A Study on a Multitasking Environment for Dynamically Reconfigurable Processors

VU MANH TUAN

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Science for Open and Environmental Systems
Graduate School of Science and Technology
Keio University

2009

MICROSOFT 1005

Preface

This thesis presents fundamental work in the area of multitasking on dynamically reconfigurable processors. The main objective is to investigate necessary mechanisms in order to build a true multitasking execution environment on such devices.

Reconfigurable devices promise to be a valuable alternative to conventional computing devices such as general-purpose processors and application specific integrated circuits. The hardware circuit of reconfigurable architectures is not static but adapted to the target applications at the run time. Through dynamic hardware customization, reconfigurable architectures potentially achieve a better performance than microprocessors while maintaining a higher level of flexibility than application specific integrated circuits.

Due to the increase of logic capacity of modern fields-programmable gate arrays expanding beyond millions of system gates, and through the broad use of coarse-grained dynamic reconfiguration, it has become feasible for several applications to share a single high density device. In other words, reconfigurable hardware devices can now execute several hardware tasks in parallel. However, developing applications that share a device is difficult as the current design flow assumes the exclusive use of the underlying reconfigurable resources, and critical mechanisms for a multitasking environment on such devices are still immature.

The use and implementation of similar mechanisms modeled on a microprocessor-based software operating system would allow the full benefits of multitasking on dynamic reconfiguration systems to be realized. In this thesis, based on equivalent concepts found in the software domain with specific characteristics of hardware implementation, several necessary mechanisms to support a multitasking environment on dynamically reconfigurable processors are presented.

Specifically, I consider following issues: (1) how to improve the throughput in data flow driven applications by using an appropriate application model and a mapping algorithm to map a target application onto a underlying coarse-grained reconfigurable device consisting of multiple hardware execution units, (2) how to implement a preemption mechanism for hardware tasks, (3) how to extend the multi-thread execution model in a tile-based architecture into a multicore architecture, and the impact of the core size on performance and internal fragmentation in a multicore reconfigurable architecture.

Efficiently mapping applications partitioned into multiple threads of control or tasks onto a coarse-grained dynamically reconfigurable processing array is crucial for further exploiting the par-

allelism of applications. Basically, a certain speedup of reconfigurable devices over traditional microprocessors can be achieved by exploiting the inherent parallelism of a target application at a larger scale apart from instruction-level parallelism. However, in many cases, the parallelism of an application is not enough for the reconfigurable array to be utilized efficiently. In such a condition, one of the methods to improve performance is to make the best use of stream-level pipelined execution; or in other words, to exploit the task-level parallelism. That is, a total process is partitioned into small independent sequential processes, which are separately mapped onto a coarse-grained underlying reconfigurable array architecture comprising of multiple hardware execution units. By investigating the trade-off between the size of a tile group and delay as well as execution time, an optimized mapping algorithm is proposed and evaluated compared with the single-process execution model and other mapping versions.

Effectively implementing a hardware task preemption scheme, which is a critical mechanism in multitasking operating systems where an executing task is temporarily suspended, and a new task is executed or a previously interrupted task is resumed, on coarse-grained dynamically reconfigurable processors is another mechanism toward a multitasking environment. Different from well-known task preemption in software operating systems based on microprocessors, hardware task preemption accompanies problems on how to suspend and resume hardware execution, how to save and restore context data within a certain overhead. Based on the resource requirement and the state transition graph of a target application, the concept of preemption points is introduced and a static preemption approach is proposed. Basically, an application can only be preempted at certain computation states where resource usage is small. Moreover, the steps of the proposed method are integrated into the design flow in order to help developers at the design time.

The development of multicore architectures in microprocessors has recently become prevalent and achieved advantages over traditional single-core devices. Multicore architectures have proved to be a suitable platform for a true multitasking environment at the task level where tasks can execute in parallel. I introduce a multicore reconfigurable architecture in which cores have a certain size and are connected by an interconnection network composed of routers. Mapping multiple hardware tasks onto the proposed architecture and designing an efficient inter-task communication mechanism become important research problems. The proposed on-chip network will act not only as an efficient communication mechanism between tasks (both hardware and software tasks), but also as a support method for mapping processes onto the underlying reconfigurable array. First of all, a tile-based architecture and the proposed multicore architecture are examined to see how each of those affects on the performance of target applications. The channel bit width to transfer data in the interconnection network of the multicore architecture is taken into account. Next, the size of core in the proposed multicore reconfigurable architecture is an important factor in balancing performance and resource utilization. Therefore, this is evaluated by implementing real applications on the proposed architecture and measuring correspondent parameters.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Professor Hideharu Amano, my supervisor at Keio University for his guidance, encouragement, support, and vision throughout my Ph.D. program. In addition to academic guidance, he has been a main motivating force in the development of my professional skills and perspective of research. In the long road to my dissertation, he provided me with extraordinary opportunities to learn professional knowledge and achieve research experience. I am indebted to him for showing me how to look forward and think without limitations. I also learned how to bring out research issues, think in a systematic ways, use different tools to assist research processes, and write scholarly papers in a professional manner. Doing research is a difficult job, especially at the beginning when you are not sure where to start from. More importantly, the difficulty could be double or more when you do not have enough background in the field of reconfigurable computing, which is a new and still immature area with limited documents. Apart from an excellent researcher, Professor Amano has a systematic way to organize his Lab and guide his students to overcome research problems. When joining Amano Laboratory, I had almost little knowledge and experience in computer architecture, particularly in reconfigurable computing. From very first exercise using Musketeer to design and develop stream applications on DRP architecture, Professor Amano gave me many useful advices and guided me to write research papers from simple issues, whose content seemed not to be important at the first sight. In addition, whenever problems appearing in daily life during the period of time I live in Japan, he has been very kind to spend his time and effort to help me solve such problems.

I am grateful to the committee members of doctoral evaluation, Professor Yoshikazu Yamamoto, Professor Takahiro Yakoh, and Professor Kenji Kono for their valuable reviews and comments.

A special thank to Japanese government for funding and giving me a great opportunity to come to Japan, to study in one of the most excellent places, Keio University, and to pursue a Ph.D. Degree in computer architecture. Receiving Monbukagakusho scholarship would be one of the most important turning points in my life to start a long way with challenges toward the dissertation and further. I also want to thank Keio Leading-edge Laboratory of Science and Technology (KLL) for providing financial support to my research during the last two years of my course.

It has been a wonderful experience to participate in collaborative research efforts with my fellow graduate students in WASMII group during my Ph.D. course. Masayasu Suzuki, who is now working in Sony corporation, was very kindly to help me start first lessons and experiences on Verilog. He

organized and attended seminars even after graduated and went to Sony for working to let me touch first knowledge on the field of reconfigurable computing, and to show me a correct way to carry out research. I also received a great deal of support from Dr. Yohei Hasegawa. He often read my research papers and gave me useful comments before papers were submitted. Dr. Hiroki Matsutaini provided me with his tools for building a network on chip, which I used in my research.

Additional thanks goes to NEC Corporation and Japan Science and Technology Agency (JST) for providing support to this research. The DRP-1 device and design tools were provided by NEC Electronics Corporation and NEC Corporation. Since my research closely relates to DRP architecture, without such an excellent device, I would not be able to achieve crucial results and complete the thesis.

Keio University has supported me in this work by providing excellent research facilities and working environment. I want to thank the management and many people especially those at the International Center of Yagami campus.

From bottom of my heart, I would like to send a special thank you to my parents for all their love, support, and psychological assistance they have given me over a long period. They always care for me, give me constant encouragement throughout my life, and have faith in me. My younger sister, Vu Thanh Trang, assisted me from the first Japanese lesson before I went to Japan. She did not send me a lot of emails, but I know she always hopes me to complete my work successfully.

I am grateful to my wife, Le Minh Phuong, for her patience especially for the long years of waiting for the completion of this work. My second son was born when I was already in Japan, and my wife has looked after him alone for such a long duration. Although staying in Vietnam, they were my inspiration to work hard with a hope that the faster I could finish the course, the earlier I would be able to return, to see their faces and smiles, and to talk to them face-to-face. My first son, Vu Tuan Minh, who has stayed with me in Japan for almost three years, also receive my heart-felt thanks for being a special encouragement for me to study Japanese language and live with joy and love in Japan.

Last but not least, my friends who used to be with me in Soshigaya International House played an important role in the duration I attended the Ph.D. course. Some of them have returned to Vietnam, others are still in Japan, but their help and encouragement add more strength to me to complete the work.

Vu Manh Tuan

Yokohama, Japan
October 2008

Contents

Preface i

Acknowledgments iii

1 Introduction 1

1.1 Motivation . . . 4

1.2 Contribution of the Thesis . . . 5

1.3 Thesis Organization . . . 5

2 Methodology and System Architecture 7

2.1 Methodology . . . 7

2.2 Concepts . . . 9

2.2.1 Application . . . 9

2.2.2 Hardware task . . . 10

2.2.3 Computational core . . . 12

2.2.4 Inter-task communication . . . 14

2.3 System Architecture . . . 14

2.4 Multitasking Execution Model . . . 16

2.4.1 Time-sharing, non-space partitioning . . . 17

2.4.2 Non-time sharing, static partitioning . . . 17

2.4.3 Time-sharing, two-dimensional partitioning . . . 18

2.4.4 Time-sharing, multicore partitioning . . . 19

3 Coarse-grained Dynamically Reconfigurable Processing Arrays 21

3.1 DRPA Architecture Overview . . . 21

3.1.1 Coarse-grained processing array . . . 21

3.1.2 Interconnection structure . . . 22

3.1.3 Dynamic reconfiguration method . . . 23

3.1.4 Coupling between CPU and DRPA . . . 25

3.1.5 C-based programming methodology . . . 26

3.2 Review of DRPAs . . . 27

3.2.1 Chameleon CS2112 . . . 27
3.2.2 PACT XPP-III . . . 29
3.2.3 NEC Electronics DRP-1 . . . 30
3.2.4 IPFlex DAPDNA-2 . . . 32
3.2.5 Hitachi FE-GA . . . 34
3.2.6 Elixent D-Fabrix . . . 36
3.2.7 Rapport Kilocore . . . 38
3.2.8 IMEC ADRES . . . 39
3.2.9 Stretch S5/S6 SCP Engine . . . 41
3.2.10 Fujitsu Cluster Architecture . . . 43
3.2.11 MuCCRA platform . . . 44
3.3 Summary . . . 47

4 Hardware Task Mapping 50
4.1 Problem . . . 50
4.2 Related Work . . . 51
4.3 Target Architecture and Application Model . . . 52
4.3.1 Target architecture . . . 52
4.3.2 Target application model . . . 53
4.3.3 Goal of mapping . . . 53
4.4 Mapping Algorithm . . . 54
4.4.1 Target task graphs . . . 54
4.4.2 Target architecture and task mapping . . . 55
4.4.3 Mapping algorithm . . . 56
4.5 Target Device . . . 59
4.5.1 Device . . . 59
4.5.2 Mapping applications onto DRP-1 . . . 59
4.6 Evaluation . . . 60
4.6.1 Target applications . . . 60
4.6.2 Mapping versions . . . 62
4.6.3 Implementation results . . . 63
4.6.4 Throughput . . . 63
4.6.5 Execution time . . . 64
4.6.6 Area utilization . . . 65
4.6.7 Two methods for topological mapping . . . 65
4.7 Conclusion . . . 66

5 Hardware Task Preemption **67**
5.1 Problem . . . 67
5.2 Related Work and Research Contribution . . . 68
5.2.1 Related work . . . 68
5.2.2 Research contribution . . . 68
5.3 Preemption Analysis . . . 69
5.3.1 Task switching . . . 69
5.3.2 Approach . . . 70
5.3.3 State transition graph . . . 72
5.4 Preemption Algorithms . . . 72
5.4.1 System design flow . . . 72
5.4.2 Preemption algorithm . . . 74
5.4.3 Illustrative example . . . 78
5.5 Target Device . . . 80
5.6 Evaluation . . . 80
5.6.1 Target applications . . . 80
5.6.2 Hardware overhead . . . 81
5.6.3 Preemption latency . . . 83
5.6.4 Hardware overhead vs. preemption latency . . . 83
5.7 Conclusion . . . 84

6 Multicore Reconfigurable Architecture **86**
6.1 Problem . . . 86
6.2 Related Work . . . 88
6.3 Evaluated Architectures . . . 88
6.3.1 Target Device . . . 88
6.3.2 Tile-based architecture . . . 89
6.3.3 Multicore architecture . . . 90
6.3.4 Application model . . . 92
6.4 Evaluation . . . 92
6.4.1 Simulation environment . . . 92
6.4.2 Two architectures . . . 92
6.4.3 Evaluation with different core sizes . . . 94
6.4.4 Internal fragmentation . . . 97
6.5 Conclusion . . . 98

7 Conclusion and Future Work **99**
7.1 Thesis Summary . . . 99

7.2 Suggestions for Future Research 101

Abbreviation and Acronyms **103**

Bibliography **105**

Publications **114**

List of Tables

2.1 Research methodology 8
2.2 A common plan 8
2.3 Methodology paths used in the thesis 9

3.1 Types of DAPDNA-2 PE 34
3.2 Summary features of surveyed Dynamically Reconfigurable Processors 48

4.1 Example of possible target architectures 53
4.2 Implementation results of target applications 61
4.3 Time for topological mapping 66

5.1 Target applications and evaluation results 82

6.1 Router implementation 91
6.2 Throughput of two evaluated architecture 94
6.3 Implementation results of target applications 96

List of Figures

2.1 Application's task graph 10
2.2 Hardware task model 11
2.3 A heterogeneous SoC 12
2.4 Multicore architecture studied in this thesis 13
2.5 System architecture 15
2.6 Time-sharing, non-space partitioning model 17
2.7 Non-time sharing, static partitioning model 18
2.8 Time-sharing, two-dimensional partitioning model 19
2.9 Time-sharing, multicore partitioning model 20

3.1 PE array of PACT-XPP 22
3.2 Example of a PE (RC of Morphosys) 22
3.3 Island-style 23
3.4 Direct interconnection 23
3.5 Switch matrix 23
3.6 Configuration Delivery Scheme 24
3.7 Multicontext Scheme 24
3.8 Coupling of Reconfigurable Fabric and Microprocessor 26
3.9 CS2112 reconfigurable fabric 27
3.10 CS2112 datapath unit (DPU) 27
3.11 XPP-III Core Architecture Sample 28
3.12 XPP-III ALU-PAE Architecture 30
3.13 DRP Tile Architecture 31
3.14 DRP PE Architecture 31
3.15 DRP-1 Architecture 32
3.16 Programming Flow of DRP 32
3.17 DAPDNA-2 Architecture 33
3.18 FE-GA Architecture 35
3.19 FE-GA ALU Cell Architecture 36
3.20 FE-GA LS Cell Operation 36

3.21 Chessboard-style ALU Array . . . 37
3.22 D-Fabrix ALU and Switchbox . . . 37
3.23 ET1 Architecture with MeP core and D-Fabrix . . . 38
3.24 Virtual Pipeline Model . . . 39
3.25 Kilocore KC256 Architecture . . . 39
3.26 Kilocore KC256 PE Architecture . . . 40
3.27 ADRES Core Architecture . . . 41
3.28 ADRES Reconfigurable Cell . . . 42
3.29 S6000 Architecture . . . 43
3.30 S6 SCP Engine . . . 43
3.31 Fujitsu Cluster Architecture . . . 44
3.32 Fujitsu Cluster Group . . . 45
3.33 MuCCRA-1 Architecture . . . 46
3.34 PE architecture of MuCCRA-1 . . . 46
3.35 MuCCRA-2 Architecture . . . 46
3.36 MuCCRA-D Architecture . . . 46
3.37 PE architecture of MuCCRA-D . . . 47

4.1 General design flow . . . 52
4.2 Target DRPA . . . 52
4.3 Task mapping for JPEG encoder . . . 54
4.4 Target task graph . . . 55
4.5 Delay and execution time vs. number of Tiles . . . 55
4.6 Tile connection patterns for a TG . . . 56
4.7 Tile assignment . . . 58
4.8 Example of APME approach . . . 58
4.9 PKN models of target applications . . . 60
4.10 Execution time vs. Number of data blocks . . . 64

5.1 Task switching . . . 69
5.2 Memory and Register usages vs. Computation steps . . . 71
5.3 System design flow . . . 73
5.4 Input and output information for proposed method . . . 74
5.5 Proposed solution . . . 79
5.6 Example code . . . 79
5.7 Hardware overhead . . . 83
5.8 Maximum preemption latency . . . 84
5.9 Hardware overhead vs. Preemption latency . . . 84

6.1 Tile-based architecture 90
6.2 Multicore architecture 90
6.3 Router architecture 91
6.4 Representation of JPEG encoder 91
6.5 Simulation environment 93
6.6 Implementation variants 95

Chapter 1

Introduction

The aim of this thesis is to investigate a suitable architecture based on feasible mechanisms for developing a true multitasking environment on dynamically reconfigurable processing arrays (DRPAs).

Recent years, a lot of effort in research and development has been dedicated to computer and processor architectures to satisfy ever-growing demands for higher performance, better power consumption and lower cost. One of the fundamental trade-offs in the design of computing systems involves the balance between flexibility and performance. On one hand, general-purpose processors (GPPs) and digital signal processors (DSPs), which are built around an instruction-set architecture, provide the possibility of processing arbitrary computations due to their general architectural concept. However, these types of processors are rather inefficient regarding performance and power consumption. On the other hand, application-specific integrated circuits (ASICs), which contain dedicated circuits specialized to a particular set of tasks, represent a kind of architecture that is optimized for a predetermined set of tasks. Nonetheless, while being very efficient regarding performance and power consumption, but ASICs lack flexibility as no programmable resources are provided.

The increasingly higher integration of transistors at an increasingly lower cost per transistor has resulted in a capability of putting over billion transistors on a single chip. This progress has led for designers to a System-on-Chip (SoC) design methodology in a wide variety of application areas. Recently, SoCs have been in widespread use as the total solution for single-chip system integration. The components in such SoCs may include embedded microprocessors, memory blocks, external interfaces, peripherals, and application-specific customized functional blocks.

Reconfigurable computing has been intended to be an alternative to possibly bridge the gap between above traditional approaches [1]. Reconfigurable systems are implemented with programmable logic in order to alter the hardware circuits on demand to achieve potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. Reconfigurable computing came to realization through the introduction of FPGAs [2,3] in mid eighties. Like ASICs, FPGAs are distinguished by their ability to directly implement specialized circuitry in hardware. Additionally, like GPPs, they contain programmable resources that may be easily modified after field deployment in response to changing operational parameters, functions or data sets.

Implementing applications on a reconfigurable fabric allows to improve performance while minimizing power consumption especially for stream applications. Such types of applications have special characteristics that the instruction stream does not change fast, but the data stream changes constantly. For example, a video encoder/decoder, often handles a very large amount of data, but the groups of instructions that execute on these large data streams are usually small. In comparing with traditional applications like a word processor running on a GPPs, many different pieces of code for performing variety of functionalities such as menus, spell checkers and formats operate on a single document file. In this application, the data stream does not change very much; but, the instruction stream is changing very often. GPPs are usually designed with such a type of applications in mind. Another difference between traditional applications and stream applications is their bandwidth requirements. Since a traditional application can fill up its cache with all instructions and data without concerning to fetch more very soon, GPPs designed for such applications usually have large caches connected by relatively low bandwidth buses. On the other hand, stream applications can be designed with smaller caches, but since they transfer so much data they need much more bandwidth. A reconfigurable platform consists of a large number of logic blocks or reconfigurable functional units, which are well-matched to the high-sample rates and distributed computation often required by stream applications. In other words, the characteristics of stream applications that have the deep impact on performance and be appropriate to implement on reconfigurable platforms are data parallelism, amenability to pipelining, arithmetic complexity and simple control requirements.

Early reconfigurable computing platforms proposed and built based on low density FPGAs with small amounts of on-board memory [4, 5, 6] could only accommodate one application, resulting in a single task environment due to limited resources. With the fast pace development of semiconductor technology predicted by the International Technology Roadmap for Semiconductors [7] for the CMOS process technology to be able to scaled down to 25nm and further to 22nm within the next seven years, the number of processing cores integrated on an SoC has been steadily increased. For example, as many as 1,024 functional blocks, each of which consists of a MIPS-like processor and a cache memory, will be realized on a single chip with a 35nm technology [8]. Modern FPGAs containing millions of system-gates [9, 10] or hundreds of thousands of logic cells [11, 12, 13] and large course-grained dynamically reconfigurable processors [14] realize the possibility of sharing a piece of reconfigurable hardware among multiple concurrently executing applications. This could potentially increase the resource utilization of such reconfigurable devices and decrease response times. In addition, reconfigurable systems that are composed of multiple FPGA chips interconnected on a single processing board have been introduced for research and commerce. This is to provide for applications that are too large to be implemented within a single FPGA, but may be partitioned over the multiple FPGAs. For multi-FPGA systems, because of the need for efficient communication between the FPGAs, determining the inter-chip routing topology is a very important step in the design process. Furthermore, it is important to efficiently partition a design into different FPGA chips.

The research, development and implementation of a multitasking environment on GPPs have

come a long way from simple *time sharing* and *cooperative multitasking* systems to *preemptive multitasking*, *real-time multitasking* and recently *multitasking on multi-processor and multicore architectures*. Nonetheless, multitasking is still immature since it is a large and diverse field. The newest trend in the processor architecture is a multicore structure, which now dominates the chip making industry [15]. In spite of that, there is still a problem for current software models because today's operating systems that use threads and cache coherency snooping will not scale well to such multicore devices.

The concept of multitasking on dynamically reconfigurable processors, in combination with an appropriate operating system support, enables efficient sharing of resources due to adequate task scheduling, allows dynamic changes of the application due to introduction of new tasks during runtime and greatly simplifies application development since tasks can be developed quite independent from each other. Specifically, Multitasking aims at providing an appropriate mechanism to allow multiple applications to efficiently share a piece of reconfigurable hardware. Nonetheless, building such an environment on hardware has unique characteristics, which make this be different from and more difficult than that on GPPs. Some problems involving multitasking on dynamically reconfigurable hardware are as follows:

- How to suspend a running hardware task, and some times later to resume it?
- How to capture the state data of a hardware task that is suspended, and to restore the state data when the task is restarted?
- How to map the tasks of a target application onto hardware execution units?
- What is the most suitable model of target applications for dynamically reconfigurable devices in order to efficiently exploit multiple execution units on such devices?
- How can hardware tasks communicate to one another and to software tasks?
- How to relocate hardware tasks to different execution units?
- What is the most relevant architecture for dynamically reconfigurable processors to support a multitasking environment?
- How does an inter-task communication mechanism affect on the performance of target applications?
- How to select the most relevant core size in a multicore architecture?

Addressing the design of such problems for coarse-grained DRPA platforms is the aim of this thesis.

1.1 Motivation

Multitasking is useful if the application is composed of several well separated computational tasks that should be processed quasi simultaneously on limited hardware resources. The concept of multitasking, in combination with the corresponding operating system (OS) support, enables efficient sharing of resources with adequate task scheduling, allows dynamic changes of applications with the introduction of new tasks at run-time, and greatly simplifies application development because tasks can be developed quite independent from each other.

Challenges

When designing a system with reconfigurable hardware, a key challenge is to implement all necessary functionalities to meet the required performance, but at reasonable cost. Therefore, the system resources have to be used efficiently. For processor-based systems, multitasking is a method to share resources. With the use of reconfigurable architectures in SoCs to accelerate certain computations formerly implemented on GPPs or in ASICs, it is critical to develop appropriate techniques to DRPAs in order to use their resources efficiently. DRPAs pose certain challenges as follows:

1. *Design of novel reconfigurable architectures for multitasking:* most of available commercial reconfigurable architectures do not provide native support for identifying, capturing and restoring the state data of a hardware task before and after interruption, which are basic requirements for preemptive multitasking. Although partial reconfiguration on FPGAs, which allows to load a new task into a part of the device while other tasks keep running on other parts, has been introduced and employed, it has only limited support.

2. *Task partitioning and mapping:* task partitioning is often used to split an application into parts; and, task mapping is to map such parts into different hardware execution units of a target device. As the number of parts an application can be divided into and the number of hardware execution units for a certain device are greatly varying, a mapping method to improve performance and device utilization is crucial.

3. *Inter-task communication:* a DRPA can be considered as a multicore architecture with hardware execution units functioned as cores. As the run-time, tasks are assigned to cores for execution. Here, the question of how large the size of cores is and how cores should be connected in order for tasks to exchange data becomes the key for efficiently exploiting DRPA resources.

4. *A runtime environment for multitasking:* multitasking on reconfigurable systems requires certain support from a run-time environment or an OS. Such an OS needs to manage resources and controls the execution of hardware at the run-time. Furthermore, the OS has to provide application programming interfaces for designers to abstract the details of the underlying hardware.

1.2 Contribution of the Thesis

The main contribution of the thesis is to try to solve the first three challenges mentioned above toward coarse-grained dynamically reconfiguration processors. By introducing, evaluating and discussing crucial architectural components for, the thesis tries to give an outline of a realizable coarse-grained dynamically reconfigurable platform that supports the multitasking capability.

- A systematic method for mapping a target application modeled as a Kahn Process Network onto a coarse-grained DRPA to exploit a task-level pipeline technique toward improving throughput is proposed and evaluated [16]. Besides exploiting parallelism within an application to achieve high throughput in data-flow driven applications, the proposed method tries to apply the computation model of processing multiple computations in parallel without decreasing performance by making the best use of the stream-level pipeline execution.

- A preemption algorithm for inserting preemption points into the scheduled task graph of a target application subject to preemption latency constraints in order to minimize the hardware overhead is developed and investigated [17]. Important factors when implementing a preemption mechanism for hardware tasks such as preemption latency, hardware overhead and performance degradation are taken into account and quantitatively evaluated.

- A multicore reconfigurable architecture consisting of multiple small computational cores connected by an interconnection network is introduced. At first, the thesis proposes a general system architecture and related matters, then a simulation environment where experiments and evaluations of the proposed system may be carried out is described. Next, a comparison of a tile-based architecture and a multicore architecture in terms of performance is examined. Last, based on the proposed architecture, an evaluation with different core sizes is implemented in order to find out how the size of cores in a homogeneous system influences on the performance and the internal fragmentation of target applications.

1.3 Thesis Organization

Chapter 2 *Methodology and System Architecture* presents the methodology and system architecture proposed in this thesis. First, the methodology is outlined, and the organization of the thesis is shown according to recommendations from the methodology. Next, several concepts and definitions are described since they might cause confused for their similarities in the microprocessor field. Then, an outline of the system architecture is described. Several multitasking models based on resource sharing are also introduced.

Chapter 3 *Coarse-grained Dynamically Reconfigurable Processing Arrays* provides background to research carried out in this thesis and related work. A general architecture of DRPAs that subject to be the target devices of the thesis is discussed. A brief introductions of currently available devices

is also mentioned and categorized according to certain criteria. The target architecture using in Chapter 4, Chapter 5 and Chapter 6, which is NEC Electronics DRP, is also described in this chapter.

Chapter 4 *Hardware Task Mapping* proposes and investigates a systematic method for mapping an application modeled as a Kahn Process Network onto a target DRPA in order to enhance throughput by trying to exploit more inherent parallelism of target applications. The proposed mapping algorithm is evaluated with several real applications to show the impact of different versions mapped onto the target device on performance.

Chapter 5 *Hardware Task Preemption* aims at studying a general method for capturing the state data of hardware tasks targeting coarse-grained DRPAs. By forming the state transition graph of a target application and using the report on resource usages, the proposed algorithm tries to generate a list of preemption points, where the application is allowed to be preempted, subject to user-specified preemption latency.

Chapter 6 *Multicore Reconfigurable Architecture* extends the task-level pipelined execution model examined in Chapter 4 by introducing a multicore reconfigurable architecture composed of multiple computational cores connected by an interconnection network. Two architecture, a tile-based architecture and a multicore architecture, are compared in terms of performance to see the effect of inter-task communication methods. Then, the problem of how the size of cores in a multicore reconfigurable architecture influences on the performance and the resource utilization of target applications is investigated.

Chapter 7 *Conclusion and Future Work* finalizes the thesis with a summary, draws conclusions and presents suggestions for future exploration.

Chapter 2

Methodology and System Architecture

2.1 Methodology

This chapter introduces and exploits a type of system engineering methodology to guide studies in the thesis. The methodology is based on a research of software engineering presented in [18], in which a methodology for solving problems in the field of software engineering is proposed. It consists of three stages:

- Following a certain path for categorizing research questions or issues into different types in order to clarify problems to be solved,
- Selecting a strategy for addressing questions raised in the previous stage. The chosen strategy can be implemented as building a model, performing experiments, or proposing a technique in order to understand the problem and obtain results.
- And applying a validation technique to verify the result obtained. Not only does the validation result confirm whether the selected strategy is correct, it may recommend a new research direction or another strategy to better answer the research issue.

For each stage, five different types of research questions, which are presented in Table 2.1, are suggested. For example, a question relating to feasibility may be asked like "does X exist and what is it?". To provide the answer to the question, several strategies such as building a qualitative model, reporting observations, or generalizing from examples can be selected. Then, to verify the answers obtained from the strategy, a validation technique can be selected such as evaluation or experience. According to the methodology, a validation technique depends on the strategy used. An example to build a common plan is as follows. The research question is "Can X be done better?"; the strategy selected to address the question may be "To build a Y"; then, a validation method is to measure Y and compare to X. If Y is better, a correct answer for solving a research problem is found, otherwise, another Y1, or Y2... could be suggested as alternatives to examine the question. This is shown on Table 2.2

Table 2.1: Research methodology

Question	Strategy/Result	Validation
Feasibility Does X exist and what is it? Is it possible to do X at all?	Qualitative model Report observations Generalize from examples Structure a problem area	Persuasion
Characterization What are the characteristics of X? What exactly do we mean by X? What are the varieties of X and how are they related?	Technique Invent new ways to do some tasks, including implementation techniques Develop ways to select from alternatives	Implementation
Method/Means How can we do X? What is a better way to do X? How can we automate doing X?	System Embody result in a system using the system both for insight and as a carrier of results	Evaluation
Generalization Is X always true of Y? Given X, what will Y be?	Empirical model Develop empirical predictive models from observed data	Analysis
Selection How do I decide whether X or Y?	Analytic model Develop structural models that permit formal analysis	Experience

Table 2.2: A common plan

Question	**Strategy/Result**	**Validation**
Feasibility	Qualitative model	Persuasion
Characterization	Technique	Implementation
Can X be done better?	Build a Y	Measure Y, compare to X
Generalization	Empirical model	Analysis
Selection	Analytic model	Experience

In this thesis, each chapter tries to address relevant research questions drawn from Table 2.1. Each chapter concentrates on a specific and separate research problem toward a common target, that is to build a multitasking environment on coarse-grained DRPAs. To do that, first, problems are clearly specified. Then, previous work related to the problems is reviewed, and the solutions are proposed. An outline of the whole system proposed to solve the problems is introduced. Next, after describing approaches, they are carefully examined by a thorough evaluation to obtain relevant results. Based on the results, relevant discussion and conclusion can be drawn. The organization of the thesis related to the methodology is show in Table 2.3.

Table 2.3: Methodology paths used in the thesis

	Methodology Path		
Thesis organization	Question	Strategy	Validation
Chapter 2 Methodology and system architecture	Feasibility	Qualitative model	Persuasion
Chapter 3 Coarse-grained dynamically reconfigurable processing arrays	Feasibility	DRPA background and architectural analysis	Persuasion
Chapter 4 Hardware task mapping	Method	Technique	Implementation and Evaluation
Chapter 5 Hardware task preemption	Method	Technique	Implementation and Evaluation
Chapter 6 Multicore reconfigurable architecture	Method	Technique	Implementation and Evaluation

2.2 Concepts

Before going into the detail of studies in this thesis, several concepts relating to multitasking on DRPAs need to be defined by drawing an analogy from the software operating system domain and examining unique features when such concepts are applied in reconfigurable computing. Concepts being mentioned are: *application*, *hardware task*, *computational core*, and *inter-task communication*.

2.2.1 Application

An application mentioned in the thesis is a computation program designed to help people to perform a certain type of work. This should be contrasted with system software often involved in services of an operating system. In this context the term *application* refers to both the application software and its implementation.

As a general understanding, target applications for reconfigurable computing are often those, from which massive amounts of parallelism can be exploited, and which have simple control requirements. Applications having large datasets with few or no data dependencies are ideal targets for implementing on reconfigurable platforms [19]. They include image and video processing algorithms, multimedia, network and communication systems, data processing applications and industrial measurement systems, and many military and aerospace systems [19, 20]. All these applications share some common key characteristics:

- They contain computations which are mostly regular and data flow oriented. This can be efficiently computed on reconfigurable devices.

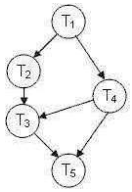


Fig. 2.1: Application's task graph

- They may consist of several different computational tasks, which could allow a task-level pipelined technique to be implemented.

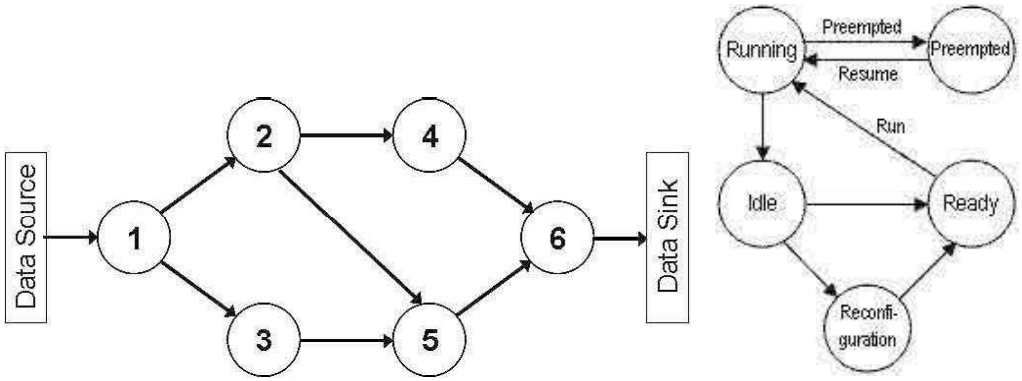
In this thesis, those application are considered as stream applications. That is, data blocks to be processed are iteratively received in a certain interval. The execution of stream applications is often deterministic, or in other words, for a determined input, the same output is always produced. In addition, a partial result at the output of an execution can be obtained from a partial information at the input. This feature is very important since it allows to exploit parallelism and pipelined processing.

An application can be partitioned into multiple tasks, which can be modeled as a set of task graphs, in which nodes represent tasks, and edges show transitions and data dependencies between tasks. Fig. 2.2.1 shows an example of a task graph consisting of five sub-tasks T_1, T_2, T_3, T_4 and T_5.

2.2.2 Hardware task

Generally speaking, hardware tasks are the parts of an application implemented in reconfigurable logic. In this thesis, since I target coarse-grained DRPAs, a hardware task can be considered as the representative of a part of an application mapped onto on a DRPA for execution. The behavior of a hardware task could be represented in the form of a state transition graph, in which nodes represent computation states, and edges shows the transition and data dependence between computation states, as shown in Fig. 2.2(a)). There are a start node with no incoming edges, and an end node with no outgoing edges. Tasks can also be modeled as a finite-state machine (FSM) to represent their executing states (Fig. 2.2).

Hardware tasks may be realized as various implementation variants with the trade-off between performance and reconfigurable resource utilization. Such various implementations can be achieved by considering different circuit architectures. For example, in Chapter 4, it is shown that using a larger tile group for a task could improve throughput and reduce execution time because more parallelism could be exploited. A hardware task requires a specific amount of reconfigurable resource. For FPGA devices, designs often provides information on how many slices are required. For coarse-grained dynamically reconfigurable processors, the number of ALUs, flip-flops and register files a design consumes are usually reported. Additional resources such as different types of memories and multipliers may also be reported.



(a) State transition graph (b) Task finite state machine

Fig. 2.2: Hardware task model

Before hardware tasks can start execution, they must be placed somewhere inside the reconfigurable array. Therefore, they can be characterized by their sizes and shapes. The size of a task can be computed as the number of logic elements the task needs to implement its computation. It can be represented in terms of slices or configurable logic blocks for fine-grained FPGAs, and in terms of processing elements (PEs) for coarse-grained reconfigurable devices.

Apart from the size, which shows a certain area requirement, a task implemented on a reconfigurable device can be characterized by its shape, which is the smallest enclosing geometry shape that contains all processing elements and routing resources used by the task. The shape of a task can be rectangular or polygonal. No matter which shape a task is, external fragmentation caused by dynamic addition and deletion of tasks in a multitasking environment always occurs. While non-rectangular task shapes can use a reconfigurable array more effectively with almost no internal fragmentation in theory, they make allocation, mapping process and defragmentation more difficult. In contrast, although causing internal fragmentation to a certain degree, rectangular task shapes simplifies allocation; and, especially, defragmentation could be performed much faster. As a result, most of research relating to the problem of task allocation, transformation, placement and area defragmentation assume tasks to be rectangular in shape [21, 22, 23, 24]. More importantly, coarse-grained DRPAs often have their reconfigurable array partitioned into larger areas often called tiles, slices or stripes, each of which has a certain amount of PEs, to simplify the routing architecture and to easy the mapping process. Those areas are usually rectangular naturally. In this section and in the thesis, I only take rectangular task shapes into consideration.

The concept of *hardware task* could be considered to be equivalent to that of *software process* in the microprocessor computer domain. However, several unique characteristics in reconfigurable computing cause the differences between these two concepts.

- The program code of a software process is a set of sequential instructions that can arbitrarily be divided into equal sized parts. However, in reconfigurable computing, a hardware task consists

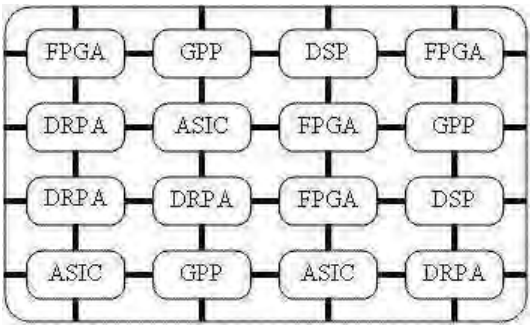


Fig. 2.3: A heterogeneous SoC

of a two-dimensional logic circuit that is loaded in a reconfigurable array for execution. Partitioning hardware is far more complicated than partitioning the sequential codes of software process.

- Maintaining the state of a hardware task is much more complex than that of a software process. When a software process is preempted, the operating system performs a context switch to save the current state of the process. This involves saving the values of a fixed number of registers. However, when a hardware task is preempted, all state information distributed in different storage elements such as embedded memories, register files and flip-flops must be saved. At each computation state, the amount of state data is considerably varying.

- The program data of a software process are contained in a separate part within the process. However, in a hardware task, instructions are circuits and the division between data and computational elements is less clear.

2.2.3 Computational core

A computational core in an SoC is a hardware execution unit or a processor-like hardware block, in which suitable applications can be mapped to for execution. For example, each component represented as a rounded rectangle in Fig. 2.3 is a core with different characteristics and functionalities. Cores are different in structures, granularities and functionalities; so, the system is heterogeneous. A core may be a fine-grained reconfigurable device like an FPGA, a coarse-grain reconfigurable unit such as a DRPA, a general-purpose programmable core, for example, a microprocessor core, or a digital signal processing device.

In a homogeneous system, cores are identical in architecture. For example, a multi-FPGA system containing multiple FPGA chips can be seen as a multicore system, where each core corresponds to an FPGA [25, 26]. In a reconfigurable system where the reconfigurable array can be divided into hardware parts, each of which can be assigned to a task for execution, and all parts can be reconfigured and executed independently, cores can be considered as equivalent to such parts. As an example, when an FPGA can be divided into slots of equal sizes, each slot can be viewed as a

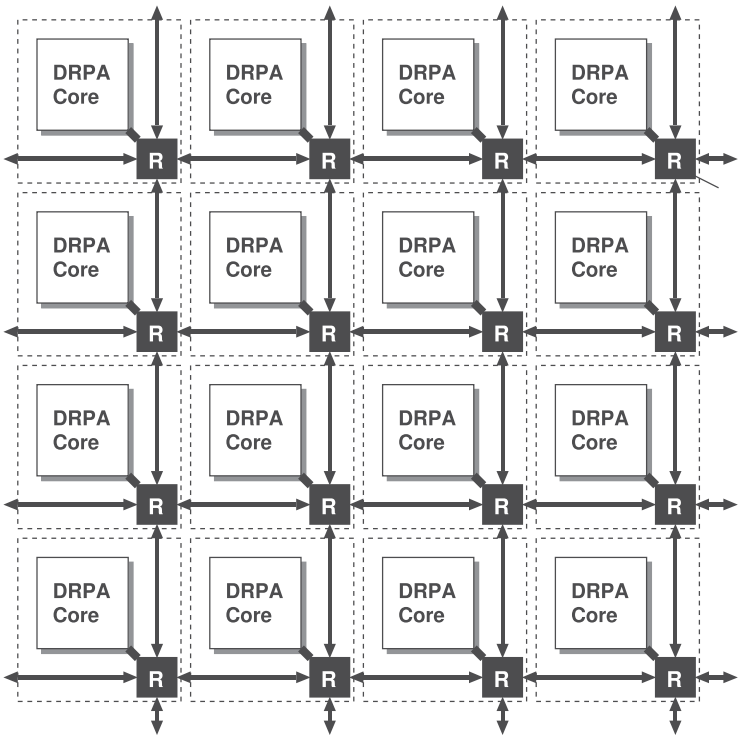


Fig. 2.4: Multicore architecture studied in this thesis

core; or, in a tile-based dynamically reconfigurable architecture like NEC's DRP (Section 3.2.3), tiles could be assumed to be cores, and a 8-tile architecture of DRP-1 is equivalent to a 8-core system. Apart from being able to execute tasks mapped to, cores might be allowed to join together to form core groups.

Many currently available dynamically reconfigurable processors are designed to easily become intellectual property (IP) cores in embedded SoCs such as NEC's DRP (Section 3.2.3), PACT XPP (Section 3.2.2) and MuCCRA (Section 3.2.11). The dynamically reconfigurable processor itself is user-programmable for target applications after the SoC fabrication. In this thesis, the proposed multicore reconfigurable architecture (Chapter 6) has coarse-grained DRPA cores connected through an interconnection network as shown in Fig. 2.4. This structure is homogeneous since each core is assumed to be identical.

Apart from the architecture, cores are characterized by their size. The size of a reconfigurable core can be computed differently. For fine-grained reconfigurable devices, size can be viewed as the number of logic elements or logic array blocks as in Altera Stratic FPGA [27], or the number of configurable logic blocks as in Xilinx Virtex FPGA families. For coarse-grained reconfigurable devices, size can be considered as the number of tiles in each slice as in Chameleon CS2112 (Section 3.2.1), the number of processing array elements as in PACT XPP (Section 3.2.2), the number of PEs or tiles as in NEC's DRP (Section 3.2.3), and the number of cells (ALU and MLT) as in Hitachi FE-GA (Section 3.2.5). In this thesis, since I use NEC's DRP as the target device for research, the

size of a core is calculated as the number of tiles or equivalent processing elements the core has. Specifically, when mentioning the size of a core is n tiles, or $n-tile$ core, a core is considered to have $n(tiles) \times 64(PEs/tile)$ processing elements.

2.2.4 Inter-task communication

In the microprocessor domain, inter-process communication provides mechanisms to allow processes to communicate and to synchronize their actions without necessarily sharing some parts of the address space. These mechanisms are divided into methods for message passing, synchronization, shared memory, and remote procedure call. Applications executing on a reconfigurable system also need to be able to communicate with one another that do not share the same address space. When an application is partitioned into parts, each of which becomes a new hardware task that do not share the address space with other tasks, so an inter-task communication mechanism is needed. In a reconfigurable system, several communication mechanisms can be exploited to transfer data between tasks such as memory, non-shared direct hardware channels, or an interconnection network.

One of the simplest methods to transfer data between two hardware tasks is to use embedded memory modules with a suitable arbitrator. For example, in Chapter 4, vertical memories organized as FIFOs are employed to send and receive processing data between tasks. These FIFOs use a simple handshake mechanism in order for two tasks involving in communication to determine if a FIFO is full or empty. If there is no data in the input FIFO, or the output FIFO is full, the execution of receiving and sending tasks, respectively, is stalled.

Another mechanism for supporting inter-task communication in reconfigurable computing is the use of an interconnection network as illustrated in Fig. 2.4. This involves configuring a communication network architecture independent from all task to connect hardware execution units. The advantage of using an interconnection network over an on-chip wiring architecture is structure, modularity and performance [28]. Moreover, using networks allow more flexibility in where tasks can be allocated. This mechanism will be used to form a multicore reconfigurable architecture in Chapter 6.

2.3 System Architecture

The architecture shown in Fig. 2.5 consists of several components responsible for user interface, OS service provider, hardware abstraction, interconnection network, and underlying reconfigurable device. Outside dashed rectangles show services and mechanisms that will be addressed in following chapters of the thesis. Each component will be described separately in more detail.

User application and interface

This component is similar to the same concept found in software operating systems based on microprocessors. It aims at providing an interface between the users, the OS and the hardware. Users can input commands and execute applications via the interface. The OS service provider is then re-

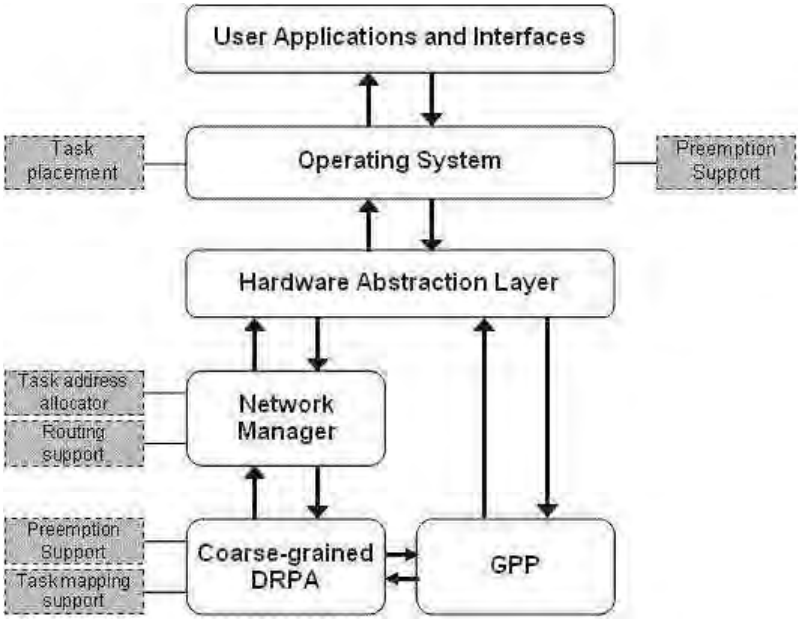


Fig. 2.5: System architecture

sponsible for converting them into appropriate calls to the application programming interface (API). The OS service provider also reports back to the user interface about the status of the requested operations.

Operating system

The OS service provider is responsible for interpreting user commands from the interface, modifying to suit APIs, and then passing them directly to the hardware abstraction layer. The services provided include embedded memory reading and writing, and platform specific configuration information. Using the OS allows to cope with the change of the hardware abstraction layer since in this case, only the services provided by the OS must be changed, and the user interface is still the same.

Hardware abstraction layer

Similar to equivalent concept in the GPP field, a hardware abstraction layer allows an operating system to interact with a hardware device at a more abstract level. However, the biggest challenge in reconfigurable computing is that there is no agreed standard interface for reconfigurable devices. This is different from devices in the GPP world such as hard disks, monitors or memories. Manufacturer have their own APIs, which require users to have a good understanding of the platform. As a result, the hardware abstraction layer mentioned here should provide a standard API that abstracts away the detail of underlying reconfigurable hardware. It should provide an API to connect to the platform, configure the device, control the clock rate and access embedded memory.

Network manager

The network manager plays an important role in supporting inter-task communication and input/output operations. In this thesis, I encourage to use an interconnection network to connect cores and form

a multicore reconfigurable architecture. Although the network itself is fixed, tasks are dynamically mapped onto any core, and once the mapping process has been done, information about where tasks are located is updated to the network manager in order to allow related tasks to communicate to one another.

Coarse-grained dynamically reconfigurable processor

This is the target hardware platform in which tasks can execute to perform their functionalities. Generally speaking, it is a multicore architecture where each core is a separate coarse-grained multi-context dynamically reconfigurable processor with a certain size. Cores are connected by a network-on-chip. The device should have the capability of partial reconfiguration in order to support the dynamic addition and deletion of tasks. A task can be mapped to any core for execution; however, in order to improve throughput and reduce fragmentation, an appropriate mapping approach should be introduced.

GPP

As being mentioned in many research, reconfigurable platforms are not efficient for control and load/store operations; instead, they are often used for accelerating certain parts of an application like data-flow kernels. Accordingly, a reconfigurable device is usually coupled with a certain type of a microprocessor. The coupling between a GPP and a reconfigurable fabric will be discussed in refsec:coupling. A GPP part in Fig. 2.5 represents such a processor for handling operations unsuitable being executed on the coarse-grained dynamically reconfigurable processor. Moreover, a GPP is needed to perform specific management jobs for the reconfigurable hardware.

Multitasking-support mechanisms

Several shaded small rectangles outside above mentioned components represent mechanisms to support multitasking on DRPAs studied in this thesis. The position of such rectangles shows appropriate layers where the mechanisms should be integrated. Some mechanisms appearing in more than one layer mean their implementations need support from these layers. For example, in order to implement a preemption scheme for hardware tasks, not only does the target coarse-grained DRPA need to provide a preemption interrupt and a method to recognize preemption points[1], the scheduler in the OS must be aware of the interrupt and the state data of preempted tasks in order to correctly schedule tasks. As a result, *Preemption support* appears both at the OS layer and the coarse-grained dynamically reconfigurable processor layer.

2.4 Multitasking Execution Model

There are several models of multitasking execution in reconfigurable computing based on different characteristics. This section introduces the multitasking execution model based on resource sharing, which is the most relevant to studies in this thesis. Several approaches attempt to determine how

[1]The concept of *preemption points* will be introduced in Chapter 5

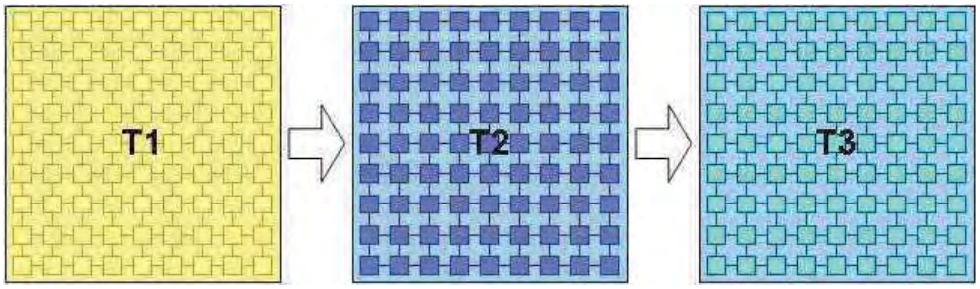


Fig. 2.6: Time-sharing, non-space partitioning model

a reconfigurable system allows multiple tasks to share its resources, especially the reconfigurable processing array. Since the purpose of reconfigurable devices is to extract as much parallelism both instruction-level and task-level parallelism from a target application as possible in order to achieve a certain speedup over microprocessor platforms, a resource sharing model is the way to allow multiple hardware tasks to use the resources of a reconfigurable device more efficiently; or in other words, this exploits task-level parallelism to further improve performance, especially when a target application does not have enough degree of instruction-level parallelism.

2.4.1 Time-sharing, non-space partitioning

The simplest method to realize multitasking is a time sharing approach without partitioning the area of a reconfigurable device, as shown in Fig. 2.6. That is the entire device resources are assigned to only one task at a time. Each task can run until completion (non-preemptive multitasking), or tasks can be preempted and resumed later to free resources for other tasks (preemptive multitasking). Switching from one task to another is carried out by fully reconfiguring the whole device. Since only one task is executed at a time, this model incurs the problem of the overhead in loading a configuration and the the limited availability of on-chip memory for caching. Moreover, not only does device utilization become inefficient since the sizes of tasks are varying and no matter how large a task is, it is always allocated the whole reconfigurable array for execution, but no parallelism between tasks can be exploited. Several systems use this approach [29, 30, 31, 32]. For example, Fig. 2.6 indicates that, at first, task $T1$ is loaded for execution, then when $T1$ is stopped either because it has completed or because it is preempted, task $T2$ is loaded to replace $T1$. Then, upon finishing, task $T3$ will replace task $T2$. Whenever a task is allocated for execution, it always takes over the whole reconfigurable array though its size might be much smaller than the size of the array; and, the whole device needs to be reconfigured.

2.4.2 Non-time sharing, static partitioning

Using the static partitioning approach, all tasks are allocated to different areas in the reconfigurable array providing that the total size of tasks does not exceed the size of the array (Fig. 2.7). Normally,

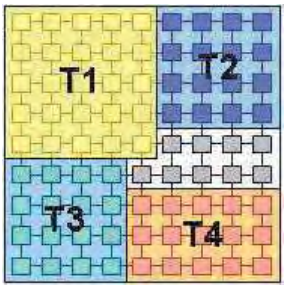


Fig. 2.7: Non-time sharing, static partitioning model

all tasks are configured once and stay active until the system shuts down, or until another set of tasks requires to load in for execution. This method makes a reconfigurable device behave similar to an ASIC and it is suitable to systems where a set of tasks can be determined in advanced. Since tasks do not need to be allocated and freed often, mechanisms for scheduling and resource management are not necessary, or they are quite simple if such mechanisms requires. This approach is efficient when tasks are predetermined and they often execute for most of the time. It is not suitable if some tasks need to be executed just once or only occasionally since all tasks get resources allocated. Fig. 2.7 shows an example where all four tasks $T1$, $T2$, $T3$ and $T4$ are present in the reconfigurable array at the same time. No matter if any of these tasks is executing or not, they all occupy a part of the reconfigurable array.

The mapping method proposed in Chapter 4 exploits this approach. According to the method, an application is partitioned into multiple tasks, each of which is assigned to a tile or a group of tiles of a underlying DRPA, and all of them are present at the same when being executed. Tasks can communicate using a FIFO model based on embedded memories. This approach allows to exploit task-level pipelined model by arranging tasks involving in a computation stream into a pipelined chain to take advantages of a pipelined model toward improving throughput. Also, for reconfigurable devices without partial reconfiguration capability, non-time sharing and static partitioning method is the best way to exploit task-level parallelism.

2.4.3 Time-sharing, two-dimensional partitioning

In this approach, tasks, which are represented as rectangles, can be dynamically allocated to any area in the reconfigurable array for execution providing that enough free space is available at the assigned area, and then removed upon completion, as shown on Fig. 2.8. When a task completes or is preempted, the area it occupies is freed, and a new task may take over that area; so, when that task is resumed for continuing execution, it is likely to be assigned to a new area. This approach enables a more flexible and efficient multitasking environment since tasks can be dynamically allocated resourses for execution and freed upon completion. This is similar to the memory allocation mechanism in microprocessor-based OS. On the down side, this method could make the problem of scheduling and fragmentation more complicated. More importantly, it requires the capability of

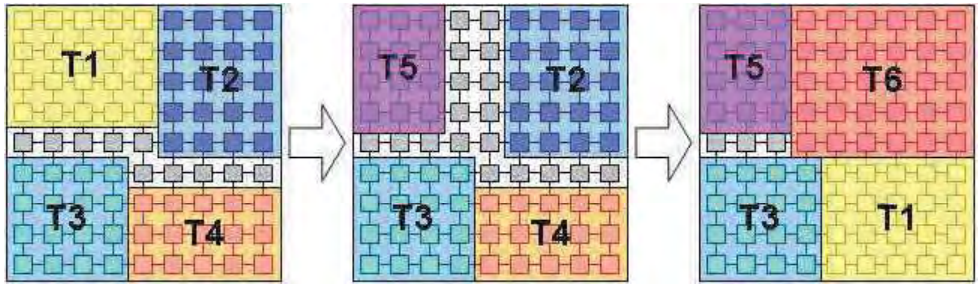


Fig. 2.8: Time-sharing, two-dimensional partitioning model

partial reconfiguration on a two-dimensional reconfigurable array. Several studies have addressed different aspects of this approach [33, 34, 35].

As an example from Fig. 2.8, at the beginning, four tasks $T1$, $T2$, $T3$ and $T4$ are allocated for execution. After some times, task $T1$ is preempted and removed, task $T5$ comes and takes over a part of area that was occupied by $T1$ previously. Then, when tasks $T2$ and $T4$ complete, task $T6$ is scheduled and $T1$ is resumed. This time, task $T1$ is allocated to a new area. As shown, the external fragmentation, which is fragmentation of area outside the rectangular boundary of all tasks, changes from time to time. That is the reason why a defragmentaion mechanism is necessary; otherwise, a task might be denied to place on the reconfigurable array for execution since sufficient contiguous area is not available though total empty area may be greater than the area required by the task.

2.4.4 Time-sharing, multicore partitioning

This approach is based on a multicore architecture, where independent computational cores are connected by a certain network. Cores may be identical and have equal sizes, so I have a homogeneous architecture. Cores could also be different in size, granularity or structure as shown in Fig. 2.3; in this case, a heterogeneous architecture is represented. Generally speaking, a task can be mapped to any core for execution; however, for a heterogeneous system, tasks are allocated to a certain core based on their sizes or functionalities in order to make the best use of advantages of cores.

Fig. 2.9 shows a heterogeneous model where cores are identical in structrure but different in size. Cores could be connected by different types of network. For example, in Fig. 2.9, if connection elements are switches, a system with an island-style network is established; in case when connection elements are routers, a different system with a network-on-chip is formed. A mapping process decides which tasks are assigned to which cores based on the size of tasks in order to optimize the usage of cores' resources. When a task completes, the core it occupies is freed and another task can be assigned to the core. This architecture assumes that each task must be able to fit in a core. If a task is too large for any core in the system, it should be partitioned into smaller tasks. This approach is the extension of the above mentioned time-sharing, two-dimensional partitioning architecture. Therefore, the approach encounters the same problems as the above solution such as scheduling, internal and external fragmentation. However, by using multiple cores, the process of partial reconfiguration

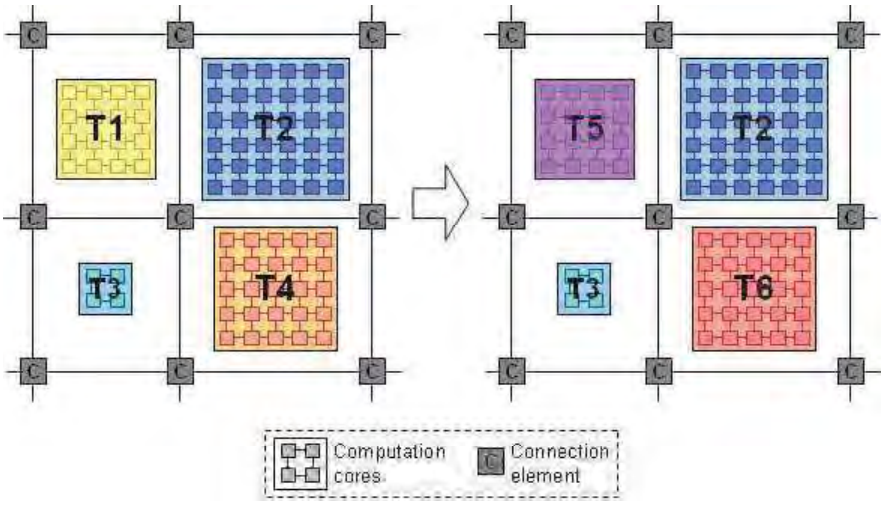


Fig. 2.9: Time-sharing, multicore partitioning model

becomes much easier since cores are independent.

As an example from Fig. 2.9, at the beginning, four tasks $T1$, $T2$, $T3$ and $T4$ are allocated for execution according to their sizes. When tasks $T1$ and $T4$ finish or are preempted, their correspondent cores are freed and task $T5$ and $T6$ are assigned to these cores. As being shown, a task always takes over a core though its size might be smaller than the core size. This causes internal fragmentation.

Chapter 3

Coarse-grained Dynamically Reconfigurable Processing Arrays

This chapter describes coarse-grained DRPAs that are the potential target devices of research in this thesis. First, the background and the architectural overview of DRPAs are introduced, then several dynamically reconfigurable processor architectures including both commercially available devices and new prototypes released and developed recently are described. Devices mentioned in this chapter is summarized in a table, in which devices are categorized according to certain criteria introduced in Section 3.1.

3.1 DRPA Architecture Overview

A large number of coarse-grained reconfigurable architectures have been developed over the years by researchers and the industry. Reconfigurable architectures can be classified based on several different parameters. In following sections, system-level architectures for coarse-grained are described by presenting various flavors of reconfigurable fabric.

3.1.1 Coarse-grained processing array

A typical DRPA often consists of a two-dimensional array of coarse-grained PEs, distributed memory modules, multipliers, state transition controllers, and I/O elements. Compared with fine-grained FPGAs, the advantage of a DRPA in terms of area efficiency comes from the capability to provide multiple-bit wide datapaths and word-level operators instead of bit-level reconfigurability. Wide datapaths allow the efficient implementation of complex operators in hardware. Thus, routing overhead caused by composing complex operators with bit-level processing units is avoided. Fig. 3.1 shows a PE array of PACT XPP-64 [36] with a 8x8 computational PE (ALU-PAE) array and a set of memory modules (RAM-PAEs) at both sides.

In general, each PE has a 4-bit to 32-bit ALU for numerical and logical calculations, logics for shift/mask operations, registers or register files, and multiplexers for switching the dataflow between

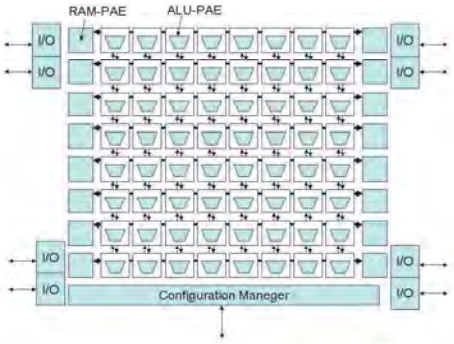


Fig. 3.1: PE array of PACT-XPP

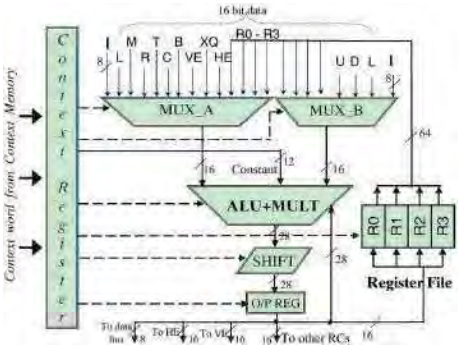


Fig. 3.2: Example of a PE (RC of Morphosys)

such components. An example of a typical PE, which is a Reconfigurable Cell (RC) of Mophosys, is presented on Fig. 3.2 [37]. As shown on the figure, an RC consists of an ALU-multiplier ($ALU-MULT$), a shift unit ($SHIFT$), two multiplexers ($MUXA$ and $MUXB$), a register file and an output register ($O/PREG$). Context data stored in $ContextRegister$ determine the functionalities of the whole RC.

The operation of ALUs, shift/mask logics, and data paths between components are controlled with configuration data stored in configuration memories. The array of multiple PEs connected with programmable routing resources provides various kinds of parallel datapaths such as single instruction streams and multiple data streama, multiple nstruction streams and multiple data streams, and pipelining. Although a coarse-grained PE array has low flexibility compared to a fine-grained reconfigurable fabric, it can provide high performance and high area-efficiency for parallel multimedia applications. The PE with a small granularity such as 4-bit or 8-bit can generally improve flexibility with low datapath speed. In contrast, a larger granularity can provide high area-efficiency and reduce configuration data only if the granularity suites to a target application.

3.1.2 Interconnection structure

Interconnection networks [38] used to connect PEs in an array are varying, but they could be categorized into three groups: an island-style interconnection, a direct interconnection, and a switch matrix interconnection.

Island-style interconnection Fig. 3.3 depicts an PE array with an island-style interconnection structure, which is very similar to that of FPGAs. Switches denoted as grayed squares with **S** character inside are provided at the intersection of horizontal and vertical data channels. Switches can be changed by dynamic reconfiguration, especially, in every clock cycle for multicontext devices to realize different connections among PEs. Although being flexible on PE-to-PE connections, an island style encounters two major problems: (1) the area for routing and switches is large, and (2) the maximum operating frequency is degraded by a long critical path.

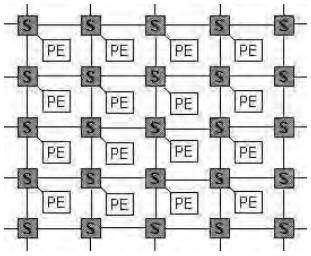


Fig. 3.3: Island-style

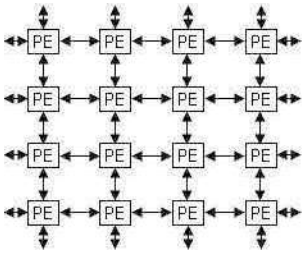


Fig. 3.4: Direct interconnection

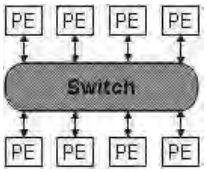


Fig. 3.5: Switch matrix

Direct interconnection Fig. 3.4 shows a typical direct interconnection structure in a PE array. In this style, each PE is directly connected to its nearest neighbors. In order to reduce the delay for communicating with remote PEs, additional long connections to other PEs in the same row and column are provided in some architectures [39]. A direct interconnection could be a solution for two problems existing with an island-style interconnection. However, it suffers a long delay to transfer data between remote PEs.

Switch matrix interconnection A large switching matrix can be used to connect a number of PEs as shown in Fig. 3.5. Since the number of PEs connected with a single switch is limited, various types indirect interconnection may be used. For example, in D-Fabrix architecture, a chess-board like interconnection between PEs and switches is applied by having three-stage indirect switching network to connect all PEs in a cluster. In FE-GA, PEs are connected to distributed memory modules through a switch, which allows to transfer data stored in memory modules to any PEs located at the edge of the PE array.

3.1.3 Dynamic reconfiguration method

Although the coarse-grained PE arrays actually provide high area-efficiency compared to fine-grained FPGAs, they are still inefficient compared to logically equivalent ASICs because of the programmability. A dynamic reconfiguration scheme is a primary technique to improve area- and power-efficiency of field-programmable logic devices like the coarse-grained PE arrays. A large amount of configuration data can be stored into on-chip and/or off-chip memories, and the coarse-grained PE array can be dynamically reconfigured by reading out a desired configuration data from the memories. The dynamic reconfiguration offers a great advantage with respect to area- and power-efficiency because the PE array can be reconfigured so as to be optimized for various kinds of functions.

In the last few decades, fine-grained FPGA-based dynamically reconfigurable devices have been released. They include partially run-time reconfigurable FPGAs [40] and multicontext FPGAs [41, 42]. Fine-grained dynamically reconfigurable devices require a large amount of configuration data, and this implies millisecond-order reconfiguration time. Therefore, dynamically reconfigurable FPGAs have been used for only restricted purposes such as a logic emulation system. Fortunately, coarse-grained PE arrays can drastically reduce the need for configuration data, and they provide less

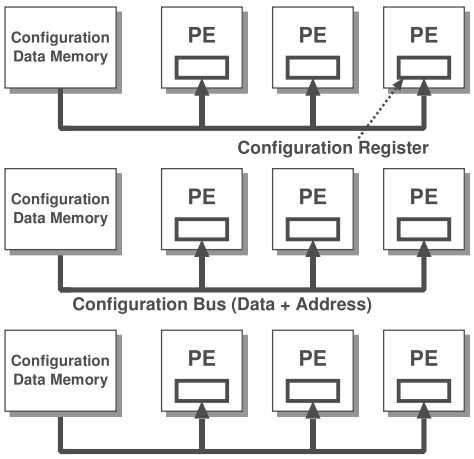


Fig. 3.6: Configuration Delivery Scheme

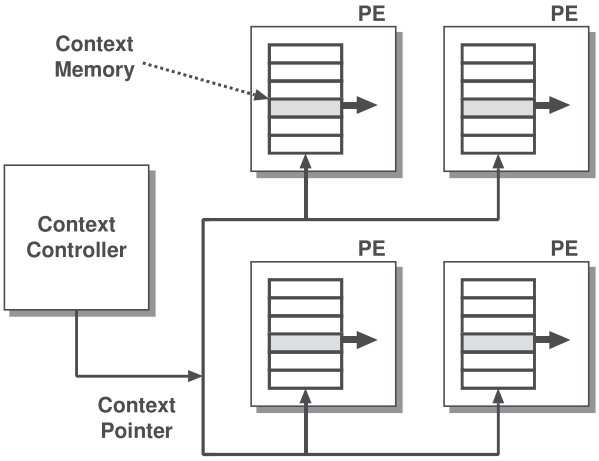


Fig. 3.7: Multicontext Scheme

than microsecond-order and practical dynamic reconfigurability.

In this thesis, dynamic reconfiguration schemes are classified into two categories: a configuration delivery scheme and a multicontext scheme.

Configuration Delivery Scheme

The *configuration delivery scheme* is used for the systems in which the dynamic reconfiguration is not performed frequently. In this method, as shown in Fig. 3.6, the configuration data for each reconfigurable element including PEs and switching resources are stored in on-chip configuration data memory modules. The configuration data is delivered to corresponding elements in sequence via a dedicated configuration bus. Since configuration data memory modules are centralized on the chip, the configuration delivery scheme has an advantage of area-efficiency, and communication with outside of the chip is readily feasible. The reconfiguration time for the configuration delivery scheme is almost microsecond order. This is preferable to FPGAs with millisecond-order reconfiguration time, but operations of the PE array have to be suspended during the reconfiguration in most cases.

Multicontext Scheme

The other dynamic reconfiguration method, the *multicontext scheme*, is supported in recent dynamically reconfigurable processors. In this method, as shown in Fig. 3.7, each reconfigurable element provides its configuration data memory called a *context memory* which stores configuration data sets for operational instructions of each PE and intra/inter-PE connection instructions. A context number is delivered to all of the reconfigurable elements and used as a pointer to the context memories. By changing the context number and reading the context memory simultaneously, all the reconfigurable elements can switch the context in parallel with only one clock cycle. This method implies that the configuration data corresponding to a context is distributed to each reconfigurable element, and a context is switched by reading out the configuration from each of the context memories.

There are different context switching control methods for the multicontext dynamically reconfigurable devices. The three methods supported by recent architectures include a data-driven control,

state transition control, and program counter-based control. At first, Ling and Amano [43] proposed the data-driven control method in their architecture WASMII. In the WASMII, a context is activated when all input data are present. Secondly, in the state transition control method, a simple controller has a state transition table which describes statically defined context switching patterns. Finally, the program counter-based control is the simplest method based on an incrementing counter. As similar to commonly used microprocessors, the controller has a counter for the context number and the context indicated by the counter is activated.

Although the data-driven method provides dynamic and flexible context switching controls, it has a design difficulty because of the hardware overhead for detecting an activated context [44]. Therefore, the simpler methods including the state transition control and the program counter-based control mechanisms are commonly used in the recent multicontext dynamically reconfigurable processors. Since the proposal of WASMII in 1992, our project has investigated the multicontext dynamically reconfigurable processor architectures, and I focus on them also in this thesis. And, I suppose the state transition control and the program counter-based control methods for the context switching control because of their generality.

3.1.4 Coupling between CPU and DRPA

A DRPA is often used as an accelerator for a host processor. A system consisting of a microprocessor and a DRPA could achieve both the high performance with hardware-accelerated execution on the reconfigurable array and the flexibility with software execution on the GPP. From the viewpoint of the coupling between a microprocessor and a reconfigurable fabric, reconfigurable system can be classified into three groups [45], as shown on Fig. 3.8.

- an attached reconfigurable processing unit,
- a tightly coupled coprocessor, and
- a reconfigurable functional unit (RCFU).

Attached Reconfigurable Processing Unit

Attached reconfigurable processing units are integrated into a system on a memory or I/O bus. Systems with attached reconfigurable processing units, for example, Splash 2 [46, 47], RASH [48], and Teramac [49], have no direct access to the processor. The primary feature of attached reconfigurable processing units is that they are easy to add to existing computer systems. However, due to the bandwidth and latency constraints imposed by memory or I/O buses, they do not suit to communication-intensive applications.

Tightly Coupled Coprocessor

A coprocessor system consists of a microprocessor and one or several reconfigurable fabrics that play a role of a coprocessor. In general, the coprocessor handles computation-intensive parts of a program, and the other parts are executed by the microprocessor. In such a system, the reconfigurable fabric

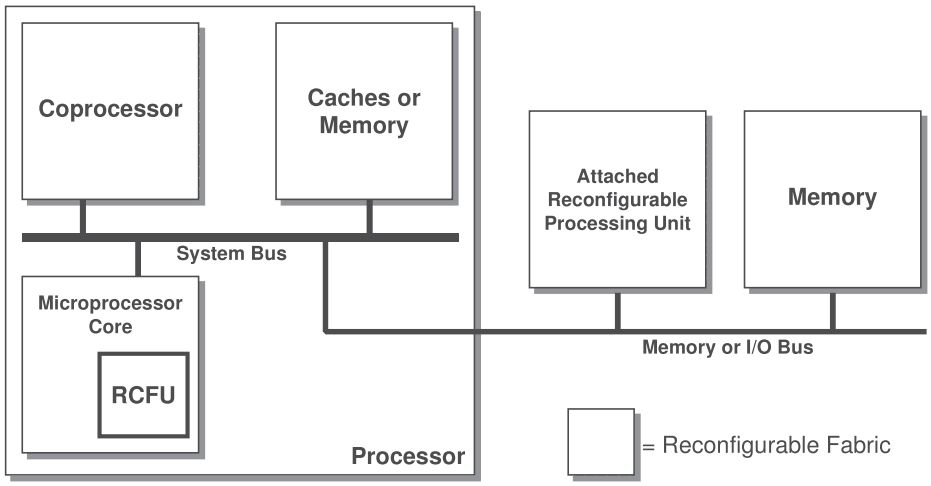


Fig. 3.8: Coupling of Reconfigurable Fabric and Microprocessor

can access to the same memory hierarchy as to the microprocessor including several levels of caches, on-chip memories, and external memories. As a result, there is a low-latency and high-bandwidth connection between the microprocessor and the reconfigurable fabric. The examples of such systems include Garp [6, 50], Napa 1000 [51], PRISM [52, 53], and ArMen [54].

Reconfigurable Functional Unit (RCFU)

The tightest coupling between a microprocessor and a reconfigurable fabric occurs when the reconfigurable fabric is on the microprocessor's datapath, as in functional unit architectures like PRISC [55], Chimaera [56], and OneChip [57, 58]. All of these allow custom instructions to be executed. The reconfigurable fabric is on the processor datapath and has access to registers. However, these implementations restrict the applicability of the reconfigurable fabric by disallowing state to be stored in the fabric and in some cases by disallowing direct access to a memory, essentially eliminating their usefulness for stream-based processing.

3.1.5 C-based programming methodology

Due to their field-programmability, dynamically reconfigurable processors can execute various kinds of parallel algorithms. For a stream-based processing like an image processing, streaming data are distributed into memory modules and executed by the PE array in data-parallel and SIMD fashion. Moreover, since the PE array has a lot of independent operators such as shifters, ALUs, and registers, and switches a context in a cycle-by-cycle manner, it can perform as like a Very Long Instruction Word (VLIW) processor exploiting instruction-level parallelism (ILP).

For dynamically reconfigurable processors, a C/C++-based programming methodology is commonly accepted [59]. Although basic C language is suitable to describe algorithmic behaviors, it's not suitable for data flow or continuous computation [60]. In order to apply C language to describe a hardware behavior of ASICs and FPGAs, several C-based hardware description languages and

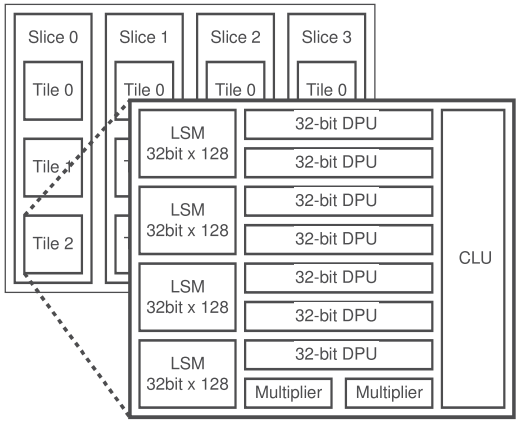


Fig. 3.9: CS2112 reconfigurable fabric

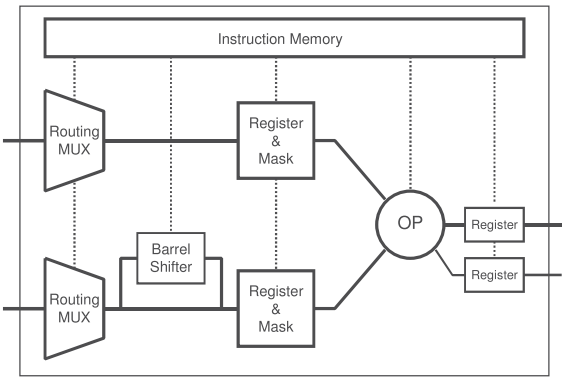


Fig. 3.10: CS2112 datapath unit (DPU)

high-level synthesis techniques have been proposed. They support extended features such as timing and communications. Fortunately, since the coarse-grained PE array consists of ALUs, shifters, and register files, an arbitrary data flow graph described in a C-based language can be efficiently mapped onto the PE array.

3.2 Review of DRPAs

This section reviews several dynamically reconfigurable processor architectures released by consumer-electronics makers in the last decade. They include not only commercially available products but also new devices under research and development.

3.2.1 Chameleon CS2112

The Chameleon's Reconfigurable Communication Processor (RCP) [61] called CS2112 [1], comprises a Reconfigurable Processing Fabric (RPF), programmable input and output banks, and an embedded microprocessor. Chameleon is a licensee of the ARC processor, which it uses as an executive controller overseeing the 128-bit internal RoadRunner split-transaction bus. RPF uses RoadRunner to communicate with the ARC core, as well as with external devices on the PCI bus.

As shown in Fig. 3.9, the device's fabric is made up of four programming slices, each of which has three tiles. Inside each tile is a control logic unit (CLU), which is a programmable logic array that controls the tile's registers; four separate local store memories (LSM) measuring 32 bits x 128 entries deep; dual 16 x 24 multipliers (MPU); and a total of seven 32-bit datapath units (DPU), similar in function to an arithmetic logic unit. One CS2112 chip thus has 84 datapath units, 24 multipliers and 48 local store memories, with an aggregate memory of 24 kbytes.

The DPU shown in Fig. 3.10 is a fundamental computational element in the fabric. It has a basic word length of 32 bits but has special operations that allow it to operate in SIMD fashion on

[1]The technologies of Chameleon's reconfigurable processors is now licensed by Intel Corporation.

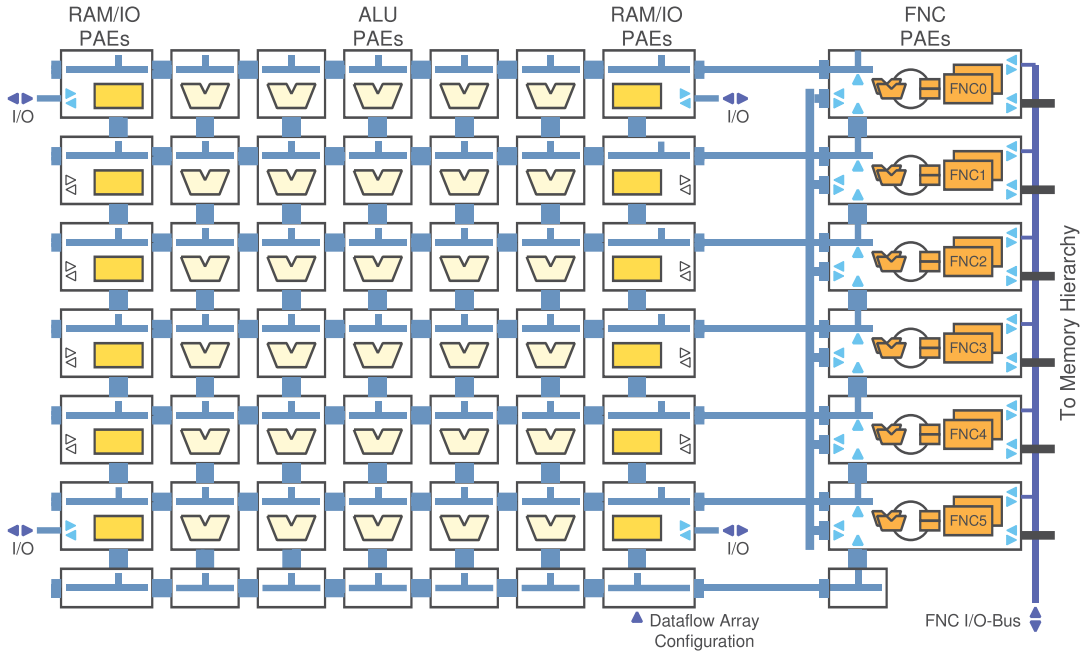


Fig. 3.11: XPP-III Core Architecture Sample

four 8-bit data streams or two 16-bit data streams. The core of the DPU is the 32-bit Operator (OP) that performs arithmetic and logical operations. The MPUs can perform 16 x 24-bit or 16 x 16-bit single-cycle multiplications. The LSM is a multi-ported 32-bit x 128-word RAM. Each DPU is programmed with eight user-definable instructions stored in its instruction memory (context memory). Each instruction specifies a complete configuration for the DPU. The CLU directly implements a FSM to select the DPU and MPU instructions stored in the instruction memory. Nearby DPUs are connected to each other in a full crossbar connection. More distant DPUs are connected with routing with a one-clock pipeline delay.

Each tile of the RCP has a background configuration plane and active configuration plane. Instructions for the architecture are dynamically programmable, since instructions load in the background plane during one clock cycle, then swap functions with an active plane. To program all four slices, a new configuration data can be loaded in less than 3 μsec. The entire system can be reconfigured in one clock cycle by switching the configuration data from the background plane to the active plane. That means multivariate problems are handled in an all-or-nothing fashion in the Chameleon design. In a cdma2000 chip-rate application, for example, tasks such as pseudrandom number generation and rake finger searches are not parceled out to various pipelined subprocessors. Instead, an entire tile is dedicated first to pseudorandoms, then to demodulation, then to finger searches and finally to access searches, with the task reassigned in a single clock cycle.

Applications are developed in a mixed environment, where routines compiled in C are developed for the ARC executive controller, while Verilog source code is created and synthesized for the reconfigurable fabric. The resulting fabric bit stream is sent to the ARC linker in the final object-code

steps and then to Chameleon's proprietary execution engine for final simulation.

The RCP has been implemented at a 0.25 μm CMOS technology with a 125-MHz clock. From the performance evaluation results to handle digital processing algorithms, CS2112 can handle a 1,024-point fast Fourier transform in 10 microseconds, while a 48-tap symmetric FIR filter has a 125-Msample/second capability. As baseband processing in 3G digital cellular markets branches out to include some IF filtering functions.

3.2.2 PACT XPP-III

eXtreme Processing Platform (XPP) [36] is a reconfigurable processor architecture based on a hierarchical array of PEs called Processing Array Elements (PAEs). An XPP Core contains a rectangular array of three types of PAEs. Those in the center of the array are ALU-PAEs. To the left and right side of the ALU-PAEs are RAM-PAEs with I/Os. Finally, at the right side of the array, there is a column of FNC-PAEs. Fig. 3.11 shows a sample array with 30 ALU-PAEs, 12 RAM-PAEs, and 6 FNC-PAEs. The PAEs can be configured while neighboring PAEs are processing data. Reconfiguration is triggered by a controlling FNC-PAE or by special event signals originating within the PE array.

The FNC-PAE comprises a 2 x 4 array of 16-bit ALUs, a Special Function Unit (SFU), a 16-bit register file, a 32-bit address generator, a local instruction cache, a tightly coupled memory, and I/O ports. The eight ALUs are designed to be small and fast because they are arranged in two non-pipelined columns of four ALUs each. SFUs operate in parallel to the ALU datapath. They support up to two 16x16-bit multiplications and functions such as a bit-field extraction. By combining the SFU multiplications with the adders of the ALU array, it is possible to execute two pipelined multiply-accumulate (MAC) operations each cycle.

As shown in Fig. 3.12, the ALU-PAE contains three XPP objects: FREG, ALU, and BREG object. All the objects have input registers which store the data or event packets for one cycle. The ALU object in the center of the PAE provides basic logical and arithmetic operations, and special arithmetic operations such as multiplication. The Forward Register (FREG) object on the left side and the Backward Register (BREG) object on the right side of the ALU-PAE are very similar. The main difference is the processing direction: top-down for the FREG and bottom-up for the BREG object. Both objects provide routing of data, dataflow operators such as multiplexing, basic arithmetic operations and look-up table for boolean operations.

The RAM-PAE consists of the FREG and BREG objects which are identical to the ones in the ALU-PAEs, a RAM object, an additional I/O object. The RAM object contains a small bank of two-ported SRAM. The RAM operates either in internal RAM (IRAM) or in a FIFO mode. The content of RAMs is preserved during reconfiguration of the array. The I/O object is integrated into the RAM-PAE, providing access to external data.

The XPP objects communicate through a packet-oriented network. An operation is performed as

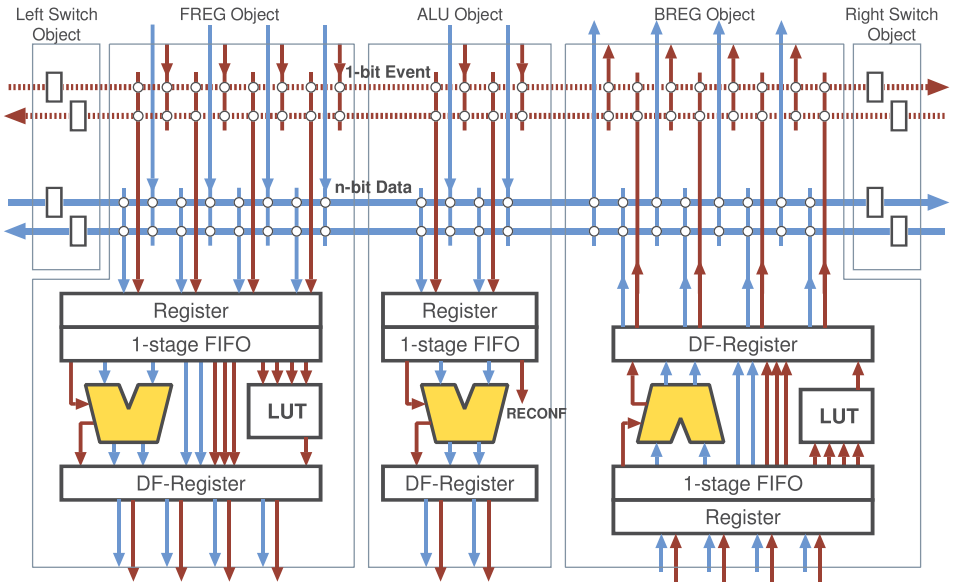


Fig. 3.12: XPP-III ALU-PAE Architecture

soon as all necessary data input packets are available. The results are forwarded as soon as they are available, provided the previous results have been consumed. Thus it is possible to map a dataflow graph directly to ALU objects and to pipeline input data streams through it. The communication system is designed to transmit one packet per cycle. Hardware protocols ensure that no packets are lost, even in the case of pipeline stalls or during a configuration process.

In [62], it is described that a video decoder on the XPP-III, in which various video sequences including MPEG-2, MPEG-4, H.264, and VC-1 (WMV9), are supported. The evaluation result shows that the XPP-III version of 40 FNC-PAEs, 16 ALU-PAEs, and 8 RAM-PAEs can perform real-time decoding of H.264 frames with VGA size at 92MHz and HD resolution (1280x720) at 174MHz.

3.2.3 NEC Electronics DRP-1

Dynamically Reconfigurable Processor (DRP) is a coarse-grained reconfigurable processor that was released by NEC Electronics in 2002 [63]. The basic building unit of the DRP architecture is a Tile shown in Fig. 3.13. Each Tile consists of 64 PEs, 8 vertical memory modules (VMEMs) with 2 controller (VMCTRs), 4 horizontal memory modules (HMEMs) with one controller (HMCTR), and a state transition controller (STC). By combining multiple Tiles, designers can construct a DRP core with a desirable size.

Fig. 3.14 shows the PE architecture which is composed of an Arithmetic Logic Unit (ALU), a Data Manipulation Unit (DMU), a Flip-Flop Unit (FFU), and a Register File Unit (RFU). The ALU performs addition, subtraction and so on. The DMU is a logic operator, and it performs the combinations of logical AND, OR, and shift. The FFU is an 8-bit wide flip-flop unit, and the RFU is

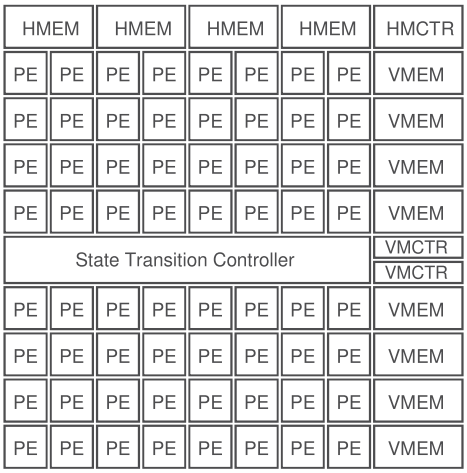


Fig. 3.13: DRP Tile Architecture

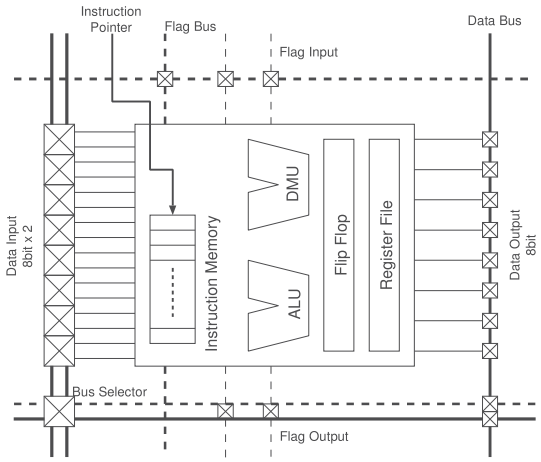


Fig. 3.14: DRP PE Architecture

an 8-bit wide and 16-entry deep register file.

In a DRP core, memory modules are laid-out in jail-bar like structure. VMEMs separate the tile columns, and HMEMs take the top and bottom. Each of VMEM modules is an 8-bit x 256-word memory with one read port and one write port. It can be also configured as a FIFO storage. Each of HMEM modules is an 8-bit x 8192-word memory with one read or write port. Together with PE's FFU and RFU, the DRP supplies an uniform distribution of memory resources.

Each of the reconfigurable elements such as PEs has its instruction memory (context memory), in which 16 contexts can be stored. According to a context pointer delivered from the STC, all of the reconfigurable units read the corresponding context and then be reconfigured. The STC is a programmable sequencer which carried out switching the context based on a simple state transition, and can store up to 64 states. Each state is associated with the context pointer which indicates a particular context. The STC can receive event signals from the PE array to take a branch conditionally. The maximum number of branches that can be specified from the PE array is four.

Fig. 3.15 depicts the prototype chip DRP-1, which is composed of a DRP core with 4 x 2 Tiles. It has been fabricated with a 0.15-μm CMOS technology. It consists of 8 Tiles, eight 32-bit multipliers, an external SRAM controller, a PCI interface, and 256-bit I/Os. The maximum operational frequency is 100MHz. Although the DRP-1 is used as an attached reconfigurable processing unit, the DRP core can be used as an IP core on SoCs with an embedded processor. In this case, the number of Tiles can be chosen so as to achieve required performance with minimum area.

An integrated design environment called Musketeer is provided for the DRP-1 [59]. The application design flow is shown in Fig. 3.16. It consists of two flows: compilation flow (shown on the left part of Fig. 3.16) and verification flow (shown on the right part of Fig. 3.16). The compilation flow is composed of three main stages: behavioral synthesis, technology mapper and place-and-route. Firstly, the behavioral synthesis tool receives a DRP programs described in a C-based hardware description language called Behavioral Design Language (BDL) as input, separates the control flow

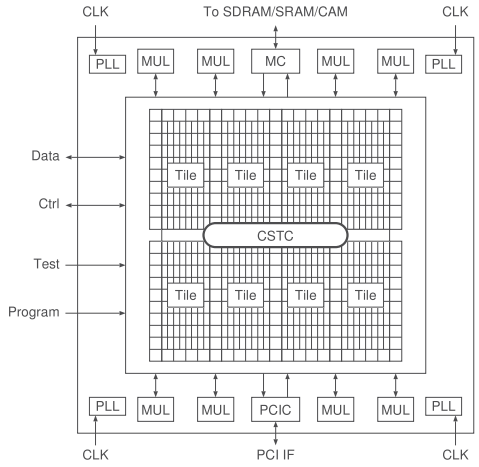


Fig. 3.15: DRP-1 Architecture

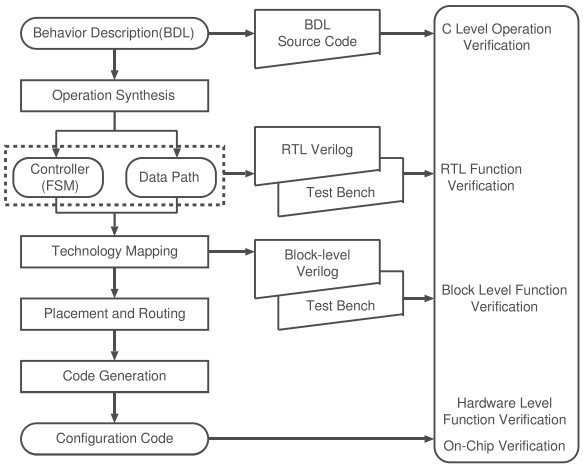


Fig. 3.16: Programming Flow of DRP

represented as the finite-state machine and data flow, and produces the output in a Verilog format that describes the datapath of each computation state. The tool also provides a report containing the number of states, the number of predicted PEs used in each state, and the predicted delay for critical paths. The technology mapper uses the Verilog code produced by the behavioral synthesis tool as input to convert the control part (finite-state machine) to STC code and the datapath part to instruction code for PEs. The mapper outputs a netlist for the next place-and-route tool and Verilog testbench code for verification. The place-and-route tool uses the netlist to determine the actual place of PEs and data transfer between PEs. Finally, the output of the place-and-route stage is converted to configuration data, which can be loaded into DRP-1 for execution.

The verification flow allows to verify the application being developed at many levels including C-level verification, RTL functional verification, block-level functional verification, hardware-level functional verification, and on-chip verification.

3.2.4 IPFlex DAPDNA-2

DAPDNA [64] is a dynamically reconfigurable processor architecture released by IPFlex in 2002. In DAPDNA, sequential processing and large-scale data processing are distinguished and executed in parallel. The sequential processing is performed by a 32-bit RISC processor core called Digital Application Processor (DAP). And, the large-scale data processing is accelerated by a two-dimensional PE array called Distributed Network Architecture (DNA) matrix. The DNA matrix can dynamically reconfigure its datapath and exploit loop- and data-level parallelism.

Fig. 3.17 depicts DAPDNA-2 architecture which is the second generation processor of DAPDNA architecture. It consists of a DAP core and a DNA-matrix. The DAP core has an independent 8-KB of instruction cache, and 8-KB of data cache. The DNA matrix has 32-bit processing elements with multicontext dynamic reconfigurability. The operating frequency of each processor core is up to 166MHz. The DAP core performs system-control and general-purpose operations. The DNA-