Runtime reconfigurable platform for embedded systems

Krishnamoorthy Baskaran

2005

Krishnamoorthy Baskaran. (2005). Runtime reconfigurable platform for embedded systems. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2336>

<https://doi.org/10.32657/10356/2336>

Nanyang Technological University

Downloaded on 26 Apr 2023 21:54:49 SGT

Runtime Reconfigurable Platform for Embedded Systems

Krishnamoorthy Baskaran

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2005

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Dr.T.Srikanthan for giving me the opportunity to do graduate studies under his supervision and for his support and encouragement. If not for the numerous stimulating discussions we had in spite of his busy schedule, this project would not have been possible.

I would like to take this opportunity to also thank Dr. Wu Jigang for his suggestions and helping me interpret the initial results, and Rosario Jagadeesh for reading part of this report and helping me with his suggestions.

I would like to thank Ms. Nah Kiat Joo and Ms. Merilyn Yap for providing required support and help.

I would like to take this opportunity to thank all my friends and my family members for their help and encouragement.

Abstract

Reconfigurable computing combines the benefits of both software and reconfigurable hardware implementations. The use of FPGA in reconfigurable computing is becoming increasingly prevalent due to its ever-increasing density and rapid turn around time. There is a growing need to maximize the utilization of FPGA resources with the help of a framework for runtime reconfigurable platforms in an attempt to reduce cost and power consumption. Another development branch leads to dynamically reconfigurable FPGAs, which allows certain areas within the FPGA to be reconfigured while the rest continues to run unaffected.

In this research work, we have examined research issues and technology approaches for dynamic reconfigurable platforms based on an in-depth survey of previous research work. We have proposed a number of optimal algorithms for both initial and run-time partitioning of the FPGA, scheduling and placement of tasks in the reconfigurable hardware. We show that the proposed partitioning techniques are superior to existing ones reported in the literature. The effectiveness of these techniques was evaluated using the first come first serve (FCFS) policy for implementing both the scheduling and placement operations. Our investigations show that a reduction of up to 17% in terms of execution time, up to 20% in response time, up to 30% in configuration delay, and up to 24% in configuration power is achievable as a direct result of incorporating the partitioning schemes. Two other non-preemptive algorithms, namely shortest job first (SJF) and priority based scheduling are also implemented and fully evaluated to

demonstrate the effectiveness of the initial and run-time partitioning schemes for these scheduling and placement techniques.

Techniques have been developed to determine the salient parameters of the reconfigurable space for application specific implementations. The proposed application-specific initial and run-time partitioning as well as scheduling strategy result in a further performance improvement of up to 34% and 20% in terms of execution and response time respectively. Finally, a framework has been proposed to incorporate the proposed schemes into a reconfigurable platform. The report ends with future work that must be carried out to implement a mature reconfigurable platform for embedded systems.

CONTENTS

ACKNOWLEDGEMENT	i
ABSTRACT	ii
LIST OF FIGURES	vii
LIST OF TABLES	ix
1.INTRODUCTION	
1.1 Background	1
1.2 A Brief Introduction To Reconfigurable Computing	1
1.3 Motivation	3
1.4 Research Objectives	4
1.5 Report Organization	5
2. LITERATURE SURVEY	
2.1 Introduction	7
2.2 Reconfigurable Computing	7
2.3 Field Programmable Gate Arrays(FPGAs)	8
2.4 FPGA Configuration	14
2.4.1 Reconfiguration Models	16
2.4.1.1 Single Context FPGAs	16
2.4.1.2 Multi-Context FPGAs	17
2.4.1.3 Partial RunTime Reconfigurable FPGAs	18
2.5 Reconfigurability Methods	19
2.5.1 Static Reconfiguration	19
2.5.2 Run-Time Reconfiguration	20
2.5.2.1 Algorithmic Reconfiguration	21
2.5.2.2 Architecturd Reconfiguration	21
2.5.2.3 Functional Reconfiguration	21
2.6 Survey of Reconfigurable Platforms	23
2.6.1. University Research Projects	24
2.7 Embedded Real Time Operating System	25
2.8 Hardware Operating System for Reconfigurable Computing	26
2.9 Summary	30
3. RECONFIGURABLE PLATFORM: ISSUES AND APPROACHES	
3.1 Reconfigurable Platform Design Issues	31
3.2 Hardware/Software Codesign Issues	33
3.3 Operating System Level Issues	34
3.3.1 Partitioning Issues	35
3.3.2 Scheduling Issues	37
3.3.3 Placement Issues	38
3.3.4 Communication Issues	39

3.4 summary	39
4 PARTITIONING ALGORITHM FOR FPGAS	
4.1 Proposed Partitioning Techniques	41
4.1.1 Initial Partitioning Algorithm	42
4.2 Run-time Partitioning Algorithm	45
4.3 Experimental Results	49
4.3.1 Simulation-Environment	49
4.3.2 Performance of Partitioning Algorithm	51
4.3.3 Effect of Runtime Partitioning on Initial Partitioning	55
4.3.4 Partitioning Algorithm Performance for Different Device Widths	57
4.4 summary	61
5.SCHEDULING AND PLACEMENT METHODOLOGY	
5.1 Task Scheduling in Reconfigurable Platform	62
5.1.1 Scheduling Policies	63
5.2 A Unified Framework for Reconfigurable Platform	64
5.2.1 Scheduling Scheme	65
5.2.2 Placement Strategy	65
5.3 Experimental Evaluation	69
5.3.1 First Come First Serve Scheduling Policy	69
5.3.2 Shortest Job First Scheduling Policy	70
5.3.3 Priority Based Scheduling Policy	71
5.3.4 Comparing single queue and multiple queue placement techniques	72
5.4 summary	73
6.APPLICATION-SPECIFIC PARTITIONING AND SCHEDULING	
6.1 Problem Formulation	75
6.2 Proposed Algorithm for Application Specific Partitioning	76
Calculating Distinct Blocks	78
Path-based Hardware Estimation Technique	79
Partitioning the Estimated Hardware	82
6.3 Application-Specific Scheduling and Placement	82
6.4 Performance of the Application Specific Partitioning and Scheduling Algorithm	84
6.5 Summary	86
7. RUNTIME RECONFIGURABLE PLATFORM	
7.1 Proposed Architecture for Runtime Reconfigurable Platform	88
Reconfigurable System-on-Chip (RCSoc)	88
Soft Core CPU	89
7.2 Hardware OPERating System [HOPES]	90
Partitioning Algorithm	91
Scheduling and Placement Methodology	91

Development Environment	92
7.3 summary	92
8.CONCLUSION AND FUTURE WORK	93
8.1 Conclusion	93
8.2 Future Work	95
REFERENCES	96

LIST OF FIGURES

Figure 1.1:	The trade-off between flexibility and performance	2
Figure 2.1:	Software and hardware based computing approaches	8
Figure 2.2:	Class of FPGA architectures [17]	10
Figure 2.3:	Programmable logic core model of an FPGA	13
Figure 2.4:	Simplified structure of Xilinx Virtex CLB	14
Figure 2.5:	Bitstream example	15
Figure 2.6:	Virtex configuration column example	16
Figure 2.7:	The structure of single context FPGA	17
Figure 2.8:	(a) The conceptual view of a four-context FPGA	18
	(b) The configuration memory structure of a four-context FPGA	18
Figure 2.9:	The structure of a partial runtime reconfigurable FPGA	19
Figure 2.10:	Global runtime reconfiguration	22
Figure 2.11:	Local runtime reconfiguration	23
Figure 2.12:	General basic architecture of a reconfigurable platform	23
Figure 3.1:	Reconfigurable platform design flow	32
Figure 3.2:	Hardware/Software co-design architecture	34
Figure 3.3:	Hardware operating system architecture	35
Figure 3.4:	Partitioned FPGA	36
Figure 3.5:	Executing TFG on a partitioned reconfigurable hardware	37
Figure 3.6:	Task placements in the partitioned FPGA	39
Figure 4.1:	Task width distribution	42
Figure 4.2:	Reconfigurable device partitioning	42
Figure 4.3:	Initial partitioning algorithm	44
Figure 4.4:	Example of initial partitioning algorithm	45
Figure 4.5:	Runtime partitioning of the FPGA	47
Figure 4.6:	Runtime partitioning algorithm	48
Figure 4.7:	Total execution time	51
Figure 4.8:	Average response time	52
Figure 4.9:	Average task configuration time	52
Figure 4.10:	Total task configuration power	53
Figure 4.11:	Total area utilization	54
Figure 4.12:	fragmentation	54
Figure 4.13:	Average response time with and without runtime partitioning	55
Figure 4.14:	Average task configuration time with and without runtime partitioning	55
Figure 4.15:	Total task configuration power with and without runtime partitioning	56
Figure 4.16:	Total area utilization with and without runtime partitioning	57
Figure 4.17:	Fragmentation with and without runtime partitioning	57
Figure 4.18:	Total Execution time for different device widths	58
Figure 4.19:	Average task configuration time for different device widths	59
Figure 4.20:	Total task configuration power for different device widths	59

Figure 4.21:	Average task response time for different device widths	60
Figure 4.22:	Total area utilization for different device widths	60
Figure 4.23:	Fragmentation for different device widths	61
Figure 5.1:	Preemptive task scheduling	63
Figure 5.2:	Non-preemptive task scheduling	63
Figure 5.3:	Hardware/ Software task computations	65
Figure 5.4:	Structure of single-queue scheduler and placer	66
Figure 5.5:	Structure of multiple-queue scheduler and placer	67
Figure 5.6:	High-level view of reconfigurable platform scheduling and placement	68
Figure 5.7:	The reconfigurable platform scheduling and placement algorithm	68
Figure 5.8:	Total execution time for FCFS	69
Figure 5.9:	Total configuration Time for FCFS	69
Figure 5.10:	Total area utilization for FCFS	70
Figure 5.11:	Total configuration power for FCFS	70
Figure 5.12:	Total execution time for SJF	70
Figure 5.13:	Total configuration Time for SJF	70
Figure 5.14:	Total configuration power for SJF	71
Figure 5.15:	Total area utilization for SJF	71
Figure 5.16:	Total execution time for PB	72
Figure 5.17:	Total configuration Time for PB	72
Figure 5.18:	Total configuration power for PB	72
Figure 5.19:	Total area utilization for PB	72
Figure 5.20:	Total execution time for FCFS	73
Figure 5.21:	Total configuration Time for FCFS	73
Figure 5.22:	Total configuration power for FCFS	73
Figure 5.23:	Total area utilization for FCFS	73
Figure 6.1:	Flow graph of an application	76
Figure 6.2:	General outline for application-specific Partitioning [Design time]	77
Figure 6.3:	Calculating distinct blocks	79
Figure 6.4:	Path-based hardware estimation technique	80
Figure 6.5:	Example of DAG for path-based hardware estimation	81
Figure 6.6:	Hardware requirement for paths in the DAG	81
Figure 6.7:	Flow of the application-specific scheduling and placement methodology	83
Figure 6.8:	Total execution time with FCFS scheduling	84
Figure 6.9:	Average response time with FCFS scheduling	85
Figure 6.10:	Total execution time with SJF scheduling	85
Figure 6.11:	Average response time with SJF scheduling	85
Figure 6.12:	Total execution time with PB scheduling	86
Figure 6.13:	Average response time with PB scheduling	86
Figure 7.1:	Reconfigurable computing target architecture	89
Figure 7.2:	Components of Hardware OPERating System	90

LIST OF TABLES

Table 2.1:	FPGA programming technologies [17]	12
Table 2.2:	Selected commercial FPGAs of different types	14
Table 2.3:	Configuration column types	15
Table 3.1:	Area requirements for FPGA cores	36
Table 4.1:	Task set information	50
Table 4.2:	Average task configuration time	56
Table 4.3:	Total task configuration power	56
Table 4.4:	Total area utilization	56
Table 4.5:	Devices and task set information	58

Chapter 1

Introduction

1.1 Background

System performance is one of the most important considerations for computer system designers. Until recent years, system designers had only two options in implementing a computing system, which largely varied in terms of flexibility and performance. The first choice was to develop special purpose hardware tailored to meet the needs of a specific task. However, as far as dedicated optimized circuits are concerned, the system performance increases, but even minor changes to the system would require redesign. As a result, flexibility of changing any parameter of the system is lost. On the other hand, we have general-purpose systems that are built to accommodate the problem of flexibility, but system performance is compromised because of the lack of dedicated hardware resource designed for specific problems. This led to the introduction of a new computing model called reconfigurable computing.

Reconfigurable computing systems have become an alternative to fill the gap between application specific integrated circuits (ASICs) and general-purpose computing systems. Reconfigurable Computing is a research area that promises to make the most of this idea. The usage of reconfigurable computing is continuously growing in the areas like wearable computing [1], mobile communication system [2], video communications [3], neural computing [4], and network processors [5].

1.2 A Brief Introduction To Reconfigurable Computing

The reconfigurable platform based design makes use of novel technologies for improving performance, decreasing cost, increasing energy-efficiency, and reducing the time-to-market. The idea behind reconfigurable computing is to have hardware that can be

configured in real time, to satisfy the computational requirements of the target application. Various approaches have been attempted and proven effective in the research community. Reconfigurable Computing can provide the high performance of ASICs with the flexibility of programmable processors. Figure 1.1 [6] shows the trade-off between flexibility and performance. Reconfigurable computing lies somewhere in the performance spectrum that spans between general purpose processor and ASICs. Much of the activity surrounding reconfigurable computing has centered on runtime reconfiguration of Field-Programmable Gate Arrays (FPGAs).

Figure 1.1: The trade-off between flexibility and performance

One of the methods that have proved largely successful in delivering good performance is the use of Field Programmable Gate Arrays (FPGAs) as the reconfigurable hardware. Field Programmable Gate Arrays (FPGAs) offer an attractive alternative to ASICs or general-purpose computing systems. FPGAs are flexible, fast, and cheap. FPGAs can be reprogrammed without a new fabrication run. This flexibility comes with a price: FPGAs are generally not as fast as ASICs. However, an FPGA's inherent parallelism can give significant performance gains when compared to a general-purpose computing system. Additionally, FPGAs are mass-produced, making them a medium cost option.

FPGAs have been used in ASIC design areas for prototype testing, and quick synthesis of datapaths for quite some time. With the advent of FPGA devices, which can be dynamically and partially reconfigured, such as the Xilinx [7] Virtex series, and the Atmel [8] AT4OK series; they have been identified as a suitable building block for Reconfigurable Computers. Partial reconfigurability [9] of an FPGA means that parts of the device can be reconfigured, while retaining the configuration information in the rest of the device. Dynamic reconfiguration means that parts of the device can be reconfigured, even as the other parts are continuing with the computation that they are configured to perform.

In addition to reconfigurable hardware, reconfigurable platforms also typically include a general-purpose processor, which execute some portions of the application in software and also runs an operating-system-like program, which manages the resources in the system. A runtime reconfigurable platform system would be able to decide – on the fly – what hardware task to execute on the FPGA area and what software task to run on the CPU.

1.3 Motivation

Quick design turn around time and flexibility are the most attractive features of a reconfigurable computing platform. In recent years, reconfigurable computing systems typically consist of a large variety of different processing elements, memories, I/O devices, sensors, and actuators. The processing elements are split into softcore CPUs, fixed-function hardware (ASICs), and reconfigurable hardware. The above mentioned system requirements have paved the way for the System-on-Chip (SoC) concept. In SoC, the entire computing system is designed to exist on a single silicon chip. The single chip solution is a vital motivation for developing a new framework for runtime reconfigurable embedded systems.

As the overall power consumption and cost of FPGAs increase, the issue of the management of these hardware resources and their sharing among many applications and

users will become more of a concern. An efficient operating system that is tailored towards reconfigurable platform can provide many benefits such as increased design productivity, portability, and resource utilization. There is a strong need for identifying a fundamental set of services that must be provided by an operating system for reconfigurable computing, which is a new line of research. The issues associated with each such service need to be studied in detail.

1.4 Research Objectives

The management of the reconfigurable hardware resources in the reconfigurable platform is crucial for achieving optimal performance. Therefore it is desirable that the reconfigurable platform has an operating system that is capable of efficiently managing the hardware resources. The vital services that should be provided by the operating system are: partitioning the reconfigurable area into multiple blocks, scheduling both the hardware and software tasks in an optimal manner, the placement of the hardware tasks on the partitioned blocks and ensuring proper communication and co-ordination between the tasks, both hardware and software.

One of the important considerations of the runtime reconfigurable platform is partitioning of the reconfigurable hardware resource. The most common approach is to partition the FPGA into fixed-sized blocks [67]. However, this approach may lead to under utilization of the reconfigurable area and an increase in the amount of time required to configure the various tasks. This is because the characteristics of the hardware tasks, such as their minimum and maximum sizes, are not taken into consideration while partitioning the FPGA. Power is also an important consideration. Therefore it is likely that the development of a partitioning methodology based on the characteristics of the task set and the size of the reconfigurable device can lead to increased area utilization and reduction in configuration time. Hence, one of the objectives of this research is to develop an efficient partitioning strategy for reconfigurable computing.

The scheduling and placement methodologies should be capable of handling hardware tasks in addition to managing the software tasks that are executed in the processor. The scheduling scheme of the hardware tasks has to be flexible. That is, it should be able to support more than one scheduling policy depending on the user application requirements. The impact of various scheduling techniques on the execution time of the application, the average response time, the configuration time and power has to be studied. The placement techniques of a hardware task on the reconfigurable surface involves selecting the optimal block among the available free blocks and configuring the block with the logic circuit that represents the hardware task. The tradeoffs involved in the selection of a suitable block in terms of configuration time and the effect on the scheduling and placement process need to be examined.

1.5 Report Organization

This report is divided into the following eight chapters.

- i) **Introduction:** This chapter presents the background on the research topic. It also states the objective and scope of the project along with a brief introduction to reconfigurable computing.
- ii) **Literature Survey:** This chapter highlights the established research so far in the field of **FPGA** technologies, reconfigurable Computing, configurable System-on-Chip (CSoC), real-time/hardware operating system functionality like partitioning, placement, and scheduling, and reconfigurable architecture for the embedded systems.
- iii) **Reconfigurable Platform: Issues and Approaches:** This chapter discusses the issues and approaches for the runtime reconfigurable platform and the requirements of the various components for the deployment of the hardware operating system are studied.
- iv) **Partitioning Algorithms for FPGAs:** The initial and runtime partitioning algorithm for the runtime reconfigurable platform is presented in this chapter. The simulation results of the proposed algorithms are also presented.
- v) **Scheduling and Placement Methodology:** This chapter presents different scheduling schemes and placement methods for the reconfigurable platform along with the experimental results.

- vi) **Application-Specific Partitioning and Scheduling:** This chapter describes the application-specific hardware requirements, partitioning, and scheduling techniques along with the simulation results.
- vii) **Runtime Reconfigurable Platform:** A new architecture for runtime reconfigurable platforms is proposed in this chapter.
- viii) **Conclusion and Future work** This chapter summarises and concludes the thesis and offers recommendations for future work.

This research work resulted in the following publications:

- K.Baskaran and T.Srikanthan, “*Issues and Approaches to Runtime Reconfigurable Computing Platform for Embedded Systems*” IADIS International Conference Applied Computing March 2004 LISBON, PORTUGAL. (Accepted).
- K.Baskaran and T.Srikanthan, “*AHardware Operating System based Approach for Run-time Reconfigurable Platform of Embedded Devices*”, 6* Real-Time Linux Workshop, pp 111-116, November 3-5 2004, Singapore.
- K.Baskaran, Wu Jigang and T.Srikanthan, “*Hardware Partitioning Algorithm for Reconfigurable Operating System in Embedded Systems*”, 6* Real-Time Linux Workshop, pp 117-123, November 3-5 2004, Singapore.
- K.Baskaran and T.Srikanthan “*Hardware OPERating Systems (HOPES) for Runtime Reconfigurable Platform for Embedded devices*”, IEEE International Workshop on Computer Architecture for Machine Perception, July 4-6 2005, Palermo, Italy (Accepted).
- K.Baskaran, Wu Jigang and T.Srikanthan, “*Application-Spec & Partitioning Algorithm for Runtime Reconfigurable Platform for Embedded Devices*”, Engineering of Reconfigurable Systems and Algorithms, June 27-30 2005, Las Vegas USA (Accepted).

Chapter 2

Literature Survey

2.1 Introduction

In order to create a runtime reconfigurable platform for embedded systems, there are many areas of research which must be considered. These include FPGA technologies, reconfigurable computing, real-time hardware operating system functionality like partitioning, placement, and scheduling, and reconfigurable architectures for embedded systems. This chapter discusses previous work in the areas related to our research.

2.2 Reconfigurable Computing

In recent years, reconfigurable computing has emerged as an area of active research. Reconfigurable computing is a novel approach that combines the strengths of software and hardware based computing approaches. Reconfigurable computing systems combine a processor with dynamically reconfigurable hardware. The main goal is to take advantage of the capabilities and feature of both resources. While the processor takes care of all the general-purpose computation, the reconfigurable hardware acts as hardware logic execution platform.

The following example shows the difference between hardware and software implementations through the evaluation of the function $y = ax^2 + by + c$. Figure 2.1(a) shows a software programme for this function, sequencing several basic operations through the processor until the function is evaluated. Figure 2.1(b) illustrates the same operation for a hardware-based design, with dedicated computation units providing an answer in a single iteration [12].

Figure 2.1: Software and hardware based computing approaches

Reconfigurable computing is intended to fill the gap between software and hardware. It achieves higher performance than software and maintains a higher level of flexibility than hardware. Most reconfigurable computing systems have been implemented on field programmable gate arrays (FPGAs) [10]. The performance improvement of FPGAs over CPUs can be significant, but approximately three times slower than Application Specific Integrated Circuits (ASICs). In [11, 12], Dehon provides both analytical and empirical comparisons of configurable and general-purpose computing. Dehon's analysis shows that FPGAs offer a much higher computational density per unit area compared to the general-purpose CPUs. The benefit of this computational feature is that the hardware may be reused by different applications. The next section describes the FPGA technology.

2.3 Field Programmable Gate Arrays (FPGAs)

The technology of FPGAs has made possible the new form of computing known as reconfigurable computing. The FPGA is a very large-scale integrated circuit (VLSI) that can be quickly programmed, i.e. configured, in order to implement digital systems consisting of millions of gates. It possesses great advantages over the factory fabricated

ASIC in

- (i) A short product turn-around time
- (ii) Low development cost.

Since the introduction of the FPGA about 20 years ago, devices and markets have greatly grown. Due to the increased complexity, performance, features and lower costs, programmable logic devices have evolved from ‘glue logic’ to a board level integration role. Today’s FPGAs, shipped by the two leading commercial programmable logic device companies Xilinx and Altera, provide more than million system gates composed of memory and datapath elements and programmable logic gates. For example, Xilinx XC2V8000 has 8 million system gates [13] while Altera EP20K1500E has 2 and 4 million system gates [14]. Future FPGAs can be expected to be much larger: As with many other integrated circuits, the number of elements on a single FPGA has doubled roughly every 18 months [15].

Exploiting the maximum parallelism of the target application, FPGA based systems often attain the performance impossible to the ordinary stored-program computers, which suffer from the von Neumann bottleneck. When run on FPGAs, the data intensive tasks such as the image processing, pattern matching, and circuit simulation tend to bring excellent outcomes. A case in point is the Splash machine achieving approximately 200 times better performance compared to supercomputer CRAY-I1 in edit distance calculation [16]. Today, the scope of applications is still being broadened. Different systems utilizing FPGAs have been proposed, but the concept is collectively addressed as reconfigurable computing. The variety includes stand-alone computation engines, custom compute accelerators, and rapid prototyping systems.

The FPGA is a type of programmable logic that can be configured for implementing a wide variety of applications. Reconfigurability is an important property of most programmable logic devices. FPGA devices cannot compete with custom ASIC hardware implementation in terms of density or speed, but their reconfigurability allows hardware

designs to be created and changed rapidly, thus reducing time-to-market and costs over custom hardware.

Commercially available FPGAs can be categorized into four main types: symmetrical array, row-based, hierarchical PLD, and sea-of-gates. They all differ in the interconnections and the arrangement of the logic blocks used. Figure 2.2 [17] shows the different architectures. The symmetrical array architecture consists of an array of programmable logic blocks with vertical and horizontal programming routing channels. The row-based architecture has logic blocks arranged in rows with horizontal routing channel between successive rows. Hierarchical architecture imply a hierarchy in the placement and routing of the connections between the logic blocks. The sea-of-gates architecture consists of fine-grain logic blocks covering the entire floor of the device and the interconnection is using a neighbor-to-neighbor routing technique.

Figure 2.2: Class of FPGA architectures [17]

Currently three technologies are in used for programming FPGAs. They are static RAM cells, anti-fuse, and EPROMEEPROM transistors. Depending upon the user

requirement, one FPGA technology may have more desirable features for that application.

Anti-fused FPGAs are only one-time programmable [10,74,75]. An anti-fuse works in the opposite way of a usual fuse. An anti-fuse resides in a high-impedance state and can be programmed into low impedance or "fused" state. This technology can be used to make one-time programmable devices that are less expensive than the SRAM technology. Faster clock rate and smaller chip area are also achieved in comparison to the other two types of FPGAs manufactured with a similar process technology.

Static RAM-based FPGA programmable [10,74,75] connections are made using pass-transistors, transmission gates, or multiplexers that are controlled by SRAM cells. This technology allows fast in-circuit reconfiguration. The major disadvantage is the size of the chip required by the RAM technology and that the chip configuration needs to be loaded to the chip from some external source (usually external non-volatile memory chip). The FPGA can either actively read its configuration data out of external serial or byte-parallel PROM (master mode), or the configuration data can be written into the FPGA (slave and peripheral mode). The FPGA can be programmed an unlimited number of times.

EPROM/EEPROM technology [10,74,75] is the same as that used in EPROM memory devices. In this technology, the programmable switch is a transistor that can be permanently disabled by injecting a charge on the gate. In the EPROM transistor's case, the charge is removed by exposing the gate to ultra-violet (UV) light. The programming is achieved without external storage of configuration. EPROM based programmable chips cannot be re-programmed in-circuit and need to be cleared with UV erasing. EEPROM based devices, which are electrically erasable, occupy twice as much area as EPROM-based FPGAs. Though EEPROM-based FPGAs tend to be slower, they possess the convenience of the both anti-fused FPGAs and SRAM-based FPGAs. Table 2.1 shows some of the characteristics of the above programming technologies.

Technology	Volatile	Reprogrammable	Chip Area
Anti-Fuse	No	No	Small [anti-fuse] Large [programming transistors]
EPROM	No	Out of Circuit	Small
EEPROM	No	Out of Circuit	2 x EPROM
Static Ram	Yes	In-circuit	Large

Table 2.1: FPGA programming technologies [17]

Figure 2.3 shows the model of the programmable logic core of an FPGA. In this model, the programmable logic core of an FPGA consists of an array of three basic elements: configurable logic blocks (CLBs), connection boxes (CBs), and switch boxes (SBs). In SRAM-based FPGAs, a CLB contains several SRAM lookup tables (LUTs) to store user-defined logic functions. A CLB also contains flip-flops, multiplexers, and dedicated circuitry to optimize the performance of user applications. CLBs are connected through horizontal and vertical wiring channels between two neighboring rows or columns. To enhance the connectivity, there are various kinds of wires with different lengths for connecting CLBs that are multiple blocks apart. For example, in Xilinx Virtex-series FPGAs, single lines connect adjacent CLBs, while hex lines connect CLBs that are three or six blocks apart [7]. Figure 2.4 shows a typical CLB. There are two types of routing devices, CBs, and SBs, to direct the signal flows among CLBs and wiring channels. CBs serve as local bridge between CLBs and adjacent wiring channels. SBs are switch matrices that connect horizontal or vertical wiring channels. In SRAM-based FPGAs, the state of connections in these routing devices is controlled by SRAM cells, which are configured according to the desired functionality.

Figure 2.3: Programmable logic core model of an FPGA

In this architecture, a CLB column includes all CLBs and the corresponding switch matrices (CBs and SBs) in the same column of the array. The actual circuitry, programmable logic and routing resources, and the configuration architecture of each CLB column are identical. A typical example of FPGA architecture used in this work is the Xilinx Virtex-series FPGA. The basic building block of the Xilinx Virtex FPGA is the logic cell (LC). A LC includes a 4-input function generator, carry logic and a storage element. Each Virtex CLB contains four LCs, organized in two slices (Figure 2.4). The 4-input function generators are implemented as 4-input look-up tables (LUTs). Each of them can provide the functions of one 4-input LUT or a 16x1-bit synchronous RAM (called "distributed RAM"). Furthermore, two LUTs in a slice can be combined to create a 16x2-bit or 32-bit synchronous RAM, or 16x1-bit dual-port synchronous RAM [7]. Table 2.2 classifies the commercially available FPGAs based on their architecture, logic block type and programming technology.

Figure 2.4: Simplified structure of Xilinx Virtex CLB

Company	Architecture	Logic Block Type	Programming Technology
Actel	Row-Based	Multiplexer-Based	Anti-fuse
Altera	Hierarchical-PLD	PLD Block	EPROM
QuickLogic	Symmetrical Array	Multiplexer-Based	Anti-fuse
Xilinx	Symmetrical Array	Look-up Table	Static RAM

Table 2.2: Selected commercial FPGAs of different types

2.4 FPGA Configuration

FPGA configuration is the process of loading a design bitstream into the FPGA internal configuration memory [18]. Xilinx Virtex configuration logic is significantly different from that of the other Xilinx series, but maintains compatibility to all FPGA families. Xilinx Virtex devices can be configured through SelectMap interface, master/slave interfaces, or the boundary-scan interface. The collection of configuration bits is called a configuration bitstream. The bitstream is a series of configuration commands and configuration data, as shown in Figure 2.5. Writing some or all of a configuration is done by issuing configuration commands to the desired interface followed by the configuration data.

Figure 2.5: Bitstream example

The configuration data format of the Virtex family is independent of the configuration mode and follows a pattern: Imagine the Virtex family configuration memory as a rectangular array of bits. The bits are grouped in one-bit wide portions called frames. Thus, frames are the smallest unit that can be written (or read back). Frames are grouped together into a large unit called column. There exist five different types of columns as mentioned in Table 2.3.

Column Type	# of frames	# per Device
Center	8	1
CLB	48	# of CLB columns
IOB	54	2
Block SelectRAM Interconnect	27	# Block SelectRAM columns
Block SelectRAM content	64	# Block SelectRAM columns

Table 2.3: Configuration column types

As mentioned above, configuration data is separated into frames and the frames are grouped together into columns. Figure 2.6 shows the configuration columns for a Virtex device. There is one center column, which consists of 8 frames. This center column contains the configuration data for the four global clock buffers and their corresponding clock nets. Two IOB columns hold the configuration for the IOBs on the left and right side of the silicon die. The CLB columns contain the configuration for the CLB and the adjacent IOBs on the top and the bottom, respectively. The remaining two column types contain information about the Block SelectRAM: one contains interconnection data and the other contains the content-related parts. The number and size of frames varies with the device size. The total number of configuration bits for a particular device is calculated

by multiplying the number of frames by the number of bits per frame.

Figure 2.6: Virtex configuration column example

2.4.1 Reconfiguration Models

In the following, we describe three reconfiguration models based on the actual programming architecture of the FPGA, and examine how it affects the configuration process. The overheads associated with each of the models are analyzed.

- (i) Single Context FPGAs
- (ii) Multi-Context FPGAs
- (iii) Partial RunTime Reconfiguration FPGAs

2.4.1.1 Single Context FPGAs

In the single context FPGA [19,20] shown in Figure 2.7, the whole CLB array can be viewed as a shift register, and the whole chip area must be reconfigured during each reconfiguration. This means that even if only a small portion of the chip needs to be reconfigured, the whole chip is rewritten. Since many of the applications being developed

with FPGAs today involve runtime reconfiguration, the reconfiguration of the single context architecture incurs a significant overhead. Therefore, much of research has focused on the new generation of architectures or tools that can reduce this reconfiguration overhead.

Figure 2.7: The structure of single context FPGA

2.4.1.2 Multi-Context FPGAs

A multi-context device [21,22] has multiple layers of programming bits, where each layer can be active at a different point in time. An advantage of the multi-context FPGA over a single context architecture is that it allows for an extremely fast context switch (in the order of nanoseconds), whereas the single context FPGA may take milliseconds or more to reprogram. The multi-context design allows for background loading, permitting one context to be configured while another is executing. Each context of a multi-context device can be viewed as a separate single context device. A conceptual view of a four-context FPGA is shown in Figure 2.8. Any one of the four bits in locations 1,2,3 and 4 can be used as the configuration bit C depending on the context.

Figure 2-8: (a) The conceptual view of a four-context FPGA (b) The configuration memory structure of a four-context FPGA.

24.13 Partial Runtime Reconfiguration FPGAs

Partial Runtime Reconfiguration (PRTR) [23,24,25] is another typical reconfiguration model. By changing only a portion of the reconfigurable logic while other sections continue operating, the reconfiguration latency can be hidden from other computations. In recent years, many commercial devices (Xilinx Virtex series, Xilinx 6200 series) and systems (Garp, Chimaera [25], DISC) have applied the PRTR model. The structure of a PRTR FPGA is shown in Figure 2.9. In a reconfigurable FPGA, the fundamental programming bit layer operates like a RAM device. By sending configuration values with row and column addresses to the FPGA, a certain location can be configured. The various reconfigurability methods are presented in the next section.

Figure 2.9: The structure of a partial runtime reconfigurable FPGA

2.5 Reconfigurability Methods

We will distinguish between two types of reconfiguration mechanisms [26], depending on the use they make of the dynamic nature of the reconfigurable device. The first one is the most broadly used, especially because it's the one supported for all reconfigurable devices, and it is called Static Reconfiguration (SR), which is often referred as compile time reconfiguration, because the configuration of the device is decided at compile-time, and never replaced until the application has ended.

We will refer to the second method as Run-Time Reconfiguration (RTR), also called dynamic reconfiguration or on-line reconfiguration. Runtime reconfiguration allows the replacement of the contents of the programmable device during the execution of the application, in a predefined order if known at design time, or randomly depending on the execution of the application.

2.5.1 Static Reconfiguration

In most present FPGA-based implementations the configuration bitstream is downloaded once the execution of the system starts, and remains unchanged until the application has finished. Then it can be reloaded for the execution of a new application. This approach is

known as Static reconfiguration, because the functionality of the FPGA is fixed at compile-time. The design process doesn't vary from that of a custom-designed ASIC, but the FPGA can be reprogrammed for its reuse in any other application.

We could summarize the main characteristics of static reconfiguration into the following:

- The FPGA configuration bitstream is downloaded only once, during the set-up of the system, and remains unchanged during execution.
- The system must be stopped each time a new configuration is downloaded (here configuration is synonymous to application).
- There is no need of runtime reconfiguration features for this type of reconfigurability. Any kind of reprogrammable FPGA can be used in a SR system. The reconfiguration speed will only affect the set-up of the system, which is performed only once per application, so it isn't very critical. Static reconfiguration involves hardware change at a relatively slow rate.

2.5.2 Runtime Reconfiguration

The main advantage of RTR compared to SR is that it allows reusing the reconfigurable device several times for the same application. To be able to do that it is necessary to partition the application into a set of configurations, instead of using spatial exclusiveness (different blocks don't share the same area, but they use different hardware resources, which can run simultaneously) as a criterion, time exclusiveness is used. (Configurations are not inter-dependant in time so they can be executed in parallel or sequentially using the same piece of hardware).

Runtime reconfigurable applications consist of a set of time-exclusive tasks that can be downloaded into the FPGA (one at each time, or several simultaneously) using a dynamic allocation scheme. The challenges associated with runtime reconfiguration are closely linked with the goal of reconfiguration. Therefore, it is important to consider the

motivation and the different scenarios of runtime reconfiguration, which are algorithmic, architectural and functional reconfiguration. They are briefly described below.

2.5.2.1 Algorithmic Reconfiguration

The goal in algorithmic reconfiguration is to reconfigure the system with a different computational algorithm that implements the same functionality, but with different performance, accuracy, power, or resource requirements. The need for such reconfiguration arises when either the dynamics of the environment or the operational requirements change.

2.5.2.2 Architectural Reconfiguration

Architectural reconfiguration is performed to modify the hardware topology and computation topology by reallocating resources to computations. The need for this type of reconfiguration arises in situations where some resources become unavailable either due to a fault or due to reallocation to a higher priority job, or due to a shutdown in order to minimize the power usage. For the system to keep functioning in spite of the fault the hardware topology needs to be modified and the computational tasks need to be reassigned.

2.5.2.3 Functional Reconfiguration

Functional reconfiguration is used to execute different functions on the same resources. The need for this type of reconfiguration arises in situations where a large number of different functions are to be performed on a very limited resource envelope. In such situations the resources must be time-shared across different computational tasks to maximize resource utilization and minimize redundancy. In contrast to SR, the FPGA will probably be reconfigured more than once during the execution of an application.

We can distinguish two types of runtime reconfiguration methods: global and local. Global reconfiguration restricts the number of concurrent executing tasks to just one, which can occupy the whole circuit, whereas in local reconfiguration there can be more

than one task executing concurrently

Another classification could be established depending on the speed of the reconfiguration, and if the device must be stopped. Partially reconfigurable devices allow the replacement of just one part of the device, but it can't operate during replacement. Dynamically reconfigurable devices have the greatest degree of flexibility of all, thus allowing dynamic local reconfiguration.

Figure 2.10: Global runtime reconfiguration

In the global reconfiguration scheme, only one configuration can be downloaded at a time (Figure 2.10). With local runtime reconfiguration, the application is not restricted to only one downloaded configuration at a time. Several can co-exist and run simultaneously on the reconfigurable FPGA, being downloaded as required, and deleted, as more free space is needed for a new configuration,

Local runtime reconfiguration framework works in the same manner as a software memory cache as shown in [27], and for that reason it is referred to as logic caching. The FPGA acts as an active cache memory [28], which instead of instructions contains circuits. When not used, those circuits can be unloaded to allow new configurations to be downloaded, trying to keep the locality of the execution, in the same way that it happens

in software with cache memories (Figure 2.11).

Figure 2.21: Local runtime reconfiguration

2.6 Survey of Reconfigurable Platforms

Reconfigurable computing platforms are set to be the next generation of general-purpose computing machines. Research groups worldwide are currently working on combining the Flexibility of general-purpose processors with the efficiency of custom application-specific hardware implemented in reconfigurable logic. These systems are in general known as reconfigurable platforms. The general basic architecture of a reconfigurable platform is shown in Figure 2.12.

Figure 2.12: General basic architecture of a reconfigurable platform

This typical architecture integrates a standard processor core with an array of reconfigurable logic and memory on a single chip. The major reconfigurable platforms and systems will be briefly discussed in the following section.

2.6.1 University Research Projects

PIPERENCH [29] was developed at Carnegie Mellon University. It is a reconfigurable fabric that belongs to the family of Reconfigurable Attached Processors since it is interfaced with a processor through a PCI bus. PipeRench consists of an interconnected network of processing elements organized as pipeline stages. Each processing element consists of registers and ALUs. An intermediate language is used to generate the fabric's configurations.

CHIMAERA [30] was developed at Northwestern University. It is a reconfigurable array that is integrated to a processor as a Reconfigurable Functional Unit. The array has access to shadow registers and consists only of configurable combinational logic.

GARP [31][32] was developed at University of California Berkeley. It belongs to the family of Reconfigurable Coprocessors as it integrates a reconfigurable array that has access to the processor's memory hierarchy. The reconfigurable array may be partially reconfigured as it is organized in rows. Configuration bits are included and linked as constants with ordinary C compiled programs. GARP is dynamically reconfigurable, with configurations schemes controlled by a program running on the main processor.

MORPHOSYS [33] was developed at University of California at Irvine. It is a Reconfigurable Cell Array architecture that includes context memory and belongs to the family of Reconfigurable Coprocessors. A DMA controller transfers data to the context memory, and to and from a frame buffer that holds the data the array will operate on.

REMARC [34] was developed at Stanford University. It is a Reconfigurable Coprocessor with 64 programmable units, that targets multimedia applications. Each 16-bit unit has an entry instruction RAM, ALUs, data RAM, instruction and several other registers. The reconfigurable array operates on the coprocessor data registers and a control unit transfers data between these registers and the processor.

PRISC [35] was developed at Harvard University. It integrates combinational reconfigurable logic as Reconfigurable Functional Units with two inputs and one output. The compiler analyzes the generated code and identifies sets of sequential instructions to execute on the RFU.

SONIC [36] is a project from the University of London. It is designed to exploit parallelism in video image processing algorithms. It consists of a set of processing elements, called PIPEs, interconnected by a bus. It belongs to the family of Reconfigurable Attached Processors since it is interfaced with a processor through a PCI bus.

ONECHIP [37] is a reconfigurable processor developed in the Department of Electrical and Computer Engineering at the University of Toronto. It is a third generation architecture that integrates a Reconfigurable Functional Unit (RFU) into a superscalar Reduced Instruction Set Computer (RISC) processor's pipeline. The architecture allows dynamic scheduling and dynamic reconfiguration. It also provides support for pre-loading configurations and for Least Recently Used (LRU) configuration management.

2.7 Embedded Real Time Operating System

An embedded system is an application-oriented special computer system, which is scalable on both software and hardware. It can satisfy the strict requirements of functionality, reliability, cost, volume, and power consumption of the particular application. With rapid development of IC design and manufacture, CPUs have become inexpensive. Lots of consumer electronics have embedded CPUs and therefore are embedded systems. For example, PDAs, mobile phones, point-of-sale devices, VCDs,

and industrial robot control, can be termed as embedded systems. As applications of embedded systems become more complex, the operating system support and development environment become crucial. The real-time operating system (RTOS) becomes extremely important to the development of applications with real-time system requirements.

Real-time operating systems are an integral part of real-time systems. RTOS provides service for better hardware abstraction, task management, task synchronization, memory management, interprocess communication and I/O. However, the manner in which they support these areas differs from conventional operating systems. An RTOS is an operating system that guarantees timely response to external and internal events of real-time systems. Problems often faced by RTOSs are CPU and task scheduling, timing event management, and resource management. The RTOS requires multitasking, process threads, and explicit interrupt levels to deal with real-time events and interrupts.

Trends in chip design show rapidly increasing levels of integration. Processors, custom hardware, reconfigurable logic and memory can all be integrated onto a single configurable system-on-chip. Current available RTOS functionality is not suitable for the reconfigurable platforms, as it does not address the issues that are specific to the reconfigurable hardware.

2.8 Hardware Operating System for Reconfigurable Computing

Reconfigurable hardware operating systems are a rather new line of research. Operating systems traditionally provide run time support for applications. Some researchers have seen the need for a hardware operating system for runtime reconfigurable environment. The use of a reconfigurable operating system results in a number of benefits, as it

- Increases productivity
- Increases portability
- Eases system partitioning
- Simplifies debugging

Merino [56] proposed a hardware operating system to tackle a number of problems that the design of runtime reconfigurable system possibly poses.

- The partitioning of a high-level system description into blocks that can be swapped into/out of the reconfigurable area.
- The search for an optimal reconfiguration schedule for those blocks.
- The determination of a set of services that helps to properly execute the required functionality.

Wigley et al. [38] outlined the fundamental set of services that should be provided by an operating system for reconfigurable computing: resource allocation, resource partitioning, application placement and routing. Furthermore, they argued that the current evolution of an OS for reconfigurable systems is quite similar to the events that already occurred in traditional computing. They also mention some fundamental differences: the fact that in the case of reconfigurable computing, the application may have its own set of registers that present its internal context. This application specific context adds to the general context common to all applications and must also be savearestored when switching applications. The paper also describes the different forms of FPGA area fragmentation that influence hardware task placement. Their experimental work provides some results detailing the time needed to do partitioning, task placement and routing.

Wigley et al. [39] present an overview of the services provided by their operating system for reconfigurable computing. In addition to [38], they briefly discuss application loading, application s
e communication
abstraction between hardware and software (i.e. between OS and application) and between the hardware applications. They also introduce a number of operating system performance metrics.

Brebner proposed the virtual hardware operating system [40] and explored some of the fundamental issues that will influence the construction of any operating system for FPGAs with dynamic reconfiguration. He proposed that the application be designed into

relocatable cores known as swappable logic units (SLU) [41]. He identified the main responsibilities of an operating system to be the placement of SLUs and providing for “bus addressable” registers for communication between SLUs. The entire system described by him was simulated in C.

Nollet *et al.* [42] proposed an operating system for reconfigurable platforms. It is noted that reconfigurable applications share a large number of low-level processes that would be better handled by a common API. Several hardware structures are described to support the operating system. The reconfigurable space is split into equally sized tiles. A packet-switched inter-connection network connects each tile. Routers handle communication over the network. At design time tasks are constrained to the size of the tile. If a task is smaller than a tile then space is wasted. The fixed tile size and flexible communication reduce the time to around 100 μ s. This leaves the actual tile configuration time of around 100ms as the bottleneck in placing hardware tasks.

Simmler *et al.* [43] investigated multitasking on an FPGA. Their paper presents a way to enable task switching on an FPGA, by using an application dependent technique to extract and restore task state information: state extraction is done by filtering out the relevant state bits from the readback bitstream. State reconstruction is done by means of manipulation of the bitstream. Obviously, this is a very architecture dependent technique. In [44] Mignolet *et al.* introduced relocatable tasks, which can be executed either in software or hardware, depending on the available resources and the performance required.

Walder *et al.* [45] report a hardware operating system for reconfigurable computing and discussed operating system services including device partitioning, allocation, placement and routing. They [46] investigated area models for reconfigurable devices and introduced a 1D block partitioned model and devised an online scheduling system that schedules tasks according to several non-preemptive and preemptive methods. However, the proposed scheduling method does not consider the time and power required for

configuring the tasks and the reconfigurable area requirements, which depend on the size of the tasks.

A major drawback of using runtime reconfiguration is the significant delay of reprogramming the reconfigurable hardware. The total runtime of an application includes the actual computation delay of the each task on the hardware along with the total time spent on hardware reconfiguration between computations. Soheil *et al* [49] proposed an algorithm for minimizing the number of reconfigurations required in an application. However, they did not discuss the effect on configuration delay and reconfiguration area in the FPGA. The problem of allocation/ fragmentation/ relocation of the reconfigurable hardware task can appear in a way almost identical to that of software memory. Recently, researchers [47, 50, 51, 52] have dealt with this problem. Diessel *et al.* [47][48] studied the techniques denoted as local repacking and ordered compaction for placing tasks on dynamically reconfigurable FPGAs by trying to rearrange some of the already executing tasks. Compton *et al* [50] also consider relocation as the answer to the fragmentation problem. They propose a 2D architecture named R/D FPGA [Relocation/Defragmentation] that is able to relocate resident tasks in a row-by-row manner, with the help of a row-buffer, in order to free up the area for the arriving tasks. In this method, the task relocation time must be taken into account. The overall configuration time is increased and relocation capability is limited. Kalte *et al.* [51] described a 1-D fixed-size partitioning of the FPGA along with the details of configuration time delay and resources wasted.

Bazargan *et al* [52] addressed the issue of placing application mappings onto a single device for hardware execution in a reconfigurable computing system. Their work presents a placement method that can be applied to dynamically adaptive hardware. The solution offered is a hybrid between the typical first-fit in a finite space and best-fit algorithms and trades quality of placement for speed. Walder *et al.* [53] proposed algorithms based on Bazargan's approach, dealing with empty rectangles. They discussed a 2D-hashing method to find feasible task placement while considering runtime issues. Walder [54]

used a typical 2D bin-packing first-fit algorithm and a best-fit using fragmentation formula that is applied to every location when looking for a place for an arriving task. Thus each alternative location is evaluated, and the one generating the FPGA state with the lowest fragmentation is chosen. However, this is a time consuming process. As an interesting novelty, they consider a hierarchical task model made of one or several rectangular sub-tasks, whose relative position can be modified (“footprint transform”) during allocation, though no clear results on the benefits of such transformations on the algorithm’s performance are given.

2.9 Summary

In this chapter, we have examined various technologies that are indispensable for reconfigurable computing. In particular, the technology behind FPGAs that form the base of various reconfigurable computing solutions were described in detail. Subsequently, various models and methods of reconfiguration were presented along with techniques for speeding up the configuration process.

This chapter provides a survey of the various reconfigurable platforms that are currently available. Subsequently a detailed review and the need for a hardware operating system were presented. It was shown that although there are contributions from researchers in this area, efforts to realize efficient and robust techniques and methodologies for the efficient deployment of a hardware operating system are at its infancy. After a careful study of the merits and demerits of the existing approaches, we have outlined our approach towards the development of the various algorithms that constitute an efficient hardware operating system. This research work will look into the various issues such as efficient partitioning methods, intelligent scheduling algorithms and placement techniques. It is desirable that the proposed techniques lead to reduction in area, time and power. Since algorithms for optimization are often NP hard, efficient mechanisms must be devised to shorten the design space exploration.

Chapter 3

Reconfigurable Platform: Issues and Approaches

In this chapter we present the issues and approaches of runtime reconfigurable platforms. For these platforms we propose the development of the various algorithms for the efficient deployment of the hardware operating system, capable of handling both software and hardware issues.

3.1 Reconfigurable Platform Design Issues

While reconfigurable platform design has a promising future, certain issues have to be taken into account if this paradigm is to drive the reconfigurable system design industry. Most of these challenges are unique to this style of design, and are active research areas. The design flow [76] for a reconfigurable platform is shown in Figure 3.1. Because of the differing advantages of hardware and software implementations, most systems consist of a mixture of the two technologies.

In reconfigurable platform design, a given application specification must be broken down to two main categories – the hardware component and the software component. This problem is called hardware-software partitioning [77] and is an important issue that can affect the performance of the overall system. The goal is to partition the components of the given application to either hardware or software so that the resulting system provides optimum performance. This process is not standardized and a typical analysis can yield multiple partitions, making the design process an iterative one where each partition is selected and evaluated one by one until the design specifications are met. Also it is difficult to automate this process completely and user intervention is usually required. The problem of optimal hardware software partitioning is an active research area, and success has been achieved in some cases.

Figure 3.1: Reconfigurable platform design flow

This design flow assumes that a hard partitioning choice is made early in the design process, after which the two parts are implemented separately and subsequently integrated. Current design techniques are moving towards independent system design and simulation, with later partitioning. The use of reconfigurable platform adds to the complexity of the codesign task due to the following reasons.

- The communication interface of the hardware blocks can be reconfigured
- The processor on which the software runs may be re-customized
- The boundary between hardware and software may be moved dynamically

3.2 HardwareSoftware Codesign Issues

Recently, embedded processors integrated on the FPGA chip have introduced a new boundary for hardwaresoftware co-design [78] with software running on the embedded core as well as hardware tasks running on the FPGA. The computational sequence of tasks in an embedded device is represented by a task flow graph (TFG). For many applications, these tasks can be distributed both as hardware area as well as software resources. Resources in the hardware side can be Application Specific Integrated Circuits (ASIC) or reconfigurable logic devices like FPGA. ASICs are not suitable for runtime reconfigurable platform because of the lack of reconfiguration function and very large design times. The FPGA devices play a major role in reconfiguration to utilize hardware resources on account of their runtime reconfiguration functionalities. The architecture for a runtime reconfigurable platform consists of an embedded processor for software operations and reconfigurable logic resources for hardware functions integrated in an FPGA and shared memory as illustrated in Figure 3.2.

In the proposed runtime reconfigurable platform, we describe the fundamental difference between the hardware and software resources as follows. The tasks in the software process are executed in a sequential way. In the FPGA, the hardware tasks can be executed concurrently. At runtime, in order to execute the hardware tasks, some time is required to configure the FPGA resource. This is not applicable to software operations. The number of tasks executed in the FPGA resource depends on the number of hardware units that can be configured on the device at runtime. The execution of the hardware tasks is normally faster compared to the software tasks. The power consumption of the software resource is much less than hardware and there is no concurrency in software.

Figure 3.2: Hardware/Software co-design architecture

33 Operating System Level Issues

For the runtime reconfigurable platform, we intend to develop efficient methodologies for various services that should be provided by the operating system, which will handle the hardware/software task management, FPGA area management, memory handling, and placement loader running on the platform. In the runtime reconfigurable platform, each hardware task is a circuit, i.e., a task typically consists of a combination of logic circuits and memory. The task is loaded and executed in the FPGA area. The operating system should be capable of performing the following function:

- Partitioning the reconfigurable hardware area depending upon the task size.
- Placement of tasks in the configurable area.
- Scheduling of the tasks.
- Loading, executing and removing tasks.
- Smart interface (input/output) technology.

Considering the above requirements, the architecture of a typical operating system for runtime reconfigurable platform is as shown in Figure 3.3. The scheduling schemes handle the scheduling of the tasks on the processor and FPGA simultaneously. The

placement techniques allocate the tasks to the processor and the partitioned reconfigurable FPGA. The interface port is used to communicate with all the devices in the reconfigurable platform.

Figure 3.3: Hardware operating system architecture

3.3.1 Partitioning Issues

The latest commercial FPGAs have several million reconfigurable gates, and are capable of running with clock speeds in the hundreds of MHz range. The FPGAs can be configured in two ways: complete and partial reconfiguration [9]. Partial reconfiguration is the ability to reconfigure a portion of the FPGA while the remainder of the design is still in operation. This configuration method is useful for applications that require the loading of different designs into the same area of the device. To make use of this technology, we need to handle the partition issues. The wide range of electronic design components including processor core, digital signal processing elements, communication and networking elements etc. typically consume only a fraction of the resources in a system-level programmable logic device. Table 3.1 presents the area requirements, given in number of CLBs, of some typical design components as reported in the literature.

The currently available FPGA technologies (Xilinx Virtex) permit partial reconfiguration only in vertical chip-spanning columns. Hence, the configuration of a task potentially interferes with another task allocated to the same columns. In runtime reconfigurable computing, we make use of a task based partitioning method [58] for partitioning the reconfigurable hardware FPGA area. A hardware design unit is called a task; it is

executable on the partitioned FPGA. In the runtime reconfigurable platform, a hardware resource is partitioned into blocks or tiles. While the vertical dimension of each block is fixed, the horizontal dimensions can be varied as shown in figure 3-4. Each partitioned block accommodates one task at a time. In order to obtain good resource utilization and to reduce configuration times, the partition size of the block should be based on the hardware task size. When we partition the FPGA fabric, we must also consider the configuration and readback time. The hardware tasks need to be configured before it is executed in the available hardware block. The configuration time varies based on the block size

Core	Area [CLB]
UART [55]	50
100-tap FIR filter [56] 12 bit data & coefficients	250
ADPCM [56]	250
DCT [57]	600
Triple-DES processor [56]	800
256 point complex FFT [56]	850

Table3.1: Area requirements for FPGA cores

Figure 3.4: Partitioned FPGA

Figure 3.5 shows the TFG of a sample application and the partitioned reconfigurable FPGA. Each task in the TFG is instantiated through partial reconfiguration and its instantiation will not affect the other existing tasks. In Figure 3.4, tasks D, E, and F, can be instantiated by reconfiguring the partitioned hardware resources, This operation will

not affect the configured existing tasks B and C. In Section 4.1, we present an efficient initial and runtime partitioning algorithm for runtime reconfigurable platforms in order to achieve better hardware resources utilization and to reduce the reconfiguration delay time.

Figure 3.5: Executing TFG on a partitioned reconfigurable hardware

3.3.2 Scheduling Issues

Task scheduling is one of the important issues in a runtime reconfigurable platform. A scheduler is a module that allocates processor and hardware resources to jobs and tasks. Tasks are scheduled and allocated resources according to a chosen set of scheduling rules and the availability of resources. Scheduling hardware tasks is different from software task scheduling because in the case of a hardware task, there exists a need to reconfigure the partitioned FPGA area. The operating system should keep track of the available resources, and the location of all the tasks running. The task scheduler decides which task has to be executed next, among all tasks ready to run. This decision is based on some scheduling policy. Most existing schedulers can support only one scheduling algorithm and therefore, the system can only cater to a narrow range of applications, which work well under the same scheduling algorithm. However, if the scheduler is configurable to support several scheduling algorithms, then the solutions for hardware task management

become more flexible. The proposed scheduler supports a wide range of scheduling policies. The scheduler not only considers the task execution time, it also takes the task execution area, task configuration time and power issues into consideration during scheduling.

A good scheduler satisfies the following conditions:

- One hardware block is assigned to one hardware task at any given instant.
- The hardware task size should be less than or equal to the size of the block to which it is assigned.
- No hardware task is scheduled before its release time.
- Depending on the scheduler, the configuration and readback time is added to the task execution time.
- The placement handler handles the placement constraints.
- Resource utilization constraints should also be satisfied.

3.3.3 Placement Issues

The placement problem is one of the important issues associated with runtime reconfigurable platform devices. A basic service provided by a traditional operating system is the loading of tasks and initiating their execution. In a reconfigurable platform this is a relatively complex problem because tasks consist of a combination of logic circuits and memory. Loading a task in a reconfigurable platform means placing the circuit and the storage elements on the reconfigurable P G A . In a runtime reconfigurable platform, the configuration time delay is an important issue. It depends on the task placement on the partitioned FPGA blocks. Figure 3.6 shows a partitioned FPGA with blocks of different widths. The task shown in gray, has a width of five CLBs. As illustrated in the figure 3.6, when the task is placed in a large-sized block, a major portion of the block area remains unutilized. Also since the entire block has to be configured irrespective of the task size, the configuration delay also increases. It is desirable to minimize the configuration time required to configure the task on the FPGA. The placement technique determines which tasks are to be placed onto which blocks of the reconfigurable FPGA and configures the tasks on the FPGA.

Figure 3.6: Task placements in the partitioned FPGA

3.3.4 Communication Issues

An important issue in runtime reconfigurable platform is the communication between the various devices. As the user application running on the reconfigurable platform is implemented in hardware and software, there is a need for a standard communication interface between the hardware (FPGA) and the processor. A flexible communication infrastructure is required to provide communication between the FPGA and the processor. The hardware tasks are stored in external memory in the form of configuration bitstreams. Creating an interface between the external memory and the FPGA is one of the important communication issues. Our interface methods should also be able to handle the communication between tasks. Three possible cases are envisaged: communication between two hardware tasks (hw-hw), communication between hardware task and software task (hw-sw) and communication between two software tasks (SW-SW).

3.4 Summary

The reconfigurable computing platform possesses many advantages over the traditional software and hardware based computing approaches in terms of performance power, cost and hardware area utilization. An application targeted towards the reconfigurable computing system can be expressed as a task flow graph consisting of both software and hardware tasks. Therefore, the reconfigurable platform requires an operating system that is especially tailored to manage the hardware tasks in addition to conventional software task management.

In this chapter we have outlined the various requirements that must be satisfied for the efficient deployment of a hardware operating system. In particular the issues associated

with the efficient partitioning of the reconfigurable FPGA area into multiple blocks and the placement of the tasks in the partitioned blocks were discussed. It was shown that the partitioning and placement techniques should seek to minimize the configuration delay and improve the reconfigurable area utilization. Also, the need for a flexible task scheduler was established and a set of basic conditions that should be satisfied by an efficient scheduler was identified. In the next chapter, we present an important component of the proposed hardware operating system namely the method for efficiently partitioning the reconfigurable FPGA surface.

Chapter 4

Partitioning Algorithms for FPGAs

Partitioning, scheduling and placement of the hardware tasks are major issues in reconfigurable computing. A hardware task is a circuit, which is executable on the partitioned FPGA blocks. Each task has a different type of behavior, area requirement, and execution time. The partitioning process is used to determine the optimal number and widths of blocks in the reconfigurable FPGA area. A proposed partitioning algorithm should yield efficient partitioning results for the given reconfigurable device, which leads to better scheduling results. It seeks to minimize power, area, and reconfiguration time in the runtime reconfigurable platforms. The limitations of the currently available partial reconfigurable FPGAs (Xilinx Virtex) do not allow alterations in the vertical dimension and therefore the only option is to partition the FPGAs in the horizontal dimension. In this chapter, we have proposed a one-dimensional partitioning methodology that takes the characteristics of the task set into consideration.

4.1 Proposed Partitioning Techniques

We introduce an efficient partitioning algorithm for runtime reconfigurable platforms to achieve better hardware resource utilization. The following notations will be employed to describe the proposed partitioning algorithm:

$w_{\max}$: Upper bound of task width.

$w_{\min}$: Lower bound of task width.

$\overline{w}$: Device width.

w_i : Block width. ($w_{\min} \leq w_i \leq w_{\max}$)

m_i : Number of Blocks having width w_i .

k : Number of distinct sized blocks, $k \geq 1$.

Let us consider a sequence of tasks with different widths. These tasks are assumed to be randomly distributed in $[w_{\min}, w_{\max}]$, where $w_{\min}$ and $w_{\max}$ are the lower bound and the

upper bound of the task width respectively. Figure 4.1 illustrates tasks of different widths. Let $\bar{w}$ be the size of the reconfigurable device. This section explains two stages of partitioning, namely initial partitioning and runtime partitioning. The initial partitioning splits the reconfigurable logic area in the FPGA into hardware task size based blocks, each of which is capable of accommodating a hardware task. The initial partitioning algorithm is described next.

Figure 4.1: Task width distribution

4.1.1 Initial Partitioning Algorithm

The goal of the initial partitioning of the runtime reconfigurable platform is to achieve efficient partitioning of the FPGA reconfigurable area. It also lays the groundwork for the runtime partitioning algorithm. The initial partitioning problem can be stated as:

Given a reconfigurable device of width $\bar{w}$, and with $w_{\min}$ and $w_{\max}$ as the minimum and maximum of size of the given tasks, partition it (Figure 4.2) such that:

$$(a) \quad \bar{w} = \sum_{i=1}^k m_i w_i$$

$$(b) \quad \max_{1 \leq i, j \leq k} \{ |m_i - m_j| \} \text{ is minimized.}$$

Where $k \geq 1$ and w_i is the width of the i^{th} block ($w_{\min} \leq w_i \leq w_{\max}$);

Figure 4.2: Reconfigurable device partitioning

We propose a heuristic algorithm for the initial partitioning problem, as it is an optimization problem. The following outlines the proposed algorithm.

1. Define k and w_i as follows: -

$$(a) \quad k = \left\lceil \frac{w_{\max}}{w_{\min}} \right\rceil$$

$$(b) \quad w_i = (i + 1) w_{\min} \\ w_k = w_{\max}$$

This is because all tasks uniformly emerge in the interval $[w_{\min}, w_{\max}]$. This implies that $[w_{\min}, w_{\max}]$ is divided into $k - 1$ nearly same sized subintervals by $w_i, i = 1, 2, \dots, k - 2$. (In the cases where $w_{\max} / w_{\min}$ is a whole number, w_{k-1} and w_k will be same. Hence we have not mentioned w_{k-1}).

$$2 w_{\min} = w_1 < w_2 < w_3 \dots \dots < w_{k-2} < w_k = w_{\max}$$

2. Determine m_i according to k and w_i

$$\text{Assume } s = \sum_{i=1}^k w_i. \text{ Define } t = \left\lceil \frac{\bar{w}}{s} \right\rceil$$

In other words, the device width is divided into t units each of width s and each unit is further subdivided into k distinct sized blocks.

Set initial value of each m_i to t .

3. For partitioning the remaining of the device width i.e. $\bar{w} - t * s$,

Define remaining width

$$r = \bar{w} - t * s$$

If current remaining width $r > w_n, n = k, k-1, \dots, 1$ then $m_n = m_n + 1$. After each partitioning, update the remainder width as follows: -

$$r = r - w_n \quad n = k, k-1, \dots, 1$$

Merge the last remaining width with a block of size w_l .

The formal description of the initial partitioning algorithm is given in Figure 4.3:

Input: $\bar{w}$ - The size of the reconfigurable device; $w_{\min}$ - The lower bound of the width of the tasks; $w_{\max}$ - The upper bound of the width of the tasks;
Output: k, m_i and w_i such that $\bar{w} = \sum_{i=1}^k m_i w_i$ and $\max_{1 \leq i, j \leq k} \{ m_i - m_j \}$ is minimized
Algorithm: Initial Partitioning ($\bar{w}, w_{\min}, w_{\max}$)
<pre> begin 1. Let $k = \lfloor w_{\max} / w_{\min} \rfloor$;   /* fix the size of k */ 2. for i= 1 to k-2 do $w_i = (i + 1).w_{\min}$;   /* fix each w_i */ $w_k = w_{\max}$; 3. $s = \sum_{i=1}^k w_i$; 4. $t = \left\lceil \frac{\bar{w}}{s} \right\rceil$; 5. for i = 1 to k do $m_i := t$;   /* Partition the remainder interval */ 6. $r = \bar{w} - t * s$ 7. for i=k downto 1 do   /* get a block of size w_i from the remainder interval */   if $r \geq w_i$ then   begin $m_i = m_i + 1$; $r = r - w_i$;   end;   end for; 8. if $r > 0$ then $m_1 = m_1 + 1$; end   </pre>

Figure 4.3: Initial partitioning algorithm

As an example, let us consider an FPGA device with reconfigurable logic area (w) equal to 120 CLBs, the lower (w_{min}) and upper bound (w_{max}) of the task sizes is 5 (CLBs) and 20 (CLBs). Then the task interval $k = 20/5 = 4$. The partitioned block widths are: $w_1 = 8$, $w_2 = 12$, $w_3 = 16$ and $w_n = 20$ respectively. Applying the initial partitioning algorithm will partition the reconfigurable device as shown in Figure 4.4.

Figure 4.4: Example of initial partitioning algorithm

4.2 Runtime Partitioning Algorithm

During the initial partitioning of an FPGA reconfigurable fabric, the minimum and maximum size of the hardware tasks are known. However during runtime, the incoming task size can be larger than the available block size. In that case, no block would be able to accommodate the incoming hardware task. Therefore, we have to efficiently combine the available free blocks to form a new block suitable for incoming hardware task. We propose a runtime partitioning algorithm for reconfigurable platforms to handle the partitioning issues. This algorithm helps to optimally merge and split the currently available free hardware blocks in FPGA area to obtain sufficient space for incoming task to execute.

The runtime partitioning algorithm satisfies the following conditions:

Merging Blocks:

- Only adjacent blocks can be merged.
- At the time of merging, no tasks should be running in those blocks.
- After merging, the block size should be greater than or equal to the incoming task size.
- Time taken for merging the blocks should be minimal.
- The configuration time is proportional to the block size.

Splitting Blocks:

- At any given time only one block can be split.
- At the time of splitting, no tasks should be running in that block.
- After splitting, the block size should be greater than or equal to the incoming task size.
- The configuration time is proportional to the block size

Let the initial arrangement of the logic blocks be $b_1, b_2, \dots, b_n$. Let B be the set of all logic blocks, i.e., $B = \{b_i : 1 \leq i \leq n\}$. Assume that a_i is the size of logic block b_i . Let the incoming block be b_{in} of size a_{in} . We denote the objective function by $F(B, b_i, b_j, b_{in})$, which stands for the cost of replacing blocks ranging from b_i to b_j , $i \leq j$, by b_{in} . This cost, in turn, is a measure of the reconfigurable area that will remain unutilized as a result of accommodating the incoming task in the new block formed by merging the adjacent blocks.

The idea of the runtime partitioning algorithm can be stated as follows. If the incoming task size a_{in} is greater than the available block size then merge the available blocks. Otherwise, if the available block sizes are more than twice the size of incoming task, then split the available logic block to accommodate the incoming task. We explain below the merging and splitting procedures.

Merging of blocks: The aim of this procedure is to obtain the optimal repartition of the available logic blocks in order to accommodate the incoming block b_{in} , whose size exceeds that of the individual available logic blocks. This is achieved by merging the available blocks of lesser size. The algorithm arrives at the optimal repartition by looking for all the feasible solutions and assigning a cost to each feasible solution. The feasible solution with the least cost is taken as optimal. The search for feasible solution is conducted in an iterative manner as described below:

Iterative Search: Beginning from the left boundary, the algorithm tests whether there is a feasible solution. If there is a feasible solution, then it is set as the current optimal repartition and the cost of the repartition is determined by the objective function. Then the left boundary is incremented and the next feasible solution is searched. If the cost assigned to current feasible solution is less than the cost of the current optimal solution, then the current feasible solution is set as the current optimal partition. The iterative search continues till the right boundary is reached.

Decision on the existence of a feasible solution: Beginning from the left boundary, the procedure decides whether a feasible solution exists bounded by the right boundary. The size of initial available block is set as the size of merged blocks. Then the procedure merges the next available block with the initial block and the size of merged blocks is updated accordingly. If the size of merged blocks exceeds or equals the size of incoming block before surpassing the right most boundary, then a feasible solution exists and the feasible solution consists of blocks starting with the left boundary till the point when the size of merged blocks exceeds or equals that of incoming block.

Splitting of blocks: The aim of this procedure is to obtain the optimal repartition by splitting an available block in order to accommodate an incoming block of smaller size. The splitting is performed on the smallest of the available blocks so that it is split into two blocks; one of the size of the incoming block and another consisting of the remaining portion of the smallest available block.

Figure 4.5: Runtime partitioning of the FPGA

Input:
 1. The initial arrangements of the logic blocks: $b_1, b_2, b_3, \dots, b_n$. Let $B = \{ b_1, b_2, b_3, \dots, b_n \}$. Assume a_i is the length of the block i . The incoming block is b with the size of a .
 2. The objective function $F(B, b_i, b_j, b)$: the cost of replacing b_i to b_j by b , $i \leq j$.

Output: The optimal arrangement of the current blocks

Function: Decision_of_feasible_solution ($B, current\_sum, left\_boundary$);
 /* find a feasible solution bounded by left_boundary and right boundary */
begin
   1. $part\_sum\_of\_blocks := current\_sum$;
   2. $i := left\_boundary$;
   3. **while** ($part\_sum\_of\_blocks < a$) and ($i < n$) **do**
      $i := i + 1$;
      $part\_sum\_of\_blocks := part\_sum\_of\_blocks + a_i$;
   end;
   4. **if** $i < n$ **then**
      $right\_boundary := i$;
      $current\_sum := part\_sum\_of\_blocks$;
      $feasible\_solution := \{ b_{left\_boundary}, b_{left\_boundary+1}, \dots, b_{right\_boundary} \}$;
     return('yes');
   else
     return('no');
end
Function: Merging_blocks ($B, b, incoming\_block\_size$)
begin
   1. $current\_sum := incoming\_block\_size$;
   2. $left\_boundary := 1$;
   3. **while** $left\_boundary < n$ **do**
     begin
       3. 1. **if** Decision_of_feasible_solution ($B, current\_sum, left\_boundary$) == 'yes';
         /* there exists feasible solution */
       then
         Calculate the current objective function $F(B, feasible\_solution, b)$;
         Update the current optimal solution if necessary;
          $left\_boundary := left\_boundary + 1$;
       else
         /* stop the while loop */
          $left\_boundary := n + 1$;
       end
     end
end
Function: Splitting_blocks ($incoming\_block\_size$)
begin
   1. $block\_size := incoming\_block\_size$;
   2. $find(current\_available\_block\_size)$;
   3. $target\_block\_size := \min(current\_available\_block\_size)$;
   /* Current_available_blocks is splitted into 2 blocks */
   4. $Splitting\_Block\_size1 := incoming\_block\_size$;
    $Splitting\_Block\_size2 := target\_block\_size - Incoming\_block\_size$;
end
Algorithm: - Runtime Partitioning (In_Coming_Block_Size)
begin
   if ($\max(available\_block\_size) < incoming\_block\_size$)
     Merging_blocks ($B, incoming\_block\_size$);
   else
     if ($\text{all\_available\_block\_size} > 2 * incoming\_block\_size$)
       Splitting_blocks ($incoming\_block\_size$);
end

Figure 4.6: Runtime partitioning algorithm

Figure 4.5 above illustrates the process of merging the available free blocks in the FPGA fabric. For example, we assume the incoming task size is 30. The available block sizes are 5[A], 10[B, D, E], 15[F] and 20[C, G]. In the figure, block A and G are occupied by tasks. The other blocks are ready to be used by incoming tasks. If the incoming task size is larger than the block size, adjacent blocks must be merged to assign space for the incoming task. There are two possible ways to do this; one is to merge blocks B and C and the other is to merge blocks D, E, and F blocks. The merging of blocks B and C is the efficient solution for the above runtime partitioning problem. When the incoming task is executed on the new block formed by merging block **B** and C, the entire area of the block is utilized. The formal description of the runtime partitioning algorithm is given in Figure 4.5.

4.3 Experimental Results

4.3.1 Simulation-Environment

We have simulated the proposed algorithm to experimentally investigate the effectiveness of the partitioning techniques. The parameters of the simulator include the width of the reconfigurable device, the configuration time, minimum and maximum size of the tasks, number of tasks, and configuration and readback times for one column. We report on a number of selected experiments conducted with this simulation environment. For our simulation, we made use of the first come first serve (FCFS) scheduling policy and the free mode placement method explained in the next chapter. The simulation environment is similar to Xilinx Virtex XC2V 6000 (6 million gates) where the configuration and readback time of one column is 193 μ s.

To the best of our knowledge, there is no benchmark currently available to characterize the workload of the reconfigurable systems. In addition, there are no statistical data available from the real-world applications to model this. Therefore, we have to resort to randomly generated tasks. We assume a sequence of tasks with no inter-task communication. Four task sets, with different number of tasks and sizes, were considered and simulated. The task widths (size) are randomly distributed based on the upper and

lower bounds of the task sets as shown in Table 4.1. In all our experiments, we use four task sets with 25,50,75 and 100 tasks. For all task sets, task execution times are assumed to be randomly distributed between 5ms and 25ms. The arrival times are chosen to be equally distributed in the range shown in Table 4.1. Different values of arrival times for different task sets are chosen to ensure that the waiting times of tasks stay within reasonable limits. For this simulation environment, we consider the maximum size of the reconfigurable area as 500 CLB columns. In order to evaluate the proposed partitioning algorithm, we use various metrics such as total execution time, average configuration time, average response time and total configuration power. The total execution time is defined as the total time involved in the execution of all the tasks. The configuration time of a given task is the time required to configure the block to which it is allocated. The total configuration power is the sum of the configuration power for all the configured tasks. We consider the amount of power required to configure one CLB column as one unit of power. The scheduler performance is determined by the average response time for the task set. The average response time r_i of a task t_i is given by $r_i = f_i - a_i$, where f_i is the task's finishing time and a_i is its arriving time. The simulations were undertaken with the assumption that the overhead associated with the runtime partitioning algorithm is negligible compared to the time required for configuring and executing the tasks. In the runtime reconfigurable platform, we assume the tasks assigned to various blocks are configured and executed in parallel. The total execution time denotes the time required for all the tasks in the task set to execute. However, the total execution time is less than the sum of individual task execution times as multiple tasks are executed concurrently.

	Minimum	Maximum
Task Width (CLBs)	20	100
Arriving Time (ms)	1	50
Execution Time (ms)	5	25

Table 4.1: Task set information

4.3.2 Performance of Partitioning Algorithm

In this section, we present and analyze the performance of the partitioning algorithm for the reconfigurable platform. First, for each metric, we present the results obtained by simulation and analyze the performance. We compare the partitioning algorithm described in Section 4.1.1, with an existing method proposed in [67] where the reconfigurable device is partitioned into fixed-sized blocks. In our proposed partitioning algorithm, the reconfigurable device is partitioned into blocks of varying sizes based on the upper and lower bounds of the task sets. In Figure 4.7 and Figure 4.8, we present the total execution time and the average response time for the task sets. Based on the results, we infer that the proposed algorithm helps to reduce the total execution time and the average response time by up to 17% and 20% respectively compared to the method proposed in [67].

Figure 4.7: Total execution time

Figure 4.8: Average response time

Reconfiguration time delay [49] is a major consideration in the runtime reconfiguration platform. As shown in Figure 4.9, our partitioning methodology reduces the average task configuration time by up to 30% compared to the fixed size partitioning. Configuration power is an important issue in the design of FPGA based systems. This is because during the configuration process the reconfigurable device requires power to reset registers, enable I/O pins, and enter operating mode. From Figure 4.10 it can be seen that our partitioning algorithm results in less configuration power compared to the existing method [67].

Figure 4.9: Average task configuration time

Figure 4.10: Total task configuration power

In the runtime reconfigurable platform, the reconfigurable space is an important concern. If the reconfigurable space is used in a very efficient manner, good device utilization and low fragmentation can be obtained. Inefficient partitioning of the reconfigurable space and placement techniques lead to a high level of fragmentation as smaller sized tasks may be assigned to larger blocks. This results in low area utilization of the reconfigurable platform. Poor utilization of the reconfigurable resources result in increased configuration time, and configuration power. In our work, we considered only the internal fragmentation. Internal fragmentation is due to a smaller-sized task being assigned to a larger block resulting in unutilized space within the block. Figure 4.11 shows that the proposed algorithm yields better device utilization by up to 30% compared to the existing work [67].

Figure 4.11: Total Area Utilization

Figure 4.12 illustrates that our proposed algorithm is better than the existing work [67] in terms of fragmentation (the average percentage of unused space within the blocks) by up to 25%.

Figure 4-12: Fragmentation

4.3.3 Effect of Runtime Partitioning on Initial Partitioning

Figures 4.13 to 4.17 and Table 4.2 to 4.4 present the effect of the runtime partitioning algorithm on the initial partitioning. It can be seen that the runtime partitioning algorithm improves the performance of the initial partitioning in terms of the average task response time, average task configuration time, area utilization and task configuration power. The results establish the need for intelligently merging and splitting the pre-partitioned blocks during runtime

Figure 4.13: Average response time with and without runtime partitioning

Figure 4.14: Average task configuration time with and without runtime partitioning

Task Set	Initial Partitioning (ms)	Initial & Runtime Partitioning (ms)	% Reduction
25 tasks	12.92	12.12	6.19
50 tasks	13.58	13.08	3.64
75 tasks	13.45	13.10	2.57
100 tasks	13.84	13.44	2.87

Table 4.2: Average task configuration time **Figure 4.15:** Total task configuration power with and without runtime partitioning

Task Set	Initial Partitioning (power units)	Initial & Runtime Partitioning (power units)	% Reduction
25 tasks	1740	1620	6.82
50 tasks	3750	3600	4.01
75 tasks	5620	5400	3.93
100 tasks	7640	7400	3.14

Table 4.3: Total task configuration power

Task Set	Initial Partitioning (%)	Initial & Runtime Partitioning (%)	% Improvement
25 tasks	81	84.6	4%
50 tasks	81.8	85.7	5%
75 tasks	81.7	85.3	4%
100 tasks	81.5	85.1	4%

Table 4.4: Total area utilization

Figure 4.16: Total area utilization with and without runtime partitioning

Figure 4.17: Fragmentation with and without runtime partitioning

434 Partitioning Algorithm Performance for Different Device Widths

The following set of experiments compares the performance of the proposed algorithm against that of the method put forth in [67] for various reconfigurable device sizes. The upper and lower bounds for the task arrival time and task width are also varied for each task set as specified in Table 4.5. The experiments are performed using four different task sets denoted as A, B, C, and D, each with 25, 50, 75 and 100 tasks respectively.

Device	Device width [CLBs]	Task Width [CLBs]	Arriving Time (ms)	Execution Time (ms)
D1	500	[20,100]	[1,50]	[5,25]
D2	800	[40,200]	[1,100]	[5,25]
D3	1200	[60,300]	[1,150]	[5,25]
D4	1600	[80,400]	[1,200]	[5,25]

Table 4.5: Devices and task set information

In Figures 4.18 to Figure 4.23, we present the execution time, the average task response time, the average configuration time, area utilization and the configuration power for the task sets for the different devices. From the results, we infer that the reduction in average response time ranges from 10% to 20% for all combinations. The reduction in the average configuration time is between 20% to 30%. The improvement in area utilization ranges from 15% to 25%. The results clearly show that the proposed algorithm is capable of achieving better performance under all conditions compared to the method proposed in [67]

Figure 4.18: Total Execution time for different device widths

Figure 4.19: Average task configuration time for different device widths

Figure 4.20: Total task configuration power for different device widths

Figure 4.21: Average task response time for different device widths

Figure 4.22: Total area utilization for different device widths

Figure 4.23: Fragmentation for different device widths

4.4 Summary

Partitioning the reconfigurable FPGA in an efficient manner is vital for minimizing the configuration delay and to achieve high area utilization. In this chapter, we have presented an initial and runtime algorithm for partitioning the FPGA and described the steps in detail. The initial partitioning algorithm partitions the FPGA surface into varying-sized blocks based on the lower and upper bounds of the widths of the expected task sequence. The runtime partitioning algorithm handles situations where the incoming is larger or much smaller than all the available blocks by merging or splitting the free blocks such that maximum area utilization is achieved. The proposed techniques were simulated for task sets with different characteristics and compared with the existing approach of uniform-sized partitioning. The results confirm that the proposed methodology yields up to 17% and 20% performance enhancement in terms of execution time and response time, increases area utilization by up to 25%, reduces the configuration delay by up to 30 %, and causes a reduction of up to 24% in configuration power, which is an important consideration in the reconfigurable platform.

Chapter 5

Scheduling and Placement Methodology

In this chapter we discuss the major and dominant scheduling theories and algorithms for runtime reconfigurable platform. The nexus of scheduling and placement is the main characteristic of the problem of mapping tasks to a reconfigurable platform. In this work, we focus on scheduling and placement techniques of hardware tasks in the runtime reconfigurable systems.

5.1 Task Scheduling in Reconfigurable Platform

In general, a hardware task in a reconfigurable platform has timing constraints that specify when the task configuration begins, how long it executes and the deadline for the completion. All the scheduling algorithms that we discuss are priority driven, and they are studied in either preemptive or non-preemptive environments. The preemptive [63] scheduling policy refers to the interruption of a task that is being executed, by the arrival of a higher priority task. The interrupted task can only continue executing if the higher priority task has finished executing. In a reconfigurable platform, preemptive schedulers include hardware task configuration time and readback time in the scheduling policy. Configuration and readback times are proportional to the hardware task width. Figure 5.1 shows an example of preemptive task scheduling in a reconfigurable platform, with task T_1 , and its interaction with another task T_2 . In this example, we assume that T_1 has a low priority than T_2 . First, T_1 is configured and begins to execute immediately. Let us assume that the task T_2 arrives while T_1 is being executed. Since, T_2 has higher priority over T_1 , T_2 preempts the ongoing execution of task T_1 . At this point, the execution of task T_1 is stopped and the current task status (contents of the storage registers etc.) is readback as a bitstream. Subsequently, T_2 is configured and executed. After T_2 completes execution, T_1 is configured with the readback bitstream and executed from the state where it was stopped.

Figure 5.1: Preemptive task scheduling

In non-preemptive [62][64] schedulers, once a task is loaded on to the reconfigurable device it must be allowed to execute until completion. Figure 5.2 illustrates the case where the computation of T_1 is non-preemptive and where T_2 has priority over T_1 . T_1 becomes configured and begins to execute. But even though T_2 has priority over T_1 , T_1 cannot be preempted, and T_2 must wait until the execution of T_1 completes.

Figure 5.2: Non-preemptive task scheduling

5.1.1 Scheduling Policies

In our work, we consider the following non-preemptive scheduling policies:

First Come First Serve [FCFS] Scheduling: This non-preemptive scheduling algorithm executes the tasks in the order they arrive and executes them to completion. As each task arrives, it is placed in a queue. During each iteration, the oldest task in the queue is selected to be configured and executed. While this scheduling policy treats all the tasks in a equal

and fair manner, a major drawback is that tasks with large execution times may cause the other tasks to wait for longer periods. This will increase average response time of the system, which is often not desirable.

Shortest Job First [SJF] Scheduling: This scheduling policy aims to overcome the drawback associated with the FCFS policy explained above by favoring the task with the shortest expected execution time. This will cause the tasks with large execution time to wait for long periods before being scheduled. However, the SJF policy helps to reduce the average response time of the scheduler.

Priority Based [PB] Scheduling: This non-preemptive scheduling method requires each task to be assigned a priority in advance. During runtime the task with highest priority among all the tasks in the queue is selected to be configured and executed. The priority based scheduling methods are classified as fixed priority, dynamic priority, or mixed priority [73]. A fixed-priority algorithm assigns all priorities at design time, and those priorities remain constant for the lifetime of the task. A dynamic-priority algorithm assigns priorities at runtime, based on execution parameters of tasks, such as upcoming deadlines. A mixed-priority algorithm has both static and dynamic components.

5.2 A Unified Framework for Reconfigurable Platform

The proposed framework concerns only the hardware tasks to be executed in the reconfigurable FPGA area. In the runtime reconfigurable platform, the scheduling can be divided into two parts. On the processor side, the scheduling maintains the traditional meaning, i.e., to find out the computation order and start time on the sequential devices. On the configurable FPGA side, the scheduling not only determines the execution time but also the configuration time of the hardware tasks on the FPGA taking into consideration the partitioned block sizes and the resource constraints. In Figure 5.3, S1 to S4 denote software tasks and H1 to H4 denote hardware tasks. The figure illustrates that while the software tasks involve only execution, the hardware tasks have to be configured first before execution.

Figure 5.3: Hardware/ Software task computations

5.2.1 Scheduling Scheme

In runtime reconfigurable platform scheduling, tasks are queued and processed according to one of the scheduling policies. The tasks in the queue are assumed to be independent tasks. The time to Load a hardware task on the reconfigurable FPGA is proportional to the area of the task. In the runtime reconfigurable platform, the scheduler selects the task from the queue depending on its scheduling policy and calls the placer to determine whether a feasible block exists in the partitioned FPGA where the tasks could be mapped. If necessary the runtime partitioning algorithm is invoked to optimally merge or split the existing free blocks. However, if no suitable block could be found to accommodate the task, it waits until free space becomes available.

5.2.2 Placement Strategy

Consideration of configuration time delay in the runtime reconfigurable platform is an important issue in the placement methodology, The amount of configuration time a task requires depends on the hardware tasks configured in the block partitioned reconfigurable FPGA In the one-dimensional partitioned FPGA, the area of the task is proportional to the allocated block size in the FPGA, The main purpose of the placement method is the management of the free partitioned blocks on the reconfigurable device. The structure of the scheduler and placer are shown in Figure 5.4, which consists of a single queue and a placer. The arriving tasks go into the queue and wait for a free block to become available. The placer selects and places the task that is to be executed next. Every time an executing task finishes, the placer chooses the next task to allocate into the freed block. The placer can operate in two modes namely free mode and restricted mode. In the restricted mode,

the task can only be placed into a block that matches the task size. This minimizes the amount of time and power required to configure the task. However, this may slightly increase the average response time of the system. In the free mode, the placer can allocate a task to any block that is able to accommodate it,

Figure 5.4: Structure of single-queue scheduler and placer

Placement with Multiple Queue

In order to further improve the performance of the system, we propose a methodology in which the placer makes use of multiple queues. Each queue corresponds to a distinct-sized block obtained through the initial partitioning algorithm explained in Chapter 4. This ensures that each block will be immediately allocated to a task that matches its size instead of remaining unutilized until the matching task is allocated based on its order of arrival. This method reduces the average task response time. The proposed multiple queue placement methodology is illustrated in Figure 5.5.

Figure 5.5: Structure of multiple-queue scheduler and placer

Figure 5.6 presents a high level view of scheduling, placement and partitioning in reconfigurable platform. Both the scheduling and placement modules can be implemented using multiple options. While the scheduler uses one among the three scheduling policies (FCFS, SJF, and PB), the placement module can employ either the free mode or the restricted mode of operation. Also, both single queue and multiple queue based approach can be adopted for placement. The partitioning module can make use of either the merging or splitting procedure based on the requirement. The inputs for the scheduling module are the hardware tasks, their time of arrival, and their configuration and expected execution times. The process is completed when all the tasks have been configured and executed. The flowchart in Figure 5.7 illustrates the interaction between the scheduling, placement and partitioning modules.

Figure 5.6: High-level view of reconfigurable platform scheduling and placement

Figure 5.7: The reconfigurable platform scheduling and placement algorithm

5.3 Experimental Evaluation

In this section, we simulate and evaluate the various techniques for scheduling and placement proposed in this chapter. The three different scheduling policies namely, first come first serve (FCFS), shortest job first (SJF) and priority based (PB) were simulated. The two placement strategies introduced in the previous section namely, free mode and restricted mode were evaluated with each of the scheduling policies. For the following analysis the device width of the FPGA is considered as 380 CLBs and with block sizes of 40,60,80,and 100. The simulation parameters are chosen as per the criteria outlined in

5.3.1 First Come First Serve (FCFS)Scheduling Policy

The FCFS scheduler was implemented and tested with underlying placer modes. Figure 5.8. and Figure 5.9 shows the total execution time and total configuration time for free mode and restricted mode. The results show that the free mode's execution time is higher than the restricted mode. Since the restricted mode selects the most appropriate block size for the task to be executed, the total area utilization of the restricted mode is better than the free mode as shown in Figure 5.10. Also the restricted mode results in reduced onfiguration time and configuration power as shown in Figure 5.9 and Figure 5.11.

Figure 5.8: Total execution time for FCFS

Figure 5.9: Total configuration time for FCFS

Figure 5.10: Total area utilization for FCFS **Figure 5.11:** Total configuration power for FCFS

5.3.2 Shortest Job First (SJF) Scheduling Policy

Figures 5.12 to 5.15 compare the free and restricted modes of placement for the SJF scheduling policy. It can be seen that the trend is similar to that of the FCFS policy for all the parameters. From the results we infer that restricted mode results in up to 5% reduction in the total execution time and up to 11 % reduction in the total configuration time compared to the free mode. The restricted mode helps to increase the total area utilization by up to 11 % compared to the free mode.

Figure 5.12: Total execution time for SJF **Figure 5.13:** Total configuration time for SJF

Figure 5.14: Total configuration power for SJF

Figure 5.15: Total area utilization for SJF

5.3.3 Priority Based (PB) Scheduling Policy

In order to simulate the priority based scheduling policy we make use of the fixed priority approach, where each task is assigned a priority during design time itself. In Figure 5.16 and Figure 5.17, we present the total execution time and the total configuration time for each placement mode for all the task sets. The results show that the restricted mode is better than the free mode in terms of execution time and configuration time. From Figure 5.18 and Figure 5.19 we infer that the restricted mode yields lower configuration power and better area utilization. The restricted mode results in better execution time and configuration time by up to 4% and 8% respectively compared to the free mode. Thus it can be seen that for all the parameters the restricted mode outperforms the free mode by a significant margin.

Figure 5.16: Total execution time for PB **Figure 5.17:** Total configuration time for PB **Figure 5.18:** Total configuration power for PB **Figure 5.19:** Total area utilization for PB

534. Comparing single queue and multiple queue placement techniques

We evaluate the effect of multiple queue placement using the FCFS scheduling policy. Figures 5.20 to 5.23 compare the single queue and multiple queue techniques. It can be seen that the multiple queue method consistently outperforms the single queue method, be it marginally, for all the parameters. This is because employing multiple queues reduces the

waiting time of the tasks and increases the utilization of the free blocks in the reconfigurable area.

Figure 5.20: Total execution time for FCFS

Figure 5.21 : Total configuration time for FCFS

Figure 5.22: Total configuration power for FCFS

Figure 5.23: Total area utilization for FCFS

5.4 Summary

In this chapter, we have discussed various scheduling policies and placement techniques. We have also presented a unified framework for reconfigurable platforms which handles three different problems, namely partitioning, scheduling and placement. To address the

scheduling component of the presented framework, a scheduling method, which includes three different scheduling policies has been proposed and experimentally evaluated. In the absence of any previous work in this area, we are unable to benchmark the performance of our algorithm. However, we have compared the free mode and restricted mode placement techniques for each of the three different scheduling policies. Our experimental results show that the restricted mode consistently outperforms the free mode in terms of execution time, area utilization, configuration time and configuration power. Since these parameters are important considerations in reconfigurable platforms, we have decided to employ restricted mode in subsequent works.

Chapter 6

Application Specific Partitioning and Scheduling

Partitioning of the reconfigurable resources in an FPGA concerns improving the area utilization, reducing the configuration delay and power consumption. At runtime reconfiguration, finding the free space in the FPGA and relocating an existing task in the FPGA are high latency processes. This chapter proposes a technique to address the latency concerns and reconfigurable area requirements for application specific CSoC. Predefined applications afford us a way to estimate its maximum hardware requirements by making use of a Directed Acyclic Graph (DAG). The knowledge of maximum hardware requirements for the given application can be in turn used to reduce configuration time requirement and power consumption of the initial and runtime reconfiguration processes. The proposed methodology is very well suited for currently available partial reconfigurable FPGAs like the Xilinx Virtex Family [9].

6.1 Problem Formulation

In this section, a task graph model [60] is used to describe the given application. The application is partitioned into a task graph, which is a directed acyclic graph, as shown in Figure 6.1. In a task graph, each node represents a task that may be implemented as hardware or software. The directed edges represent data dependencies between tasks. An edge say $1 \rightarrow 4$, implies that task 4 cannot start execution until task 1 is finished. The data dependency edges ensure the correct order of execution. Each edge is associated with a scalar describing the amount of data that must be transferred between the two connected tasks, which decides the communication time between the tasks, and it is referred to as inter-task period. It takes a certain amount of time to configure the reconfigurable platform before a hardware task can be executed. This is referred to as the task configuration time. We know the number and the size of the tasks for a given application. From the task graph, we determine its hardware area requirement. Our main

objective is to find the reconfigurable hardware requirements of the given application with minimal reconfiguration time delay. Our methodology aims to improve the reconfigurable hardware utilization and to minimize the reconfiguration time delay. One way of eliminating the reconfiguration delay could be the usage of preconfiguration, which means reconfiguring the FPGA before such a configuration is actually needed. This problem model concerns only the hardware tasks to be executed in the reconfigurable FPGA area.

Figure 6.1: Flow graph of an application

6.2 Proposed Algorithm for Application Specific Partitioning

Figure 6.2 describes the general outline of our approach for estimating and partitioning the reconfigurable area for a given application. It is worth noting that the operations depicted in the figure need to be performed only once during design time. The first step is the hardware estimation phase, where the reconfigurable hardware requirement of the given application is determined. The hardware estimation is followed by the partitioning of the estimated reconfigurable hardware into varying-sized blocks as per the initial partitioning algorithm described in Section 4.1.

Figure 6.2: General outline for application-specific partitioning [Design time]

The following notations will be employed to describe the proposed application specific partitioning algorithm:

- T_n : Total number of tasks;
- S_T : Size of the task;
- w_{max} : Upper bound of task width;
- w_{min} : Lower bound of task width;
- k : Number of distinct sized blocks, $k \geq 1$;

The goal of the application specific partitioning algorithm for the runtime reconfigurable platforms is to determine the reconfigurable area requirements of a given application in the FPGA and partitioning the estimated hardware. Let us assume that there is a task flow graph and it has a sequence of tasks of different sizes and the configuration time of the hardware task is less than the inter-task communication time. From the task flow graph,

we know the number of tasks involved in the given application, minimum & maximum task sizes, and the arriving and execution times of the tasks.

The application specific partitioning algorithm consists of the following steps:

- Calculating the size and the number of distinct hardware blocks for given application;
- Estimating the maximum hardware requirement for the given application using a path-based technique;
- Partitioning the estimated hardware of the given application;
- Scheduling of the tasks in the given application

Each of the above is explained in the following sections.

Calculating Distinct Blocks

As was the case with initial partitioning algorithm described in Section 4.1.2, fix the size of k and w_i by $k = \lfloor w_{max}/w_{min} \rfloor$ and $w_i = (i+1) * w_{min}$ for $i = 1, 2, \dots, k-2$, and $w_k = w_{max}$, respectively. Hence

$$2 * w_{min} = w_1 < w_2 < w_3 \dots < w_{k-2} < w_k = w_{max}$$

The number of the blocks thus determined is further optimized for the given application as follows. In application-specific partitioning the total number of tasks involved is known. Allocate each task to the block, whose size equals or just exceeds the size of the task. After all the tasks have been assigned to blocks, discard the blocks with the number of assigned tasks less than a threshold. In our implementation, the threshold is chosen as 2. The new value of k is the number of distinct blocks after removing the blocks with one or no task assigned to it.

Figure 6.3 illustrates the formal description of the procedure for calculating the distinct blocks. For each application, we determine the number of blocks and the size of the blocks.

Procedure: Calculate distinct blocks

Begin

- ```

1. Let $k := \lfloor w_{max} / w_{min} \rfloor$; /* fix the size of k */
2. for $i := 1$ to $k-2$ do $w_i := (i+1) \cdot w_{min}$; /* fix each w_i */
3. $w_k := w_{max}$;
4. for $j=1$ to T_n do /*find no. of blocks required */
    to_allocate_task(); /* allocate tasks to related blocks */
5. find (number of tasks in each blocks);
6. if (number of task less than threshold) then
    remove(block);
7. $k = \text{get distinct blocks}$;

```

**End**

**Figure 6.3:** Calculating distinct blocks

## Path-Based Hardware Estimation Technique

The proposed technique for estimating the optimal amount of reconfigurable hardware required for a given application is based on an adaptation of concepts employed in path-based scheduling [61]. In an application represented as a directed acyclic task flow graph, the control flow from the start to the end can be along any of the possible paths in the graph based on the inputs provided. Each path is composed of a different set of tasks and may have different hardware requirements. Therefore, we estimate the optimal hardware requirement for each path in the task flow graph in order to determine the given application's reconfigurable hardware requirement.

Before applying the path-based technique, each task in the task flow graph has to be replaced with the closest-sized block that can accommodate it among all the distinct blocks that were determined as explained in the previous subsection. Subsequently, all the possible paths in the graph from the start to the end have to be determined. In graph theory, all paths between two given vertices are normally found using a recursive depth-first search method [57]. For each path, the optimal hardware requirement is the sum of the sizes of all the distinct-sized blocks that are part of it. Since the reconfigurable platform should be able to support the path with the largest requirement, the maximum value among the hardware requirements for all the paths is adopted as the amount of

reconfigurable hardware required for the given application. The above steps are enumerated in Figure 6.4.

**Procedure:** Path-based hardware estimation

**Begin**

1. *#paths := compute number of paths (DAG);*

2. *for i = 1 to #paths do*

*hardware requirement of path i = sum of all distinct-sized blocks in path i;*

3. *calculate maximum hardware requirement of all paths;*

**End**

**Figure 6.4:** Path-based hardware estimation technique

An example of DAG for path-based hardware estimation is shown in Figure 6.5. The numbers in the square boxes represent the block size corresponding to each task. The arc shown in the graph highlights one of the execution paths of the given application. Calculating all paths in the graph representation in Figure 6.5 leads to five paths. The five paths are: ABEJM, ABFJM, ACGLM, ADHKM, and ADIKM. The hardware requirement of the each path is shown in the Figure 6.6. The block sizes involved in the path ABEJM are 10,15,10,25 and 15, which totals to 75. However, there are two tasks A and E requiring a block of size 10 for execution. Once task A has finished execution, it can be reused for task E. Thus the total hardware requirement for path ABEJM sums to 50 when only distinct-sized blocks are considered. Therefore, the estimated hardware requirement for the five paths are: 50,70,45,55 and 55. The maximum hardware requirement of the given application is therefore 70.

**Figure 6.5:** Example of DAG for path-based hardware estimation

**Figure 6.6:** Hardware requirement for paths in the DAG



## **Partitioning the Estimated Hardware**

After estimating the maximum reconfigurable hardware required for the given application as explained in the previous subsections the initial partitioning algorithm is applied to partition the reconfigurable space into varying-sized blocks. The estimated optimal hardware requirement of the application serves as the width of the reconfigurable device that is subjected to initial partitioning. The other inputs to the initial partitioning algorithm such as the minimum and maximum task widths are derived from the set of distinct blocks that was determined as explained earlier. Thus, the initial partitioning algorithm is tailored to the needs of the given application in order to increase the overall performance.

## **6.3 Application-Specific Scheduling and Placement**

In the following, we describe the actions that take place in the application-specific system during runtime. The operations of the scheduling, placement and runtime partitioning processes in the application-specific methodology is similar to that of the non-application-specific case. However, there are some important differences. Firstly, since the details of the application and the nature of the hardware tasks in it are known in advance, a scheduling policy that is most suited to the application can be implemented. Also, different priorities could be assigned to the various tasks based on the application requirements.

The most important feature in application-specific scheduling is that since the task flow is known in advance, based on the tasks that are being currently executed, the tasks that are expected to be executed next can be prefetched and added to the task queue. The scheduler then selects a task from the queue based on the scheduling policy and supplies it to the placement module for configuration and execution. Thus, in the application-specific system a task can be configured and made ready even before it is needed for execution. We refer to this as preconfiguration. It is envisaged that this would

significantly improve the total execution time of the given application. Figure 6.7 presents the flow of the application-specific scheduling and placement methodology.

**Figure 6.7:** Flow of the application-specific scheduling and placement methodology

## 6.4 Performance of the Application-Specific Partitioning and Scheduling Algorithm

In this section, we simulate and evaluate the performance of the application-specific partitioning and scheduling algorithm. The simulation parameters were as described in Section 4.3. We compare the proposed application-specific algorithm with the general-purpose partitioning, scheduling and placement schemes proposed in the previous chapters in order to quantify the improvement achieved. The simulations were performed for four different task sets using the three scheduling policies and the two placement modes. From Figure 6.8 to Figure 6.13, it can be concluded that the application specific partitioning algorithm along with preconfiguration reduces the total execution time and average response time of the tasks. The reduction in execution time was between 15 % to 34 % and the same in response time was between 10 % to 24 %. The results clearly confirm that incorporating application-specific techniques significantly improves the performance of the reconfigurable platform.

**Figure 6.8:** Total execution time with FCFS scheduling

Figure 6.9: Average response time with FCFS scheduling

Figure 6.10: Total execution time with SJF scheduling

Figure 6.11: Average response time with SJF scheduling

**Figure 6.12:** Total execution time with PB scheduling

**Figure 6.13:** Average response time with PB scheduling

## 6.5 Summary

In this chapter, we have presented an application-specific partitioning and scheduling algorithm to further optimize the use of the reconfigurable space. The proposed algorithm makes use of the knowledge of the maximum hardware requirement of the given application, which is estimated by a path-based technique. Our algorithm is employed to

maximize reconfigurable hardware utilization and to minimize the reconfiguration time delay. The results confirm that the proposed algorithm yields up to 34 % and up to 24% performance enhancement in terms of total execution time and average response time when compared to the case where application-specific information is not considered. In the next chapter, the architecture of the proposed reconfigurable computing platform is described.

# Chapter 7

## Runtime Reconfigurable Platform

---

In this chapter, we present a new architecture for a runtime reconfigurable platform. The runtime reconfigurable platform incorporates a specialized operating system capable of handling both software and hardware issues. For these platforms we have proposed and developed various algorithms and methodologies for partitioning, scheduling and placement. These algorithms and methodologies are necessary for the efficient deployment of the hardware operating system for runtime reconfigurable platform.

### 7.1 Proposed Architecture for Runtime Reconfigurable Platform

The proposed reconfigurable computing target architecture is shown in Figure 7.1. The Configurable System-on-Chip (CSoC) is used as the platform. It consists of one soft-core CPU, Hardware OPERating System [HOPES], external memory, and communication bus. The embedded processor is implemented in the RCSoC as a fixed (non-reconfigurable) core. HOPES manages the RTOS facilities, hardware task management, FPGA partitioning, hardware scheduling schemes and the placement methodology. The communication block serves to communicate among the hardware blocks, HOPES, and the embedded processor.

#### ReConfigurable System-on-Chip (RCSoC)

ReConfigurable System-on-Chip (RCSoC) has become a reality now, driven by the fast development of CMOS VLSI technologies. The RCSoCs are sometimes called platform FPGAs. The basic concept of the RCSoC is to make use of reconfigurable resources along with a conventional microprocessor. The main goal is to take advantage of the capabilities and features of both resources. ReConfigurable SoCs (RCSoC), consist of a processor, memory, and on-chip reconfigurable hardware parts for customization for a particular application. The RCSoC device consists of an FPGA that incorporates a CPU. In the RCSoC, the CPU is located in the fixed logic area. The hardware tasks are



executed in the reconfigurable logic area, so that CPU is not interrupted. The CPUs used in the reconfigurable computing systems can be characterized as either hardcores or softcores. Hardcores are designed into the FPGA fabric by the vendor as custom VLSI layouts and are connected to the FPGA through predefined wiring. A softcore processor core is described in a hardware description language. This gives more flexibility when interfacing the softcore processor to the rest of the devices in the reconfigurable platform. The proposed runtime reconfigurable platform incorporates a soft core CPU.

**Figure 7.1:** Reconfigurable computing target architecture

### Soft Core CPU

There are several soft CPU cores currently available that can meet the performance demands of real time applications. Commercial CPU cores such as the NIOS [66] and Microblaze [68] are FPGA vendor dependent and hence cannot be ported onto other FPGA platforms. For our runtime reconfigurable platform, an open source CPU that is capable of supporting future enhancements is proposed. The CPU needs a supporting assembler, simulator and C compiler. The LEON SPAFX Processor in VHDL [69] and



Opencores OR1200 in Verilog [65] are both open source CPUs. We propose to use the OR1200 as it fares better in terms of performance and FPGA resource requirement [65].

## 7.2 Hardware OPERating System (HOPES)

The hardware operating system for runtime reconfigurable platform is a new line of research and involves several challenges. HOPES manages the available hardware resources of the reconfigurable platform, i.e., CPU, reconfigurable hardware unit, I/O and memory. The main function of HOPES is to manage the tasks. Generally, an application will contain a combination of tasks. While software tasks can be run on the CPU, hardware tasks execute on the reconfigurable hardware area. HOPES should keep track of available hardware resources in order to be able to allocate the incoming hardware tasks to the reconfigurable area. The important components of HOPES are shown in the Figure 7.2.

**Figure 7.2:** Components of Hardware OPERating System

In this work, we proposed and developed algorithms and methodologies for the various modules in the operating system namely partitioning, scheduling and placement. Our algorithms and methodologies are very useful for the efficient deployment of the hardware operating system for runtime reconfigurable platform.

### **Partitioning Algorithm**

Our proposed initial and runtime partitioning algorithm is perfectly suited for the hardware operating system. The initial partitioning algorithm would partition the FPGA surface into varying-sized blocks based on the lower and upper bounds of the widths of the expected tasks. The runtime partitioning algorithm handles situations where the incoming task is larger or much smaller than all the available blocks by optimally merging or splitting the free blocks such that maximum area utilization is achieved. Our algorithm reduces the configuration time and the configuration power and increases the reconfigurable area utilization in the FPGA.

### **Scheduling and Placement Methodology**

The combination of scheduling and placement is the main characteristic of the problem of mapping tasks to the runtime reconfigurable platform. We proposed and developed scheduling and placement techniques for hardware tasks in the runtime reconfigurable system. The proposed scheduler supports a wide range of scheduling policies. The scheduler not only considers the task execution time, it also takes the task execution area, task configuration time and power issues into consideration during scheduling. The proposed placement technique determines tasks that are to be placed onto the various blocks of the reconfigurable FPGA and configures the task on the FPGA.

Our scheduling scheme supports different scheduling policies namely first come first serve (FCFS), shortest job first (SJF), and priority based (PB) scheduling. The proposed scheduling technique and placement methodology would help to improve the efficiency of the hardware operating system.

## Development Environment

We envisage the use of Celoxica RC200 board [70] as the implementation platform. This board integrates a Virtex XC2V1000 SRAM based FPGA and a variety of different *U 0* devices such as Ethernet, Audio, Video, SmartMedia, Parallel port, RS-232 and PS/2 keyboard and mouse. The Xilinx Virtex XC2V1000 FPGA in this board has one million gates and supports partial reconfiguration [9] methodology. Because of these various onboard resources, a wide range of applications can be executed. The Opencores OR1200 soft-core CPU should be installed in the FPGA fabric as a fixed logic. Xilinx ISE Foundation 6.1 in combination with the Modular Design Package [71] serves as the development environment for circuit stream generation: All host software are to be created with C or C++.

## 7.3 Summary

In this chapter, the architecture of the proposed reconfigurable computing platform was described. We discussed the various components of the target architecture. We also discussed the various services that should be managed by the hardware operating system [HOPES]. The methodologies developed and discussed in the earlier chapters possess the features required by HOPES and are thus eminently suitable for this purpose. We have also proposed the softcore CPU based platform for the purpose of implementing a highly integrated reconfigurable computing environment within a configurable space.

## Chapter 8

# Conclusion and Future Work

---

In the previous chapters we have presented the research work concerning the partitioning, scheduling and placement techniques for runtime reconfigurable platforms. In this chapter a summary of the work described in the thesis is presented followed by possible future areas of research.

### 8.1 Conclusion

System-on-a-chip designs on FPGA are necessitating the development of novel techniques for the hardware operating system for runtime reconfigurable platforms. Various issues, and approaches for developing a comprehensive runtime reconfigurable platform have also been presented.

We have proposed and evaluated a new initial and runtime partitioning algorithms for the reconfigurable logic area. The initial partitioning algorithm is capable of segmenting the reconfigurable FPGA area into near optimal number of blocks. The initial partitioning algorithm also serves as the groundwork for runtime partitioning and helps to achieve better hardware utilization. The runtime partitioning algorithm handles situations where the incoming task is larger or much smaller than all the available blocks by optimally merging or splitting the free blocks such that maximum area utilization is achieved. The simulation results confirm that the proposed methodology yields up to 17% and 20 % performance enhancement in terms of execution time and response time respectively, reduces the configuration delay by up to 30 %, and causes a reduction of up to 24 % in configuration power, which is an important consideration in the reconfigurable platform. The proposed partitioning algorithms have been verified fully and extensive simulations confirm that they outperform existing methods.

The effectiveness of the partitioning algorithms has been evaluated with various scheduling policies and placement techniques. In particular, we have compared the free

mode and restricted mode placement techniques for three different scheduling policies namely first come first serve (FCFS), shortest job first (SJF), and priority based (PB) scheduling. We have also presented a unified environment for the reconfigurable platform, which treats three different problems, namely partitioning, scheduling and placement strategy as its components. To the best of our knowledge, there is no evidence of similar efforts to propose an integrated environment for a reconfigurable platform. Our experimental results show that the restricted mode consistently outperforms the free mode in terms of execution time, area utilization, configuration time and configuration power. Since these parameters are important considerations in reconfigurable platforms, we have decided to employ restricted mode in subsequent works.

We have also developed application-specific partitioning and scheduling algorithms to tailor the reconfigurable space and block sizes for a given application. The proposed algorithm makes use of a path-based technique to compute the maximum reconfigurable hardware requirements. We have clearly demonstrated that the proposed algorithms maximize the reconfigurable hardware utilization and eliminate the reconfiguration time delay to further improve the overall efficiency. Simulations confirm that the proposed algorithm yields up to 34 % and up to 24% performance enhancement in terms of speed of execution and response time respectively when compared to the case where application-specific information is not considered.

A soft-core CPU based reconfigurable computing platform was also presented. We have identified the components of the target architecture. A hardware operating system [HOPES] and its various functional modules are also described. This provides for a systematic approach to implement a highly integrated reconfigurable computing environment.

## 8.2 Future Work

This research brings forth a number of possibilities for future work

- Developing a hardware operating system for runtime reconfigurable platform, which manages hardware/software task management, reconfigurable area management, memory handling, placement techniques for software and hardware tasks, and all types of communications.
- Implementation of different scheduling policies in the runtime reconfigurable platforms. It is envisaged that the task execution area, execution time, configuration time, readback time, and power issues have to be taken into consideration while implementing new scheduling policies. Also scheduling techniques for handling tasks with hard and soft deadlines in the runtime reconfigurable platform have to be developed.
- A growing class of embedded systems needs to execute multiple applications concurrently rather than just a single application. The runtime reconfigurable platform is well suited for multiple applications to execute concurrently. For this purpose, new methodologies for reconfigurable area management and task handling have to be explored. For a start, the global FPGA space can be partitioned to allocate dedicated space for each application. With this, the proposed techniques could be employed to support multiple applications.
- One of the important research areas in runtime reconfigurable platform is fault tolerance. The application space of reconfigurable devices has grown to include many applications in a single SoC platform with a strong need for fault tolerance. These systems should possess hardware redundancy to allow for continued operation in the presence of operational faults. For this purpose, fault tolerance techniques for runtime reconfigurable platform have to be developed.

## REFERENCES

- [1] C. Plessl et al. "Reconfigurable Hardware in Wearable Computing Nodes". In Proceedings of the 6th International Symposium on Wearable Computers (ISWC), IEEE Computer Society, pp. 215–222, October 2002.
- [2] IMEC Interuniversity Micro Electronic Center, T-ReCS Gecko, <http://www.imec.be>.
- [3] J.Villasenor, C.Jones, and B.Schoner. "Video communication using rapidly reconfigurable hardware," IEEE Trans. Circuits Syst. Video Technol., vol5, (6), pp.565-567,1995.
- [4] J.G. Eldredge, and B.L.Hutchings, "Density enhancement of a neural network-using FPGAs and runtime reconfiguration," IEEE workshop on FPGAs for custom computing machines, IEEE Computer Society, Los Alamitos, CA, pp. 180-188,1994.
- [5] S.Chakraborty et al. "Design Space Exploration of Network Processor Architectures," Network Processor Design: Issues and Practices, Morgan Kaufmann, Vol 1, pp.55-89 October 2002.
- [6] S.A. Guccione and M.J. Gonzalez. "Classification and performance of Reconfigurable Architectures," Proceedings of 5<sup>th</sup> International workshop on Field Programmable Logic and Applications, Lecture Notes in Computer Science, Springer Verlag, pp.439-447,1995.
- [7] Xilinx Inc., Virtex Series <http://www.xilinx.com>.
- [8] Atmel Corporation, Field Programmable Gate Array – Devices <http://www.atmel.com>
- [9] Xilinx Inc., "Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations", Application Note, <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>
- [10] S.Brown and J.Rose. "FPGA and CPLD Architectures: A Tutorial," IEEE Design & Test of computers, vol 13(2),pp. 42-57, summer 1996.
- [11] A.DeHon, "Comparing Computing Machines," in Proceedings of SPIE Vol.3526, pp. 124-133,1998.
- [12] A.DeHon, "Why Configurable Computing? The Computational Density Advantage of Configurable Architectures," at [www.cs.berkeley.edu/~amd/cs294f98/papers/Whycc.ps.zip](http://www.cs.berkeley.edu/~amd/cs294f98/papers/Whycc.ps.zip).
- [13] Xilinx Inc., "Virtex II Platform Field Programmable Gate Arrays Overview" [http://www.xilinx.com/xlnx/xil-prodcat-product.sp?title=v2\\_overview](http://www.xilinx.com/xlnx/xil-prodcat-product.sp?title=v2_overview)

[141] Altera Corporation, "Apex 20K Programmable Logic Device Family" ["http://www.altera.com/literature/ds/apex.pdf"](http://www.altera.com/literature/ds/apex.pdf).

[151] John villasenor et al, "Configurable Computing" in Scientific American, June 1997.

[16] M. Gokhale et al., "Splash: A Reconfigurable Linear Logic Array", International Conference on Parallel Processing, pp.526-532, 1990.

[171] Virtual computer Corporation, <http://www.VCC.com>

[181] Xilinx Inc., "Virtex Series Configuration Architecture User Guide" <http://direct.xilinx.com/bvdocs/appnotes/xapp51.pdf>

[19] Xilinx Inc., "The Programmable Logic Data Book", 1994.

[20] Lucent Technology Inc., "FPGA Data Book", 1998.

[21] A. DeHon. "DPGA-coupled microprocessors: Commodity ICs for the early 21st century". In Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp. 31-39, April 1994.

[22] Steve Trimberger, Khue Duong, and Bob Conn. "Architecture issues and solutions for a high-capacity FPGA", In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 3-9, February 1997.

[23] C. Ebeling, D.C. Cronquist, P. Franklin. "RaPiD – Reconfigurable Pipelined Datapath", In Lecture Notes in Computer Science 1142—Field programmable Logic: Smart Applications, New Paradigms and Compilers. R. W. Hartenstein, M. Glesner, Eds. Berlin, Germany: Springer-Verlag, pp. 126-135, 1996.

[24] H. Schmit, "Incremental Reconfiguration for Pipelined Applications", IEEE Symposium on FPGAs for Custom Computing Machines, pp. 47- 55, 1997.

[25] S. Hauck, T. Fry, M. Hosler, J. Kao. "The Chimaera Reconfigurable Functional Unit", IEEE Symposium on FPGAs for Custom Computing Machines, pp. 87-96, 1997.

[26] B. L. Hutchings and M. J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications," International Workshop on Field Programmable Logic and Applications, Oxford, pp.293-202, 1995.

[27] P. Lysaght and J. Dunlop, "Dynamic Reconfiguration of Field Programmable Gate Arrays," 3rd. International Workshop on Field Programmable Logic and Applications (FPL93), Oxford, 1993.

[28] J. Vuillemin, et al., "Programmable Active Memories: Reconfigurable Systems Come of Age," IEEE Transactions on VLSI Systems, vol.4, no. 1, pp. 56-69, 1996.



- [29] Goldstein, S. C., H. Schmit, M. Budiu, S. Cadambi, M. Moe and R. R. Taylor. "Piperech: A reconfigurable Architecture and Compiler", Computer: Innovative Technology for Computer Professionals, IEEE Computer Society, vol 33, no 4, pp. 70-77, April 2000.
- [30] Hauck, S., T. W. Fry, M. M. Hosler, and J. P. Kao. "The Chimaera Reconfigurable Functional Unit", Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97), pp. 87-96, April 1997.
- [31] Hauser, J. R., and J. Wawrzynek. "GARP: A MIPS Processor with a Reconfigurable Coprocessor", Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'97), pp. 12-21, April 1997.
- [32] Callahan, T. J., J. R. Hauser and J. Wawrzyneck. "The Garp Architecture and C Compiler", Computer: Innovative Technology for Computer Professionals, vol 33, no 4, IEEE Computer Society, pp. 62-69, April 2000.
- [33] Lu, G., H. Singh, M. Lee, N. Bagherzadeh, F. Kurdahi and E. M. C. Filho. "The MorphoSys Parallel Reconfigurable System", Proceedings of Euro-Par 99, Toulouse, France, September 99.
- [34] Miyamori, T., and K. Olukotun. "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications", Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98), pp. 2-11, April 1998.
- [35] Razdan, R., and M. D. Smith. "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", International Symposium on Microarchitecture (MICRO 27), pp.172-180, November 1994.
- [36] Haynes, S. D., J. Stone, P. Y. K. Cheung, W. Luk. "Video Image Processing with the Sonic Architecture", Computer: Innovative Technology for Computer Professionals, Vol 33, Number 4, IEEE Computer Society, pp. 50- 57, April 2000.
- [37] Wittig, R. D., and P. Chow. "Onechip: An FPGA Processor with Reconfigurable Logic", In proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'96), pp. 126-135, March 1996.
- [38] Grant Wigley and David Kearney, "The Development of an Operating System for Reconfigurable Computing". In proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM). IEEE CS Press, April 2001.
- [39] G. Wigley and D. Kearney. "Research Issues in Operating Systems for Reconfigurable Computing", In proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms (ERSA), CSREA Press, pp.10-16, June 2002.

- [40] G. Brebner, -- A Virtual Hardware Operating System for the Xilinx XC6200 “, In proceedings of the 6th International Workshop on Field Programmable Logic and Applications (*FPL*), Springer, pp. 327–336. 1996.
- [41] G. Brebner and O. Diessel, “Chip-Based Reconfigurable Task Management”, In Proceedings of the 11th International Workshop on Field Programmable Gate Arrays (FPL), Springer, pp.182–191.2001.
- [42] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins “Designing an Operating System for a Heterogeneous Reconfigurable SoC” Reconfigurable Architectures Workshop (RAW) April 2003.
- [43] H. Simmler, L. Levinson, and R. Manner, -- Multitasking on FPGA Coprocessors”. In proceedings of the 10th International Workshop on Field Programmable Gate Arrays (FPL), Springer, pp. 121–130.2000.
- [44] **J.-Y.** Mignolet et al, “Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip,” In proceedings of Design, Automation and Test in Europe (DATE), IEEE Computer Society, pp.986–991. March 2003.
- [45] Herbert Walder, Marco Platzner, “Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations”, Engineering of Reconfigurable Systems and Algorithms, pp.284-287, 2003.
- [46] Herbert Walder, Marco Platzner, “Online Scheduling for Block-Partitioned Reconfigurable Devices”, pp. 10290-10295, DATE 2003
- [47] O.Diessel and Hossam ElGindy, -- On Scheduling Dynamic FPGA Reconfigurations”, In proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems (PART), pp.191–200,1998.
- [48] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and **B.** Schmidt, “Dynamic scheduling of tasks on partially reconfigurable FPGAs,” In IEE Proceedings on Computers and Digital Techniques, vol 147, pp.181–188, May 2000.
- [49] Soheil et al, “An Optimal Algorithm for Minimizing Run-Time Reconfiguration Delay,” ACM Transactions on Embedded Computing Systems, Vol 3, pp.237-256, May 2004.
- [50] K.Compton et al. “Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing,” IEEE Trans. Very Large Scale Integration (VLSI) Systems, voll0, no.3, pp.116-122, 2002.

[51] H.Kalte et al., "System-on-Chip Approach Enabling Online Fine-Grained 1D-Placement," 18<sup>th</sup> IEEE Parallel and Distributed Processing Symposium, April 2004.

[52] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems", In IEEE Design and Test of Computers, vol 17, pp.68–83, 2000.

[53] Herbert Walder, Christoph Steiger, Marco Platzner, "Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D Hashing", pp. 178, IPDPS 2003.

[54] Herbert Walder and Marco Platzner, "Non-preemptive Multitasking on **FPGA**: Task Placement and Footprint Transform", In proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA), CSREA Press, pp.24–30. June 2002.

1551 Michael Lerjen, "Reconfigurable Bluetooth Ethernet Bridge," Master's thesis, ETH Zurich, 2002.

[56] Xilinx Inc., Virtex Core Generator.

[57] A.Aggarwal et al., "Parallel depth-first search in general directed graphs" Proceedings of the twenty-first annual ACM symposium on Theory of computing, USA, pp.297-308, 1989.

**[58]** Pedro Merino, Margarida Jacome, and Juan Carlos Lopez, "A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems," In proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), IEEE CS Press, pp.324–325. April 1998.

[59] Li Shang et al. "Dynamic Power Consumption in Virtex- II FPGA Family," FPGA'02, California, USA, pp.157-164, February-2002.

[60] Wayne Wolf and Jorgen Stanunstrup, "Hardware/Software co-design Principles and Practice," Kluwer Academic Publishers, 1997.

[61] R. Camposano, "Path-Based Scheduling for Synthesis," IEEE Trans. On Computer-Aided Design, vol 10, no.1, pp.85-93, January 1991.

[62] K Ramamritham et al., "Efficient Scheduling algorithms for Real-Time Multiprocessor systems," IEEE Trans. Parallel and Distributed Systems, vol. 1, no.2, pp. 184-194, April. 1990.

[63] J.Xu and L.Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," IEEE Trans. Software Eng., vol. 16, no.3, pp.360-369, March 1990.

[64] K.Ramamritham, "Allocation and Scheduling of Precedence Related Periodic Tasks," IEEE Trans. Parallel and Distributed Systems, vol. 6, no.4, pp.412-420, April 1995.

[65] OpenRISC Architecture <http://www.opencores.org/projects/or1KArchitecture>

[66] Altera Corporation, Nios Embedded Processor Homepage <http://www.altera.com/products/ip/processors/nios/nio-index.html>

[67] H.Kalte et al, "System-on-Chip Approach Enabling Online Fine-Grained ID-Placement," 18th IEEE Parallel and Distributed Processing Symposium, April 2004.

[68] Xilinx Inc., MicroBlaze Soft Processor Homepage <http://www.xilinx.com/xlnx/xil-prodcat-product.jsp?title=microblaze>

[69] J. Gaisler, "LEON SPARC Processor", <http://www.gaisler.com/main.html>, June 2001.

[70] Celoxica Limited, Celoxica RC200 Development Board, <http://www.celoxica.com>.

[71] Xilinx Inc., Advanced Design Techniques, Modular Design, <http://www.xilinx.com>.

[72] Amphion Semiconductor Ltd., [www.amphion.com](http://www.amphion.com)

[73] Michael Barr, "Introduction to Rate Monotonic Scheduling" Embedded System Programming, vol.15, no.3, March 2002 at [www.embedded.com/story/OEG20020221S0089](http://www.embedded.com/story/OEG20020221S0089)

[74] J.Rose et al, "Architecture of Field-Programmable Gate Arrays", Proceedings of the IEEE, vol. 81, no.7, pp.1013-1029, July 1993.

[75] M.J.S. Smith, "Application-Specific Integrated Circuits" - Addison-Wesley Longman Inc., 1997

[76] Pradeep Adhipathi, "Model based approach to Hardware/Software Partitioning of SOC Designs", Master Thesis, Virginia Polytechnic Institute and State University, June 2004.

[77] Greg Stitt et al, "Dynamic Hardware/Software Partitioning: A First Approach", Anaheim, California, USA, pp.250-255 DAC 2003

[78] Giovanni De Micheli et al., "Hardware/Software CO-Design", Proceedings of the IEEE vol. 85, no.3, pp.349-365, March 1997