG to an FPGA device was done from a designer's PC, however, this has lately changed.

In recent years most of industrial/commercial platforms manufacturers began to include the bitstream loaders that are capable of loading configuration bitstreams from an external FLASH memory card [112, 6]. This made field upgrades as easy as switching/replacing a memory card. Loaders, such as SystemACE from Xilinx [112], allow to implement completely stand-alone solutions with simple upgrade option of a CompactFlash card. SystemACE acts as a PC replacement, since it configures FPGA JTAG interface [117], and essentially implements whole JTAG protocol on a chip. However, the maximum speed of configuration is limited by the off-the-shelf memory. Currently highest speed of CompactFlash memory SystemACE is limited to 30 Mbit/sec, uploading configuration of a Virtex 4 Xilinx FPGA would take up to 2 seconds [112]. Hence, such solution is suitable for systems that do not require rapid start-up or reconfiguration times. There are other configuration options available that are

proprietary to the FPGA vendor (Xilinx-SelectMAP, Lattice-sysConfig, Altera- Fast Passive Parallel configuration port (FPP)).

The MARS platform uses Xilinx Virtex 4 FPGA withs a serial SelectMAP configuration interface. The interface includes a data signal D0, a clock signal, and several control signals such as chip enable. The advantage of this interface in comparison to JTAG is that it can be used in embedded systems. In this case a designer can create a custom configuration header operating at higher speed than JTAG provides and not worry about supporting JTAG protocol. Loader can be implemented as ASIC, there are some available from Xilinx XC04S. At the same time, it can be implemented on microcontroller/microprocessor with a FLASH memory or directly on an FPGA soft processor. MARS platform design was implemented with a combination of a microcontroller and a CPLD to allow a rapid data readout from the SSP library memory. This approach is mostly used in embedded solutions and is not designed for a PC-to-platform configurations, so it is well suited for MARS platform.

Lately there was an advancement in high speed configuration interfaces in several families of high-end FPGAs [83]. This was driven by the customer demand of because of the ever increasing size of configuration bitstreams. Due to large bitstream sizes the start-up time for some FPGAs reached several seconds. A Parallel configuration interface with the 8 bit and recently 32 bit bus was introduced by Xilinx. In addition, the configuration clock speed was increased from 25 MHz to 100MHz. As mentioned in Chapter 5 increasing configuration speed to 3.2 Gbit/s, is extremely beneficial for a system that has to support rapid reconfiguration or adaptation. This was one of the key requirements for MARS platform, so it was included in the design. Parallel SelectMAP32 configuration interface has same control and protocol of operation as Serial SelectMAP. Instead of single D0 data line there are 32 (D0-D31) data lines, which transmit data in parallel. Hence, by using same setup, as was done for a Serial SelectMAP, and programming the configuration loader/scheduler to output 32 bit words instead of 1 bit word, the maximum performance could be achieved.

With the three interfaces MARS platform can be configured with the standard JTAG interface, and a proprietary slow speed interface, proprietary high speed interface. The performance and comparison of effectiveness of these configuration interfaces can be compared between each other and other existing platforms.

The configuration loader and scheduler was chosen as suggested in Chapter 5. It involves a combination of the FLASH memory, CPLD, and microcontroller with various interfaces. Simplified architecture block diagram is shown in Figure 8.3. Since SelectMAP32 was chosen due to its bandwidth, the FLASH memory modules had to be organized in a way to accommodate 3.2 Gbit/sec bandwidth. For that purpose four 16 bit width NOR-FLASH memory modules, with capability of 50MHz operation were chosen. At the point of a rapid reconfiguration 64 bits of data are read in parallel at 50MHz of the FLASH modules. This information is packaged into 32 bit words and sent over the SelectMAP32 parallel configuration bus at twice the speed. As mentioned previously, CPLD is used for two main reasons: repetitive operation at high speed, and vast number of flexible I/O assignment. First of all,16 bits of data, 26 bits of address, and 5 control lines where needed in order to connect to four FLASH modules. For the actual SelectMAP32 interface there are 32 data and 5 control lines. Microcontroller to CPLD parallel bus required additional 10 lines. Overall configuration device needed total of 111 I/O lines. The only types of the reconfigurable devices that allowed such I/O count were CPLDs and FPGAs. At the time of MARS development CPLD was the only configurable device that operated on a single supply of 3.3V, and contained large number of I/O pins and did not require external loader. The reason for the 3.3V constraint was requirement of design portability. The portability means that the same loader can be integrated on different platforms with various families of FPGAs without a need for additional voltage regulation and loader redesign. Most of small sized FPGAs at that time required additional 3.3V and 1.5V or 1.2V voltage sources for auxiliary and core powers respectively. In addition, all SRAM based FPGAs required an external loader and associated peripherals.

Some microprocessors and microcontrollers that have the required pin count may seem to qualify as candidates for the configuration loader. However, when 100MHz of configuration frequency is taken in consideration, microcontrollers simply cannot keep up with the MIPS limitation. On the other hand, microcontroller plays an essential role as a configuration memory manager. Since CPLD is significantly limited in the amount of logic in comparison to an FPGA, it can only perform simple repetitive operations. So, in MARS the requests from the FPGA, core updates from PC, and scheduling are handled by the microcontroller. Microcontroller is based on the Microchip PIC18F8XXX microcontroller family. It interfaces to CPLD over the parallel bus and peripheral interfaces, such as USB, dual RS232, SPI, parallel interfaces, and button switches for direct user input. Microcontroller also provides a user with a very flexible and simple implementation of desired controller operations. This is because the microcontroller programming can be done in embedded C or assembly language.

The microcontroller in the MARS platform implementation involves several functions:

1. Downloading SSPs in a form of bitstreams over CPLD from a PC to a predefined FLASH memory virtual slot.

2. Storage and execution of a reconfiguration schedule.

3. Initiation of reconfiguration by requesting CPLD to configure a specific core residing in one of the FLASH memory virtual slots.

4. Providing user with feedback of current operation for monitoring purposes.

To provide the flexibility of application development a parallel communication bus was included between the FPGA, CPLD, and microcontroller devices, as shown in Figure 8.3, where 8 bits are data and 2 bits are for handshaking and control. This device interconnection provides flexibility in scheduler operation. A schedule of SSP configurations can be uploaded to the microcontroller by a request from a PC/external source, or directly by a request of FPGA.

The way TPM typically operates in MARS is as follows:

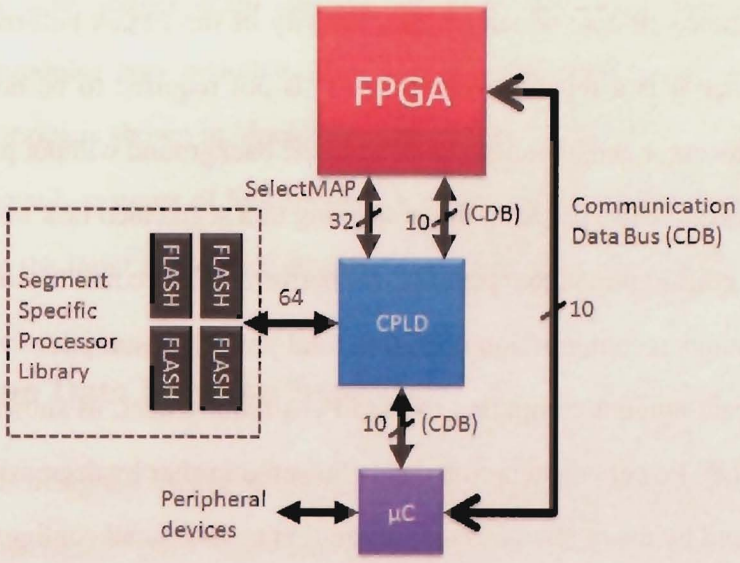


Figure 8.3: Communication Bus Structure Between: FPGA, CPLD, and Microcontroller

1. Microcontroller issues a signal for the reconfiguration to the CPLD and the starting memory address of the SSP core.

2. Reconfiguration signal causes CPLD to begin FPGA re-configuration procedure and follow the schedule of control operations listed in Chapter 3.

3. CPLD reads an SSP bitstream from the FLASH memory beginning from the address received by the microcontroller until the end of the bitstream.

4. Processing core retrieves temporal data from the temporal data memory (SRAM) and after processing, it writes a new set of data.

5. Based either on the timing schedule, or signaling from the FPGA the microcontroller fetches an address of the next SSP bitstream in the schedule queue to the CPLD, and the operation repeats again.

MARS architecture allows for dynamic operation not only by the programmed schedule, but also by a request of the reconfigurable device itself [56]. The bus between the CPLD and FPGA bus allows for direct request of a particular SSP bitstream. Such bitstream could

be a fault tolerance IP-core which checks integrity of the FPGA before/after SSP a reconfiguration. Since it is a repetitive operation, it is not required to be done by a microcontroller/microprocessor and, therefore, is done in the background without placing extra load on the microprocessor. This speeds up the processing of a segmented task even more.

The power consumption measurement was performed to account for total power consumption. In continuous reconfiguration operation total power consumption was in 66mW, which is an insignificant amount comparing to the FPGA requirement, as shown in the results section of Chapter 9. Power consumption can be lowered further by decreasing the frequency of configuration and by using low power configuration manager and configuration controller.

The other part of the configuration memory manager that was developed is the software application, as was mentioned in Section 8.2.

8.3.3 MARS Temporal Data Memory

The temporal partitioning it involves sequential reconfiguration of the FPGA device with the temporal data stored on the temporal data memory. Best suited memory for this task is SRAM, due to its rapid access time and control simplicity. After reconfiguration the processing core rapidly retrieves the stored data and proceeds with further processing. At every stage of processing the data being processed is placed into predefined memory spaces. For this purpose dual 72 Mb SRAM memory banks (CY7C1472) were included in the design. Since this platform is used in the development of multi-modal stream processing applications, it would also use this SRAM memory for frame buffering, hence, the dual bank configuration and this particular memory size. Since acquisition of video frames occurs simultaneously with their processing and the output of the processed images, the buffering of the video stream had to be implemented. The SRAM modules were implemented with separate address and data buses. One SRAM chip saves the data from the image sensor, while the other processes a previously saved frame. On the completion of the image capture/processing the banks switch operation.

For a larger and longer term storage of processed data two 256Mb SDRAM (MT46V64M4) modules was included, also were in dual bank configuration. Organization of these memories is shown in block diagram in Figure 8.4.

The remaining sub-systems of MARS platform are various interfaces that are needed for interconnection to the input and output devices, discussed in the next section.

8.3.4 Platform Data I/O Interfaces

MARS platform is designed for stream processing applications, hence, it requires several different types of interfaces. These interfaces and peripherals include: LVDS 400MHz 16-bit interface, 400Mbit serializer with coaxial interface, VME bus interface, 4 SVGA output ports, USB, RS232, LEDs, push-buttons, and other service interfaces.

The processing platform was specialized for video-stream applications, so the video input and outputs were required. For video output a standard video digital to analog converter (DAC) was selects. ADV7125 DAC supported up to 330 MSPS which allows to display a TrueHD at $120fps$. To perform a parallel processing of a number of video streams, four DACs were included in the design.

Next chapter talks about the "Fast Track" project which involved development of the high-speed stereo image acquisition system. For this purpose a special interface was designed for MARS platform. The overall block diagram and an actual photo of the platform is shown in Figure 8.4.

Due to the nature of the stream processing, two types of high-speed interfaces were included: a 400 Mbit/s serializer/deserializer and a high speed low voltage differential (LVDS) interface. The interfaces are used for input of a stream of data either from a local input over LVDS interface or from a remote location connected over a coaxial cable to the serializer. As

it will be described in the experimental section of Chapter 9, a stereo-camera module was designed with the LVDS interface. Stereo-camera operates at 200 fps and provides a high speed stream of video frames which are processed by the algorithms running on the FPGA device.

Communication over USB and RS232 is performed using the microcontroller that was interfaced to the FPGA device as well. Microcontroller communicates to well known interfaces and does not need a special core to be designed for FPGA device. USB interface is used for communication with the PC from which SSP bitstream and schedule is uploaded. This interface is also used for download of temporal and final data for verification purposes. VME interface is mostly used for the connection to the expansion board, as well as interconnection with the other MARS platforms over the VME bus.

MARS platform was specifically designed to be rack mountable, as shown in Figure 8.5, to have an aggregating capability for joint parallel processing.

8.4 Summary

One of the novel ideas behind the research is the configuration loader/scheduler which performs all of the SSP core management and re-configuration of the FPGA device. This architecture was specifically developed for the temporal partitioning operation.

After completion of the hardware and software implementation of the MARS platform with support for temporal partitioning, several architectural modifications became apparent. Appendix ?? proposes a number of different approaches to increase the efficiency, as well as to decrease the cost of systems with temporal partitioning support.

This chapter focused on the implementation of all previously proposed and developed methodologies in software, hardware, and firmware. The implementation of CAD tool for task algorithm segmentation and synthesis of SSP set with associated SPC GUI software was described. This chapter also described in detail the hardware architecture for the temporal

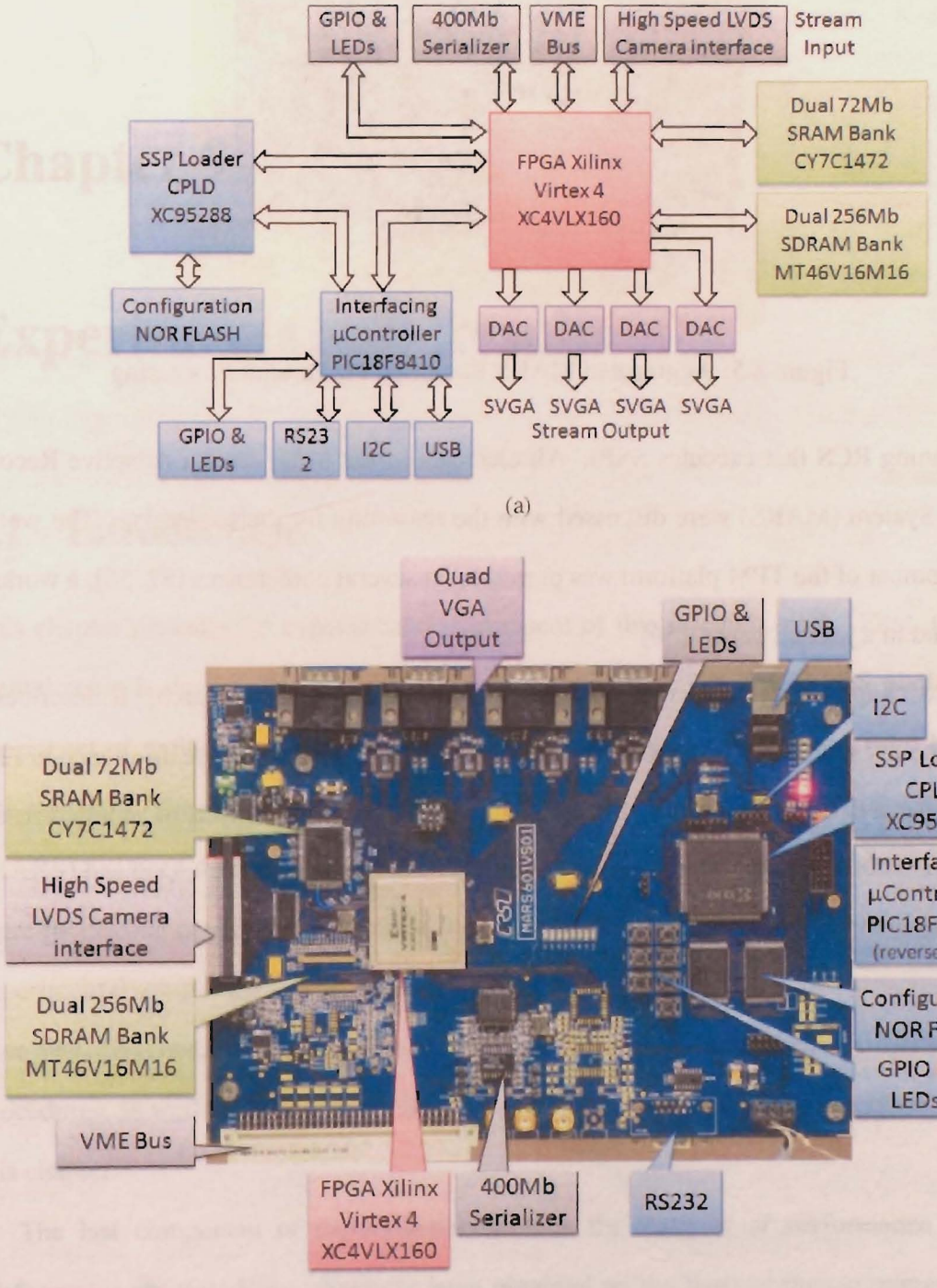


Figure 8.4: Multi-stream Adaptive Reconfigurable System (MARS): (A) Block Diagram (B) Component Placement

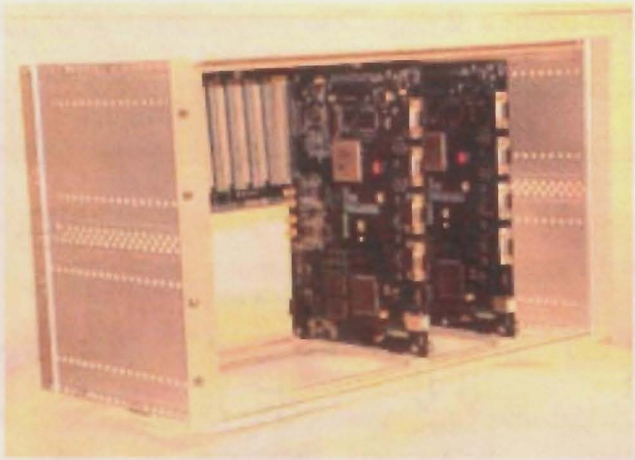


Figure 8.5: Aggregated MARS Platforms for Parallel Processing

partitioning RCS that executes SSPs. All elements of the Multi-stream Adaptive Reconfigurable System (MARS) were discussed with the reasoning for their selection. The work on development of the TPM platform was presented in several conferences [52, 56], a workshop [55], and in a journal paper [54].

Next chapter concentrates on the experimental aspect of the research. It describes the experimental setups of the MARS platform with other platforms that were developed in ERSL. It provides a detailed explanation of conducted experiments and analysis of acquired result.

Chapter 9

Experiments and Results

9.1 Introduction

This chapter presents the experimental component of this research work. First, the experimental setup is described. This experimental setup is based on the MARS platform and a special set of multi-video capturing and pre-processing platforms (stereo-vision high-frame rate cameras). The architecture organization of Multi-Mode Adaptive Reconfigurable system (MARS) has been described up to the component level in Chapter 8. The video capturing on a pre-processing platform organization is presented in this chapter. The next stage after the experimental setup is performance verification. For this stage special video-processing VHCs have been designed, integrated, and tested in operation with MARS platform. The verification procedures, as well as the algorithm implemented in the above VHCs is described in detail in this chapter.

The last component of experimental work is the analysis of performance and cost-performance characteristics, that have been obtained on the basis of the experimental multi-core (multi-VHC) segment-specific processors (SSPs). These SSPs were designed for the above analysis only and are not associated with any specific application. The experiments were done with the goal to analyze the following parameters:

1. Resource utilization.

2. Power consumption.

3. Data-execution timing parameters.

4. Bitstream compilation timing.

5. Cost-performance analysis.

All of the above experiments were conducted for the MARS platform based on the Xilinx Virtex 4 FPGA family. The choice for this FPGA family was based on the fact that it provides the highest bandwidth of configuration bus, an ability for partial configuration, and spans across the devices of the same package.

9.2 Experimental Setups

Experiments for temporal partitioning mechanism (TPM) included several different steps. This section describes the experimental setup and the experiments performed. First and foremost, the experiments had to be performed to test TPM methodology on a hardware platform with temporal partitioning support of FPGA resources. As described in Chapter 8, Section 8.3.2 MARS platform was designed specifically for this purpose. With the support of 3.2 Gbit/sec bitstream upload bandwidth, programmable configuration memory manager was particularly suited for temporal partitioning experiments.

In order to visually illustrate the idea of the experiments few image processing tasks were selected. The selection of video-stream processing applications was also motivated by the R&D project associated with the development of the next generation of space-borne machine-vision platforms. This project is called "FastTrack High-Frame Rate Stereo-Vision Sensor" and was funded by MDA Space Missions and Ontario Centres of Excellence (OCE). It was conducted in cooperation with the research groups from Queen's University (object tracking

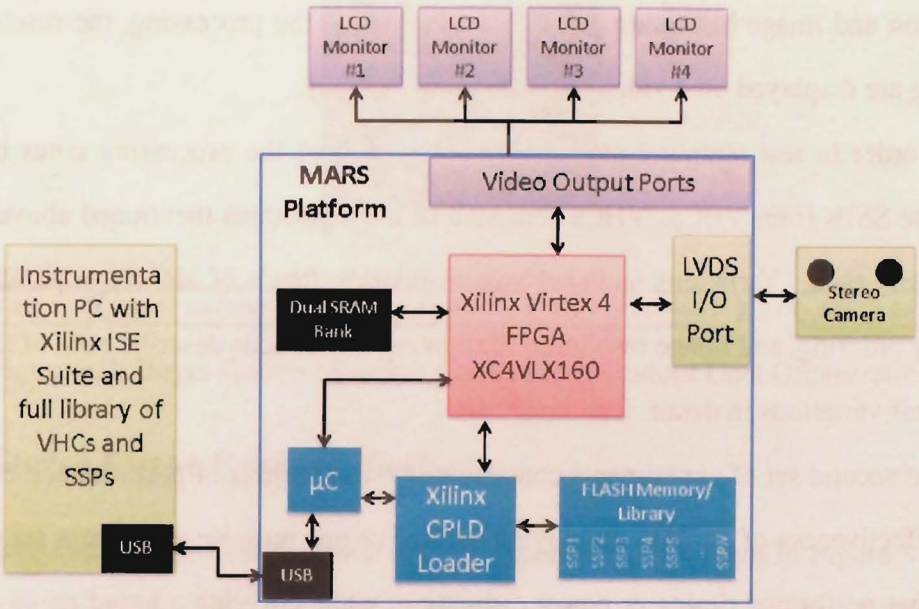


Figure 9.1: Experimental Setup Based on MARS Platform and Stereo-vision Capturing Module

algorithms) and University of Toronto (3D-vision algorithms). The goal of the project was to create an FPGA based multi-mode 3D machine vision platform that would be able to process multiple (3 and more) video streams with relatively high frame rate capability (up to 200 frames per second). Therefore, the stereo camera which was developed as a part of the "FastTrack" project was used for the high speed image capturing and was attached to the LVDS I/O port of the MARS platform. Detailed description of the "FastTrack" platform is given in the next subsection. However, in accordance with Research Collaboration Agreement and associated NDA, it was not possible to utilize the developed segment specific processors (IP-cores of 3D vision and object tracking algorithms) in this thesis without a written permission from all of the above organizations. Therefore, some relatively simple-video processing algorithms have been used (in a form of IP-cores) for test and verification purposes. These algorithms and their implementations are also described in the following subsections. The overall setup of the experimental platform and peripherals is shown in the Figure 9.1.

The overall structure of experiment consists of capturing stereo images, followed by processing. Processing algorithms consisted of Sobel edge detection, colour intensity, image

inversion and image histogram generation. Following the processing, the results of the processing are displayed on 4 video outputs simultaneously.

In order to test temporal partitioning setup each of the processing cores designed into separate SSPs from VHCs. VHCs consisted of the algorithms mentioned above with several variations each. Variations included area avoidance, levels of algorithm parallelism, video display ordering, and image resolution. Following subsections describe each of the algorithms and their variations in detail.

The second set of experiments concentrated on the aspect of performance evaluation and cost-effectiveness of an RCS employing TPM. For this purpose a different set of tools were used that performed timing & power estimation, while covering a broad range of FPGA devices. Based on the obtained results the cost-performance analysis was done and cost-per-logic evaluation was performed for devices to identify the most cost-effective solutions for TPM.

For these experiments a highly parallel VHC stream processing component was designed. It involved a highly parallel input of two 128 bit vectors that underwent several mathematical manipulations (e.g. multiplication, addition/subtraction, bit shift) and an output from the system. All operations were pipelined, and after the initial delay produced a result on every clock cycle. This VHC was used in parallel with other VHCs to form an SSP that was compiled and tested for various devices. For these experiments, besides the MARS platform which ran some of the generated SSPs, the Xilinx XPower Analyzer was used for power calculation on a broad range of different devices and frequencies. Xilinx ISE Timing Analyzer was used for obtaining the result of time analysis for the experiments.

Next subsections present details of the experimental setup and the experiments that were conducted throughout the research work.

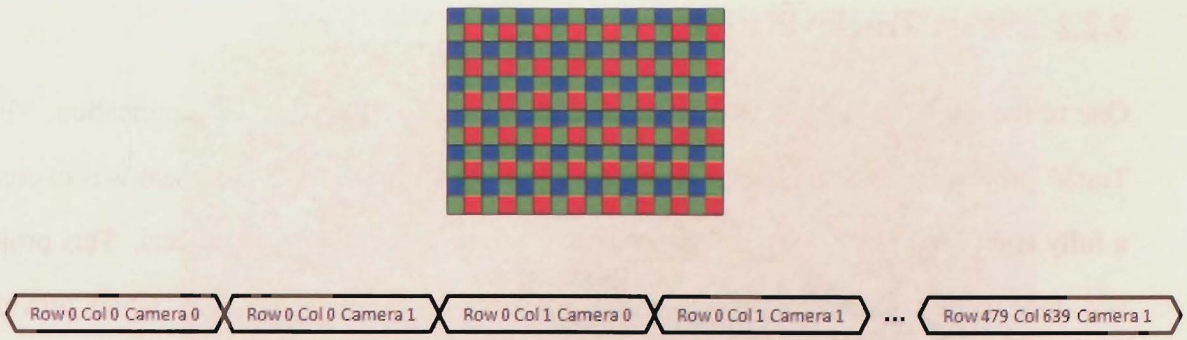


Figure 9.2: Bayer Pattern of Stereo Camera and Readout Data Organization

9.2.1 Stereo Image Capture Platform

Stereo image capture platform that was used in experiments is shown in Figure 9.3. It was developed in ERSL lab and was part of the "FastTrack" project. It is capable of taking stereo images at 5 *ms* intervals (200 fps) by a system request, or by providing constant stream of images along with the synchronization signals (e.g. slave/master modes). Resolution of a single image is 640×480 pixels. Each pixel has 8 bit colour depth arranged in a Bayer pattern format as shown in Figure 9.2. Data provided to the MARS platform from two cameras is arranged in a form of row and column addressing with an output of two bytes per address.

Each byte represents a pixel value from the two image sensors. The data from both sensors is saved simultaneously into one of the SRAM banks. Upon completion of capturing the SRAM bank contains two images which are used by the consecutive processing SSP cores. At the final step of the capture the SSP core is signaling to the configuration manager to reconfigure the FPGA with the next SSP. At this point FPGA is reconfigured with the next SSP core in the schedule with the knowledge of the image location in SRAM. Several different SSPs were created that differ in the image location on SRAM banks, in case of different memory configurations.

9.2.2 "Fast Track" Platform

One of the application of MARS platform was a project related to space application. "Fast Track" project involved tracking objects at speeds of 200 fps. The requirement was to create a fully embedded platform, which would perform run-time tracking of objects. This project was done based on the requirements presented by the MDA Space Missions, which provided funding for the project along with the Ontario Centres of Excellence.

Throughout the development of the project three versions of the "FastTrack" platform were decided and they are described briefly in this section. In order to visually track objects it was devised to use stereo image approach. Stereo images are used to extract disparity information and, then, to provide the depth information about the surrounding environment. This information is passed to the object extraction module which identifies the object based on the original model. This information about the object identity is sent to the object tracking algorithm which performs prediction and tracking of the object. For this purpose, all of the "FastTrack" platforms include a pair of image sensors.

First platform was developed as a prototype and utilized only $30fps$ image sensors. This in turn allowed to use only one LVDS interface. In order to transfer both image streams of data over 8 bit data bus, these streams were multiplexed. Multiplexing was done at twice the speed of operation with the overall bandwidth of:

$$8bits/pixel \times 640pixels \times 480lines \times 30fps \times 2images = 147.456Mbits/sec$$

Transfer data is in the raw format and represents an image in Bayer pattern. Similar format is also present in both consequent versions. The second and third version of the "FastTrack" platforms include image sensors with $200fps$ performance, which translates into a much higher bandwidth of:

$$8bits/pixel \times 640pixels \times 480lines \times 200fps \times 2images = 983.04Mbits/sec$$

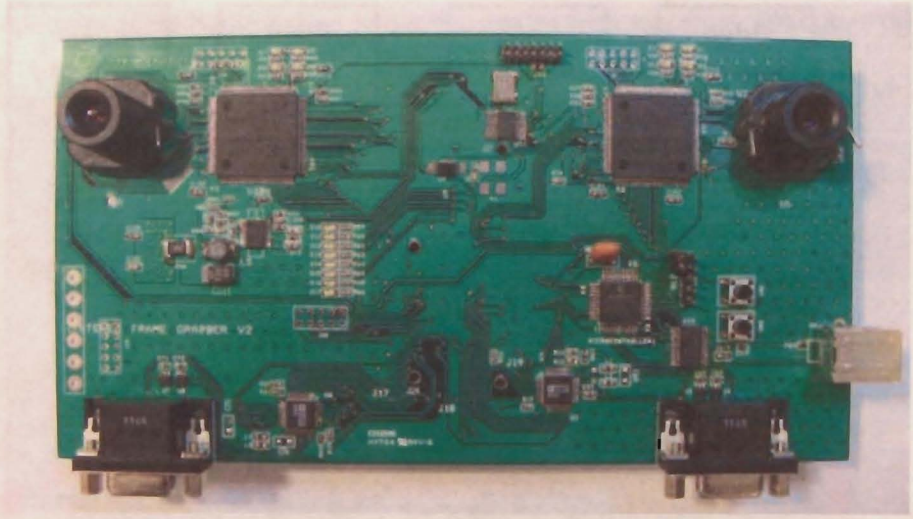


Figure 9.3: "FastTrack" Stereo-Vision Platform.

This required to expand the communication bus to 16 data bits, and to a second LVDS interface to the MARS platform. For this purpose an expansion card was built and attached to the VME expansion bus. The final version of the platform, which is shown in Figure 9.3,was capable of capturing stereo images at $\sim 200 fps$ and synchronously upload them to the MARS platform [19]. The advantage of using MARS platform is in its capability of run-time mode adaptation.

Since MARS supports both spatial and temporal partitioning and reconfiguration, it provides a unique opportunity to change modes of operation as soon as system would detect requirement for such change. Considering the Space applications, image processing algorithm requirements can change rapidly based on the light exposure. So the system can be implemented to use the temporal partitioning. Operation of temporal partitioning performs capture of several frames at $200 fps$, and then reconfigures to perform processing operation. Processing could be done in one or several cores. The number of processing cores mostly depends on the complexity of processing at a particular instance of time. Scheduling and processing cores can vary during the system operation. Based on these capabilities the MARS platform presented a unique opportunity for development of systems for Space application with various capabilities, as shown by "FastTrack" project. One of the important factors why MARS

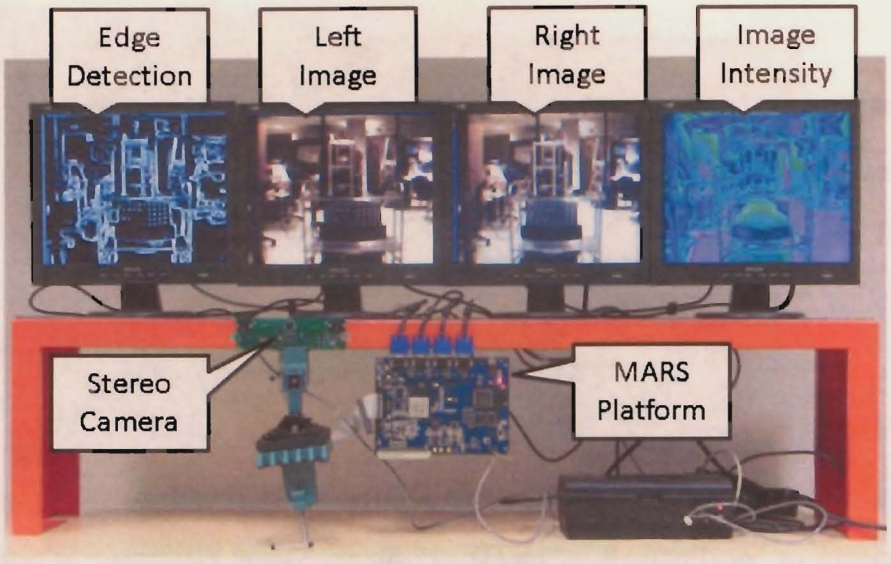


Figure 9.4: Photo of the Experimental Setup with MARS Platform, "FastTrack" Stereo-Camera, and 4 LCD Displays

platform architecture is especially beneficial for "Fast Track" project is the fault tolerance. Because the target platform is oriented for Space application the effects of cosmic radiation must be considered. As it was mentioned in Chapter 4, fault tolerance can be achieved by loading a test SSP into an FPGA to verify its integrity. For the experiments in this work the images received from the "FastTrack" stereo camera were passed down to several image/video processing algorithms. The complete experimental setup is shown in Figure 9.4. In the next three subsections these algorithms are described in more detail.

9.2.3 Results and Verification of Workload

As was mentioned in Section 9.2 a special set of SSPs had to be designed for verification of the above experimental setup. This workload should test the complete system including:

- High-frame rate capturing component.
- Multi-channel parallel video-output part.
- Video-processing component.

These components were deployed on MARS platform with run-time TPM. For this purposes the following algorithms have been selected due to their suitability for testing and verification of all above mentioned components of the experimental setup:

1. Sobel Edge detection algorithm
2. Image histogram calculation procedure
3. Image colour intensity calculation procedure

The above algorithms and procedures are associated with real-time stream processing and, therefore, easily can demonstrate correct performance of all components of the above experimental setup.

In the following subsections the descriptions of these algorithms are given, as well as the results of their implementation on the MARS platform.

9.2.3.1 Sobel Edge Detection Core

One of the image processing cores that was designed for the test experiment was the Sobel edge detection algorithm. It operates on the images that were saved by the stereo image capture SSP core. Based on the algorithm described in [31], a 3×3 matrix of data is taken from an image and each of the $\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$ matrices is multiplied by $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$, and $\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$, which results in the following equation:

$$\begin{aligned} S_1 &= a_{00} + 2 \times a_{10} + a_{20} - a_{02} - 2 \times a_{12} - a_{22} \\ S_2 &= a_{00} + 2 \times a_{01} + a_{02} - a_{20} - 2 \times a_{21} - a_{22} \\ R &= |S_1 + S_2| \end{aligned} \tag{9.1}$$

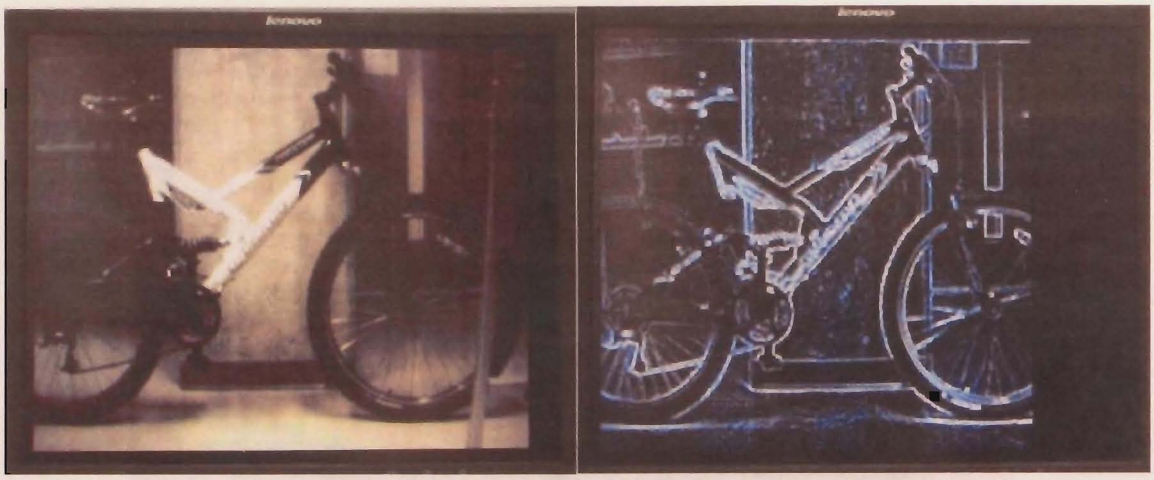


Figure 9.5: Photo of the Original Captured Image and Image after Processing on Sobel Edge Detection SSP Core

In the third line of equation 9.1 the partial sums S_1 and S_2 are added together and the absolute values of the answer is saved into a new image. This operation is done over the whole image. Because it operates with stereo images, the processing algorithm is duplicated and, therefore, allows processing in parallel. This SSP was implemented in several variations by calculating Sobel edge detection algorithm on images in parallel, and in series. The first variation obviously provides lower latency, while the second saves area, and could be generated for a smaller size logic device. Upon completion of the processing, a request is sent to the loader/scheduler for the next SSP. Figure 9.5 shows the result originally captured image and the image processed with the Sobel algorithm.

9.2.3.2 Image Histogram Calculation

One of the algorithms that was designed for image processing was a run-time histogram calculation. Histogram calculation is used in many digital cameras nowadays and performs a function of displaying a graph of the light intensity distribution. This algorithm requires three steps, which have to be done in sequence, due to the algorithm's sequential nature. It is needed to go through the whole image and record the intensity of every pixel to the intensity array counter. Since pixel intensities are fixed to 8bit resolution there are 256 intensity levels. The

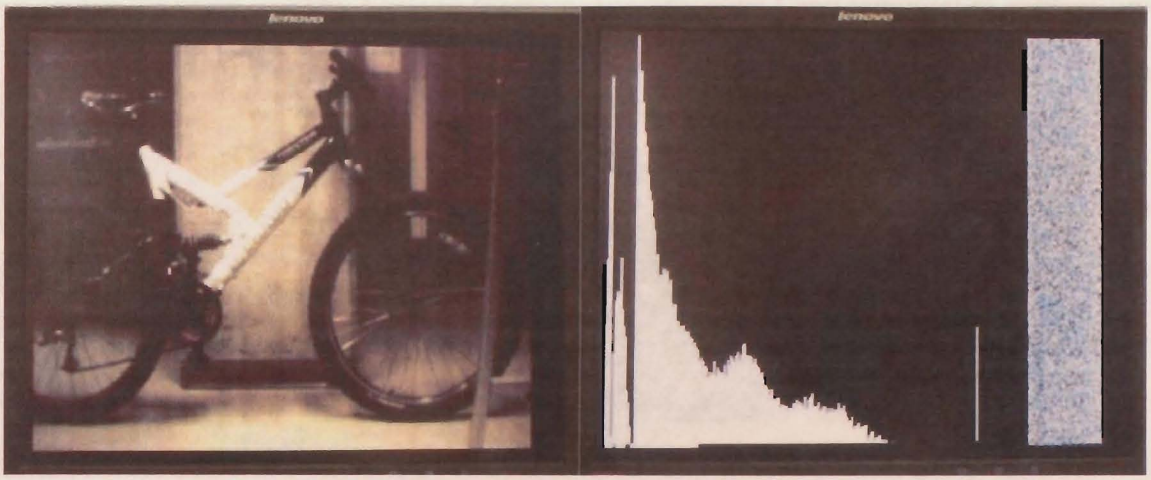


Figure 9.6: Photo of the histogram image processing SSP core

next step of operation is to scale the histogram to adapt the results to the screen. Since the resolution of the screen is 640×480, the vertical histogram value had to be adjusted to 480 pixels. Therefore, the overall formula of transformation of each intensity level is:

$$\frac{Current\ intensity}{MAX\ intensity\ counter} \times 480 \quad = \quad Modified\ intensity\ level \tag{9.2}$$

At the completion of calculation there are 256 cells, each representing a level of the intensity. These levels have to be represented in the form of a graph which is done by the next step. Since there are only 256 levels and the maximum horizontal resolution is 640 pixels, each bar representing a single level is stretched to 2 pixels in width. The final image is formed as a series of vertical white bars and stored to the SRAM as an image. This is done so that the video display core would be able to read it as a video frame without any additional processing. Example of this image processing can be segmented even more and these three steps can be saved in three separate SSP cores. Figure 9.6 shows the result of histogram image processing of the original image from the previous figure.

As in previous SSP cores, the following step is to request reconfiguration with the next SSP core from the configuration loader/scheduler.

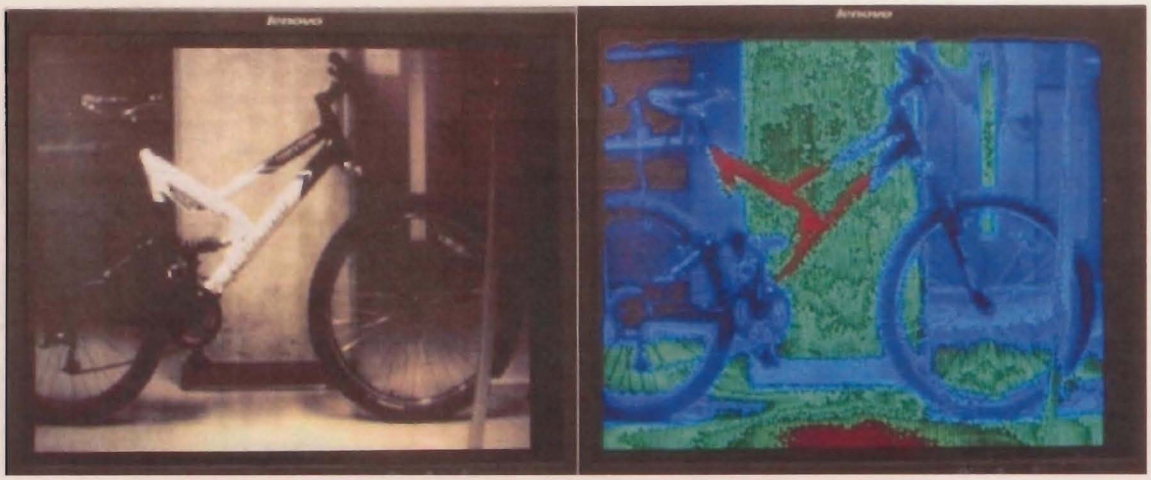


Figure 9.7: Photo of the Original Captured Image and Image after Processing on Image Intensity SSP Core

9.2.3.3 Image Colour Intensity

The purpose of image colour intensity algorithm is to display the intensity of an image by different colour representation. Algorithm scans an image and based on the intensity of each pixel, assigns an appropriate colour. The image data is read from the SRAM bank that was assigned as a source location. After pixel is read it is compared to the look-up table and a new value from the look-up table is saved into the corresponding image located at a different SRAM bank location. Figure 9.7 shows the originally captured image and the image processed with the image intensity algorithm.

9.2.4 Results of Experimental Setup Verification

The purpose of the experiments was to verify the methodology of temporal partitioning for a stream processing application and perform the power and timing analysis. Temporally partitioned SSPs described in the previous sections were executed on the MARS platform. All of the SSPs performed on the images captured in run-time by the "FastTrack" stereo-camera.

This image data was used by SSPs to perform data processing and temporal storage of processed data. Final SSP core has displays the processed images on 4 video outputs simultaneously. Whole operation takes about $\sim 110ms$ and upon completion is ready for next cycle.

Therefore, the verification stage has been successfully completed including verification of the proper functionality of TPM deployed on the MARS platform. The obtained performance parameters were registered and included in the Appendix B. Since these SSP cores were oriented only for functional verification, they did not occupy a substantial area of the FPGA device to provide performance measurements. Therefore, specific workload components (SSPs) based on highly paralleled stream processing elements have been developed and implemented to obtain the quantitative parametric characteristics of the multi-stream processing platform (MARS) with TPM.

9.3 Experimental Quantitative Performance Characteristics

The results obtained from experimentation on highly paralleled VHC stream processing core are organized in a form of tables and graphs in several sections, and are used in the analysis section. These sections are: Logic Utilization, Power consumption, Timing results, and Bitstream compilation timing. These results are from experiments conducted on large SSPs that were assembled from the VHCs described below.

9.3.1 Experimental Workload: Highly Paralleled Stream Processors

As was mentioned in introduction to this chapter, stream processing VHC was designed to perform parallel computation on two 128 bit data vectors. Since the design is highly parallel, it requires a significant amount of logic and routing resources. Due to that, it is ideal for testing the resource utilization, power consumption, and timing on various FPGA devices of

Virtex 4 LX family. The reason why LX family was selected for the MARS platform was that FPGAs of LX family have mostly homogeneous micro-architecture (consists of configurable logic and do not contain any of the embedded PowerPC hard cores and many DSP slices).

The test involved joining several of the VHC cores in parallel into two and four of these VHC to form the associated SSPs. These SSPs were generated into bitstreams for XC4VLX40, XC4VLX60, XC4VLX80, XC4VLX100, XC4VLX160 devices with a package size of FF1148. Since results can vary for the same FPGA size with a different number of available I/Os, the analysis was done on the same package of the FPGA devices to keep fair evaluation. The experiments were not conducted on XC4VLX25 and lower devices for the reason that the VHC (even as a single unit) in SSP could not be generated for this device due to the lack of sufficient logic.

In addition, power analysis was conducted for all of the generated cores by the Xilinx XPower Analyzer. For every single SSP core the performance analysis was done based on three operating frequencies: 50MHz, 100MHz, and 200MHz. Power reading was recorded as a quiescent power, dynamic power, and total power. For the analysis of cost-performance the timing results were obtained from Xilinx ISE Timing Analyzer. Results consisted of worst case data-path delays, as well as, worst clock to destination latencies.

To compare utilization of logic resources, they were recorded and presented in analysis section. Captures of post-routed diagrams were obtained using Xilinx FPGA Editor image and attached in the Appendix B of the thesis. The place and route times were also recorded and their quantitative analysis confirmed the currently growing problem of increasing of bitstream compilation times as the size of FPGA increases. These findings further support the use of TPM approach. It should be noted that because all of the SSP core compilations were performed on the same PC, they can be compared together. PC specifications were: Intel Core 2 Duo E6600 2.4 GHz processor, 4GB RAM. All compilations were done using Xilinx ISE

Table 9.1: Logic Use in 4-LUTs per Each Device for Single, Dual, and Quad VHC SSPs

	XC4VLX40	XC4VLX60	XC4VLX80	XC4VLX100	XC4VLX160	Average	Logic Per VHC
Single VHC SSP	1414	1414	1414	1414	1414	1414	1414
Dual VHC SSP	19238	20121	20099	20240	19996	19939	9969
Quad VHC SSP	Can't Fit	Can't Fit	57375	57534	57497	57469	14367

Table 9.2: Logic Use in 4-LUTs per Each Device for Single, Dual, and Quad VHC SSPs

	XC4VLX40	XC4VLX60	XC4VLX80	XC4VLX100	XC4VLX160	Average	Signals Per VHC
Single VHC SSP	6326	6326	6330	6330	6330	6328	6328
Dual VHC SSP	34647	34875	34867	35026	34707	34824	17412
Quad VHC SSP	Can't Fit	Can't Fit	91535	91777	91693	91668	22917

Design Suite 10.1 with all the latest service packs applied. Next sections present the results of the experiments and then presents the discussion of these results.

9.3.2 Logic Utilization

In first series of experiments with SSPs that incorporated Single, Dual and Quad VHCs are described in previous section. The amount of logic and signals (routing resources) utilization, have been recorded post place & route. Results are shown in Tables 9.1, 9.2.

In these tables the number of 4-LUTs and signals used were almost the same across all devices for the same type of SSP. However, when the average is divided by the number of VHCs inside an SSP, the resulting number is not the same and it is increasing. It should be noted that for a single VHC configuration the resource and signal use is disproportionally smaller in comparison to Dual and Quad implementations. This fact is due to the use of embedded DSP slices first, prior to use of logic resources, which do not require much of routing or any of extra logic. In Quad VHC configuration XC4VLX40 and XC4VLX60, the devices did not have sufficient amount of logic and signal resources to be able to fit such an

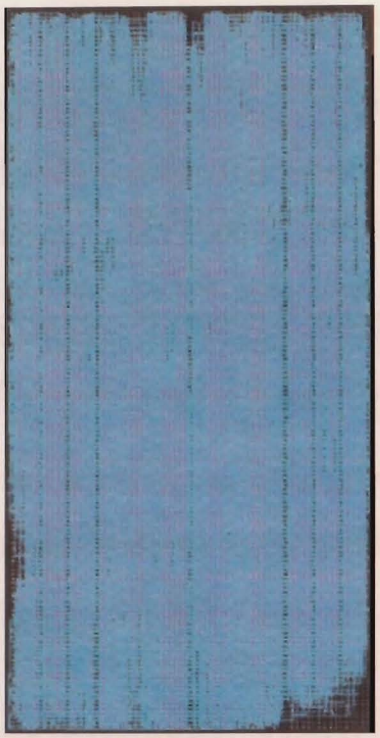


Figure 9.8: Floor Plan for Post Place and Rout of XC4VLX80 with Quad VHC Core

SSP, therefore results were not obtained. As can be seen from FPGA floor plan in Figure 9.8, Quad VHC SSP core occupied almost the entire XC4VLX80 device.

Floor plans of all the other combinations of Single, Dual, Quad VHC SSP cores are included in the Appendix B for reference.

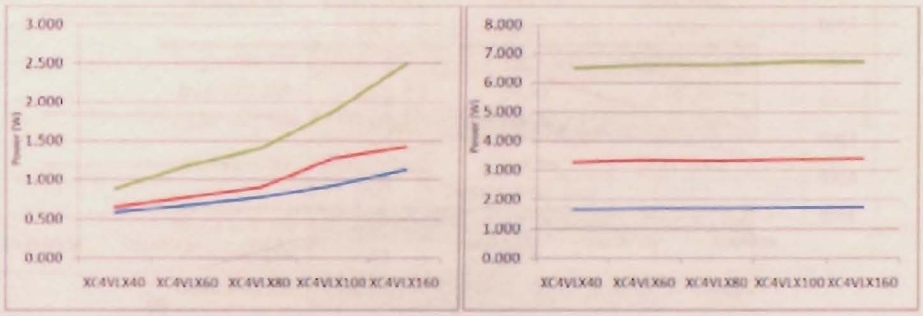
9.3.3 Power Consumption

The next series of experiments were focused on the power consumption data for the same set of SSPs. The power consumption was divided onto quiescent (static) and dynamic power. Power consumption was computed for three different frequencies: i) 50MHz, ii) 100MHz, and iii) 200MHz.

The Tables 9.3, 9.4, and 9.5 present the results obtained after running the power analysis tool Xilinx XPower. Results consisted of quiescent power, dynamic power, and total power. As in the previous section, the experiments were conducted on the SSP containing Single, Double, and Quad VHC SSP cores.

Table 9.3: Power Consumption (Watt) of Single VHC SSP Core Operated at 50MHz, 100MHz, and 200MHz

Device	Quiescent Power			Dynamic Power			Total Power		
	50MHz	100MHz	200MHz	50MHz	100MHz	200MHz	50MHz	100MHz	200MHz
XC4VLX40	0.584	0.655	0.880	1.650	3.270	6.500	2.234	3.925	7.380
XC4VLX60	0.670	0.775	1.180	1.690	3.340	6.610	2.360	4.115	7.790
XC4VLX80	0.770	0.910	1.400	1.690	3.338	6.610	2.460	4.248	8.010
XC4VLX100	0.922	1.278	1.875	1.727	3.390	6.700	2.649	4.668	8.575
XC4VLX160	1.123	1.419	2.490	1.748	3.409	6.712	2.871	4.828	9.202

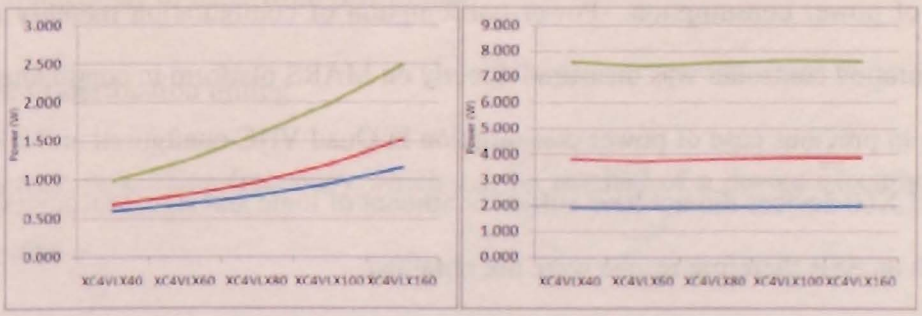


(A) (B)

Figure 9.9: Quiescent (A) and Dynamic (B) Power Consumption (Watt) for a Single VHC SSP Core Operated at 50MHz (Blue), 100MHz (Red), and 200MHz (Green)

Table 9.4: Power Consumption of Dual VHC SSP Core Operated at 50MHz, 100MHz, and 200MHz

Device	Quiescent Power			Dynamic Power			Total Power		
	50MHz	100MHz	200MHz	50MHz	100MHz	200MHz	50MHz	100MHz	200MHz
XC4VLX40	0.594	0.683	0.990	1.922	3.807	7.560	2.516	4.490	8.550
XC4VLX60	0.680	0.805	1.242	1.890	3.730	7.410	2.570	4.535	8.652
XC4VLX80	0.791	0.966	1.587	1.931	3.808	7.540	2.722	4.774	9.127
XC4VLX100	0.947	1.199	1.985	1.965	3.859	7.627	2.912	5.058	9.612
XC4VLX160	1.158	1.527	2.490	1.982	3.867	7.610	3.140	5.394	10.100

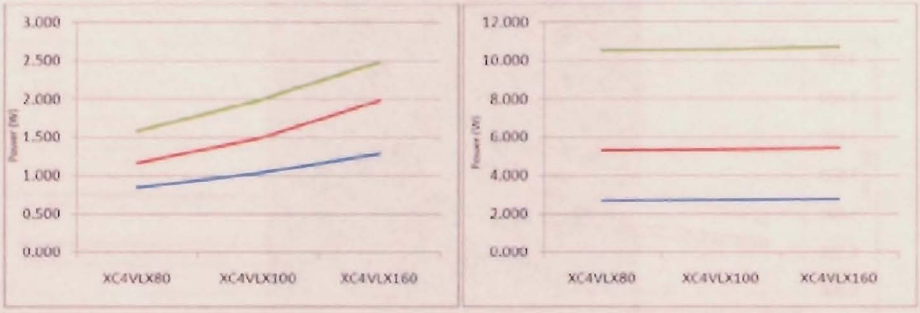


(A) (B)

Figure 9.10: Quiescent (A) and Dynamic (B) Power Consumption for a Dual VHC SSP Core Operated at 50MHz (Blue), 100MHz (Red), and 200MHz (Green)

Table 9.5: Power Consumption of Quad VHC SSP Core Operated at 50MHz, 100MHz, and 200MHz

Device	Quiescent Power			Dynamic Power			Total Power		
	50MHz	100MHz	200MHz	50MHz	100MHz	200MHz	50MHz	100MHz	200MHz
XC4VLX40	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit
XC4VLX60	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit	Can't Fit
XC4VLX80	0.854	1.163	1.587	2.687	5.312	10.530	3.541	6.475	12.117
XC4VLX100	1.034	1.489	1.985	2.717	5.354	10.600	3.751	6.843	12.585
XC4VLX160	1.289	1.992	2.490	2.770	5.433	10.730	4.059	7.425	13.220



(A) (B)

Figure 9.11: Quiescent (A) and Dynamic (B) Power Consumption for a Quad VHC SSP Core Operated at 50MHz (Blue), 100MHz (Red), and 200MHz (Green)

From the obtained results the figures depicting the power use per device were generated. Dynamic power consumption across all of the SSP cores on the same frequency have been determined. However, in the case of quiescent power, a steady increase for the same operating cores was observed. As described in Chapter 8, the power consumption of the configuration memory manager was not included. This is due to the fact that its contribution for all devices is very small (maximum of 66mW in continuous reconfiguration) and does not impact the overall result of power consumption. Power consumption of configuration memory manager and configuration controller was measured directly on MARS platform in continuous operation.

As in previous case of power consumption in Quad VHC configuration, XC4VLX40 and XC4VLX60 devices did not have sufficient amount of logic and signal resources to be able to fit such an SSP, therefore results were not obtained.

Table 9.6: Timing Operation Results for FPGAs Running SSP with Single, Dual, and Quad VHCs

XC4VLX40	Max Delay (ns)	Clock to Destination (ns)	Max Freq (MHz)
Single VHC SSP	6.81	25.94	146.84
Dual VHC SSP	6.96	28.572	143.68
XC4VLX60	Max Delay (ns)	Clock to Destination (ns)	Max Freq (MHz)
Single VHC SSP	7.03	28.2	142.25
Dual VHC SSP	7.19	29.253	139.08
XC4VLX80	Max Delay (ns)	Clock to Destination (ns)	Max Freq (MHz)
Single VHC SSP	7.46	27.15	134.05
Dual VHC SSP	7.92	29.71	126.26
Quad VHC SSP	8.08	34.55	123.76
XC4VLX100	Max Delay (ns)	Clock to Destination (ns)	Max Freq (MHz)
Single VHC SSP	7.35	28.147	136.05
Dual VHC SSP	7.42	29.4	134.77
Quad VHC SSP	7.93	33.91	126.10
XC4VLX160	Max Delay (ns)	Clock to Destination (ns)	Max Freq (MHz)
Single VHC SSP	7.54	25.95	132.63
Dual VHC SSP	8.16	30.92	122.55
Quad VHC SSP	8.66	34.3	115.47

9.3.4 Timing Results

The goal of next series of measurements was to determine the variations of timing parameters when running the same set of SSPs on all FPGA devices from Xilinx Virtex-4 LX family. The following timing parameters were estimated:

1. Maximum delay of data (from pad-to-pad).
2. Clock-to-destination timing.
3. Maximum processing frequency which can be reached of a device executing a given SSP-core.

Table 9.6 shows the timing parameters for SSP cores including Single, Dual, and Quad VHC cores.

Table 9.7: SSP Cores Compilation Times for Single, Dual, and Quad VHC SSP Cores

XC4VLX160	Compilation time (min)
Single VHC SSP	8.5
Dual VHC SSP	24.5
Quad VHC SSP	58.4

From the results it can be seen that as the logic resources increase, the delay also increases. This, in turn, decreases the maximum operating frequency for the associated SSP. At the same time, the increase in clock-to-destination delay is even more drastic.

9.3.5 Bitstream Compilation Timing

In Table 9.7 the timing of SSP core compilations is shown. These timing results were obtained from compilation of all of the above SSP cores on the PC with configuration mentioned in Section 9.3.

9.4 Analysis of Results

The overall assessment of RCS performance with TPM is discussed in this section. An analysis is done to determine the devices and size of SSP cores, that are most effective. Cost-performance analysis demonstrate which devices are most cost-efficient for the cost sensitive applications utilizing TPM. Power consumption analysis addresses the suitability of particular devices used in TPM for different types of applications. The subsection on resource utilization talks about the real overall utilization of resources by large designs, and problems that arise with FPGAs growing ever more in logic size. Finally, the compilation process of SSPs is discussed with the analysis of how the size of a monolith design greatly impacts the design time.

9.4.1 Performance Analysis of RCS with TPM

For the performance analysis of RCS, the results from Table 9.6 were used. First observation that can be made from Table 9.6 is a decrease of processing speed of in a case of increase in size of an FPGA device. As was shown in the results section, the larger FPGA devices used on the same SSP core, the longer is the maximum signal-to-signal delay, which reduces the frequency of operation from 143.68MHz to 122.55MHz on a dual VHC SSP core (over 17% decrease). This effect is intensified further for all of the devices when design becomes more complex. This is most apparent for the large devices (e.g. XC4VLX160) where from Single to Quad VHC SSP core frequency of operation drops from 132.63MHz to 115.47MHz. The same effect is observed in clock-to-destination node timing, where latency increases with the use of a larger device and increases even more with a larger design on a large device, as was shown increasing from 25.95ns to 34.3ns (over %32 increase). This is expected, since with a larger design it takes more routing lines to get to all of the resources.

Hence, due to these reasons, utilizing a smaller FPGA device with the use of TPM approach allows acceleration of the associated SSP execution. To compare the performances of small and large FPGA, one can calculate processing bandwidth of two data packets 1GBit each with the dual VHC SSP core running on a XC4VLX40 and quad VHC SSP core running on XC4VLX160, respectively:.

$$T_{frame_{Dual-XC4VLX40}} = \frac{2^{30}}{2 \times 128 \times 143.68 \times 10^6} = 29.1\,ms$$

$$T_{frame_{Quad-XC4VLX160}} = \frac{2^{30}}{4 \times 128 \times 115.47 \times 10^6} = 18.2\,ms$$

$$Speedup = \frac{29.1}{18.2} = 1.59\,times$$

Even though number of VHCs in SSP core was doubled, the real performance increased only 1.59 times. At the same time, the cost of the target FPGA increased from $570 for XC4VLX40 to $5625 for XC4VLX160 which is close to 10 times cost increase. One can see that simply doubling the number of smaller FPGAs can increase the cost-effectiveness by more than 6 times in comparison to using a larger FPGA:

$$Cost\,Performance\,increase = \frac{T_{frame_{Quad-XC4VLX160}} \times C_{XC4VLX160}}{\frac{T_{frame_{Dual-XC4VLX40}}}{2} \times 2 \times C_{XC4VLX40}} = \frac{18.2 \times 5625}{\frac{29.1}{2} \times 2 \times 570} = 6.17 \tag{9.3}$$

By the same token dividing a large design into several smaller designs and processing them with the notion of TPM, greatly benefits the cost-performance of a system. An obvious argument for this solution is that a design can be constrained to a particular area of a large device in order to achieve similar performance results as in smaller devices. However, this is not possible in most cases due to physical I/O restrictions of the device. In general, the performance on the critical path delay can be optimized. However, when a large design is fitted into a large FPGA consuming 85-90% of available resources complexity of routing doesn't often allow reaching the same level of performance as for smaller designs in smaller FPGAs. Reduction of resource utilization in a large FPGA drops the cost-efficiency for such designs, as was discussed in previous chapters.

All aspects of the design should be evaluated, not only the IP-core synthesis, and the following sections explore that approach.

9.4.2 Cost Performance Analysis of Different FPGA Devices

In Chapter 5 the cost-performance ratio was discussed and evaluation of the cost of 1K 4-LUTs per each device was done, as shown in Table 5.1. When it comes down to the actual

Table 9.8: Cost in $USD per VHC for Single, Dual, and Quad VHC SSP Core Configurations Across a Range of FPGA Devices

	XC4VLX40	XC4VLX60	XC4VLX80	XC4VLX100	XC4VLX160
Single VHC SSP	19.80	25.45	24.04	36.76	49.49
Dual VHC SSP	134.67	181.09	170.84	263.12	349.93
Quad VHC SSP	No Data	No Data	243.84	373.97	503.10

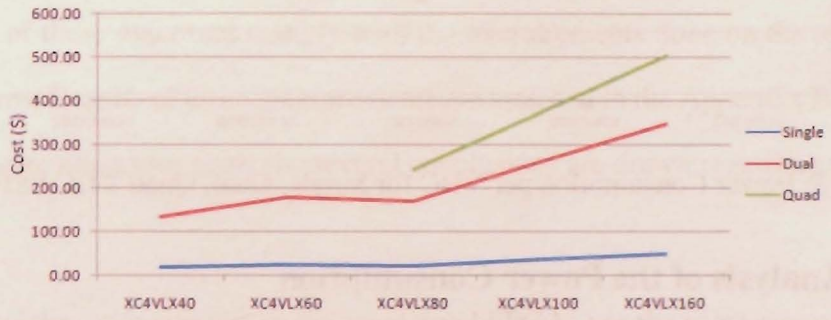


Figure 9.12: Cost-Effectiveness per VHC of Different FPGA Devices with Single, Dual, and Quad VHC SSP Cores

evaluation of the cost of the VHC in different configurations on a whole range of devices it evaluates in $/VHC core, as shown in Table 9.8.

For better visual representation, the costs are plotted on a graph, shown on Figure 9.12.

The data in the table and graph show that the most optimal and cost-effective device, on which Dual and Quad VHC SSPs can be generated, is XC4VLX80 since it the minimal cost per VHC. This cost is lower than that of XC4VLX60 and occurs on the graph at a point just before the cost per 1K 4-LUT goes up significantly. As was mentioned in other sections, it should be noted that the results for a single VHC SSP should not be taken in account, since in the beginning of place and route DSP slices of device are used, and logic resources are not utilized significantly. Due to that reason, one Dual and Quad VHC SSP core configurations have been considered.

The analysis of cost-effectiveness of obtained results now proves quantitatively what was proposed in Chapter 5. Results verify that utilization of a smaller device may be more cost-effective if it is considered in $ per 1K 4-LUTs or in $ per VHC.

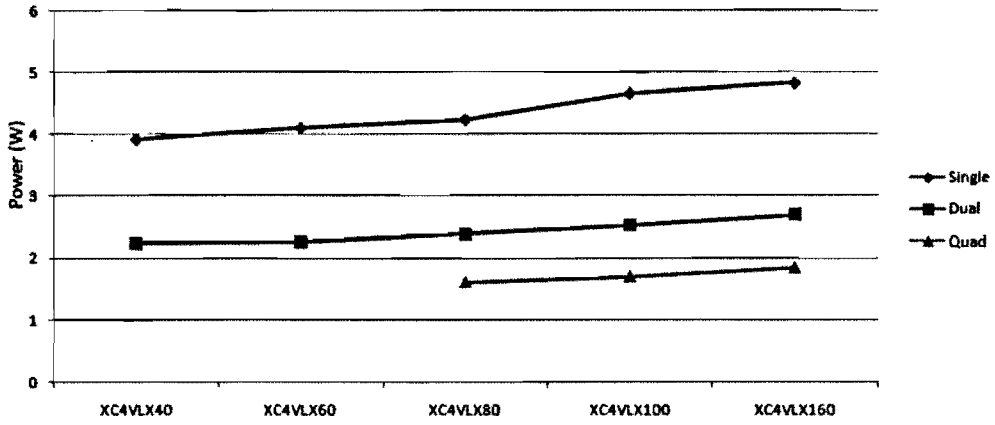


Figure 9.13: Power Consumption per VHC for Single, Dual, Quad VHC SSP Configurations

9.4.3 Analysis of the Power Consumption

Power consumption results have to be considered as one of the very important factors in the evaluation of system architecture. For the analysis of power performance Tables 9.3, 9.4, 9.5 and Graphs 9.9, 9.10, 9.11 are used and will be referred to in this section. The results of the Figures 9.9(B), 9.10(B), 9.11(B) show that dynamic power consumption is the same across all of the devices with the same number of VHCs per SSP. This fact holds at different frequencies of operation. If two functions utilize similar amount of resources the power consumption would be similar. However, the quiescent power (as seen in Figures 9.9(A), 9.10(A), 9.11(A)) increases linearly with the size of device. This can be explained with the fact that a larger device contains more logic and, thus, more static power drains in comparison to a smaller FPGA device. It should also be noted, that the increase of quiescent power is relatively insignificant and is measured around 10% for most of the scenarios going from single to quad VHC SSP core. These results also reveal that overall power use per VHC drops as more VHCs are packed into an SSP. This is an expected result, since initial power drain of the device is spread over the VHCs that occupy FPGA device. This aspect of power consumption is shown in Figure 9.13.

The Figure 9.13 also shows that power consumption per VHC for Single VHC per SSP is different from Dual VHC per SSP configuration. This effect is greatly diminished when Quad VHC SSP. This leads into the conclusion that there is a balance between power per VHC and size of the device. Because a mid-range FPGA device, such as XC4VLX80 is tightly packed as shown in Figure 9.8, it can utilize most resources and, therefore, minimize the power use per VHC. All of these outcomes comply with the measurements done on the temporal video processing cores. Results of these measurements are attached in the Appendix B of this thesis.

To summarize the power analysis, several conclusions are drawn regarding the device selection:

1. Design of the same size ran on different sized FPGAs use the same amount of dynamic power, however, devices of smaller size use much less quiescent power.

2. Smaller devices have much smaller increase of quiescent power with the increase of frequency of operation (e.g. 0.59W to 0.99 W for XC4VLX40) in comparison with large devices (e.g. 1.15W to 2.49W for XC4VLX160). This can be attributed to the fact that on larger device more logic is leaking static power. This further enforces the advantage of smaller FPGA devices with TPM architecture.

Hence, for power critical solutions smaller devices may be a better choice. On the other hand, power effectiveness of smaller or medium devices strongly depends on the number of VHCs per SSP loaded to FPGA. In other words, if SSP occupies most of FPGA resources and contains as much functionality as possible, the power effectiveness will reach its maximum. This is another motivation factor for development of the CAD tool which allows optimization of SSP architecture. This CAD tool (described in Chapter 7) allows to select and pack the SSP to utilize FPGA device as much as possible while balancing the power consumption by distributing different types of VHC across different segments.

9.4.4 Analysis of Resource Utilization

One of the important factors to be considered in evaluation of effectiveness is the resource utilization of the system. For this analysis the data acquired from compilation of single, double, and quad VHC SSP, presented in Tables 9.1, and 9.2 was used. As it was demonstrated in the Section 9.4.1, the larger the design the lower is the performance. The same is true for the resource utilization. By examining complex designs of Dual VHC SSP and Quad VHC SSP an interesting fact is revealed: resource usage per VHC core is not the same. As shown in the tables both required 4-LUTs and number of signals per VHC increases as the design becomes more complex. Furthermore, if evaluation is conducted on a cost-per-VHC basis, the results become even more advantageous for the TPM approach and a smaller FPGA. By using 1K 4-LUT costs from the Table 5.1 following comparison of VHC costs can be made:

$$C_{VHC_{Dual}-XC4VLX40} = 9.969KLUT \times \$18/KLUT = \$179.42\,perVHC$$

$$C_{VHC_{Dual}-XC4VLX160} = 9.969KLUT \times \$35/KLUT = \$348.91\,perVHC$$

$$C_{VHC_{Quad}-XC4VLX160} = 14.367KLUT \times 35KLUT = \$502.85\,perVHC$$

In the case of dual VHC SSP, implemented on XC4VLX40, the cost is $179.42 per VHC core, and when the design is placed on the XC4VLX160, the overall cost almost doubles due to higher per 4-LUT cost of the larger FPGA. On the other hand, when the quad VHC SSP is used, the cost per VHC goes up to $502.85 per VHC. Overall, these results show that for cost sensitive applications can be produced a lower total cost by exploiting smaller or mid-range devices from the same FPGA family.

9.4.5 Analysis of Compilation Process

Last but not least compilation time of the designs should be looked at, since it is becoming a more and more pressing issue for most of the companies in the industry. Compilation times in Table 9.7 show the alarming tendency, of compilation time increasing almost proportionally with the amount of resources used. By taking the amount of 4-LUTs utilized for the Dual or Quad VHC SSP core and the time it takes to perform place and route can be computed:

$$Time/LUT_{DualVHCSSP} = \frac{24.5min}{19.2KLUT} = 1.25\frac{min}{KLUT}$$

$$Time/LUT_{QuadVHCSSP} = \frac{58.4min}{57.37KLUT} = 1.02\frac{min}{KLUT}$$

With a large design occupying whole 160K 4-LUTs of XC4VLX160 it will require $160min = 2h40min$ of compilation time for a single core of a completely unconstrained design. If constraints are present, this time can increase 5-10 times depending on how many cycles of re-routing have to be performed. With the latest Virtex-6 XC6VLX760 offering 759K 6-LUTs the compilation time would increase to at least $\sim 760min = 12h30min$, which is not a reasonable time for a "rapid" digital design. Hence, the cost of overall system increases even higher due to the cost of the place and route time of the design. This also supports TPM approach which stresses utilization of smaller devices of size where recompilation has to be done only on, the single segment component and not on the whole design. This way, if a design that was divided into 10 segments processed on TPM platform, requires a modification in one of the segments, it would need only tenth of the compilation time in comparison to a monolith design. Therefore, the further tendency of increasing the on-chip FPGA resources keeping methodology of monolithic (ASIC-type) design will dramatically increase instrumentation resources and cost.

9.5 Summary

This chapter presented the set of experiments that were designed for the verification of the TPM, as well as quantitative evaluation of the FPGA devices with different configurations of VHCs in SSP cores. TPM operation was verified with several video processing cores operating on captured stereo images. The successful tests prove that not only three or four but many more processing cores can be requested by the embedded system itself, hence, giving the system the capability of run-time architecture-to-task adaptation. Results of tests of multiple VHCs on SSP core showed an overwhelming support for use of small to mid-size FPGAs with utilization of TPM approach. Benefits of utilization of small to mid-size devices were shown in power consumption, specifically in quiescent power, speed of processing, and resource utilization. At last, the analysis of place and route time showed, that when large designs are used the modification/recompilation task cannot be performed as fast and effective as it can be done on mid-size FPGAs.

Described experiments demonstrated and proved the effectiveness of temporal partitioning of run-time reconfigurable computing resources using small/mid-size FPGA devices.

Chapter 10

Summary

10.1 Summary of Research

Recent changes in the area of high-performance computing systems and their application are the major motivation for the research of dynamically reconfigurable and adaptive computers. The proposed research focuses on one of the most promising directions in this area of research - dynamically reconfigurable systems with temporal partitioning of homogeneous logic, routing, and memory resources. The approach of temporal partitioning of FPGA resources is not new. However, previously the main advantage of this approach was considered to be an ability to execute tasks that required much more computing resources than an FPGA device could provide. This approach was motivated by limited resources that FPGA vendors provided a decade ago. Nowadays, the advances in process technologies allowed to increase on-chip FPGA resources by many orders of magnitude, so now other aspects of temporal partitioning can be exploited. Major aspects are potential virtualization of computing resources and dynamic synthesis of processing data-paths by large macro-operators, associated with hardware implementations that form processing algorithms. The first aspect of virtualization of computing resources is similar to virtualization of memory in conventional computers. It became possible because of homogeneous nature of logic and routing resources in FPGA devices,

making possible the utilization of the same logic and routing resources multiple times. The dynamic synthesis of data-paths potentially can provide the system with very high flexibility. The flexibility means run-time adaptation of computing architecture to different changes in task algorithm (e.g. change for the mode of operation), as well as rapid recovery from the hardware faults. All these aspects, as well as the understanding of changes in the field and potential benefits of novel technologies have motivated this research.

It was clear from the beginning that there are several problems that must be solved before getting a first working prototype which can utilize the concept of virtualization of computing resources with the existing FPGA technology. Solving these problems became the main objective of the proposed research. First of all, there was a need for a methodology for high-level (architectural) synthesis of virtual hardware components that became the component basis for data stream processing, according to complex functions - macro-operators (MOs). Addressing this issue would allow to compose of application specific processors (ASPs) from pre-compiled hardware modules, and to program using macro-functions. However, the real life constraints, of logic resources, power, timing and area significantly complicate the ASP synthesis. It is necessary to consider multiple constraints to optimize the requested performance parameters and to do that in relatively short time. Another problem is the sensitivity of SRAM based FPGA devices to radiation effects and frequent hidden manufacturing defects.

The mechanism of fault identification and mitigation was considered from the aspect of run-time reconfiguration of FPGA with modification of ASP architecture. The CAD software provides an ability to do task programming using the macro-functions similar to object oriented programming in high-level graphical form. This software part would contain certain CAD support to conduct proper segmentation of programmed task algorithm and perform synthesis of segment specific processor (SSP), while being optimized in the above mentioned multi-parametric constraint space. Both had to be implemented in a form of platform prototype and associated CAD tool and verified. The self-restoration mechanism which would provide

mitigation from transient and permanent hardware faults in platform FPGA also expected to be developed and embedded to the above platform prototype.

As the result of the research presented in this thesis, all of the above problems have been successfully solved and the methodology for synthesis and design of cost-effective dynamically reconfigurable computing systems with temporal partitioning of computing resources has been completely developed and tested. The main contributions of this research can be divided in theoretical and proof-of-concept contributions as described in detail in the next section.

10.2 Contributions

Contributions of this work are described on a per chapter basis. Since Chapters 1 and 2 introduced the computing architectures and reconfigurable computing, the contributions begin with the introduction of architectural organization in Chapter 3.

Chapter 3 proposed a novel approach to the design of TPM architecture, including architectural organization of pipelined and non-pipelined architectures for support of the TPM mechanism. The methodology of architectural design of a configuration controller also was proposed and developed. The architecture of TPM has been presented and published in several conferences and journal publications [58], [51], [54].

Chapter 4 covers the fault tolerance aspect of the reconfigurable systems and proposed novel mitigation algorithm, that allows run-time recovery from transient and permanent faults. Two novel methods for run-time mitigation faults in FPGA systems with SRAM configuration memory were proposed and developed. Both methods utilize the concept of area avoidance. The first method operates on the response from the previous diagnostic IP-core. It performs scrubbing or loads an IP-core from the library that performs similar operation and avoids the faulty area. The second method provides a recovery from permanent faults with or without functional degradation by sequential reconfiguration of a set of IP-cores. These methods were published in the Journal for Spacecrafts and Rockets [50].

Task segmentation was introduced in Chapter 5. The notion of cost-effectiveness of RCS with TPM was presented with methodology of its evaluation based on system components. This chapter also showed that design of a TPM system has to be balanced and reflect the performance requirements in order to achieve the most cost-effective architecture. Methodology for evaluation of cost-effectiveness of RCS with TPM was presented in the conference and published in conference proceedings [52].

Chapter 6 presented a novel methodology of creation of Virtual Hardware Components (VHCs) from Macro-Operators that were in turn composed of elementary operators. Major contribution of this chapter was development of the method for generation of a VHC set from an Architecture Configuration Graph (ACG) and ACG pruning. Creation of the VHC set is done by variation in resource binding and scheduling, which produced a diversity of VHCs with different performance parameters. By identification of multi-parametric restrictions, ACG tree is pruned to result in a limited set of VHCs associated with the same macro-operator. It was shown that set of VHCs that were arranged in ascending or descending order for each parameter provide an efficient method of selection the optimal variant. Methodologies described in this chapter were published in journal and conference publications [51], [57], [58].

Chapter 7 proposed a novel methodology for synthesis of a set of Segment Specific Processors (SSPs). The methodology involved generation of SSPs optimized over several parameters of a given algorithm/task. Task segmentation is main steps were covered, such as: assignment of dependency relation to each of the MOs in the sequencing graph representing a task, MO selection methodology and creation of sub-ACG graphs which result in the composition of SSPs. Depending on the parametric constraints, segmentation may vary resulting in different SSP compositions. All of the proposed algorithms from this chapter were implemented in the GUI application that performed automatic segmentation of the created sequencing graph.

Chapter 8 presented contribution of hardware and software implementation towards this work. Hardware and software were specifically developed, implemented and debugged for verification of the methodologies in this work. Chapter 8 described in detail all components of the Multi-stream Adaptive Reconfigurable System (MARS) and the reasons why they were selected for implementation. The implementation of the CAD tool for task algorithm segmentation and synthesis of SSP set is described with associated SPC GUI software. This chapter also described in detail the hardware architecture for the temporal partitioning of RCS that executes SSPs and its building blocks. Implementing of TPM platform and associated developed software were presented in several workshops (SVAR 2007, 2008), and published in conference and journal publications [52, 56], [55], [54].

Chapter 9 deal with experimental portion of this research. Specific experimental setup was developed for the series of experiments and for the verification of the TPM. Quantitative evaluation of the FPGA devices with different configurations of VHCs in SSP cores was done. This chapter also presented analysis of the obtained results, and based on the analysis concluded with recommendations for the design of TPM architectures. Experiments have proven the methodologies and concepts. Results of multiple VHCs on SSP IP-core revealed an overwhelming support for the use of smaller to mid-sized FPGAs with utilization of TPM approach. Benefits of using smaller to mid-sized FPGAs were in power consumption, specifically in quiescent power, as well as, in speed of processing, and resource utilization.

Overall, the obtained results have proved the proposed methodology and allowed to develop a new class of reconfigurable computing systems that can provide several benefits based on utilization of computing resource virtualization such as:

1. Flexible computing architecture that can satisfy multiple parametric requirements.

2. Capability for automatic restoration from hardware faults and, thus, a longer life-time for a system.

3. Acceleration in application programming by using macro-functions instead of regular HDL programming.

4. Maximization of cost-performance parameters.

5. Application flexibility, such as multi-modal and multi-task workload instead of ASIC-type uni-task applications.

Nonetheless, there are some further research components which can improve and extend functionality of the proposed class of RCS.

10.3 Future Works

As was summarized in this chapter, substantial amount of research for this thesis lead to several ever more expandable fields that can be researched, designed, and implemented. Some of the areas of future work is described in this closing section. There are four main areas:

1. Development of an integrated CAD tool.

2. Incorporation of the segmentation mechanism with the run-time VHC selection in a real-time hardware OS.

3. Fault tolerance testing in real conditions of energized particles and further verification of the proposed methodologies.

4. Automation and creation of an extensive VHC and SSP library.

Following subsections describe this work in more detail.

10.3.1 Integrated CAD Tool

The developed CAD tool, that was presented in Chapter 8, provides a user with a capability of design estimation and selection of the appropriate VHCs for SSPs. This tool can be expanded

to be a much more integrated tool for design of systems with TPM architecture. The a fully integrated tool would be used to create full design of SSPs with further compilation into the bitstreams with the options similar to the Xilinx ISE Design Suite, Altera Quartus II, and other designer suites. In addition, the CAD tool could have a capability of generating different types of library files suited for many options of configuration controller architectures. Also, this CAD tool should include an expert VHC creator that would allow user to generate several VHCs with different performance by providing only one version of VHDL or Verilog code. The CAD software would perform all of the remaining operations to create variance of VHCs that are constrained within the given parameters. This would allow such CAD tool to evolve and aggregate as do object classes in object oriented design.

10.3.2 Segmentation in Real-Time Hardware OS

Methods for task segmentation and SSP selection were designed in a way that they can be adapted to any computing platform. This platform is typically assumed to be a user's PC, however, with the current availability of powerful embedded processors segmentation can be executed on these processors. This would give a unique opportunity of creating self-adaptable hardware systems that, based on the given processing algorithm skeleton, can perform all of the SSP selection operations. With an availability of the extensive SSP libraries which are stored on a non-volatile memory, real-time hardware OS would be able to create and modify a configuration schedule of the TPM platform. This is also very effective for fault tolerance systems and would give the TPM a capability of constant self-recovery, and thus make it virtually indestructible.

10.3.3 Fault Tolerance Verification & Testing

One aspect of this work that was not field tested is the hardware fault mitigation. In order to verify the SEE mitigation operation, the platform with the TPM support has to be tested

at proton and ion irradiation facility. Such facility can provide bombardment of the platform by proton or ion particles with specified fluence and dosage, as was described in the paper by David Hiemstra *et al.* [44]. Series of such tests would be able to provide a valuable insight into how TPM self-recovery mechanism performs. By identifying failure trends, the granularity of the area avoidance could be better tuned and, hence, allow for more optimal design from the stand point of fault recovery and overall performance. Also, further work can be done in the area of SEE mitigation for terrestrial applications, since the future FPGA devices with an even smaller scale of the process technology, would encounter SEU on the terrestrial level.

10.3.4 Extensive VHC and SSP library

One of the very important requirements for the TPM architecture is a library of pre-built VHC modules. Since VHCs are the essential building block of the SSPs the extensive library of VHCs provides a greater flexibility of architecture design, and an optimal task-to-architecture adaptation. Such library could be created with the general support of common processing functions that are used in the industry. A model for the initial library can be MATLAB package, who's initial library of functions can be expanded by the user or by purchasing specialized packages. In addition, the creation of the automated VHC generator from the algorithm representation would give users a tool for rapid creation of custom VHCs for their specialized needs.

Bibliography

[1] N. Abel, L. Kessal, and D. Demigny. Design flexibility using FPGA dynamical reconfiguration. *ICIP'04. International Conference on Image Processing*, 4:2821–2824, October 2004.

[2] M. Abramovici, J.M. Emmert, and C.E. Stroud. Roving STARs: an integrated approach to on-line testing, diagnosis, and fault tolerance for FPGAs in adaptive computing systems. *Proceedings of The Third NASA/DoD Workshop on Evolvable Hardware*, (7):73–92, July 2001.

[3] M. Aksit and Z. Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. *In Proceedings of 23rd International Conference on Distributed Computing Systems Workshops*, pages 84–89, May 2003.

[4] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, A. Marmo, S. Pastore, and G.R. Sechi. A Tool for Injecting SEU-like Faults into the Configuration Control Mechanism of Xilinx Virtex FPGAs. *Proceedings of 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 71–78, Nov. 2003.

[5] A. Alsolaim, J. Becker, M. Glesner, and J. Starzyk. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–214, 2000.

[6] Altera. *Using the Nios II Configuration Controller Reference Designs*, March 2009.

[7] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. Dynamic and Partial FPGA Exploitation. *Proceedings of the IEEE*, 95(2):438–452, Feb 2007.

[8] M. Berg. Fault tolerance implementation within SRAM based FPGA designs based upon the increased level of single event upset susceptibility. In *IOLTS 2006. 12th IEEE International On-Line Testing Symposium*, page 3, 2006.

[9] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT Compilation for Off-the-Shelf Dynamically Reconfigurable FPGAs. *Compiler Construction*, 4959:178–192, April 2008.

[10] Christophe Bobda. Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement. Master's thesis, University of Paderborn, 2003.

[11] K. Bondalapati and V.K. Prasanna. Reconfigurable computing systems. *Proceedings of the IEEE*, 90(7):1201–1217, Jul 2002.

[12] M. Borgatti, A. Capello, U. Rossi, J.-L. Lambert, I. Moussa, F. Fummi, and G. Pravadelli. An integrated design and verification methodology for reconfigurable multimedia systems. *Proceedings of Design, Automation and Test in Europe*, 3:266–271, March 2005.

[13] Paulo S. Brand, Nascimento, and Manoel Eusebio de Lima. Temporal partitioning for image processing based on time-space complexity in reconfigurable architectures. In *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe*, pages 375–380, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[14] Brendan Bridgford, Carl Carmichael, and Chen Wei Tseng. Correcting Single-Event Upsets in Virtex-II Platform FPGA Configuration Memory. February 2007.

[15] C. Carmichael. *Virtex FPGA series configuration and readback.* Xilinx Inc., 2.8 edition, March 2005.

[16] C. Chantrapornchai, E.M. Sha, and X.S. Hu. Efficient design exploration based on module utility selection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(1):19–29, Jan 2000.

[17] Sumanta Chaudhuri, Jean-Luc Danger, Sylvain Guilley, and Philippe Hoogvorst. FASE: An Open Run-Time Reconfigurable FPGA Architecture for Tamper-Resistant and Secure Embedded Systems. In *ReConFig 2006. IEEE International Conference on Reconfigurable Computing and FPGA's*, pages 1–9, Sept. 2006.

[18] Weisheng Chong, S. Ogata, M. Hariyama, and M. Kameyama. Architecture of a Multi-Context FPGA Using Reconfigurable Context Memory. *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium*, page 144a, April 2005.

[19] Pil Woo Chun, Jamin Islam, Valeri Kirischian, and Lev Kirischian. Implementing a cost-effective run-time reconfigurable system for stream applications. In *ICEE 2008. Second International Conference on Electrical Engineering*, pages 1–5, March 2008.

[20] Pill Woo Chun, Valeri Kirischian, Sergei Zhelnakov, and Lev Kirischian. Reconfigurable Multiprocessor with Self-optimizing, Self-assembling, and Self-restoring Micro-architecture. In *WARFP2005 Proceedings of Workshop on Architecture Research using FPGA Platform*, February 2005.

[21] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):967–979, Aug 1988.

[22] Altera Corporation. *Stratix II Device Handbook.* Altera Inc., 3 edition, May 2005.

[23] V. Correia and A. Reis. Advanced technology mapping for standard-cell generators. In *Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on*, pages 254–259, Sept. 2004.

[24] D.R. Czajkowski, P.K. Samudrala, and M.P. Pagey. SEU mitigation for reconfigurable FPGAs. *IEEE Aerospace Conference*, page 7, March 2006.

[25] A. Dasu and S. Panchanathan. Reconfigurable media processing. *Proceedings of International Conference on Information Technology: Coding and Computing*, pages 300–304, Apr. 2001.

[26] A. Dasu and S. Panchanathan. A survey of media processing approaches. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(8):633–645, Aug 2002.

[27] E. Davies. *Machine Vision: Theory, Algorithms and Practicalities.* Academic Press, 1990.

[28] R.F. DeMara and Kening Zhang. Autonomous FPGA fault handling through competitive runtime reconfiguration. *Proceedings of 2005 NASA/DoD Conference on Evolvable Hardware*, pages 109–116, July 2005.

[29] Paulo Sergio B. do Nascimento, Manoel E. de Lima, Stelita M. da Silva, and Jordana L. Seixas. Mapping of image processing systems to FPGA computer based on temporal partitioning and design space exploration. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 50–55, 2006.

[30] A. Doumar and H. Ito. Detecting, diagnosing, and tolerating faults in SRAM-based field programmable gate arrays: a survey. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):386–405, June 2003.

[31] Peter E. Hart Duda, Richard O. *Pattern classification and scene analysis.* John Wiley-Sons, 1973.

[32] Michalis D. Galanis, Gregory Dimitroulakos, and Costas E. Goutis. Partitioning Methodology for Heterogeneous Reconfigurable Functional Units. *The Journal of Supercomputing*, 38:17–34, October 2006.

[33] Maya Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation With Field-Programmable Gate Arrays.* Birkhauser, 2006.

[34] Kim Golblatt. *The Express Configuration of SpartanXL FPGAs.* Xilinx Inc., 1.0 edition, November 1998.

[35] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor. PipeRench: a reconfigurable architecture and compiler. *Computer*, 33(4):70–77, Apr 2000.

[36] S. Golshan and E. Bozorgzadeh. Single-event-upset (SEU) awareness in FPGA routing. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 330–333, New York, NY, USA, 2007. ACM.

[37] I. Gonzalez, S. Lopez-Buedo, and F.J. Gomez-Arribas. Implementation of secure applications in self-reconfigurable systems. *Microprocessors and Microsystems*, May 2007.

[38] Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins. *Digital Image Processing using Matlab.* Prentice Hall, 2004.

[39] P. Graham, M. Caffrey, J. Zimmerman, D. E. Johnson, P. Sundararajan, and C. Patterson. Consequences and Categories of SRAM FPGA Configuration SEUs. In *Military and Aerospace Applications of Programmable Logic Devices*, 2003.

[40] Masanori Hariyama and Michitaka Kameyama. A Multi-Context FPGA Using a Floating-Gate-MOS Functional Pass-Gate and Its CAD Environment. *Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on*, pages 1803–1806, Dec. 2006.

[41] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. *Proceedings of Design, Automation and Test in Europe, Conference and Exhibition*, pages 642–649, 2001.

[42] P.J. Hatcher, M.J. Quinn, A.J. Lapadula, B.K. Seevers, R.J. Anderson, and R.R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, Jul 1991.

[43] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, November, 2000.

[44] David M. Hiemstra, Fayez Chayab, and Zaeem Mohammed. Single Event Upset Characterization of the Virtex-4 Field Programmable Gate Array Using Proton Irradiation. *IEEE Radiation Effects Data Workshop*, pages 105–108, July 2006.

[45] R.D. Hudson, D.I. Lehn, and P.M. Athanas. A run-time reconfigurable engine for image interpolation. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 88–95, Apr 1998.

[46] Michael Huebner, Tobias Becker, and Juergen Becker. Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 28–32, 2004.

[47] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D.C. Cronquist, and M. Sivaraman. PICO: automatically designing custom computers. *Computer*, 35(9):39–47, Sep 2002.

[48] K.M. Kavi, B.P. Buckles, and U.N. Bhat. A Formal Definition of Data Flow Graph Models. *IEEE Transactions on Computers*, C-35(11):940–948, Nov. 1986.

[49] L. Kirischian. Optimization of parallel task execution on the adaptive reconfigurable group organized computing system. *PARELEC 2000, Proceedings of International Conference on Parallel Computing in Electrical Engineering*, pages 100–105, 2000.

[50] Lev Kirischian, Vadim Geurkov, Valeri Kirischian, Jacob Kleiman, and Irina Terterian. Multilevel Radiation Protection of Partially Reconfigurable Field Programmable Gate Array Devices. *Journal of Spacecraft and Rockets*, 43:523–529, 2006.

[51] Lev Kirischian, Vadim Geurkov, Valeri Kirischian, and Irina Terterian. Multiparametric optimisation of the modular computer architecture. *International Journal of Technology, Policy and Management*, 6:327–346, 2006.

[52] V. Kirischian, V. Geurkov, and L. Kirischian. Cost effective reconfigurable architecture for stream processing applications. In *CCECE 2008. Canadian Conference on Electrical and Computer Engineering*, pages 541–546, May 2008.

[53] Valeri Kirischian, Vadim Geurkov, Pill Woo Chun, and Lev Kirischian. Reconfigurable Macro-processor - Cost-efficient Platform for Rapid Prototyping. In *FAIM2007: Flexible Automation and Intelligent Manufacturing*, volume 2, pages 781–788, June 2007.

[54] Valeri Kirischian, Vadim Geurkov, Pill Woo Chun, and Lev Kirischian. Macro-programmable reconfigurable stream processor for collaborative manufacturing systems. *Journal of Intelligent Manufacturing*, 19:723–734, 2008.

[55] Valeri Kirischian, Vadim Geurkov, and Lev Kirischian. A Cost Efficient Reconfigurable Video Processing Platform for Machine Vision. In *Seventh International Workshop on Advanced Manufacturing Technologies - ATM2007*, page 44, June 2007.

[56] Valeri Kirischian, Vadim Geurkov, and Lev Kirischian. A multi-mode video-stream processor with cyclically reconfigurable architecture. In *CF '08: Proceedings of the 5th conference on Computing Frontiers*, pages 105–106, 2008.

[57] Valeri Kirischian, Irina Terterian, and Lev Kirischian. Optimization of Architecture Selection in the Multi-parametric Design Space. In *17-th International Conference on Systems Research, Infomatics & Cybernetics: InterSymp-2005*, volume 4, pages 30–35, August 2005.

[58] Valeri Kirischian, Sergei Zhelnakov, Pill Woo Chun, Lev Kirischian, and Vadim Geurkov. Uniform Reconfigurable Processing Module for Design and Manufacturing Integration. In *Fifth International Workshop on Advanced Manufacturing Technologies - ATM2005*, pages 77–82, May 2005.

[59] D. Koch, C. Beckhoff, and J. Teich. Bitstream Decompression for High Speed FPGA Configuration from Slow Memories. *ICFPT 2007. International Conference on Field-Programmable Technology*, pages 161–168, Dec. 2007.

[60] Y.E. Krasteva, A.B. Jimeno, E. de la Torre, and T. Riesgo. Straight method for reallocation of complex cores by dynamic reconfiguration in Virtex II FPGAs. *RSP 2005. The 16th IEEE International Workshop on Rapid System Prototyping*, pages 77–83, June 2005.

[61] V. Krishnan and S. Katkoori. A genetic algorithm for the design space exploration of datapaths during high-level synthesis. *IEEE Transactions on Evolutionary Computation*, 10(3):213–229, June 2006.

[62] Alexei Kudriavtsev and Peter Kogge. Generation of permutations for SIMD processors. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 147–156, 2005.

[63] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Compile-time area estimation for LUT-based FPGAs. *ACM Transactions on Automation of Electron Systems*, 11(1):104–122, 2006.

[64] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field Programmable Gate Arrays*, pages 21–30, 2006.

[65] Marco Lanuzza, Paolo Zicari, Fabio Frustaci, Stefania Perri, and Pasquale Corsonello. An Efficient and Low-Cost Design Methodology to Improve SRAM-Based FPGA Robustness in Space and Avionics Applications. *Reconfigurable Computing: Architectures, Tools and Applications*, 5453/2009:74–84, 2009.

[66] Lattice Inc. *Lattice ispTRACY Usage Guide*, tn1054 edition, February 2006.

[67] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hannikainen, and T.D. Hamalainen. A parallel MPEG-4 encoder for FPGA based multiprocessor SoC. *International Conference on Field Programmable Logic and Applications*, pages 380–385, Aug. 2005.

[68] A. Lodi, L. Ciccarelli, A. Cappelli, F. Carnpi, and M. Toma. Decoder-based multi-context interconnect architecture. *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, pages 231–233, Sept. 2003.

[69] R. Lyseckya, F. Vahid, and S.X. Tan. Dynamic FPGA routing for just-in-time FPGA compilation. *Proceedings of 41st Design Automation Conference*, pages 954–959, 2004.

[70] T. Makimoto. The hot decade of field programmable technologies. *Proceedings of 2002 IEEE International Conference on Field-Programmable Technology*, pages 3–6, Dec. 2002.

[71] V. Manohararajah, S.D. Brown, and Z.G. Vranesic. Heuristics for area minimization in lut-based fpga technology mapping. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(11):2331–2340, Nov. 2006.

[72] Gary S. May and Simon M. Sze. *Fundamentals of Semiconductor Fabrication.* John Wiley & Sons, Inc., 2003.

[73] Michael Wirthlin D. Eric Johnson Nathaniel Rollins Maya Gokhale, Paul Graham. Dynamic reconfiguration for management of radiation-induced faults in FPGAs. *International Journal of Embedded Systems*, 2:28–38, 2006.

[74] S. Toutounchi M.B. Tahoori, S. Mitra and E.J. McCluskey. Fault Grading FPGA Interconnect Test Configurations. *International Test Conference*, pages 608–617, 2003.

[75] E.J. Mcdonald. Runtime FPGA partial reconfiguration. *IEEE Aerospace and Electronic Systems Magazine*, 23(7):10–15, July 2008.

[76] F. Mehdipour, M.S. Zamani, H.R. Ahmadifar, M. Sedighi, and K. Murakami. Reducing reconfiguration time of reconfigurable computing systems in integrated temporal partitioning and physical design framework. *IPDPS 2006. 20th International Parallel and Distributed Processing Symposium*, page 8, 25-29 April 2006.

[77] B. Mei, A. Lambrechts, J.Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *IEEE Design and Test of Computers*, 22(2):90–101, April 2005.

[78] S. Merchant, G.D. Peterson, and D. Bouldin. Improving embedded systems education: laboratory enhancements using programmable systems on chip. *(MSE '05). Proceedings of 2005 IEEE International Conference on Microelectronic Systems Education*, pages 5–6, June 2005.

[79] Giovani De Micheli. *Synthesis and Optimization of Digital Circuits*. McGrow-Hill, 1994.

[80] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to technology mapping for LUT-based FPGAs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 41–49, New York, NY, USA, 2006. ACM.

[81] Abdellatif Mtibaa, Bouraoui Ouni, and Mohamed Abid. An efficient list scheduling algorithm for time placement problem. *Hardware/Software System on Chip Co-design: Approach and Application*, 33(44):285–298, July 2007.

[82] P.L. Murray and D. VanBuren. Single Event Effect Mitigation in ReConfigurable Computers for Space Applications. *IEEE Aerospace Conference*, pages 1–7, March 2005.

[83] Mark Ng and Mike Peattie. *Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode*. Xilinx Inc., 1.5 edition, December 2007.

[84] K. Paulsson, M. Hubner, G. Auer, M. Dreschmann, L. Chen, and J. Becker. Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self-Reconfiguration on Xilinx Spartan III FPGAs. In *FPL 2007. International Conference on Field Programmable Logic and Applications*, pages 351–356, Aug. 2007.

[85] K. Paulsson, M. Hübner, and J. Becker. On-line optimization of FPGA power-dissipation by exploiting run-time adaption of communication primitives. In *SBCCI*

'06: Proceedings of the 19th annual symposium on Integrated circuits and systems design, pages 173–178, 2006.

[86] R. Perez. Methods for Spacecraft Avionics Protection Against Space Radiation in the Form of Single-Event Transients. *IEEE Transactions on Electromagnetic Compatibility*, 50(3):455–465, Aug. 2008.

[87] Brian Pratt, Michael Caffrey, Paul Graham, Keith Morgan, and Michael Wirthlin. Improving FPGA Design Robustness with Partial TMR. *44th Annual., IEEE International Reliability Physics Symposium Proceedings*, pages 226–232, March 2006.

[88] K.M.G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, 48(6):579–590, June 1999.

[89] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. A parallel configuration model for reducing the run-time reconfiguration overhead. In *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe*, pages 965–969, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[90] A. D. George R. Hymel and H. Lam. Evaluating Partial Reconfiguration for Embedded FPGA Applications. In *Proceedings of High-Performance Embedded Computing Workshop*, Sept. 2007.

[91] J. Resano, D. Mozos, D. Verkest, and F. Catthoor. A reconfigurable manager for dynamically reconfigurable hardware. *IEEE Design & Test of Computers*, 22(5):452–460, Oct. 2005.

[92] L. Rockett, D. Patel, S. Danziger, B. Cronquist, and J.J. Wang. Radiation Hardened FPGA Technology for Space Applications. *2007 IEEE Aerospace Conference*, pages 1–7, March 2007.

[93] J. Rose and S. Brown. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE Journal of Solid-State Circuits*, 26(3):277–282, Mar 1991.

[94] E. Sanchez, M. Sipper, J.-O. Haenni, J.L. Beuchat, A. Stauffer, and A. Perez-Uribe. Static and dynamic configurable systems. *IEEE Transactions on Computers*, (6):556–564, June 1999.

[95] P. Schumacher, M. Mattavelli, A. Chirila-Rus, and R. Turney. A Virtual Socket Framework for Rapid Emulation of Video and Multimedia Designs. *ICME 2005. IEEE International Conference on Multimedia and Expo*, pages 872–875, July 2005.

[96] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in Virtex FPGAs. *IEEE Proceedings of Computers and Digital Techniques*, 153(3):157–164, May 2006.

[97] D. Seto and M. Watanabe. Reconfiguration performance analysis of a dynamic optically reconfigurable gate array architecture. *ICFPT 2007. International Conference on Field-Programmable Technology*, pages 265–268, Dec. 2007.

[98] Nikunj Shroff. Memory Hierarchy for Microblaze and PowerPC based Systems. Master's thesis, Indian Institute of Technology Delhi, May 2007.

[99] Miguel L. Silva and Joao Canas Ferreira. Support for partial run-time reconfiguration of platform FPGAs. *Journal of Systems Architecture*, 52:709–726, December 2006.

[100] D.P. Singh, V. Manohararajah, and S.D. Brown. Two-stage physical synthesis for FPGAs. *Proceedings of the IEEE 2005 Custom Integrated Circuits Conference*, pages 171–178, Sept. 2005.

[101] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and

computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[102] Gerard J.M. Smit, André B.J. Kokkeler, Pascal T. Wolkotte, and Marcel D. van de Burgwal. Multi-core architectures and streaming applications. *Proceedings of the 2008 international workshop on System level interconnect prediction*, pages 35–42, 2008.

[103] G Snider. Spacewalker: Automated Design Space Exploration for Embedded Computer Systems. *HP Laboratories Palo Alto HPL-2001-220*, (1), september 2001.

[104] Suresh Srinivasan, Prasanth Mangalagiri, Yuan Xie, N. Vijaykrishnan, and Karthik Sarpatwari. FLAW: FPGA lifetime awareness. *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 630–635, 2006.

[105] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, Nov. 2004.

[106] E. Stott, P. Sedcole, and P. Cheung. Fault tolerant methods for reliability in FPGAs. In *FPL 2008. International Conference on Field Programmable Logic and Applications*, pages 415–420, Sept. 2008.

[107] I. Taniguchi, K. Ueda, K. Sakanushi, Y. Takeuchi, and M. Imai. Task Partitioning Oriented Architecture Exploration Method for Dynamic Reconfigurable Architectures. *2006 IFIP International Conference on Very Large Scale Integration*, pages 290–295, Oct. 2006.

[108] C. Tanougast, Y. Berviller, P. Brunet, and S. Weber. Automated RTR temporal partitioning for reconfigurable embedded real-time system design. *Proceedings of International Parallel and Distributed Processing Symposium*, page 8, 22-26 April 2003.

[109] R. Tessier and W. Burleson. Reconfigurable Computing for Digital Signal Processing: A Survey. *The Journal of VLSI Signal Processing*, 28:7–27, May 2001.

[110] Andres Upegui and Eduardo Sanchez. Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs. *Evolvable Systems: From Biology to Hardware*, 3637:56–65, 2005.

[111] Nikolaos S. Voros and Konstantinos Masselos. *System Level Design of Reconfigurable Systems-on-Chip*. Springer, 2005.

[112] Xilinx Inc. *Xilinx Configuration Solutions*, 1.1 edition, May 2006.

[113] Xilinx Inc. *ChipScope Pro Software and Cores User Guide*, v9.1.01 edition, 2007.

[114] Xilinx Inc. *Embedded System Tools Reference Manual*, 10.1 edition, February 2008.

[115] Xilinx Inc. *Radiation-Tolerant Virtex-4 QPro-V Family Overview*, 1.2 edition, December 2008.

[116] Xilinx Inc. *Single-Event Upset Mitigation for Xilinx FPGA Block Memories*, 1.1 edition, March 2008.

[117] Xilinx Inc. *System ACE CompactFlash Solution*, 2.0 edition, October 2008.

[118] Xilinx Inc. *Virtex-5 FPGA User Guide*, 4.2 edition, May 2008.

[119] Xilinx Inc. *Virtex-5 FPGA Configuration User Guide*, 3.6 edition, February 2009.

[120] Hui-Jae You, Sun-Tae Chung, and Souhwan Jung. Optimization of SAD Algorithm on VLIW DSP. *Engineering and Technology World Academy of Science*, 27:1307–1314, February 2008.

[121] Chi Wai Yu, Julien Lamoureux, Steven J.E. Wilton, Philip H.W. Leong, and Wayne Luk. The Coarse-Grained / Fine-Grained Logic Interface in FPGAs with Embedded

Floating-Point Arithmetic Units. *4th Southern Conference on Programmable Logic*, pages 63–68, March 2008.

[122] Hamid R. Zarandi and Seyed Ghassem Miremadi. Dependability evaluation of Altera FPGA-based embedded systems subjected to SEUs. *Microelectronics Reliability*, 47:461–470, 2007.

Appendix A

Border Variant Search

This is an example of the border variant search for a VHC with a limitation of 40mW, based on the resources shown in Figure 6.17. The comparison of Figures A.1 and A.2 shows that after only 5 iterations of the algorithm the border variant of the VHC was found. By identifying the border variant, the rule $R3,2 \rightarrow R1,1 \rightarrow R2,3$ is determined which is used in the subsequent searches of the VHC variants based on other parameters.

A border variant search for the VHC with a limitation of 225CLB is shown in the Figure A.3. Similarly as with the power parameter comparison of Figures A.3 and A.4 shows that after 6 iterations of the algorithm the border variant of the VHC was found. A new rule is identified to be $R2,2 \rightarrow R3,2 \rightarrow R1,1$.

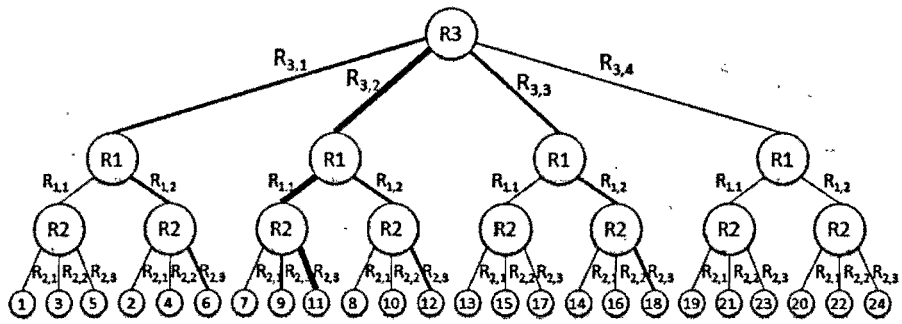
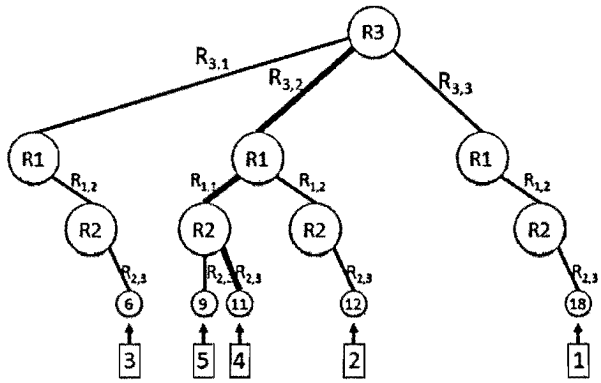


Figure A.1: Example of ACG with Selected border Variant of VHC with a 40 mW Limit Restriction



#	Result	Path	Fixed	<, >, =
1	24 mW	R3,3→R1,2→R2,3	-	>
2	36 mW	R3,3→R1,2→R2,3	-	>
3	46 mW	R3,3→R1,2→R2,3	-	<
4	39 mW	R3,2→R1,1→R2,3	R3,2; R1,1	>
5	41 mW	R3,2→R1,1→R2,2	R3,2; R1,1	<

Figure A.2: Sequence of search for power consumption border variant with a 40 mW limit

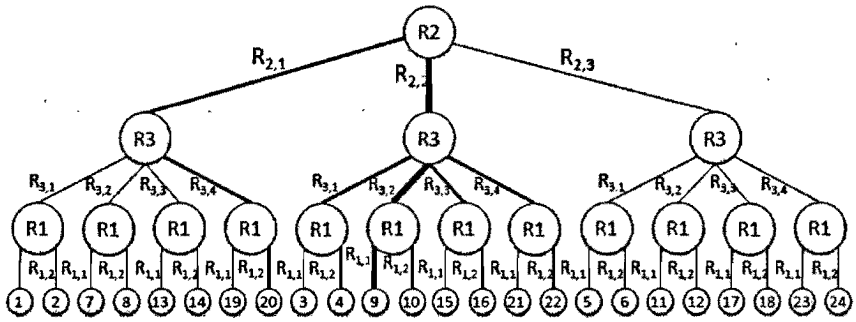
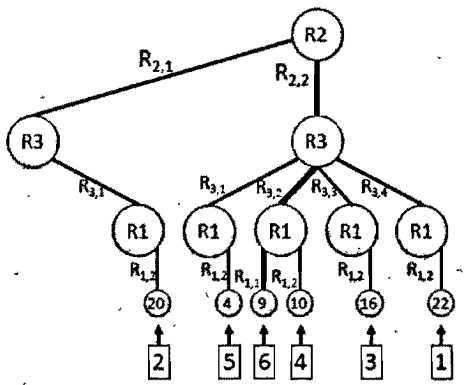


Figure A.3: Example of an ACG with Selected border Variant of VHC with a 225 CLB limit



#	Result	Path	Fixed	<, >, =
1	212 CLBs	R2,2→R3,4→R1,2	-	>
2	312 CLBs	R2,3→R3,4→R1,2	-	<
3	217 CLBs	R2,2→R3,3→R1,2	R2,2	>
4	222 CLBs	R2,2→R3,2→R1,2	R2,2	>
5	232 CLBs	R2,2→R3,1→R1,2	R2,2	<
6	225 CLBs	R2,2→R3,2→R1,1	R2,2; R3,2	=

Figure A.4: Sequence of search area requirement with a 225 CLB

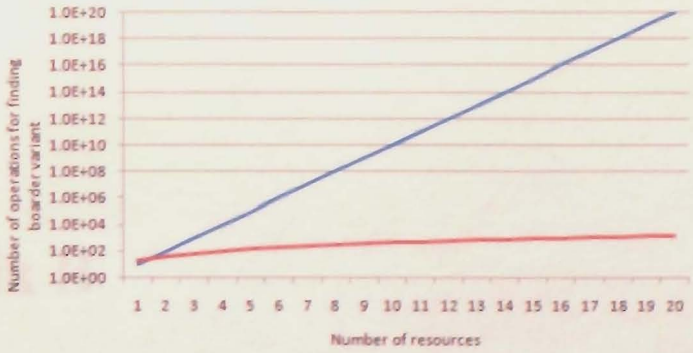


Figure A.5: Logarithmic Comparison Between Number of Variants in Exhaustive ACG Generation and border Variant Search Algorithm

Figure A.5 shows the comparison of effectiveness of this algorithm. It shows, in logarithmic scale, the number of operations required for border variant search by exhaustive (blue) method and proposed non-exhaustive (red) method.

Appendix B

Power Consumption

Following figures show complete power calculations that were done by the Xilinx XPower calculator for the experiments a described in Chapter 9. Figure B.1 shows the power consumption of the different FPGAs at three different frequencies of operation. From these graphs it is apparent that static power increases with the size of the device and the dynamic power increases with the frequency of operation and it is consistent across all of the devices. This trend is constant across all of the experiments with various types of implementations. Implementations varied in use of BlockRAM modules, output/input blocks and amount of utilized logic. The only exception was XC4VLX40 in the experiment with integrated ChipScope Pro shown in Figures B.3 and B.4, who's power consumption was higher than that of larger devices, and can be attributed to the internal architecture of the ChipScope Pro.

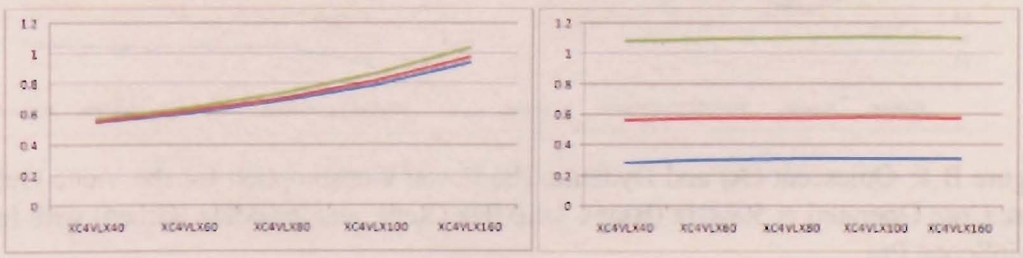


Figure B.1: Quiescent (A) and Dynamic (B) Power Consumption for the Sobel image processing SSP core. Core is operated at 50MHz (Blue), 100MHz (Red), and 200MHz (Green)

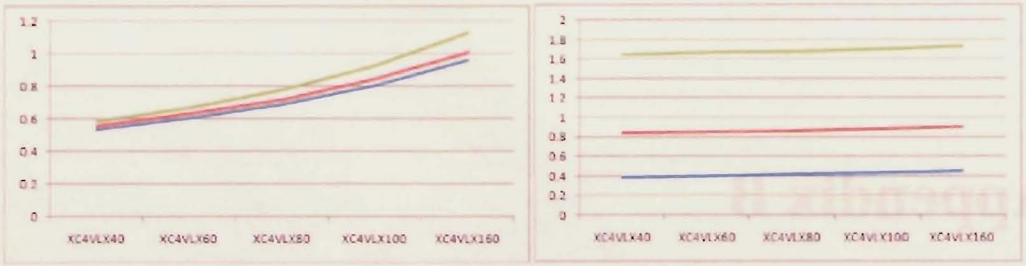


Figure B.2: Quiescent (A) and Dynamic (B) Power Consumption for the Video Output SSP core. Core is operated at 50MHz (Blue), 100MHz (Red), and 200MHz (Green)

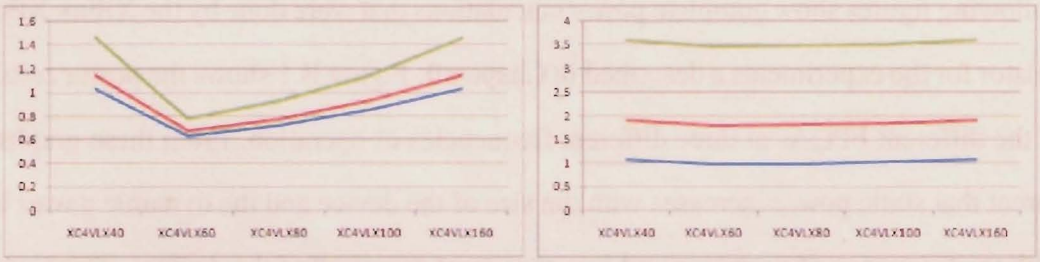


Figure B.3: Quiescent (A) and Dynamic (B) Power Consumption for the Sobel image processing SSP core. Core is operated at 50MHz (Blue), 100MHz (Red), and 200MHz (Green) with Integrated ChipScope Pro.

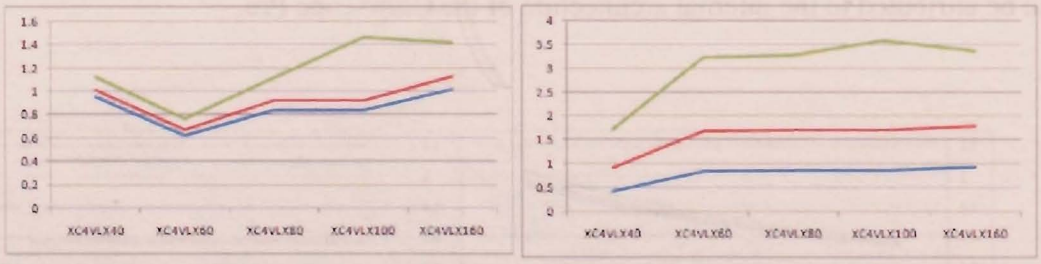


Figure B.4: Quiescent (A) and Dynamic (B) Power Consumption for the Video Output SSP core Core Operated at 50MHz (Blue), 100MHz (Red), and 200MHz (Green) with Integrated ChipScope Pro

Appendix C

Resource Utilization

Figures C.1,C.2,C.3,C.4,C.5 show the resource consumption of a single, dual, and in case of large FPGAs quad VHC cores. Experiments are described in Chapter 9 and these figures are included for completeness, in order to show the floor plans of the FPGA devices after place and route procedure.

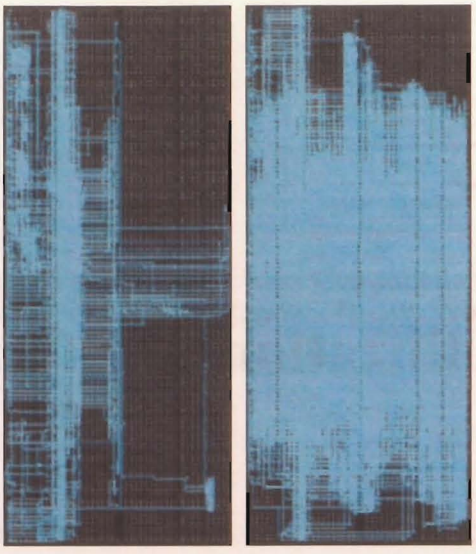


Figure C.1: Floor Plans after Place & Route for Single and Dual XC4VLX40 FPGA

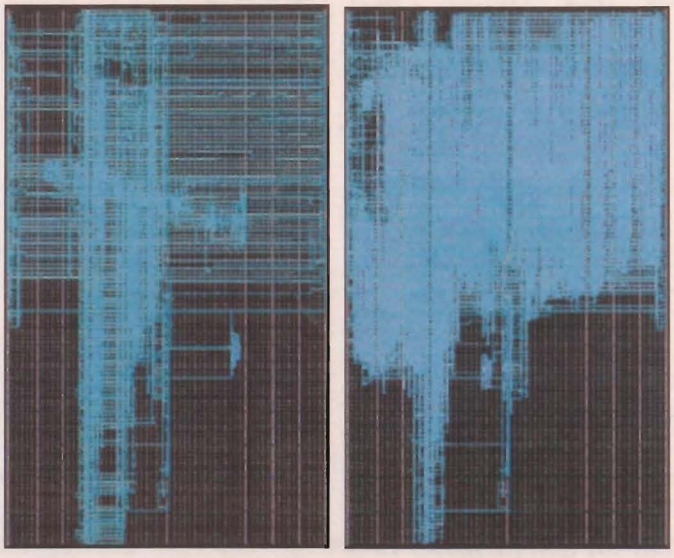


Figure C.2: Floor Plans after Place & Route for Single and Dual XC4VLX60 FPGA

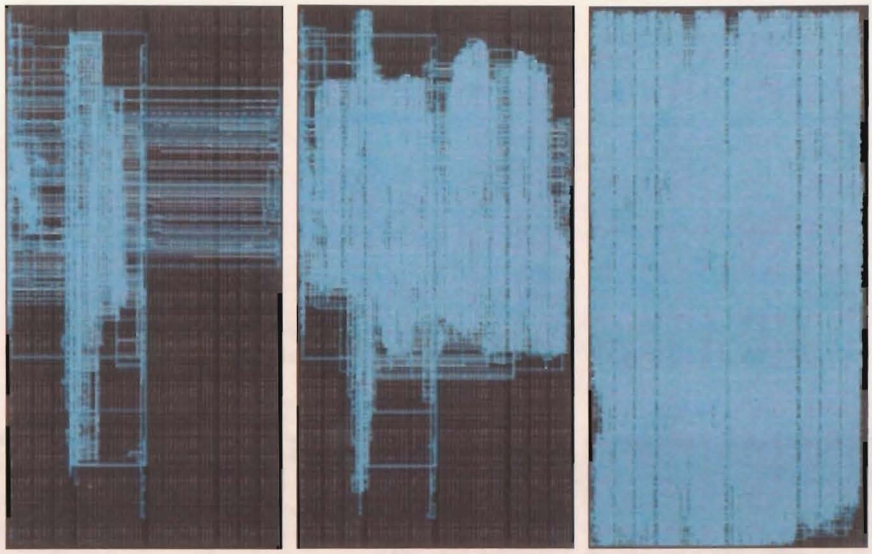


Figure C.3: Floor Plans after Place & Route for Single, Dual and Quad XC4VLX80 FPGA

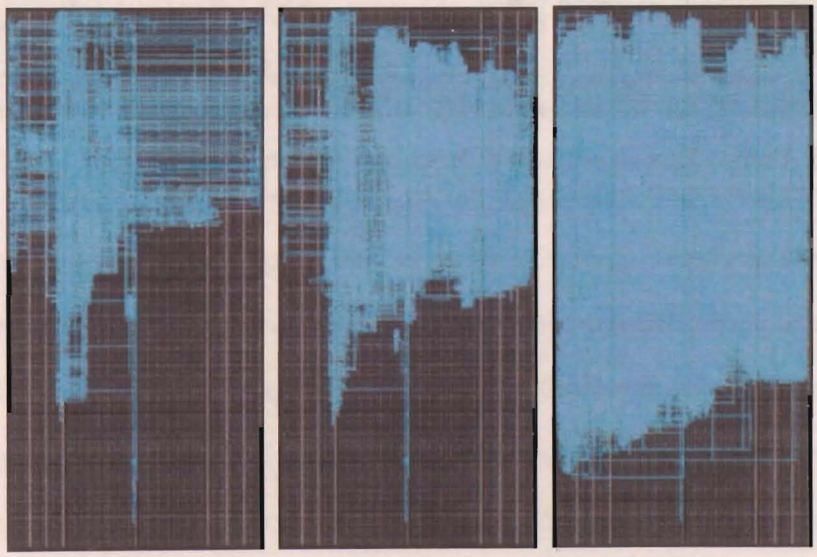


Figure C.4: Floor Plans after Place & Route for Single, Dual and Quad XC4VLX100 FPGA

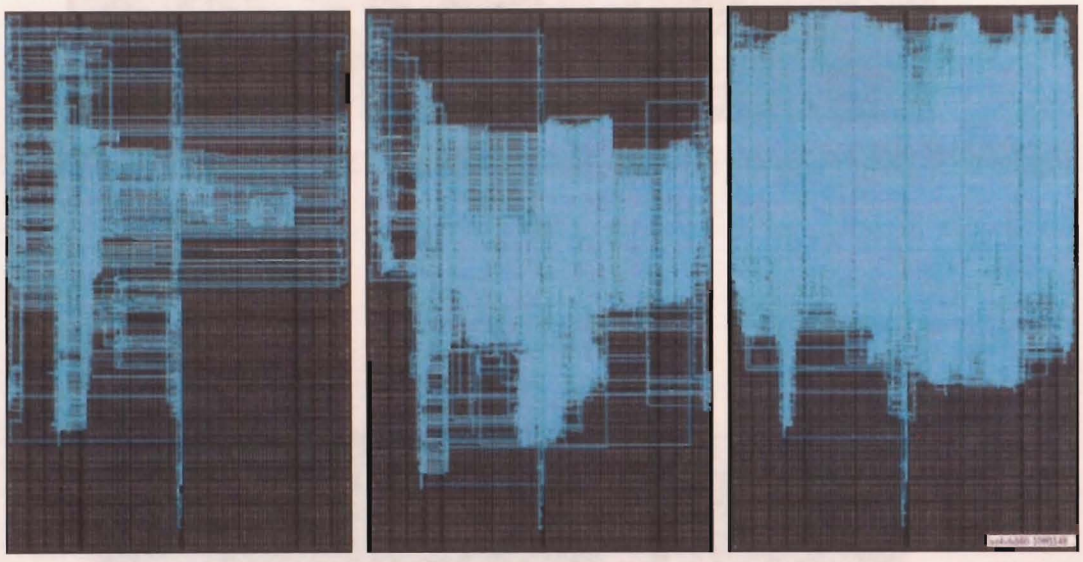


Figure C.5: Floor Plans after Place & Route for Single, Dual and Quad XC4VLX160 FPGA

Appendix D

Comparison of System design with Virtual Hardware Components utilizing TPM to Standard-cell Approach

Chapter 7 talked about the MOs and VHCs, corresponding to them that get selected by the CAD software to form the SSP cores. This methodology of conversion of SG to VHCs and segmenting them into SSPs in some ways can be compared to the mapping technology of the standard-cell approach.

Standard-cell design technique uses parts of the design that have been created ahead of time or, possibly, used in other designs. The collection of cells is called cell library. Each cell has detailed specification of its characteristics (such as: schematic/diagram, description, logic area, delay, HDL code), which is used by a designer or a CAD software in system design.

Technology mapping performs three main steps on an SG: decomposition, matching, and covering. The decomposition step transforms the initial SG into several smaller trees which are further segmented [23]. Technology mapping may focus on number of dimensions of the design space, such as: cost, performance, area, power. Area also can be used as a simple measure of cell's costs which is optimized depending on the restriction requirement.

There are several significant differences between TPM approach and standard-cell approach:

1) The first major difference is the actual concept of temporal partitioning which is the driving factor in algorithm segmentation.

2) The standard-cell approach is dealing with spatial partitioning of logic resources. The selection of the cells from the library is based on the algorithm which is described as the low level synthesis form of the logic operations. The logic operators are grouped and transformed into the equivalent operations that exist in the cell library [80; 71]. On the contrary, the proposed approach is oriented towards high-level synthesis of SSP architecture where SSP is created from large macro operators (MOs). These MOs are later associated with the VHC implementations of MOs.

3) VHC selection is based on several factors associated and optimized for a particular segment and its performance parametric restrictions. Standard-cell approach is oriented towards the fine grain implementation. In spatial partitioning, the design is restricted to the size of FPGA, unlike the temporal partitioning where the size of the FPGA is not a hard restriction, and is mostly constrained by the timing requirements.

Together with many approaches oriented towards utilization of pre-compiled components collected in libraries, the standard cell approach seems to be similar to the proposed methodology of SSP synthesis in a way of automation of algorithm segmentation and IP-core generation. However, the type of segmentation is conceptually different and addresses different aspects of automated design, hence the two approaches cannot be objectively compared.

Appendix E

Proposed Reconfigurable Device Architectures

Based on the conducted research work one of the greatest influences on system performance is the bitstream configuration bandwidth. This is important both for single and multiple FPGAs system designs. An increase in configuration interface bandwidth allows to use of smaller granularity of task divisions in multiple FPGAs system designs, and for smaller downtime in single FPGA system designs. There are several design approaches that significantly impact the overall configuration bandwidth. This section will cover them briefly, and describe pros and cons of these approaches.

E.1 Wide Configuration Bus Architecture Operating at High Configuration Clock Speed

As was described in previous chapters, currently the maximum bitstream configuration bandwidth is limited to 3.2Gbit/sec. In addition, this bandwidth is only available in Xilinx Virtex 4, 5, and 6 families. None of the other FPGA vendors provide such high configuration speeds.

Since bandwidth is linearly proportional to the bus width and clock speed, doubling either of these parameters will double the configuration bandwidth. Therefore, most designs of the future FPGAs that are aimed to support temporal partitioning should strive to increase the number of configuration interface pins and increase configuration frequency.

In addition, DDR configuration memory should be explored as a possible alternative to SRAM. Latest DDR3-1600 memory that operates at 200MHz can provide a bandwidth of up to 12800MB/sec = 102.4Gbit/sec. Due to its capacity and significantly lower price such memory could be much more cost-effective for TPM solutions. However, because control of DDR memory is typically challenging, implementing DDR controller directly in the FPGA configuration controller would be an ideal solution. This would provide a 30 times speedup over the currently existing solution and allow to configure FPGA device in orders of microseconds instead of milliseconds.

E.2 Internal Configuration Buffer with One Clock Cycle Upload.

Another proposal for configuration SRAM architecture is to create an internal pre-fetch buffer. Pre-fetch buffer represents a pre-fetch SRAM cell which is placed right before the configuration cell. The idea behind a pre-fetch SRAM cell is to be able to update its contents using same address bus lines without drastically changing/complicating the FPGA routing. The only other component present is a pass transistor or tri-state buffer. This transistor or tri-state buffer is connected to a global enable line organized in a form of a clock tree. A global enable line allows to enable all of the pass transistors or tri-state buffers to transfer values of the pre-fetch SRAM cell to the configuration SRAM cell. The purpose of having a global enable line in a clock tree-like fashion is to have a close to one cycle simultaneous switch to a pre-fetched

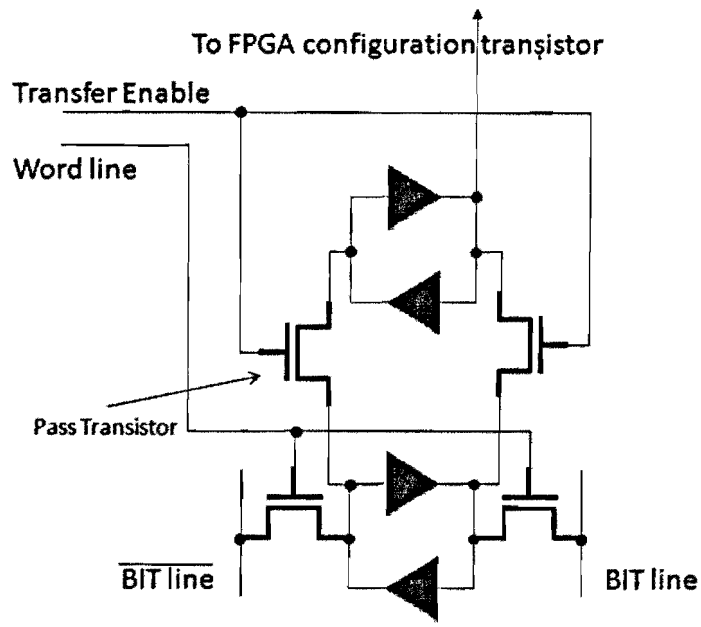


Figure E.1: Configuration SRAM Cell with Pre-fetch

configuration. The illustration of the sample schematic of an SRAM cell with pre-fetch is shown in Figure E.1.

The key difference of this architecture in comparison to the multi-context FPGAs mentioned in Section 3.2.1, is the actual switch between the configurations. In multi-context FPGAs input of the configuration transistor is switched from one SRAM cell to the other. In the configuration SRAM cell with a pre-fetch however, by enabling *Transfer Enable* line, the contents of the pre-fetch SRAM cell are transferred to the configuration SRAM cell. Transferring of the pre-fetched value to the configuration SRAM overwrites the configuration value, where in the context switch FPGAs configuration SRAM cells are multiplexed. Advantage of this architecture in comparison to the multiplexed one is the amount of logic and routing. In this configuration there is an increase in transistors, but only by an additional pre-fetch SRAM cell. In multiplexed configuration of an SRAM cell and a multiplexer are required. Considering only two configuration contexts are used in multi-context FPGA transistor for pre-fetch drops to 12 transistors in comparison of 16 for the multiplexed. Most importantly, the multi-context FPGA requires much more routing resources for the additional addressing

and data lines to the second SRAM cell. The pre-fetch configuration, on the other hand, does not require any additional addressing and data lines. Since pre-fetch SRAM cell is located on the same address/data lines, its own value is updated over the address and data lines of the configuration SRAM. This gives two modes of operation: the straight through and pre-fetched. In straight through, configuration SRAM obtains a value directly during the configuration cycle, like a traditional FPGA. For that a pass transistor is turned on and both pre-fetch SRAM and configuration SRAM get the same value. During the pre-fetched mode, which configuration SRAM cell is not affected since the pass transistor is turned off, and the pre-fetch SRAM cell gets a value update over the common address and data lines. After completion of the pre-fetch SRAM load, data can be transferred to configuration SRAM at any point simultaneously by setting the *Transfer Enable* line.

The overall number of transistors increases from 6 to 12 per configuration SRAM cell. However, since the new pre-fetched SRAM cell does not require any other control connections, its implementation on silicon does not complicate the FPGA design. This approach allows an FPGA device to continue operation while the next SSP bitstream is preloaded to the pre-fetch SRAM. At the same time, no addition of the address lines are needed and the only architectural modification required is the modification of the configuration SRAM to a pre-fetched configuration SRAM cell, as shown in the Figure E.1. It also has to be mentioned that sizing of the pre-fetch SRAM cell is also required in order to be able to transfer a value to the configuration SRAM cell. In a case of two identically sized SRAM cells transferring value from one to the other will not be possible. Therefore, four transistors in the pre-fetch SRAM will have to be 3-4 times larger in area, depending on the process technology.

There is also a slightly different approach, which builds on the pre-fetch and context switch idea to produce a more elegant solution. This approach is discussed in the next section.

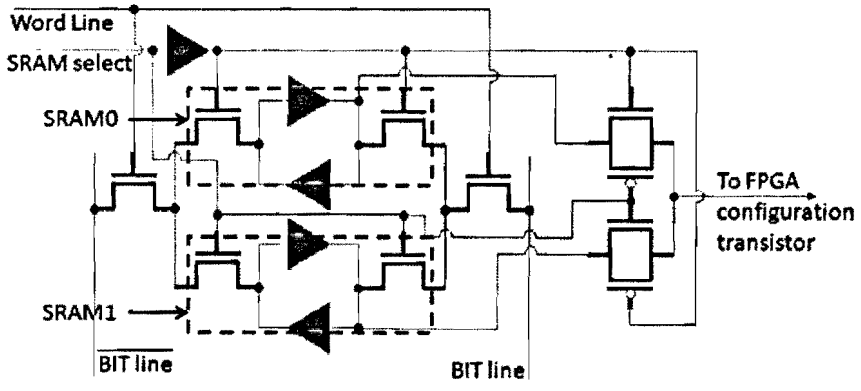


Figure E.2: Dual-context Configuration SRAM Cell Configuration

E.3 Dual Context FPGA with Address and Data Pin Reuse

Similar to the previous approach, this architecture does not require modification of the routing resources, and there are no additional address or data lines. The only additional SRAM selection line needed. It is organized similar to clock tree architecture, as mentioned in Section E.1. In this configuration both SRAM cells are connected to the same address and data lines. The difference is that there is an additional pass transistor which is activated in complement between the SRAM0 and SRAM1 cells. At the same time SRAM selection line sets the multiplexer to output either of the SRAM cells to the configuration transistor. Note that while one SRAM cell is enabled for writing the other is used as a configuration SRAM. This operation is mutually exclusive, therefore, when new configuration is written to the SRAM0 cell the SRAM1 is not disturbed, and vice versa. Schematic of this setup is provided in the Figure E.2.

There are several advantages of this configuration compound to the one presented in the Figure E.1. Even though this configuration requires 20 transistors per single configuration memory cell in comparison to 12 in the previous proposal, it actually might take less area depending on the technology used. Transistors in previous configuration pre-fetch SRAM cell have to be four times larger in area. This is necessary to be able to change the value of the configuration SRAM cell. Therefore, considering that 4 transistors from the previous configuration require four times the area, the overall area would roughly be equivalent to

$(16-4)+4\times4=24$ transistors. However, both approaches achieve the same objective of minimizing the impact on routing by addition of only single *Enable/Select* line, while providing capability of upload of a new SSP without interruption of device operation. These two approaches remove the need for the second FPGA (e.g. Figure 3.7) and signal switching peripherals. Therefore, if included in the architecture design, it would simplify the overall design of the temporal partitioning systems. This approach allows to hide the configuration overhead, as was discussed in Section 3.3.8.

E.4 Isolated Multi-Core FPGA Design with Common I/O Interface

There are FPGA devices that are capable of partial re-configuration, as was mentioned in previous chapters. The partially configurable FPGAs allow to reconfigure a frame or a tile of an FPGA device, without interrupting the operation of the remaining cores. However, support for the partial design currently lacking and there are tight restrictions on development of partial cores. In case of temporal partitioning, division of an FPGA even in half would allow for the configuration of one half of the FPGA while the second one is processing. For the multi-core FPGA two identical cores are included in a single device and multiplexed to the same I/O pins, as shown in Figure E.3. Their configuration is performed independently and development of the IP-core can be done without any dependence on the other core. By including even more cores it is possible to achieve a much more flexible architecture of interconnected independent FPGA cores. This, however, increases the overall size of the FPGA device and would only be applicable for a specific type of application.

Therefore, by including support for either a simple division of the FPGA in half, or multi-core FPGA implementation temporal partitioning mechanism will be well supported.

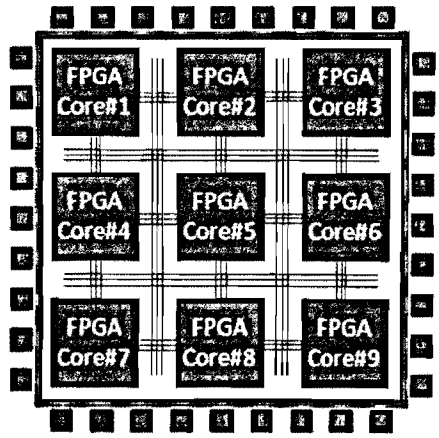


Figure E.3: Multi-core FPGA

E.5 Proposed Remote SSP Generation on a Compilation "PC Farm"

A task of SSP synthesis of a set of VHCs, into a final configuration bitstream requires a substantial amount of time. The reason for that is the large amount of logic on recent of FPGAs and the number of iterations taken by the development CAD tool for place and route of a given circuit. To address this issue, there is a possibility to create a dedicated "PC Farm", which would perform the actual task of bitstream generation. There are several advantages of such approach. First of all, "PC Farm" can be specifically tuned for the task of running synthesis, place & route. This can be achieved by dedicating a series of machines with multi-core processors and adequate amount of RAM. A user would be able to send a request from a remote workstation over a web interface. "PC Farm" would respond with the current status of the request, and sets of generated bitstream files.

The other advantage of such system is that it allows the true reuse of SSPs. Since all of the previously generated SSP bitstream instances are stored on the "PC Farm" servers, these SSPs can be reused. If a particular combination of VHCs is requested and if it already exists as a compiled SSP bitstream on the server, user will get an advantage of an immediate reply with a set of bitstreams. This aspect can tremendously increase the productivity and cut down

development time. In off-peak hours common types of requested combinations of VHCs can be compiled and stored for possible future use. Overtime, as more users generate various SSPs, this system would become even more efficient.

Regarding the storage space, currently, as a single hard drive can exceed 2TB in size, it is possible to store about 500000 bitstreams of the largest FPGA device on one drive. Therefore, a single storage rack of 16 hard drives can store about 8 million bitstream SSP cores. Various data management techniques can be employed for the previously compiled bitstreams to optimize the space used. This aspect is not discussed here, since this is not the focus of this section.

Another advantage of PC Farm is that the synthesis and place & route software is not run on the user side, therefore, user can continue development without interruption of bitstream compilation. Also, "PC Farm" development tools are always up-to-date, optimized and maintained. Hence, the user will not be concerned with such routine tasks. Last but not least, user's PC only requires to be able to run SPC GUI, and therefore, can be significantly inexpensive. Such approach would benefit users in accelerating development by performing compilations off-site and by reuse of SSP bitstreams. In addition, "PC Farm" will guarantee up-to-date software and minimal resource requirement on user's side.

E.6 Summary of Proposed Architectures

In this section several architectures were proposed and described including: increasing bitwidth of the configuration bus along with the clock speed; utilizing a configuration pre-load buffer with a single cycle upload. FPGA vendors can increase the speed of configuration by employing any of these proposed approaches. At the same time, the combination of several of these approaches at various degree of complexity can increase the bandwidth further. By the use of these approaches temporal partitioning systems would become easy to integrate into

designs, thus providing more cost-effective and adaptive solutions for ever more increasing processing demands.