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

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

List of Tables

List of Figures

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

Thesis organization	Methodology Path		
	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.

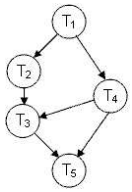


Fig. 2.1: Application's task graph

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

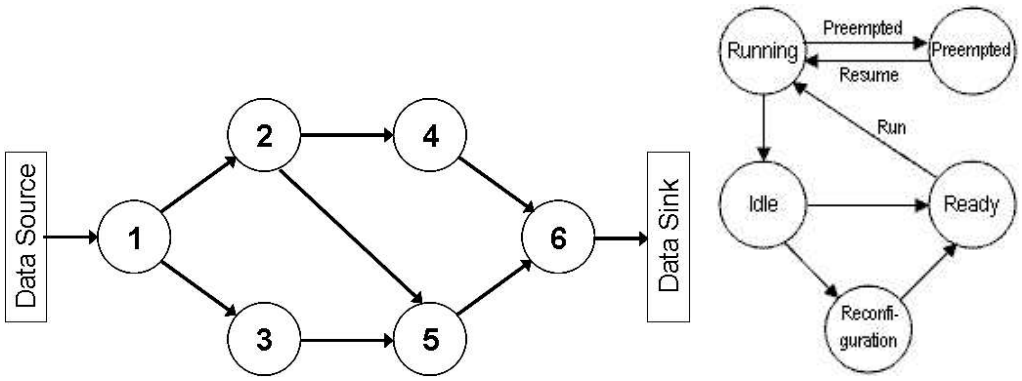
In this thesis, those application are considered as stream applications. That is, data blocks to be processed are iteratively received in a certain interval. The execution of stream applications is often deterministic, or in other words, for a determined input, the same output is always produced. In 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

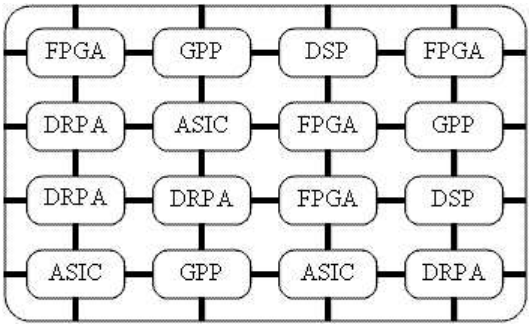


Fig. 2.3: A heterogeneous SoC

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

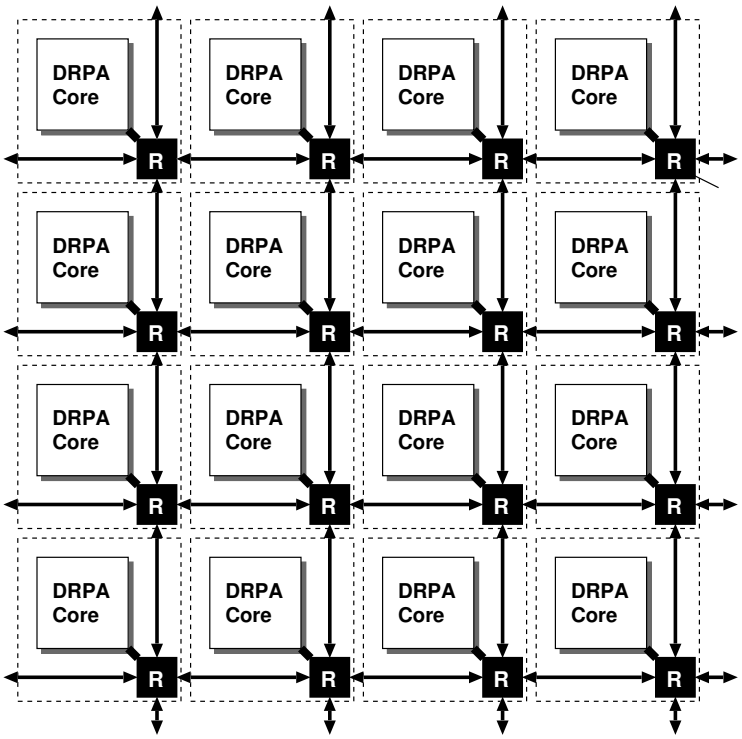


Fig. 2.4: Multicore architecture studied in this thesis

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-

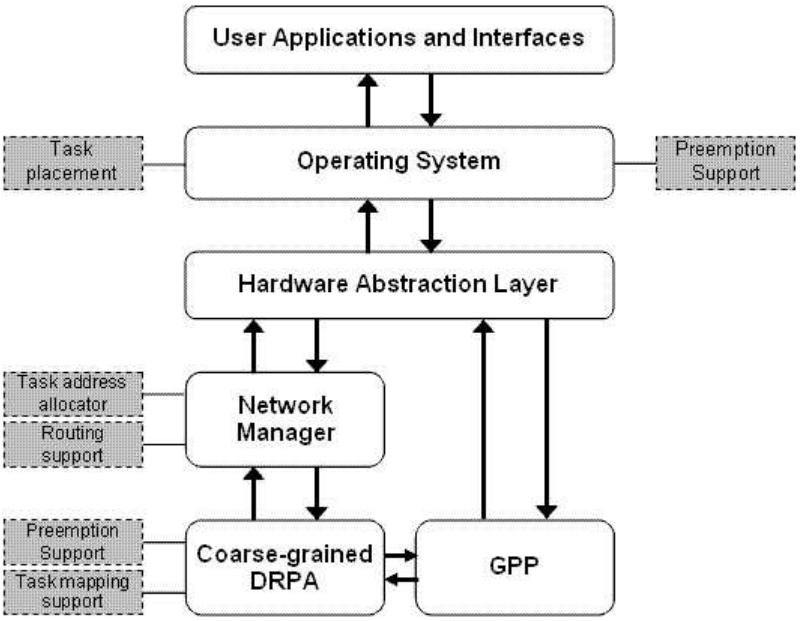


Fig. 2.5: System architecture

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

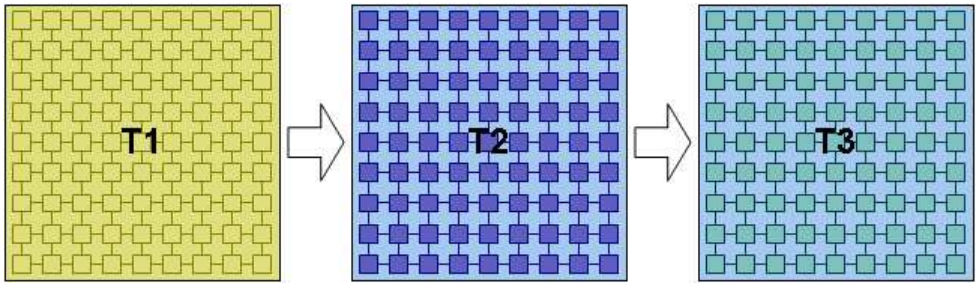


Fig. 2.6: Time-sharing, non-space partitioning model

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,

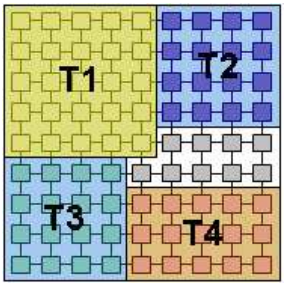


Fig. 2.7: Non-time sharing, static partitioning model

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

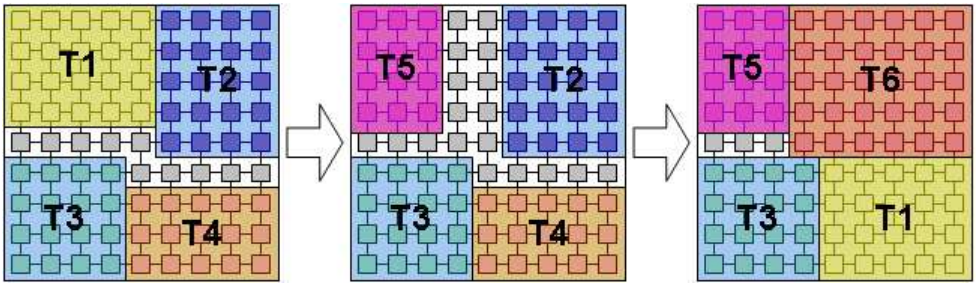


Fig. 2.8: Time-sharing, two-dimensional partitioning model

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

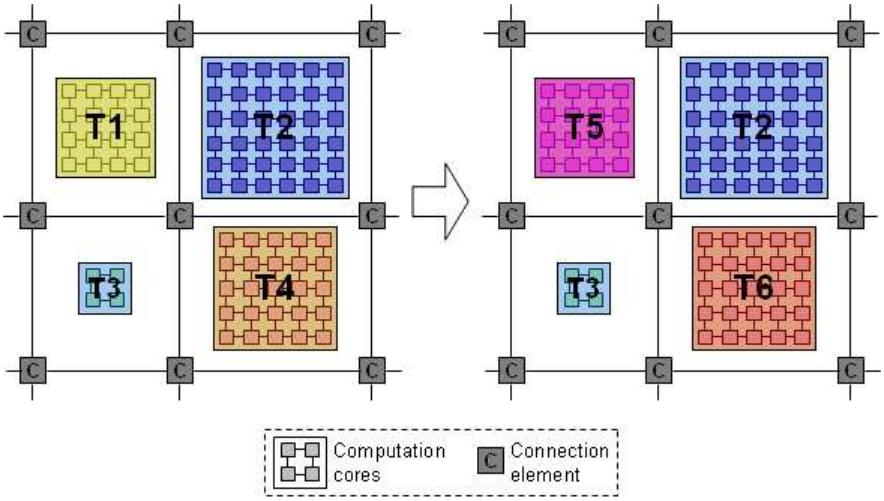


Fig. 2.9: Time-sharing, multicore partitioning model

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

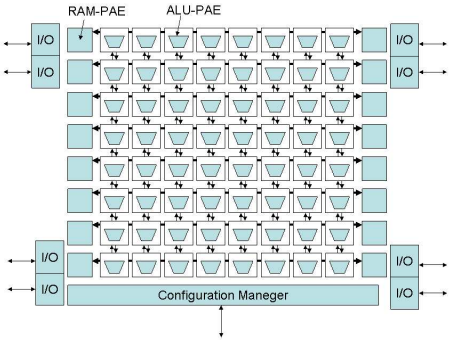


Fig. 3.1: PE array of PACT-XPP

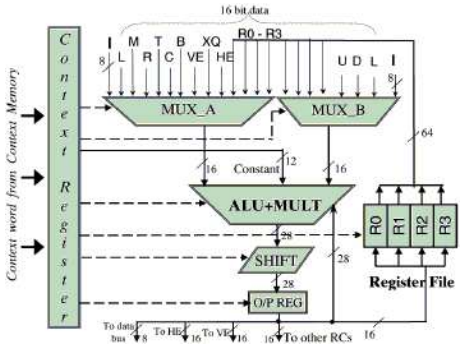


Fig. 3.2: Example of a PE (RC of Morphosys)

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.

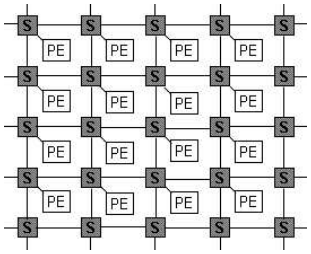


Fig. 3.3: Island-style

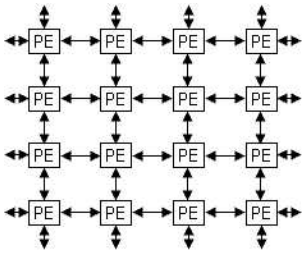


Fig. 3.4: Direct interconnection

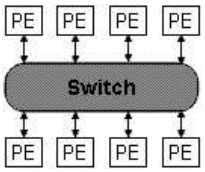


Fig. 3.5: Switch matrix

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

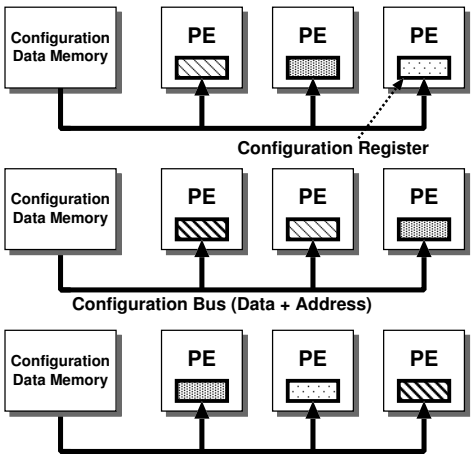


Fig. 3.6: Configuration Delivery Scheme

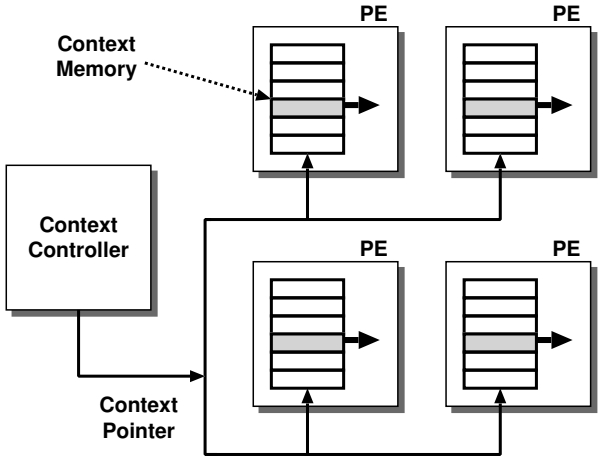


Fig. 3.7: Multicontext Scheme

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

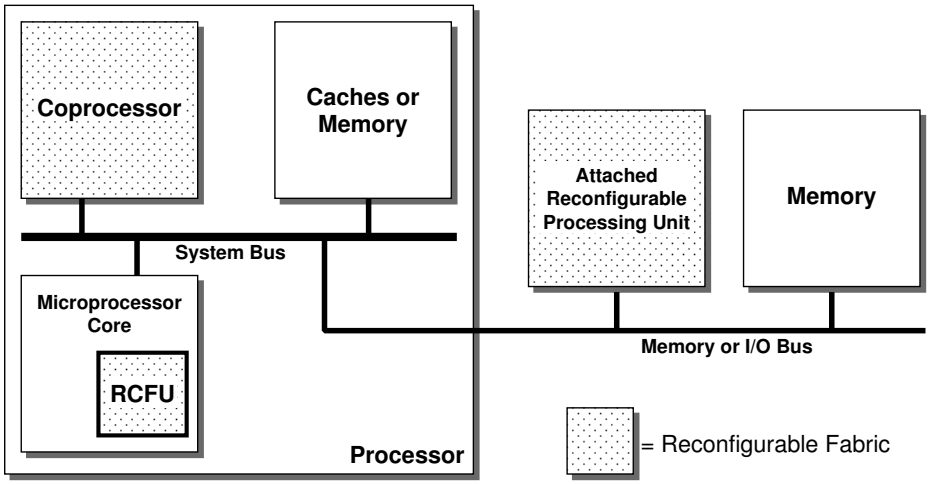


Fig. 3.8: Coupling of Reconfigurable Fabric and Microprocessor

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

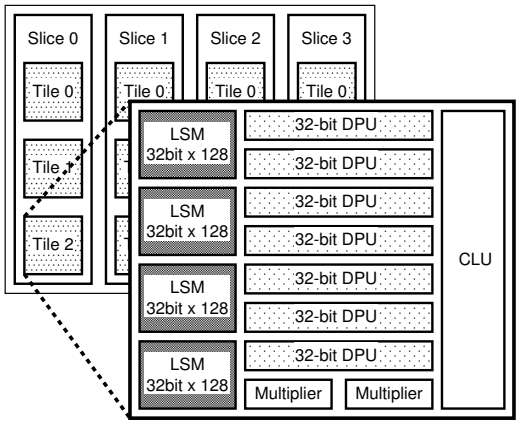


Fig. 3.9: CS2112 reconfigurable fabric

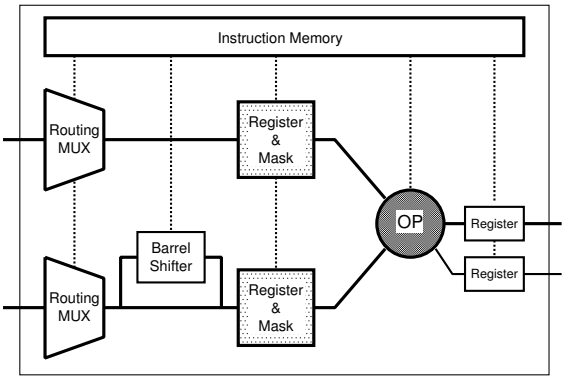


Fig. 3.10: CS2112 datapath unit (DPU)

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.

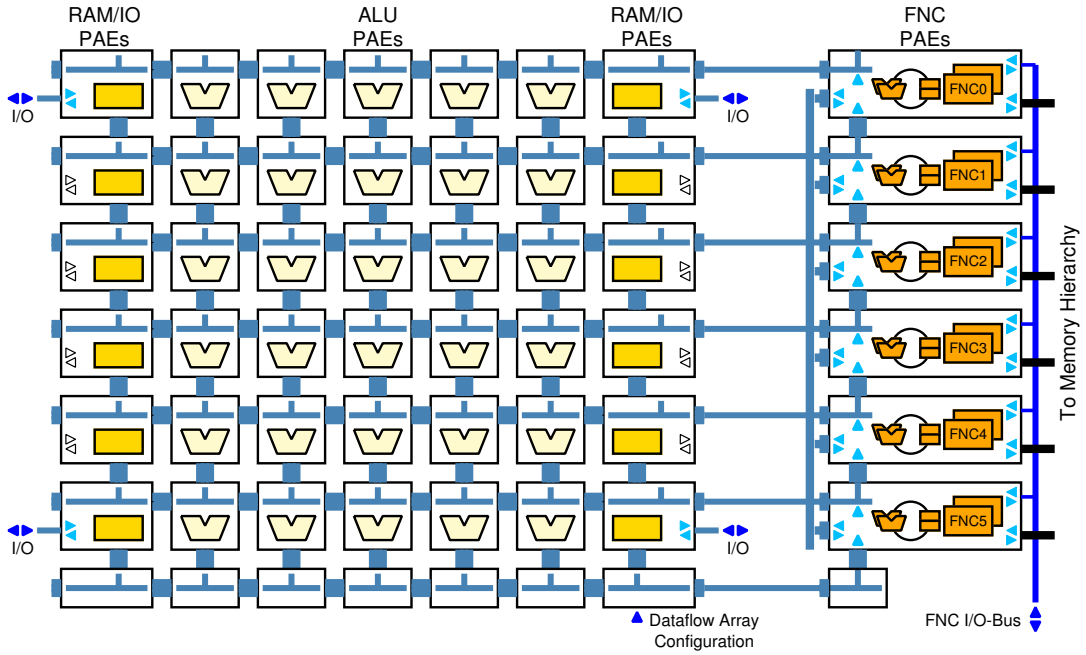


Fig. 3.11: XPP-III Core Architecture Sample

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

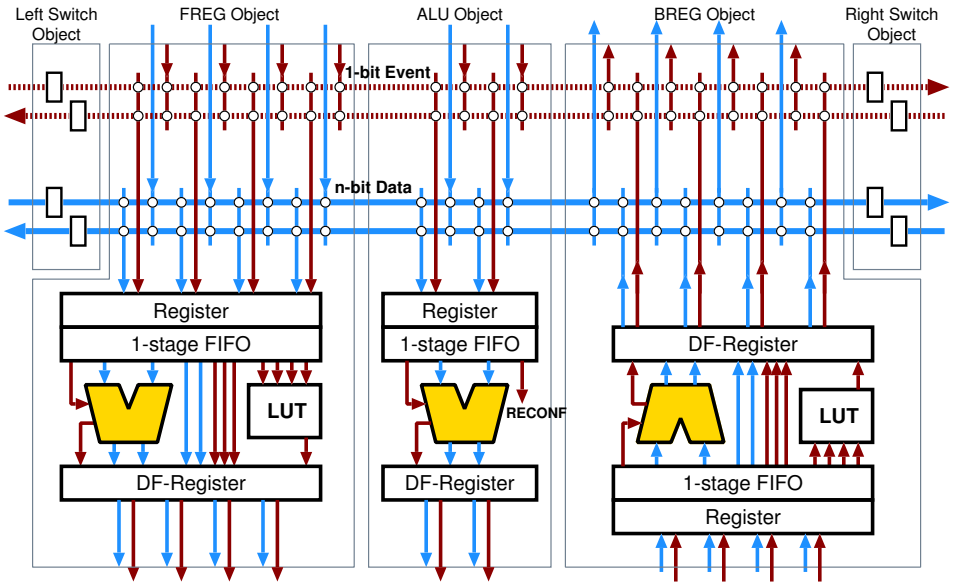


Fig. 3.12: XPP-III ALU-PAE Architecture

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

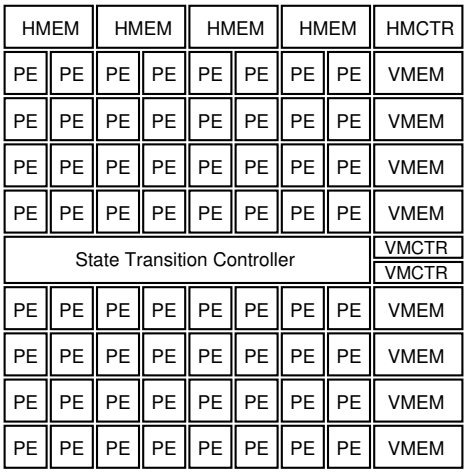


Fig. 3.13: DRP Tile Architecture

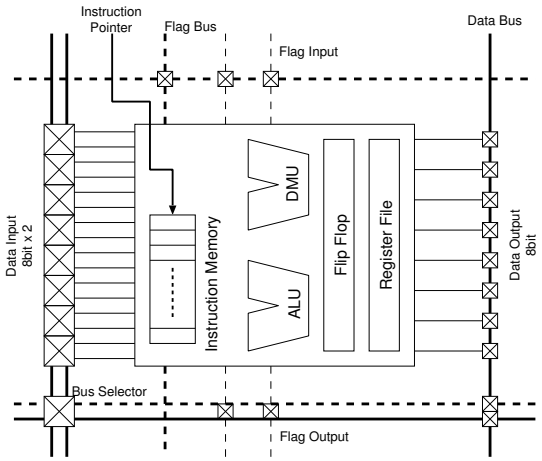


Fig. 3.14: DRP PE Architecture

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

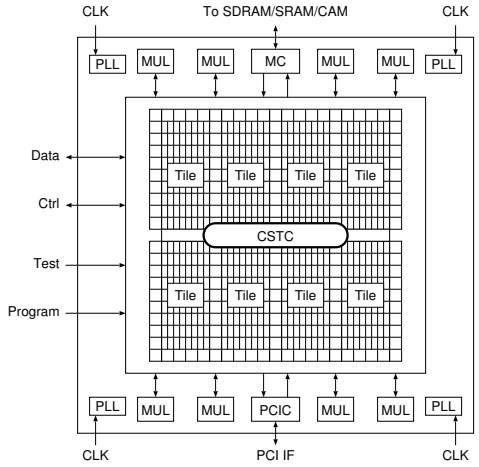


Fig. 3.15: DRP-1 Architecture

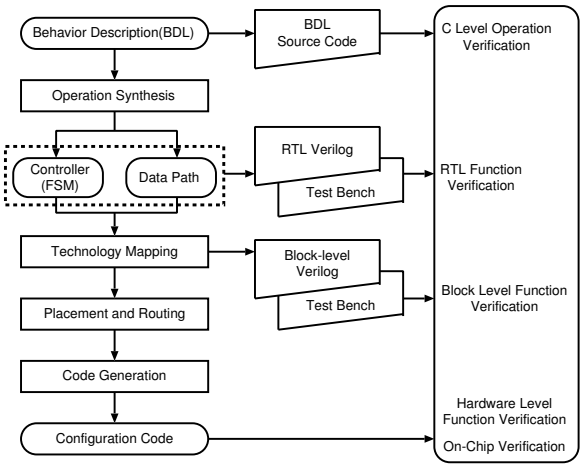


Fig. 3.16: Programming Flow of DRP

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-

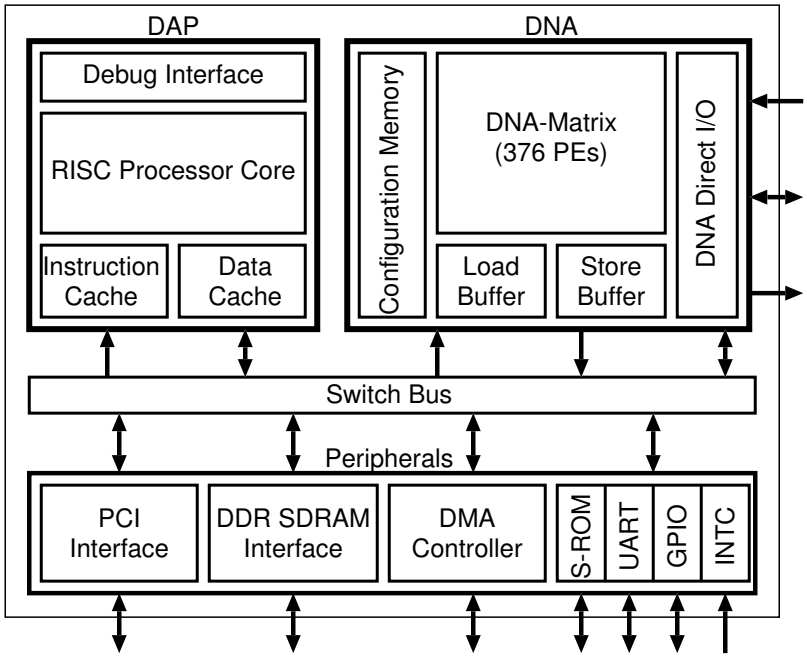


Fig. 3.17: DAPDNA-2 Architecture

Matrix provides capability for high speed data processing and scalability, with parallel computing and dynamic reconfigurability, respectively. This tightly coupled dual-core architecture realizes both flexibility and high performance.

The DNA-Matrix includes several types of processing elements as shown in Table 3.1. To map an algorithmic data flow graph directly onto the hardware, several types of PEs are placed in a two-dimensional PE array consisting of 8 x 8 PEs. This 8 x 8 unit is considered to be a segment, and the DNA-Matrix includes 6 segments, the total number of PEs being 376. There are two major categories of PEs. The first category is data processing where the combination of EXE/DLE/RAM has a complete capability to execute various operations on a data stream. The second is data I/O, and it is possible to generate data streams autonomously.

The DNA-Matrix has rich bus structure to realize various types of algorithms, and the DNA-Matrix Bus consists of a column-wise vertical bus which includes 8 x 32-bit wires corresponding to the row number. The DNA-Matrix Bus covers all rows in a segment, and the source PE can output its signal to all columns in a segment. Therefore, connections between arbitrary PEs are available, regardless of the relative position between source and destination PEs in a segment. The segment boundary connections are restricted slightly to realize a fixed operating frequency. The boundary PEs generate their output at the position of boundary PEs within the neighboring segment.

Configuration data of the DNA-Matrix is in two orders of magnitude smaller than fine-grained FPGAs. With this configuration size, the DAPDNA processor can hold entire DNA configuration data within context memories of each PEs. The DAPDNA-2 holds four contexts in the context memories: one is the foreground and three in the background. The proximity of configuration mem-

Table 3.1: Types of DAPDNA-2 PE

PE	Qty	Function
EXE	168	Arithmetic and logical Ops.
DLY	138	Variable delay for data sync.
RAM	32	Temporary memory
C16E	12	Address generator
C32E	12	Address generator
LDB	4	Data Input from processor bus
STB	4	Data Output to processor bus
LDX	4	Data Input from Direct I/O
STX	4	Data Output to Direct I/O

ory enables the DNA-Matrix to switch configuration in one clock cycle. The trigger of a dynamic reconfiguration event can be generated by two sources: the control signal from DAP core and reconfiguration signal from the DNA-Matrix itself. The total reconfiguration time using the DAP core is usually less than 1μs.

The result of performance evaluations shows the DAPDNA-2 performs more than 28,000 million instructions per second (MIPS) at a clock speed of 166MHz and is capable of more than 9,000 million MACs per second (MMACS) of multiply-accumulate operations with 16-bit inputs and a 32-bit output. And, the DAPDNA-2 can execute image filtering operations such as a Laplacian filter at 1 giga pixels per second. The same operation on a Pentium 4 (3.06 GHz) is topped at 16 mega pixels per seconds and this implies the DAPDNA-2 has 63-times performance gain.

In recent years, IPFlex continues to release enriched DAPDNA-2-based architectures. DAPDNA-3A [65] is based on multicore structure, and consists of 4 DAPs and 4 DNA-Matrices. Each DAP and DNA-Matrix is connected by a reconfigurable interconnection called AXION. Furthermore, the DNA-matrix in DAPDNA-IMS and DAPDNA-IMX [66] is enlarged to 955 16-bit PE array for the special purpose of image processing.

3.2.5 Hitachi FE-GA

Hitachi proposed Flexible Engine/Generic ALU Array (FE-GA) in 2005 [67]. The FE-GA is implemented using TSMC 90-nm Low Power CMOS technology, and it operates at 266MHz with power consumption of 200mW and with an area of 5mm^2.

The FE-GA consists of 24 ALU cells, 8 MLT cells, 10 load/store (LS) cells with a local RAM, a sequential manager (SEQM), a configuration manager (CFGM), a crossbar switch cell (XB), and a system bus interface as shown in Fig. 3.18. Each computing cell (ALU and MLT cells) is connected to its neighboring four cells. Therefore, the length of wires can be extremely short, making for a short delay that provides a high clock frequency. In addition, each of the LS cells, which are interface cells for local RAMs, can be connected to any computing cell on the utmost-left or utmost-right sides via XB connection.

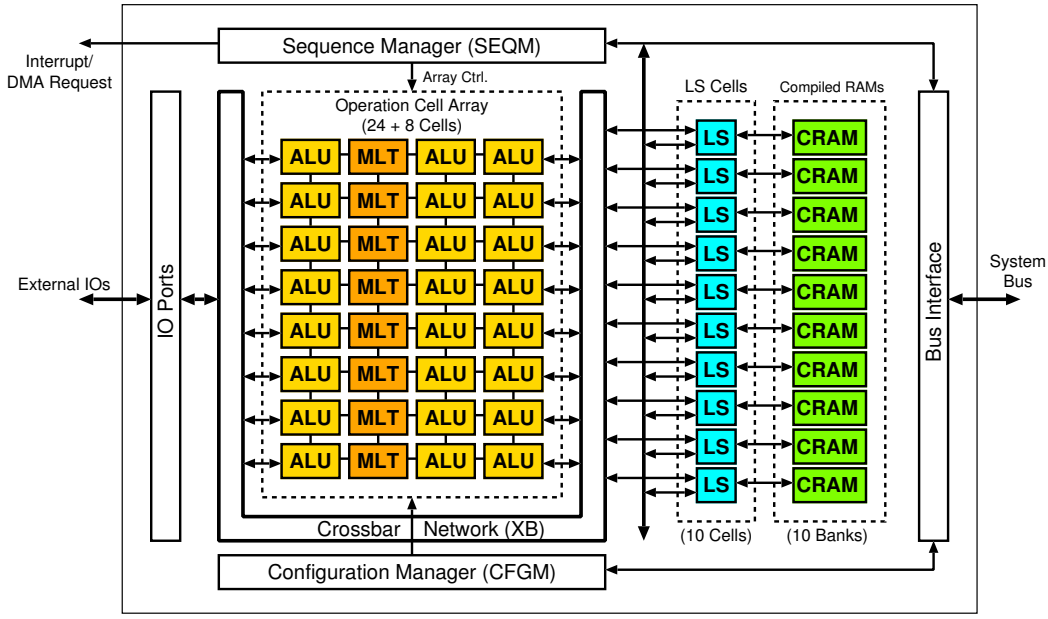


Fig. 3.18: FE-GA Architecture

The configuration data for each cell is managed by the CFGM and transferred to configuration registers (i.e., context memories) of each cell in the background. This configuration data is referenced by a pointer so that it can be compressed to an average of 20% its original size. The context switch of each cell is controlled by the SEQM, which generates a switching trigger using its built-in counters or the computation result of counters or comparisons of ALU cells. The SEQM can consider a conditional branch for the trigger, so the FE-GA can perform run-time context switches flexibly.

Fig. 3.19 shows an ALU architecture of the FE-GA. Each ALU cell consists of a general 16-bit ALU, a shifter (SFT), a data through logic circuit (THR), and an input delay logic circuit. All components of the ALU cells can be operated simultaneously. The data through logic circuit transfers the data from a cell's output to its neighboring cell's input without blocking any ALU operation. In addition, some ALU cells can send their result to the SEQM as a switching trigger that switches the context of all the cells by the result.

Each MLT cell consists of, as similar to a ALU cell, a general multiplier, a data through logic circuit, and an input delay logic circuit. All components of the MLT cells can be operated at the same time as ALU cells. Each LS cell shown in Fig. 3.20 consists of a port (A0/A1) connected to the XB, a port (B) connected to the bus interface and a port (0/1) connected to a dual-port RAM (local RAM). Because the LS cells can generate load and store addresses without using the ALU cells, the ALU cells can be saved for user applications.

Evaluation results based on TSMC 90-nm LP CMOS technology demonstrates that the operational frequency is 266MHz in worst cases, and the performance is 17.0 GOPS for 16-bit integer operations. The estimated area of the FE-GA is 5.0mm^2 including 40-KB local RAM, and power consumption is estimated as 210mW. In addition, several signal processing applications such as FIR

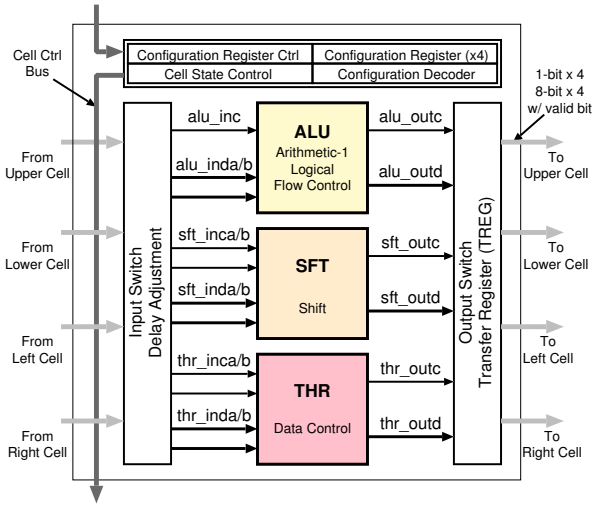


Fig. 3.19: FE-GA ALU Cell Architecture

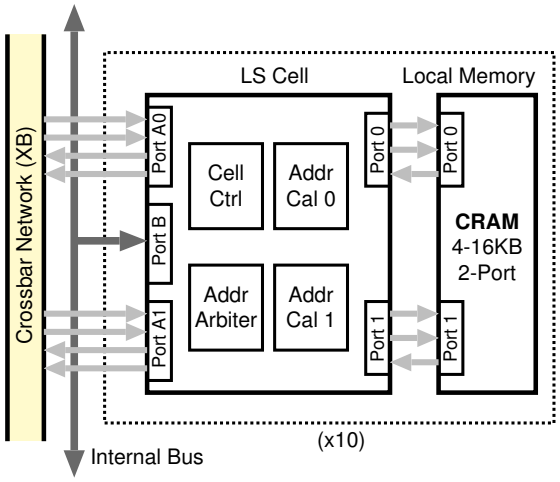


Fig. 3.20: FE-GA LS Cell Operation

filters and DCT have been implemented on the FE-GA, and it can perform at about 10 times fewer clock cycles than a DSP.

3.2.6 Elixent D-Fabrix

Elixent D-Fabrix[2] [68] reconfigurable processor is an evolution of the CHESS architecture [69] developed by Hewlett-Packard Laboratories.

The goal of the CHESS architecture was to increase both arithmetic computational density and the bandwidth and capacity of internal memories significantly beyond the capabilities of current FPGAs, while enhancing flexibility. Toward the goad, the fundamental computational unit is a 4-bit ALU with a primary set of 16 instructions. This provides efficient arithmetic capabilities suitable for cascading to useful media operations, or for supporting nibble-serial implementation styles. The entire user-visible routing structure is also based on 4-bit buses.

Fig. 3.21 shows the CHESS chessboard- or checkerboard-style reconfigurable arithmetic architecture. Each ALU is adjacent to four switchboxes, and each switchbox is adjacent to four ALUs. This allows very powerful local connectivity, with each ALU having input and output buses on all four sides, and being able to send data to or receive data from any of the eight surrounding ALUs as shown in Fig. 3.21 by the dotted arrows.

At run time, any switchbox in a CHESS array can be used as a 4-bit x 16-word memory. In this mode, all of the switches in the switchbox are disconnected, although the buses running over the switchbox can still be used. These memories are distributed through the array, attached to user plane wiring buses, and typically controlled by surrounding ALUs which generate an address and R/W control. The above design above provides one block RAM per 16 ALUs[3], using about 25% more

[2]The technologies of the D-Fabrix is now licensed by Matsushita Electric Industrial Co., Ltd.

[3]As like MeP-combined version described later, in recent D-Fabrix arrays, the basic block called Tile consists of 2 ALUs and 2 Swichboxes, and every 2 x 4 Tile has one 256 bytes block RAM.

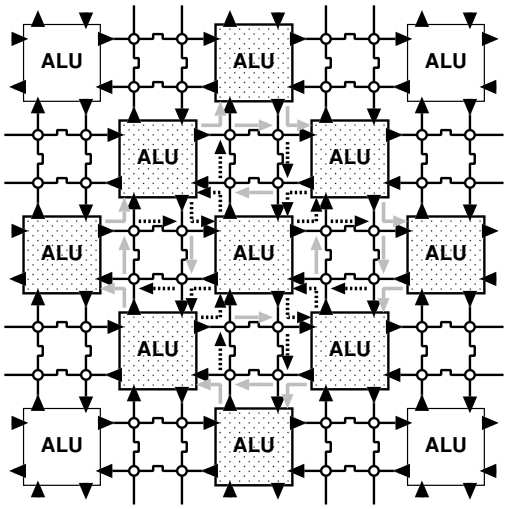


Fig. 3.21: Chessboard-style ALU Array

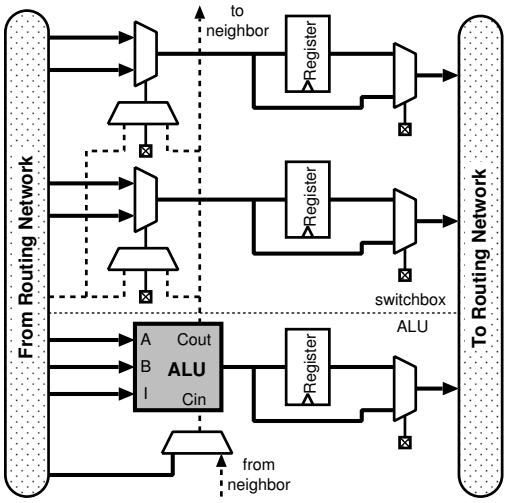


Fig. 3.22: D-Fabrix ALU and Switchbox

area than the basic array without the block RAMs.

Fig. 3.22 illustrates the ALU and Switchbox pair of Elixent D-Fabrix architecture. The architecture is almost the same as the CHESS architecture, for examples, 4-bit ALU array and chessboard-style interconnections. There is, however, one significant difference between the D-Fabrix and the CHESS. The multiplexers that select signals from the routing network to pass to the pipeline registers are statically configured in the CHESS, but the D-Fabrix allows them to be dynamically controlled, using either an ALU Cout or a signal from the routing network as a multiplexer control signal.

The area of the basic cell (ALU plus switchbox) of the CHESS architecture is 0.045 mm^2 in a process with 0.35μm feature size. The simulation results show that an operational frequency is 200MHz, and the CHESS architecture has a substantial performance advantage over some FPGAs in an equivalent semiconductor technology. In the D-Fabrix architecture, dynamic switchbox multiplexers have resulted in a slight overall increase in the area of a combined ALU and Switchbox. This increase is due to the addition of the circuits that select control inputs to switchbox multiplexers and their configuration memories. The area increase is no more than 10%. Compared to the ALU, the switchbox multiplexer is one-third of the size and it has half the propagation delay and 40% of the power consumption.

The D-Fabrix ALU array is actually integrated into Toshiba's Media Embedded Processor (MeP) [70] and Matsushita's UniPhier platform [71, 72] as a flexible hardware engine. Toshiba developed a reconfigurable LSI called ET1, in which the D-Fabrix works as an extension unit of MeP Core. Fig. 3.23 shows a block diagram of ET1. The D-Fabrix of the ET1 consists of 576 Tiles, which means 1152 ALUs, and 72 Block RAMs. In [70], it is demonstrated that the area-efficiency of D-Fabrix compared to an ASIC ranges from 1/2.8 to 1/16.3, and the maximum operational frequency ranges from 1/1.7 to 1/5.8.

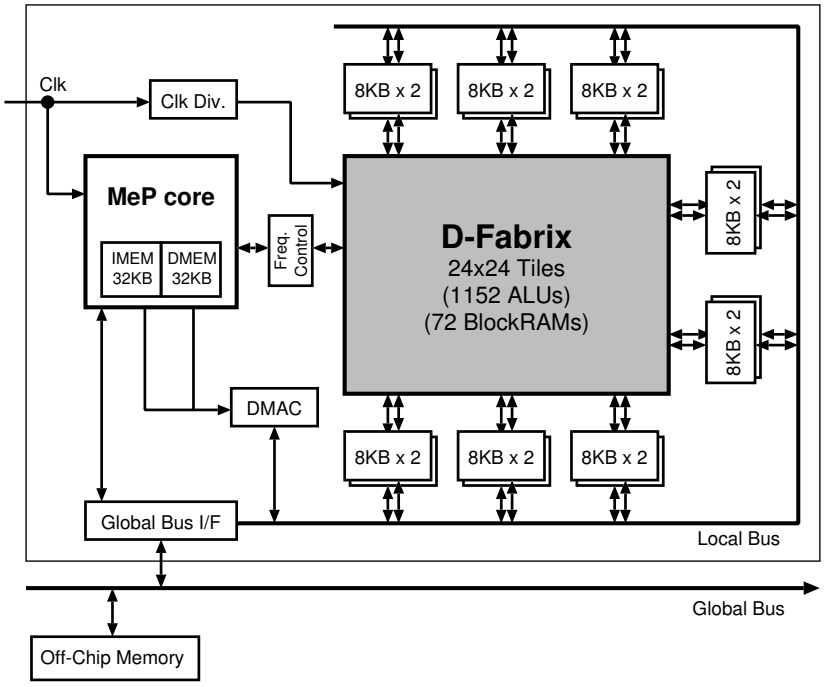


Fig. 3.23: ET1 Architecture with MeP core and D-Fabrix

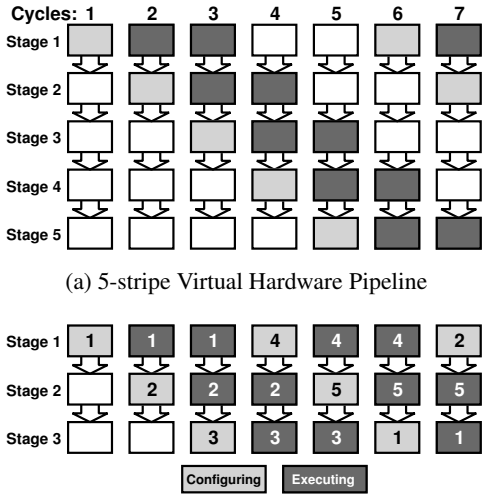
3.2.7 Rapport Kilocore

Rapport Kilocore [73] architecture is a commercialized reconfigurable processor derived from PipeRench [74], which has been proposed at Carnegie Mellon University in late 1990's.

PipeRench is based on an unique virtual hardware model called virtual pipelines described in Fig. 3.24. Fig. 3.24(a) shows a 5-stage (or stripe) virtual pipeline. Fig. 3.24(b) illustrates the first 7 cycles of reconfiguration of a 3-stage physical hardware pipeline executing the 5-stage virtual pipeline. Reconfiguration is performed by storing the configuration data of the entire virtual pipeline in an on-chip configuration data memory, and transferring them from the memory to the physical fabric every cycle.

Fig. 3.25 depicts a reconfigurable fabric of KC256, which is the first generation of the Kilocore architecture and virtual pipeline model. In Kilocore architecture, the functionality in each stripe consists of 16 8-bit PEs. All PEs within a stripe are interconnected within that stripe, which facilitates easier placement and routing of operations.

Fig. 3.26 is a block diagram of KC256's 8-bit PE. All selected inputs to multiplexers and shifters in this figure are connected to configuration bits stored in that PE. Special purpose interconnects are used to combine adjacent PEs to perform operations more than 8 bits in width. The shifters are also connected to the corresponding shifters in adjacent PEs to allow for efficient multi-PE shift operations. Each PE contains a register file with eight registers. The output from the functional unit can be written to any one register in the register file in that PE. If the value from the functional unit is not written to a register, the value from the corresponding register in the previous stripe is latched



(a) 5-stripe Virtual Hardware Pipeline

(b) 4-Stripe Physical Hardware Pipeline

Fig. 3.24: Virtual Pipeline Model

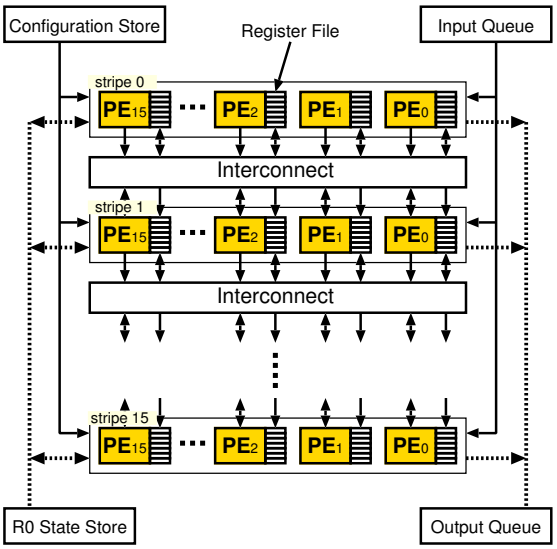


Fig. 3.25: Kilocore KC256 Architecture

into that register. The functional unit in each PE consists of eight 3-input LUTs that are identically configured. The functionality of the PE is specified by 42 configuration bits, which means that each stripe has 672 bits of configuration.

The process of configuring a stripe takes one cycle, so the pipeline can be configured one cycle before the first data of the pipeline arrives at that stage. If the virtual pipeline is larger than the real hardware, physical stripes will eventually be reconfigured with new virtual stripes. The state of the over-written virtual stripes are preserved by writing the value in R0 into the R0 state store memory. The state will be restored when the escaped virtual stripe is returned to the fabric. The process of reconfiguration continues until the last stripe in the virtual pipeline is configured. After that, reconfiguration of a physical stripe starting with the first virtual stripe will occur, and the computation proceeds with new inputs.

The KC256 is fabricated in a six-metal layer 0.18 μm CMOS technology. The total die area is 7.3 x 7.6 mm^2, and transistor count is 3.65 million. The fabric clock is designed to operate at 120 MHz under worst-case voltage and temperature conditions. At 120 MHz, the KC256 executes a 40-tap 16-bit FIR filter at 41.8 mega samples per second, which means that its performance is in the same range as Texas Instruments C64x family DSPs. The KC256 performs the IDEA encryption and decryption at 450 Mbps. By comparison, a 800-MHz Pentium-III processor executes it at a rate of 75.4 Mbps.

3.2.8 IMEC ADRES

IMEC ADRES [75] has two functional views as shown in Fig. 3.27: a VLIW view and an accelerator view as an array of reconfigurable cells (RCs). While the VLIW processor is optimized for control and load/store operations, the accelerator is optimized for data-parallel processings. In acceleration

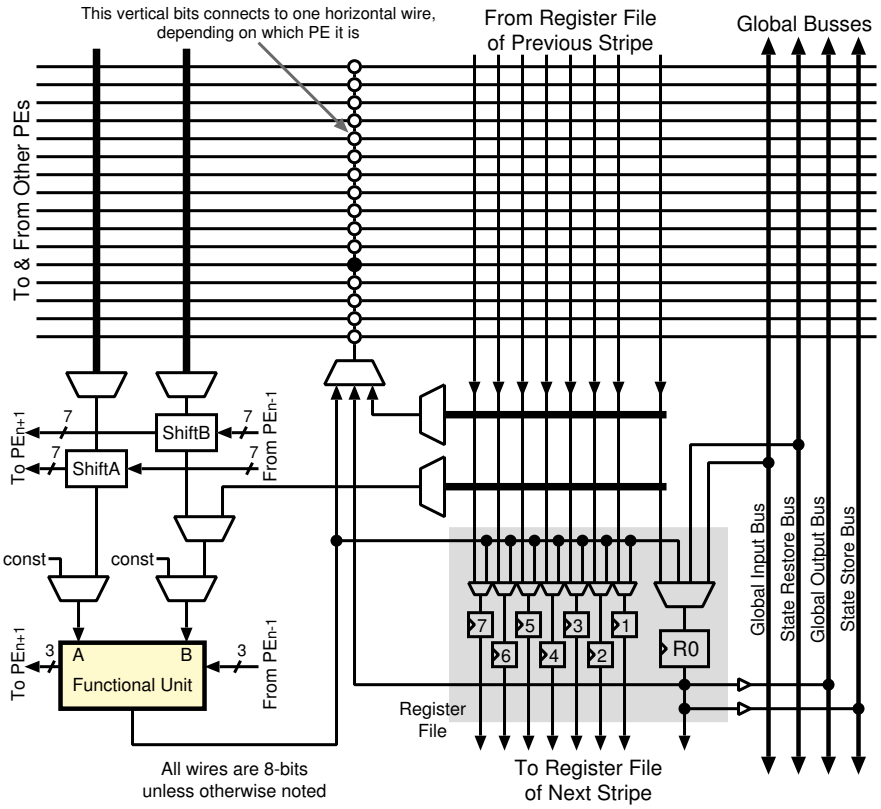


Fig. 3.26: Kilocore KC256 PE Architecture

mode, the functional units of the VLIW form the first row of the array. Orthogonal buses facilitate data transport within the array. The VLIW part includes up to eight functional units organized in a row. The units communicate with each other through a horizontal data bus for data exchange. A part of the units can communicate vertically with a common register file for data load and store.

The attached array is composed of several rows of RCs organized in a matrix form. The behavior of each RC is controlled by a locally stored set of several contexts. During the execution phase, a central controller allows dynamic reconfiguration of the RCs within a cycle. Each RC, as shown in Fig. 3.28, includes a local context memory (denoted by configuration RAM in the figure), an ALU, input and output multiplexers, and a register file for local data storage.

The data exchange between RCs is done with orthogonal interconnect-networks. There are two levels of interconnect for internal data exchange between the units: a global bus for each row or column spans the entire array. Additionally, the array is subdivided into four segments. Within a segment, a local interconnect is provided such that an RC can get input data directly from each of its horizontal or vertical neighbors.

The RC has been implemented with the Infineon Technologies CMOS 130nm 6 metals process technology. A context memory is designed with an 40-bit x 32-word SRAM. The schematics show that one RC requires 63266 transistor counts, and the total area is 0.196 mm^2. The area breakdown of the RC components is as follows: 50% for the context memory, 6% for external interfaces for

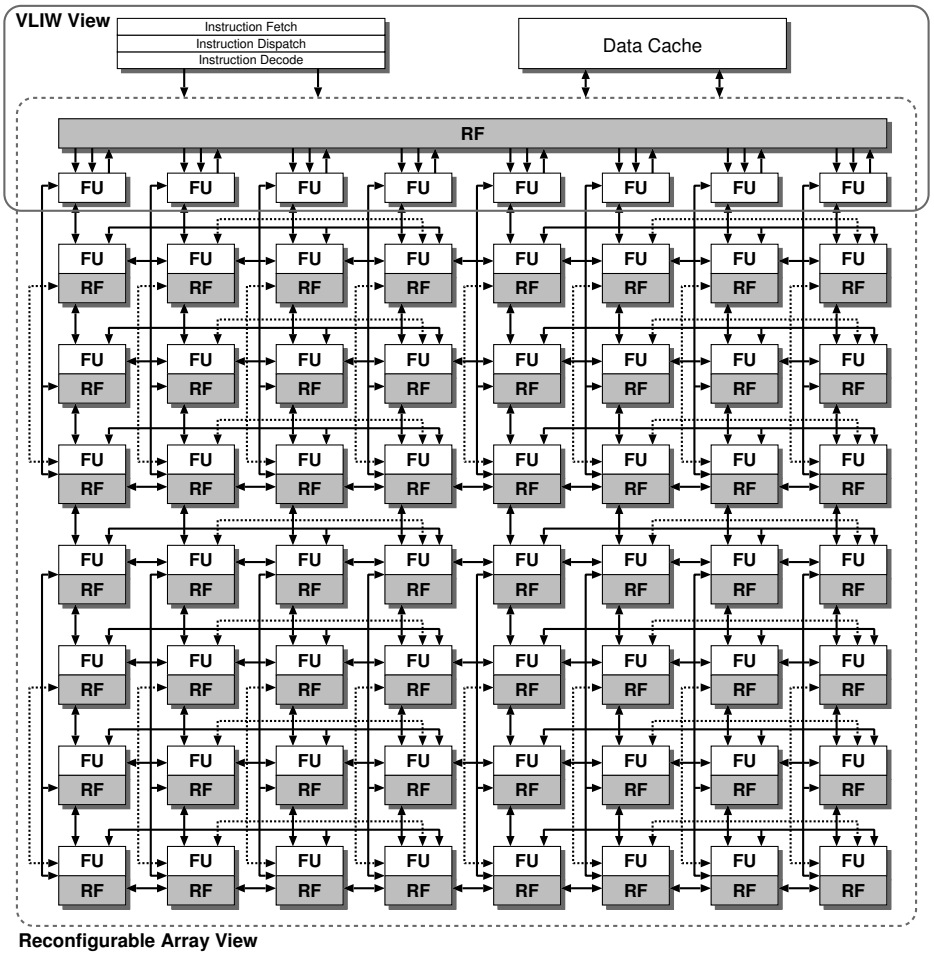


Fig. 3.27: ADRES Core Architecture

input and output, 9% for a register file, and 19% for an ALU. About 15% of the area of an RC are consumed by the interconnect between the components.

The maximum operational frequency of the RC is 60MHz or 16-ns cycle time. At a target frequency of 100 MHz, the power consumption for one RC is about 1.7mW. For performance evaluations, an H.264/AVC video decoder has been implemented on the ADRES architecture. The evaluation results show that the decoding cycle for the array view is about 88% faster than the VLIW view. The required clock frequency is 184MHz in the VLIW view and 98 MHz in the array view. The average power consumption at 100MHz would be around 46.3mW.

3.2.9 Stretch S5/S6 SCP Engine

The S6000 family configurable processors [76] are powered by Stretch S6 Software Configurable Processor (SCP) Engine. As shown in Fig. 3.29, they incorporate a Tensilica Xtensa LX dual issue VLIW processor core and a second-generation Instruction Set Extension Fabric (ISEF). The ISEF is a software-configurable computing fabric that contains 64KB of embedded RAM (IRAM). Using the ISEF, system designers can extend the processor instruction set and define new instructions using

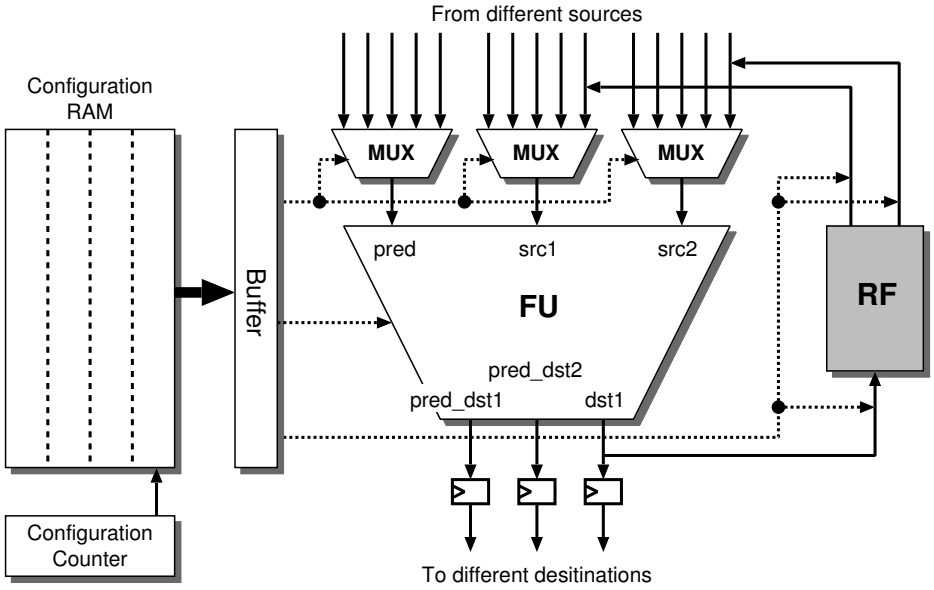


Fig. 3.28: ADRES Reconfigurable Cell

only their C/C++ code. As a result, developers get the performance of custom hardware with C/C++ development simplicity.

The S6 SCP Engine within the S6000 family, described in Fig. 3.30, contains a Xtensa LX VLIW core and an ISEF. It is the ISEF that provides the dramatic application acceleration by allowing user algorithms to be instantiated in hardware and called by the processor as single instructions. The Stretch ISEF, being tightly coupled to the processor, only needs to host compute-based and logic functions.

The S6000 ISEF contains 4096 ALU-based PEs that, in addition to traditional ALU functions, can be configured to perform 2 x 4 multipliers and cascaded for larger data width. In addition, there are 64 dedicated multipliers capable of 8 x 16 operations that can be cascaded to increase data width. Distributed state registers provide local storage for intermediate values and coefficients. Connectivity of the PEs is enhanced with distributed multiplexers, priority encoders, and shifters.

For computation-intensive applications, the S6 ISEF is fed by the same 32 128-bit wide registers carried over from previous generations of Stretch devices [77]. These registers are used for loading data into the ISEF, and their presence in the S6000 ensures maximum compatibility and code reuse from previous software configurable processor designs. The S6000 ISEF also contains 64KB of embedded ISEF RAM (IRAM) distributed throughout the fabric in 32 banks of 2KB each. The IRAM is the memory mapped into the S6SCP address space, so can be loaded directly by the processor.

The ISEF supports dynamic reconfiguration based on the configuration delivery scheme. If a fetched opcode corresponds to an extension instruction that is not resident in the ISEF, an instruction fault is raised. The operating system will then save the contents of any internal state registers, find the extension instruction group containing the missing instruction, and initiate an ISEF reconfiguration

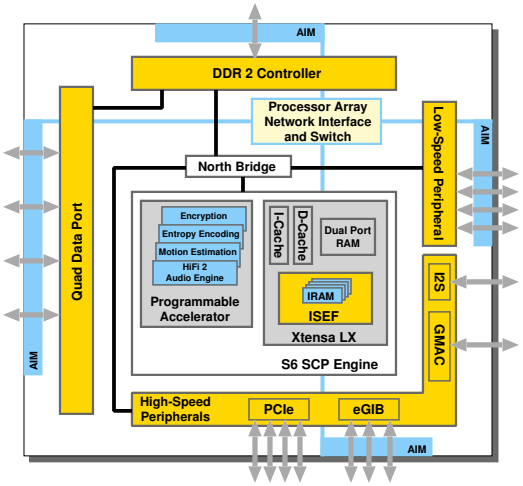


Fig. 3.29: S6000 Architecture

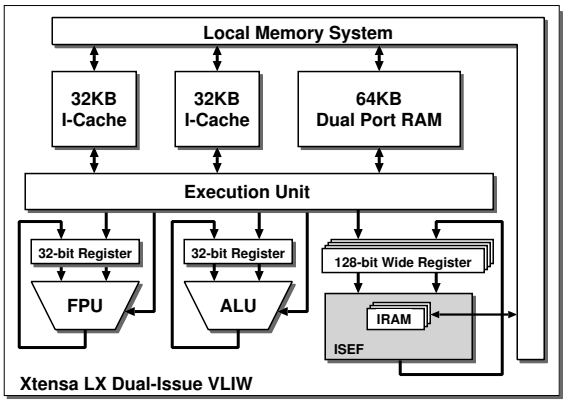


Fig. 3.30: S6 SCP Engine

before resuming the application program. The ISEF can be completely reconfigured in 27μs.

3.2.10 Fujitsu Cluster Architecture

Fujitsu has proposed a coarse-grained multicontext dynamically reconfigurable processors suitable for wireless communications [78]. The architecture consists of a hierarchical PE array called a cluster.

Fig. 3.31 shows a block diagram of the cluster. It contains several kinds of PEs, inter-PE networks, and a sequencer. To increase the cluster's area-efficiency, heterogeneous PEs are used in a cluster. The PE can be a 16-bit ALU, a MAC, a selector, a shifter, an address generator (counter), a one-port memory, a two-port memory, a register file, and variable delay line. The ALU supports logical, arithmetic, compare, and shift instructions, similar to DSPs. The "compare" instruction generates a signal that is used as a select signal by the selector and/or a transition signal by the sequencer. The MACs support 16 x 16 bit multiplication and accumulation.

The topology of inter-PE networks is a kind of indirect three-cube network and consists of three selector levels. The number of logical steps is the same in all paths. In the first implementation, the network corresponding to a 64 x 64 switch that consists of 4 x 4 switches. The sequencer controls the dynamic reconfiguration of the PE and inter-PE networks. The sequencer consists of a sequence control program memory, program counter (PC), and branch control unit. The value of the program counter is identical to the address of context memories in the PEs.

In this architecture, a hierarchical structure of the cluster as the cluster group is adopted to process larger algorithms that do not fit within a cluster. Fig. 3.32 illustrates the cluster group. Inter-cluster networks are organized through crossbars. The sequencer in each cluster also controls the context switch of each crossbar which has five direction inputs/outputs (from/to upper, lower, left, right, and into/out of the cluster). There are no restrictions on the data transfer direction between the clusters,

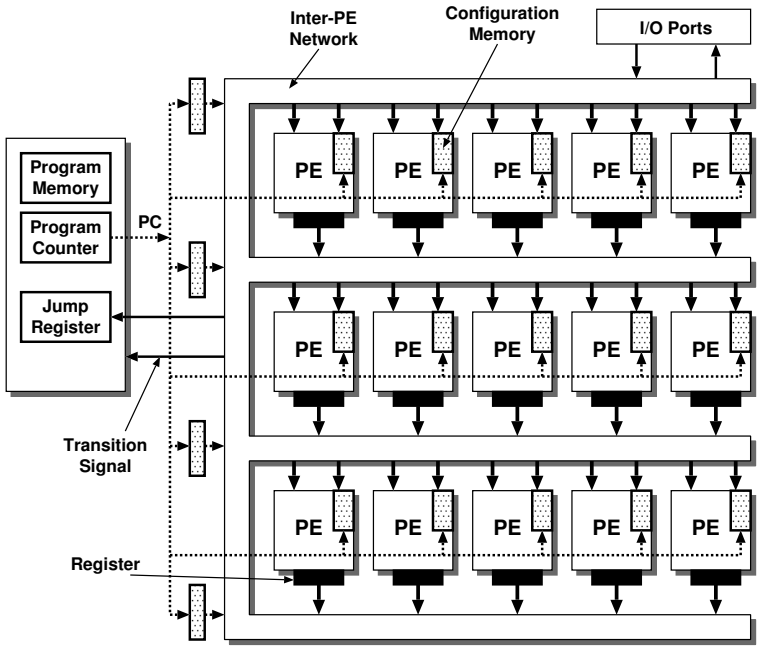


Fig. 3.31: Fujitsu Cluster Architecture

so the feedback paths that often appear in wireless communication baseband signal processing can be used. Each cluster can operate both independently and cooperatively in arbitrary combinations. Inter-cluster data transfer needs two or three additional cycles.

The shared resources located on the inter-cluster network are used by each cluster. There are two kinds of shared resources. One is a memory that can be shared by plural clusters. The other is dedicated hardware to accelerate signal processing.

The Fujitsu cluster architecture has been evaluated with algorithms that appear in the physical layer of IEEE802.11a and b wireless LAN processing. The evaluation results review that the cluster architecture provides a short latency compared to the DAPDNA-2 introduced in Section 3.2.4.

3.2.11 MuCCRA platform

The Multi-Core Configurable Reconfigurable Architecture (MuCCRA) project aims at developing a methodology and a framework to design highly configurable DRPAs toward varieties of target applications [79]. Because MuCCRA is developed for examining an optimized architecture for a given set of applications, several prototypes toward different needs have been designed and evaluated. Namely, prototypes introduced in this section are MuCCRA-1, MuCCRA-2 and MuCCRA-D.

3.2.11.1 MuCCRA-1

MuCCRA-1 is the first prototype chip of MUCCRA project adopting an island-style interconnection architecture for connect PEs [79, 39]. The MuCCRA-1 architecture shown on Fig. 3.33 consists of a 4×4 24-bit PE array, four 24-bit dedicated multipliers (MULTs) on the left side and four 24-bit

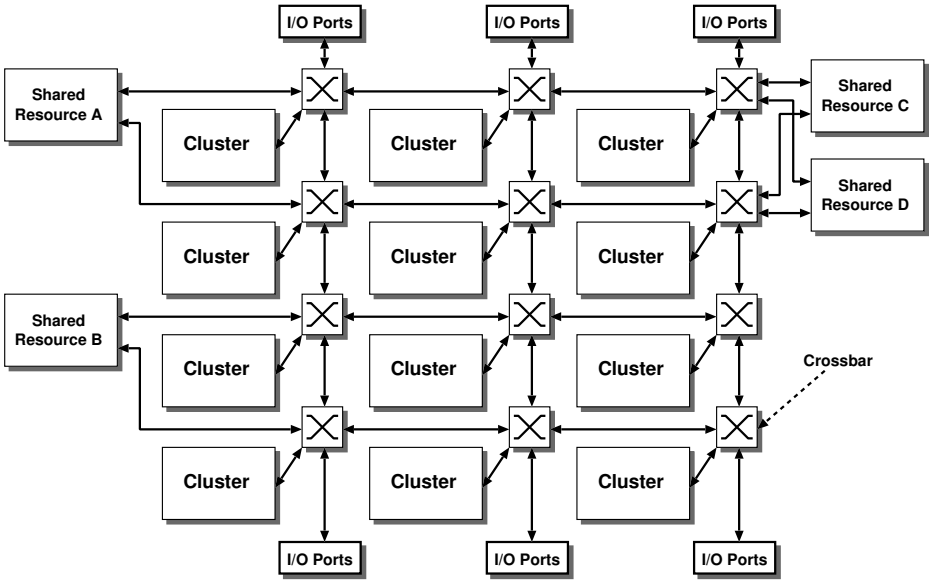


Fig. 3.32: Fujitsu Cluster Group

distributed memory modules (MEMs) at the bottom of the PE array. Each MULT unit can multiply two 24-bit data words. Each MEM is 24-bit $\times$ 256-word and 2-ported module.

A PE consists of a shift and mask unit (SMU), an arithmetic logic unit (ALU) and a register file unit (RFU) as shown on Fig. 3.34. A SMU supports various types of shift and mask operations; and, an ALU provides basic arithmetic operations, comparisons, and logical operations. Both a SMU and an ALU can perform half-word operations, in which a 24-bit datum is treated as two 12-bit data. A RFU is a 24-bit wide and 8-entry deep register file with a read/write port and a read-only port. For avoiding combinatorial loops, an output of the ALU can be connected to an input of the SMU, but the opposite connection is not allowed. On the other hand, each RFU can connect with all of inputs and outputs of the ALU and the SMU.

Each PE has a 64-entry context memory to provide multiple sets of configuration data for configuring SMA, ALU, RFU and interconnection. Being a multicontext device, context pointers are provided to every PE to control context switching. In every clock cycle, a configuration data pointed to by a context pointer is read out from the context memory. Doing this allows hardware context to be changed in a clock cycle.

MuCCRA-1 uses an island-style interconnection network. The input and output of each PE are connected to a near channel via a connection block. Switching Elements (SEs) are placed at the intersections of channels to route data to different directions according to configuration data provided.

3.2.11.2 MuCCRA-2

In order to reduce the size of the chip, MuCCRA-2 has the bit width and the context size smaller than correspondent parameters in MuCCRA-1 [79, 80]. Specifically, MuCCRA-2 uses a 16-bit architecture and allows only 16 contexts for context memories. In addition, instead of providing separated

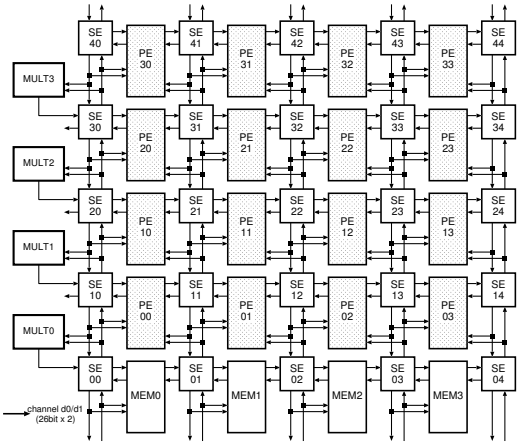


Fig. 3.33: MuCCRA-1 Architecture

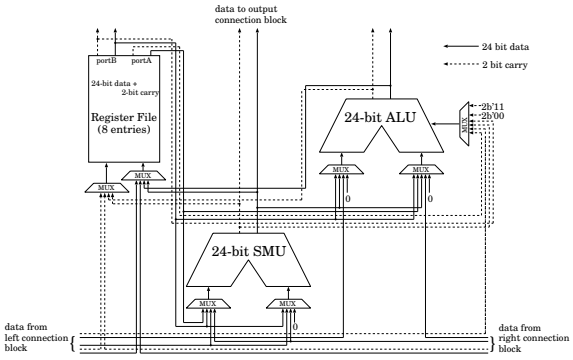


Fig. 3.34: PE architecture of MuCCRA-1

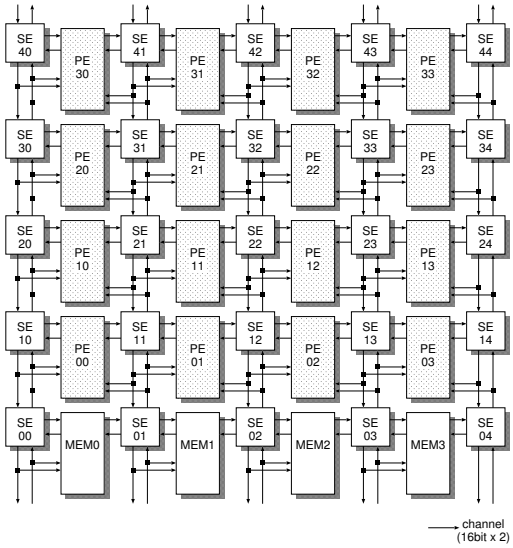


Fig. 3.35: MuCCRA-2 Architecture

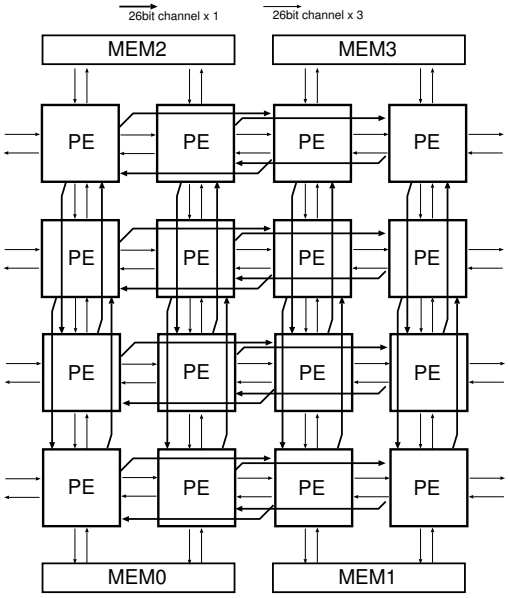


Fig. 3.36: MuCCRA-D Architecture

multipliers as in MuCCRA-1, multiply operations can be performed in every PE. Apart from that, the architecture of a PE in MuCCRA-2 is the same that in MuCCRA-1. The architecture of MuCCRA-2 is described in Fig. 3.35.

Similar to MuCCRA-1, an island-style interconnection network is exploited in MuCCRA-2 to connect PEs. However three routing channels are available on MuCCRA-2 instead of two routing channels on MuCCRA-1.

3.2.11.3 MuCCRA-D

Using an island-style interconnection network in the PE array of MuCCRA-1 and MuCCRA-2 causes two problems: the area overhead for routing wires and switches is large, and the operating frequency is reduced because of long delay caused by switches. To solve these issues, a direct interconnection

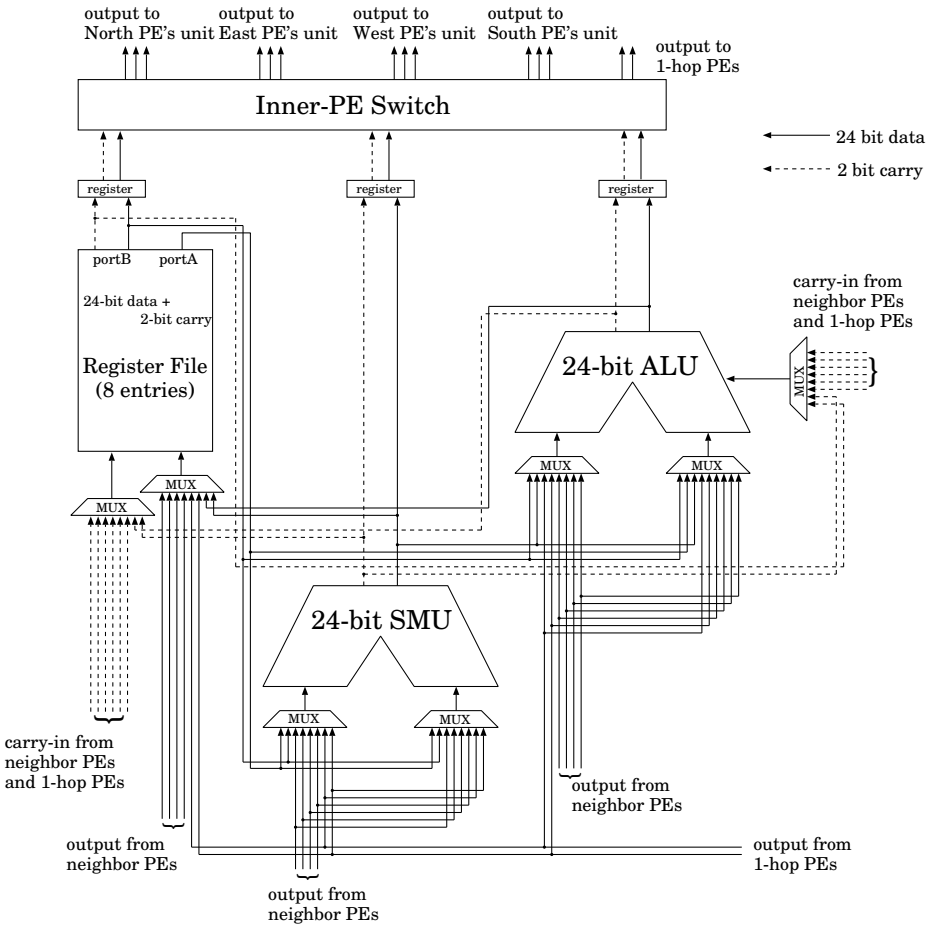


Fig. 3.37: PE architecture of MuCCRA-D

is applied in MuCCRA-D [39].

The architecture of MuCCRA-D is presented on Fig. 3.36. It composes of a 4 ×4 PE-array and 4 distributed memory modules on the top and bottom of the array. Distributed memory modules are the same as those in MuCCRA-1. Each PE is directly connected to the nearest neighboring PEs. Moreover, connections to the other PEs in the same row and column are also provided to reduce the delay for transferring data to remote PEs. Three independent channels are provided for the the nearest neighboring PEs while there is only one channel for one-hop distant PE.

Each PE of MuCCRA-D consists of a PE core, a 64-entry context memory and an inner-PE switch for selecting the destination of output data. Similar to MuCCRA-2, PEs in MuCCRA-D are equipped with multiply capability. The architecture of a PE is presented on Fig. 3.37.

3.3 Summary

Table 3.2 summarizes the characteristics of the coarse-grained dynamically reconfigurable architectures described in this chapter. As shown in this table, a wide variety of architectures have been

Table 3.2: Summary features of surveyed Dynamically Reconfigurable Processors

Number	Device	Vendor	Ref.	*G*	*PE*	*R*	*CX*	*I*
1	CS2112	Chameleon	[61]	16/32	108	M	8	Tiled, 2D Bus
2	XPP-III	PACT	[36]	16	64	C	-	2D Bus, Direct
3	DRP-1	NEC/NECEL	[63]	8	512	M	16	Tiled, 2D Bus
4	DAPDNA-2	IPFlex	[64]	32	376	M	4	Tiled, 2D Bus
5	DAPDNA-IMX	IPFlex	[66]	16	955	M	4	Tiled, 2D Bus
6	FE-GA	Hitachi	[67]	16	32	M	4	2D-mesh Direct
7	D-Fabrix	Elixent	[68]	4	576	C	-	Chessboard
8	KC256	Rapport	[73]	8	256	C	-	Crossbar for Rows
9	ADRES	IMEC	[75]	16	64	M	32	2D-mesh Direct
10	S6 SCP Engine	Stretch	[76]	4/8	4096	C	-	-
11	Cluster machine	Fujitsu	[78]	16	15/Cluster	M	-	3-Stage Switch
12	MuCCRA-1	Keio University	[79, 39]	24	16	M	64	Island style
13	MuCCRA-2	Keio University	[79, 80]	16	16	M	16	Island style
14	MuCCRA-D	Keio University	[39]	24	16	M	64	Direct

G = Granularity [bits],
PE = PE-Array Size (number of PEs),
R = Reconfiguration Method,
CX = Number of Contexts,
M = Multicontext Scheme,
C = Configuration Delivery Scheme,
I = Inter-PE Connection Network

proposed and designed, but they seem to be designed for a particular purpose such as digital signal processing devices, multimedia or wireless communications. The main reason is that the dynamically reconfigurable processors are expected to provide high area- and power-efficiency for the target application even with decreasing flexibility or programmability compared to FPGAs. Consequently, I can find various kinds of architectures, and the straightforward architectural classification has become complicated [14].

Dynamically reconfigurable processors have a very wide design space including PE structures, channel bit widths, intra/inter-PE connection networks, the number of available multipliers and distributed memory modules, dynamic reconfiguration schemes, context memory capacity, PE array sizes, and so on. Thus, unlike conventional FPGAs [3, 81], it is difficult to explore a general-purpose and preferable architecture among the dynamically reconfigurable processors. Although they are just being in practical use, designers need to perform a time-consuming and inefficient architectural exploration for its target application at design time. The configurable processors, in which several optimizations or customizations including instruction set extensions and memory size scaling can be performed, are fully exploited for embedded microprocessor designs. Also for dynamically reconfigurable processors, a configurable architecture design methodology is needed in order to provide an efficient design space exploration to SoC designers.

The variety of architectures for DRPAs also leads to a challenge in developing computer-aided designs and compilation tools that can map an application to a target reconfigurable device. This involves determining which parts of the application should be mapped to the reconfigurable array and which should be mapped to the attached processor, determining when and how often the reconfigurable device should be reconfigured, which changes the functional units implemented in hardware,

as well as the specification of algorithms for efficient mappings to the reconfigurable system. To date, a standard interface to DRPAs has not yet been solved, and each company still has its own set of tools tailored to its device.

Chapter 4

Hardware Task Mapping

4.1 Problem

To date, a large number of researches in the area of reconfigurable computing have resulted in a number of academic and commercial DRPAs. These devices play an important role in balancing high performance demands and low power consumptions, especially in embedded devices. One of the trends in developing reconfigurable devices is the dynamic reconfigurability based on a multicontext mechanism such as DRP, DAPDNA-2, FE-GA and ADRES introduced in Chapter 3 in order to minimize the reconfiguration overhead and greatly improve the performance of reconfigurable systems. The datapath mapped to a piece of physical hardware is called a *context*. A target application is divided into a set of different contexts, and a multicontext DRPA executes them by changing contexts with each clock cycle. Basically, with such multicontext DRPAs, an application is designed and mapped into hardware as a single thread of control. At any time, only one required context is activated and executed. In order to increase the throughput, some techniques such as software pipelining and loop unrolling can be applied to exploit more parallelism.

For a particular hardware architecture, the performance improvement depends on the inherent parallelism of a target application. According to Hasegawa [82], the optimal size of the PE array in a DRPA is fixed related to the target application, and PEs exceeding the optimal number are not efficiently used. In many cases, the parallelism of an application is smaller than the number of PE array in a context. Accordingly, a large number of PEs is not used efficiently.

One of the methods to improve the performance in such a condition is making the best use of stream-level pipelined execution. Most target applications of such devices are stream processing, that is, data blocks to be processed are iteratively received in a certain interval. By dividing an application into small independent sequential tasks, and executing in a pipelined manner, a large number of PEs can be used efficiently.

Beside the popular *single-process* execution, where an application is designed and implemented as the only one thread of control for being assigned and executed on a reconfigurable array, some architectures like DRP-1 [63] by NEC Electronics support the "multi-process execution", which

allows multiple threads of control to run concurrently. An application is divided into several *tasks*; a large reconfigurable array is partitioned into some small arrays, each of which is called Tile; and each *task* is assigned to different Tiles of DRPAs, and executed in parallel. An inter-task communication mechanism using internal memories is defined for exchanging data between tasks. Each Tile, to which a task is assigned, is independently controlled for executing and exchanging configuration data.

Although the introduction of the *multi-process* execution may lead to an effective way of partitioning applications in order to improve throughput and energy efficiency, systematic method for efficiently mapping tasks into hardware execution units has not been well researched. This chapter proposes a systematic mapping method to map target *tasks* into Tiles, and shows examples applying it for DRP-1.

4.2 Related Work

There are a lot of research efforts aimed at models of computation such as Synchronous Dataflow (SDF) [83], Dataflow Process Networks [84], and Kahn Process Networks [85]. In the Ptolemy project [86], several models have been combined together to create a structure for the Ptolemy framework. However, this framework mainly focuses on application modeling and simulation without supporting the real mapping of application models onto models of architectures.

Partitioning applications is a well-known technique that has been thoroughly researched. Hardware/Software partitioning specifies which parts of an application should be mapped to hardware or software components [87]. Partitioning applications between reconfigurable hardware blocks of different granularity tries to map some parts of an application into fine-grain reconfigurable units and others into coarse-grain reconfigurable arrays for exploiting the advantages of various granularity reconfigurable modules [88]. In a multiprocessor environment, partitioning applications into multiple tasks and threads, each of which could be mapped to an execution unit for running, is a well researched topic.

There are also a number of researches for job mapping and scheduling into the partitioned area of FPGAs with partial reconfigurable capabilities. In these studies, the area and execution time requiring for a job are fixed, and scheduling algorithms decide the order and place where multiple jobs should be mapped. Since a three-dimensional (x, y, and temporal) placement problem must be solved for efficient placement, a number of theoretical researches have been done [89] [90] [91].

However, my target problem has the following differences, which prevent previous researches from being directly applied.

- In the *multi-process* execution treated here, each task being a part of a single job works in a pipelined manner, and handles a large number of streaming data arriving continuously. Thus, the throughput of a single job with multiple tasks is a target for optimization.

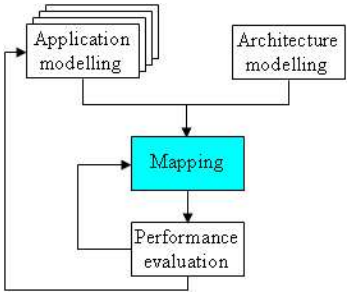


Fig. 4.1: General design flow

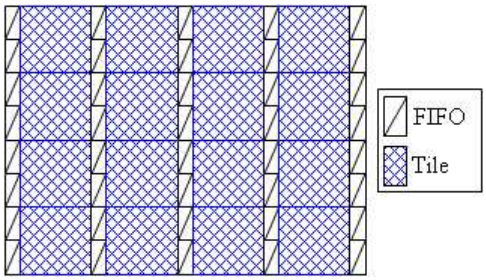


Fig. 4.2: Target DRPA

- By a multicontext execution, a Tile can execute multiple *tasks* sequentially as far as the number of contexts is sufficient.
- Multiple Tiles can be assigned to a *single task* for enhancing the execution speed by using parallel execution with a large number of PEs.

4.3 Target Architecture and Application Model

Fig. 4.1 shows a general design flow for a target DRPA. By explicitly specifying a mapping step, multiple target applications could be mapped onto a candidate architecture for evaluation. Typically, a set of applications is proposed for a certain architecture, then, they are mapped onto such an architecture. Through the performance analysis of each application-architecture-mapping combination, some improvements could be made for application implementation or mapping algorithm.

This paper will focus on a method for efficiently mapping an application modeled as a Kahn Process Network onto shown below target reconfigurable architectures.

4.3.1 Target architecture

In this paper, a two-dimensional tile-based DRPA consisting of identical $M \times N$ hardware execution units is assumed to be the target architecture. This type of target architectures can be considered as a homogeneous structure, which allows more flexible in mapping.

As an example, a target DRPA (4×4) shown in Fig. 4.2 consists of a number of PE array units, each of which is called a Tile. A Tile has a certain size, distributed shared memory modules and its own controller for dynamic reconfiguration. The total number of Tiles in the target DRPA is referred as N_{all}. Here, a multicontext DRPA is assumed, that is, each Tile has a certain size of context memory and changes its hardware context according to a context pointer. The number of contexts that can be stored in the context memory (C_{max}) is limited in a certain number. Some examples of a candidate DRPA with correspondent parameters can be seen in Table 4.1, where PEs and $Tiles$ show the total number of PEs and Tiles in each DRPA respectively; $PEs/Tile$ denotes the number of PEs per a Tile; and $Contexts$ is the maximum number of contexts that can be stored in a PE.

Table 4.1: Example of possible target architectures

Parameter	DAPDNA-2 [92]	ADRES [75]	DRP-1 [63]
PEs	376	64	512
Tiles	6	4	8
PEs/Tile	56	16	64
Contexts	4	32	16

Multiple neighboring Tiles can be joined to form a Tile group for executing a single task together. In this case, joined Tiles work synchronized with a single controller, and the number of available PEs becomes the sum of Tiles in the Tile group. Neighboring Tile groups can be communicated with each other through FIFOs formed with the distributed memory modules. That is, streaming data can be transferred between Tiles with a simple handshake mechanism.

4.3.2 Target application model

I assume that a target application can be represented with a Kahn Process Network (KPN), which is similar to a model proposed for streaming processors [93]. In this model, a total job is represented with multiple processes tasks that can be executed in a pipelined manner. That is, data streams continuously arrive at a certain interval, and the results of a task are transferred to adjacent tasks. A KPN has following characteristics:

- The execution of a KPN is deterministic, or in other words, for a given input, the same output is always produced. This model is suitable for stream-based multimedia and signal processing applications since it allows to model such types of applications, and guarantees that no data is lost.
- According to the model, tasks are monotonic, that is, they require only a partial information from input for generating a partial result to output. Therefore, the model allows parallelism and pipelined processing, which are suitable to be implemented on reconfigurable arrays.
- The model makes it easier to partition an application into a set of parallel communicating tasks.

For example, the upper part of Fig. 4.3 shows the KPN graph of a JPEG encoder. In this case, the graph becomes a linear structure. Media processing programs could be easily translated into KPNs [93], and here, I assume that the KPN corresponding to a target application has been already formed.

4.3.3 Goal of mapping

Each task of a target KPN (p_i) can be mapped into a Tile Group TG_j of the target DRPA, and executed in a pipelined manner. The lower part of Fig. 4.3 shows an example of mapping for a JPEG

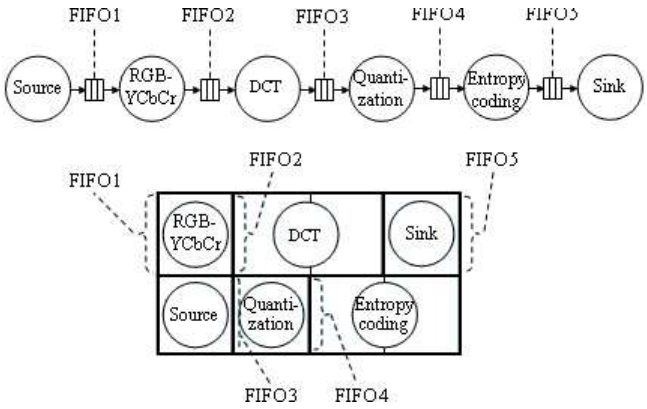


Fig. 4.3: Task mapping for JPEG encoder

encoder, in which, tasks mapped into each Tile Group get data stream from the input FIFO, execute their own computation, and produce results to the output FIFO. If there is no data in the input FIFO or the output FIFO is full, the task execution is stalled. The data stream is assumed to arrive in a certain interval corresponding to the total throughput of the DRPA.

In order to improve the total throughput and the execution time of an application, it is critical to balance computation stages in the interrelation with other tasks in a pipelined chain. In a pipelined processing model, the total throughput is bottlenecked with the most time consuming task. Here, by increasing the size of TG_js, the throughput can be enhanced by parallel processing with more number of processing elements. If, for example, the task *DCT* is the bottleneck of the JPEG encoder in Fig. 4.3, the total throughput can be improved by mapping *DCT* into a TG with a large size because the number of execution clock cycles of the bottleneck task (DCT) could be reduced.

The goal of mapping is to find the best combination of tasks and Tile Groups in order to improve the throughput and to reduce the execution time of each pipeline stage while preserving system limitations: (1) the total number of Tiles used in TG_j must be smaller than or equal to the number of Tiles supported in the target DRPA, and (2) the sum of contexts required for tasks mapped in a TG must be smaller than or equal to the number of contexts supported in the target DRPA.

4.4 Mapping Algorithm

4.4.1 Target task graphs

In this research, target task graphs are limited in a simple unidirectional linear graph with a fork-join structure. As shown in Fig. 4.4, a task can send a data stream to multiple tasks (fork) and the results are gathered in the next task (join). Each task is connected with a FIFO, and can work independently. Stream data arrive in a certain interval to the starting task, and the total tasks can be executed in a pipelined manner. Although complicated graphs cannot be represented because of the above limitations, most graphs of streaming processing are rather simple and fall into this limitation.

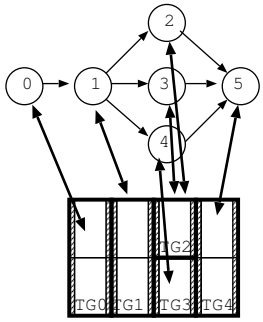


Fig. 4.4: Target task graph

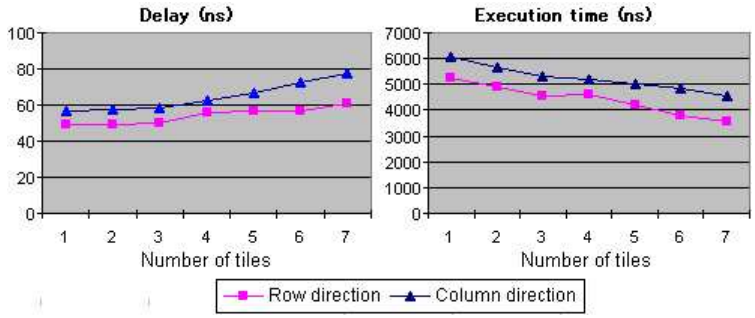


Fig. 4.5: Delay and execution time vs. number of Tiles

Here, task number p_i is assigned into each task from the starting task to the terminal one. Parallel execution tasks (tasks 2, 3, and 4 in Fig. 4.4) can be assigned in any order.

4.4.2 Target architecture and task mapping

With the target architecture introduced in Section 4.3.1, neighboring Tiles can form a Tile Group TG_j, and neighboring TGs can communicate via FIFOs provided on the edge of a Tile. The number of Tiles in a TG_j is denoted with $Size(TG_j)$.

When a task p_i is mapped on a TG_j, different assignment strategies could affect the quality of an implementation. Namely, mapping depends on three following factors: 1) The number of Tiles allocated to each task. 2) Absolute position of Tiles. 3) And, absolute position of FIFOs for exchanging data between tasks. Of which, the most important factor is the number of Tiles assigned to a task. In general, the execution time represented with $FT(p_i, Size(TG_j))$ is reduced when the task is executed with TG formed by a large number of Tiles. However, the relationship between the execution time and the number of Tiles is not simple. For example, Fig. 4.5 shows the relationship between the number of execution clock cycles and the delay with various numbers of Tiles when two main computation steps of two dimensional Discrete Cosine Transform (DCT), "Row direction" and "Column direction", are separately implemented on NEC Electronics DRP-1. While the number of execution clock cycles is decreased with a large number of Tiles, the delay tends to be stretched. So, the total execution time is reduced but not relational to the size of TGs. Moreover, if a task does not have enough degree of parallelism, the execution time is not improved at all by using Tiles larger than a certain size.

Fortunately, by using design tools like Musketeer for DRP-1 [59], the relationship between the size of TGs and the execution time can be evaluated in advance. Here, I assume that the execution time with the various size of TG has been evaluated and the result represented with $FT(p_i, Size(TG_j))$ can be obtained by table reference. Exactly speaking, $FT(p_i, Size(TG_j))$ depends not only on the size but also on the connected shape of Tiles. For example, when four Tiles are connected, the execution time slightly differs depending on the patterns shown in Fig. 4.6. However, implementation experience on DRP-1 shows that the difference is small when the number of

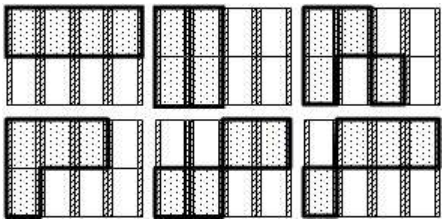


Fig. 4.6: Tile connection patterns for a TG

connected Tiles are less than eight. So, I will ignore the difference here.

When multiple tasks $p_i \ldots p_{i+k}$ are assigned into a TG_j, they are executed sequentially, (for example, task 2 and 3 mapped in TG2 in Fig. 4.4), that is, the total execution time T_{ik} becomes $T_{ik} = \sum_i^{i+k} FT(p_i, Size(TG_j))$ $(k > 0)$

Also, the number of contexts required for executing a task is depending on the number of Tiles in TG_j, and represented with $FC(p_i, Size(TG_j))$. It can be also analyzed before scheduling by the design tools. When multiple tasks are assigned into a TG_j, total required contexts becomes $C_{ik} = \sum_i^{i+k} FC(p_i, Size(TG_j))$

Here, a task fork can be included in the graph (e.g. tasks 2, 3 and 4 in Fig. 4.4). Such forked tasks can be mapped into the same TG even if they have non-neighboring numbers (e.g. tasks 2 and 4 in Fig. 4.4). However, if the graph does not have any fork, mapping tasks with non-neighboring numbers will increase the communication between two tasks, and this often requires the communication between non-neighboring Tiles. So, in order to make the algorithm simple, I only map the neighboring tasks into a TG.

4.4.3 Mapping algorithm

The proposed algorithm consists of three steps: task grouping, adjusting and topological mapping.

4.4.3.1 Task grouping

Task grouping clusters the tasks of an implementation in the specification model to n groups in a way that the computation amount in each group is balanced. Neighboring numbered tasks are mapped into a Tile as possible, keeping the limitation represented with $\sum_i^{i+k} FC(p_i, 1) < C_{max}$ for each Tile. Here, the number of mapped Tiles is called N_{used} and so the number of unused Tiles is $N_{all} - N_{used}$. When there are multiple candidates for mapping, the one with less N_{used} is selected. If $N_{used} > N_{all}$, the total job is too large to implement on the target DRPA. In this case, some of light-weight tasks must be moved to the software executed on the embedded CPU with the DRPA. In the initial mapping, every mapped Tile Group TG_j has only a Tile.

4.4.3.2 Adjusting

This step aims at adjusting the size of Tile groups assigned in the previous step toward reducing the execution time of those taking a large time. First of all, the execution time of each TG_j, $\sum_i^{i+k} FT(p_i, Size(TG_j))$, is evaluated in order to find TG_js whose execution time is the largest. To improve the execution time of such TG_j, unused n Tiles are assigned. Here, the limitation of the target DRPAs is considered. If the size of TG must be 1 or even numbers (1, 2, 4, 6 and 8), the assigned number of Tiles must be the smallest one keeping the limitation.

There are three possibilities:

- All tasks in TG_j are executed sequentially with a larger size of Tile $Size(TG_j)+n$. In this case, the execution time is $\sum_i^{i+k} FT(p_i, Size(TG_j)+n)$ if the execution time is smaller than those of the other two possibilities, add n Tiles to TG_j to increase the size.

- The first x tasks $p_i, \ldots p_{i+x-1}$ are executed with additional n Tiles and the other tasks are executed with TG_j. In this case, the execution time is $FT(p_i, n)+\cdots+FT(p_{i+x-1}, n)+FT(p_{i+x}+Size(TG_j))+\cdots+FT(p_{i+k}, Size(TG_j))$. If the execution time is smaller than those of the other two possibilities, generate a new Tile Group and move $p_i, \ldots p_{i+x-1}$ from TG_j.

- The last x tasks $p_{i+k-x}, \ldots p_{i+k}$ are executed with separated n Tiles and the other tasks are executed with TG_j. In this case, the execution time is $FT(p_i, Size(TG_j))+\cdots+FT(p_{i+k-x}, Size(TG_j))+FT(p_{i+k-x+1}, n)+\cdots+FT(p_{i+k}, n)$. If the execution time is smaller than those of the other two possibilities, generate new Tile Group and move $p_{i+k-x}, \ldots, p_{i+k}$ from TG_j.

After that, increase N_{used} and iterate the above steps until all Tiles are used; that is N_{used} is equal to N_{all}.

4.4.3.3 Topological mapping

With the previous described steps, tasks are mapped into TG_j, and the last step is to fit TG_j into the physical shape of an $M \times N$ structure. Two approaches can be applied in this step.

All possible mapping exploration (APME) Since the number of Tiles in a system is limited into small numbers (for example, eight in DRP as seen in Table 4.1), choosing the best topological mapping by this approach is possible in a reasonable amount of time by searching the complete solution space to retrieve all possible mapping variants.

In order to limit the search space, I decide the best allocation of Tiles for each list ($Size(TG_0)$, $Size(TG_1), \ldots, Size(TG_k)$), which consists of the sizes of TGs after adjusting where k denotes the number of used TGs. There are multiple possibilities of Tile assignment for each list. For example, the allocations in Fig. 4.7 are all corresponding to the list (2,1,1,2,2), since the arbitrary combination of Tiles can be allowed in DRP-1. However, allocating a TG into separating Tiles increases long wires

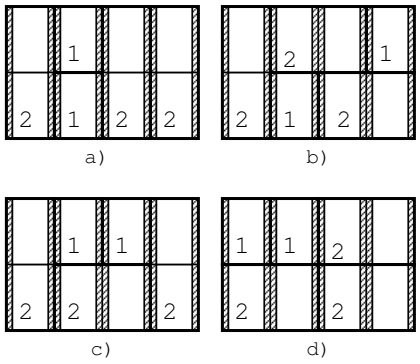


Fig. 4.7: Tile assignment

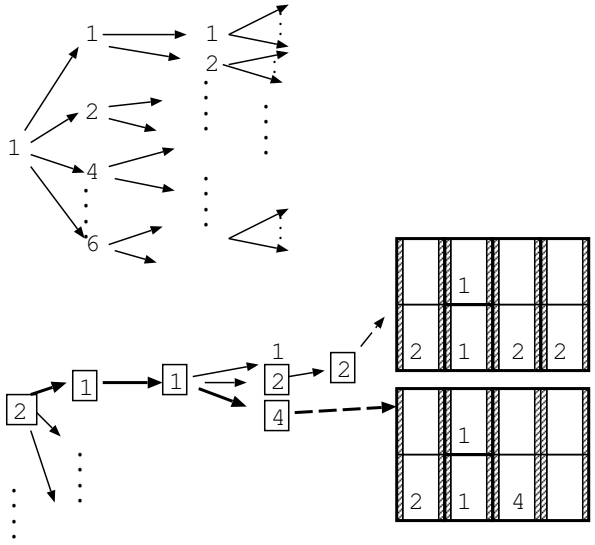


Fig. 4.8: Example of APME approach

which connect distant Tiles, and degrades the operational frequency. Moreover, the communication between TGs is done through the FIFOs allocating edges of a Tile, and so the neighboring TGs that need communication should be mapped into neighboring Tiles. Considering them, I selected only one allocation for all possible patters in each list beforehand. For example, Fig. 4.7 (a) was selected for the combination (2,1,1,2,2). It is called *prepared pattern* in this study.

Fig. 4.8 shows an example of the list (2,1,1,4) and another example of (2,1,1,2,2). In this method, for every branch of the list, a pattern is pre-assigned. Apparently, this method will cause the explosion of the possible patterns if the number of Tiles becomes large. This approach has been proved to be NP-complete, and it requires exponential time in order to find an optimal solution. However, for many up-to-date DPRAs with a realistic size, the number of patterns are reasonable.

Dynamic programming approach In the possible solution space, the same sequence of physical Tiles for a specific sequence of TG_j often appears as a part of many mapping solutions. Instead of searching the whole solution space, the dynamic programming technique can be utilized to find the optimal topological mapping for the complete sequence of TG_j by using solutions for smaller subsequences. Once an optimal solution for mapping up to TG_i is determined, the execution time for executing up to TG_{i+1} can be determined. This step is applied recursively to compute the final optimal mapping.

Given a sequence of Tile Groups ($Size(TG_0)$, $Size(TG_1)$, ..., $Size(TG_k)$), the minimum execution time for executing up to task i in a TG_j, E_{ij} can be computed using the following recursive expression.

$$E_{ij} = t_{ij} + min(E_{i-1})$$

In the expression, t_{ij} is the execution time of the task i in the physical TG_j, and $min(E_{i-1})$ is the total of the minimum execution time of other tasks up to task $(i-1)$. The expression shows that all possible ways of mapping task i are examined once the mapping of task $(i-1)$ has completed.

4.5 Target Device

4.5.1 Device

In this research, for evaluation, I used a real DRPA named DRP-1 from NEC Electronics. The architecture of DRP-1 has been described in Section 3.2.3. The *multi-process* execution in DRP-1 is almost the same as the model introduced in Section 4.3. A task is assigned manually to a basic execution unit called a Tile. Since DRP-1 contains eight Tiles (Fig. 3.15), the maximum number of *tasks* that can be concurrently executed is eight. The DRP architecture is flexible enough to allow a task to be allocated to a group of connected Tiles so as to provide a greater possibility for large tasks, which cannot be assigned to a single Tile, to be implemented. STCs inside Tiles can be operated and controlled independently to provide different instruction pointers for each task, so this allows tasks to run in parallel. A first-in first-out (FIFO) memory mechanism, which employs VMEMs between Tiles, is used as an inter-task communication method. A FIFO is for one-way communication and acts like a pipe. Writing to and reading from a FIFO are blocking, that is, a task needs to be stalled because of the data shortage.

4.5.2 Mapping applications onto DRP-1

After an application is partitioned and modeled as a KPN, it is mapped into the 8-Tile architecture of DRP-1 according to the mapping algorithm described in Section 4.4 with following constraints taking into account the specific features of DRP-1.

- The limit number of Tiles that can be allocated to an application no matter how many tasks it is partitioned into is eight.
- Separate tasks must be mapped into different Tiles, or in other words, the scheduling of two or more tasks sharing the same Tile is forbidden in this implementation. This is because NEC's DRP-1, which is used as the target device for evaluation, does not support different tasks to execute on the same Tile sequentially. In order to solve this problem using the current design tool, all possible combinations of merged tasks must be manually prepared in advance. It is not practical especially when the number of case studies becomes large. With devices providing a task controll mechanism (for example MuCCRA-1 or MuCCRA-2 [79]), this problem can be solved easily.
- A task can be mapped to a Tile Group formed in any shape (like the 4-Tile example in Fig. 4.6).

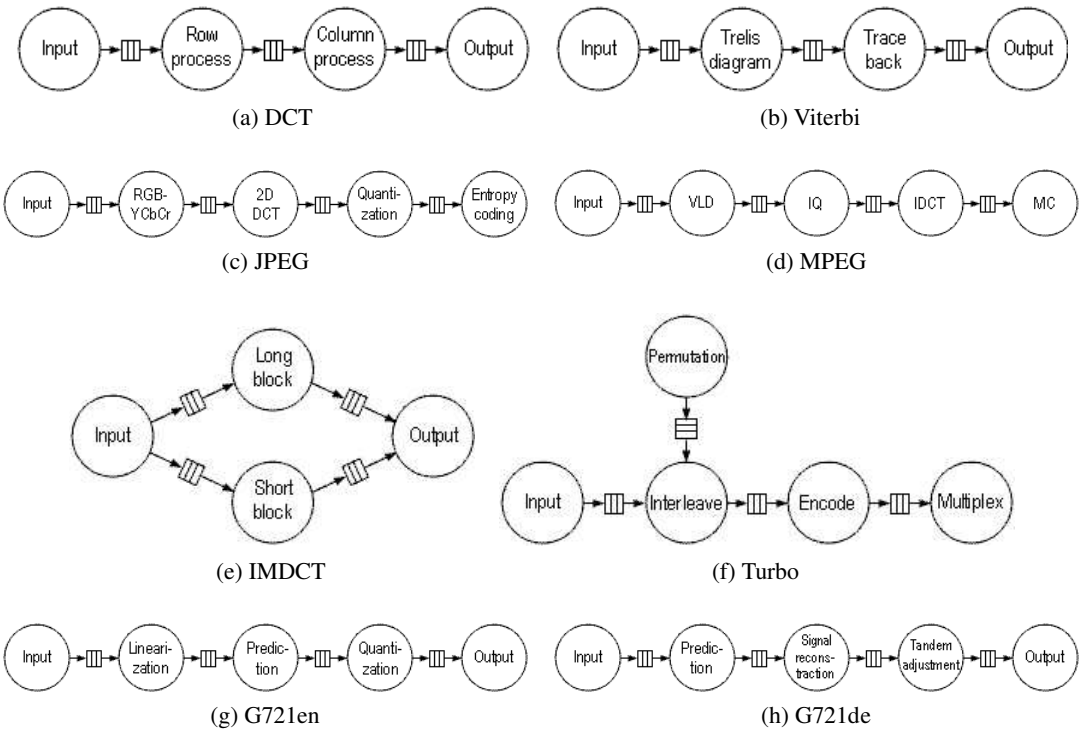


Fig. 4.9: PKN models of target applications

An application is mapped onto the DRP architecture in a way that each task is mapped onto a Tile or a Tile Group according to the mapping algorithm and each task's FIFO is mapped one-to-one onto an DRP's FIFO. Moreover, tasks that need to communicate are mapped to adjacent Tile Groups. An example mapping of the JPEG encoder is shown in Fig. 4.3.

4.6 Evaluation

4.6.1 Target applications

Target applications listed in Table 4.2 have been implemented on DRP-1. Column "#" denotes the number of tasks each application is partitioned into. Column "Variant" shows different mapping versions for an application. Of which, the first version with the name of the application without number added is the one generated by the proposed mapping algorithm. Column "Mapping" shows the number of Tiles each task in a KPN is mapped onto. For example, for the DCT, the column "Mapping" of the first line contains 1, 3, 3, 1. That means the DCT is modeled with four tasks that are mapped to groups of 1, 3, 3, and 1 Tile(s) respectively.

Target applications are partitioned into the tasks of KPNs shown in Fig. 4.9 as follows.

- DCT: input, row-direction computation, column-direction computation, and output
- IMDCT: input, long-block computation, short-block computation, and output

Table 4.2: Implementation results of target applications

Application (Abbr.)	#	Variant	Mapping	Delay [ns]	Clock cycle	Throughput [Mbps]	PEs/ context	Memories/ context
2D DCT [20] (DCT)	1	Single		67.3	95	160.2	98.1	11.0
	4	DCT	1, 3, 3, 1	46.8	112	341.9	141.3	49.3
		DCT1	1, 1, 5, 1	63.7	182	143.5	142.5	48.2
		DCT2	1, 1, 3, 1	46.9	184	194.9	115.0	35.7
		DCT3	1, 2, 4, 1	59.5	136	195.6	139.1	46.7
		DCT4	1, 2, 2, 1	45.1	166	258.0	111.8	45.1
		DCT5	1, 1, 1, 1	44.3	238	180.6	93.7	26.4
IMDCT [20] (IMDCT)	1	Single		140.6	1349	97.2	158.5	36.0
	4	IMDCT	1, 3, 3, 1	94.4	1452	182.3	185.9	39.0
		IMDCT1	1, 4, 2, 1	97.2	1421	178.6	188.0	40.1
		IMDCT2	1, 2, 2, 1	89.5	1592	169.8	178.3	39.0
Viterbi decoder [20] (Viterbi)	1	Single		30.4	11	3.0	52.7	0
	4	Viterbi	1, 2, 2, 1	25.9	21	4.3	73.0	10.8
		Viterbi1	1, 3, 3, 1	33.5	18	3.7	84.0	10.8
		Viterbi2	1, 4, 2, 1	37.2	18	3.8	87.7	10.8
		Viterbi3	1, 5, 1, 1	48.1	17	3.5	85.2	10.8
		Viterbi4	1, 1, 1, 1	25.2	27	2.6	42.1	10.8
JPEG encoder [94] (JPEG)	1	Single		56.3	636	42.9	136.3	19.8
	5	JPEG	1, 2, 2, 1, 2	53.1	894	106.3	196.4	65.2
		JPEG1	1, 1, 2, 1, 2	53.0	956	97.3	178.7	56.3
		JPEG2	1, 1, 3, 1, 2	58.6	882	100.8	214.5	65.2
Turbo encoder [95] (Turbo)	1	Single		81.3	16065	1.4	32.5	12.4
	5	Turbo	1, 1, 2, 2, 2	58.3	17642	4.0	48.8	39.5
		Turbo1	1, 1, 1, 3, 2	69.3	17498	3.9	54.2	39.5
		Turbo2	1, 1, 1, 2, 1	58.1	18312	3.4	36.9	32.4
		Turbo3	1, 1, 2, 2, 1	58.3	17936	3.9	31.1	39.5
MPEG-2 decoder [96] (MPEG)	1	Single		73.9	1564	3.9	144.9	16.3
	5	MPEG	1, 2, 2, 1, 2	62.9	1680	10.4	275.1	45.1
	5	MPEG1	1, 2, 1, 1, 3	69.7	1744	7.6	289.0	52.3
G721 encoder (G721en)	1	Single		91.2	618	16.6	98.9	21.1
	5	G721en	1, 1, 2, 2, 2	78.8	754	45.1	126.0	52.1
	5	G721en1	1, 1, 3, 1, 2	85.7	766	25.2	115.3	47.2
	5	G721en2	1, 1, 2, 3, 1	86.1	812	29.5	119.1	48.2
	5	G721en3	1, 2, 2, 1, 2	77.2	784	34.3	131.4	51.3
	5	G721en4	1, 2, 2, 2, 1	77.5	788	37.7	132.0	51.1
G721 decoder (G721de)	1	Single		94.7	669	4.1	74.6	13.5
	5	G721de	1, 2, 2, 2, 1	77.8	818	10.3	113.1	37.7
	5	G721de1	1, 3, 2, 1, 1	87.0	796	5.9	105.7	32.3
	5	G721de2	1, 3, 1, 2, 1	85.1	806	7.3	103.2	34.2
	5	G721de3	1, 2, 1, 2, 2	79.3	858	8.1	111.0	38.5
	5	G721de4	1, 2, 2, 1, 2	77.4	832	8.7	117.3	38.5

- Viterbi: input, trelis diagram, trace back, and output
- JPEG: input, RGB-YCbCr conversion, two-dimensional DCT, quantization, and entropy coding

- MPEG: input, variable length decode (VLD), inverse quantization (IQ), inverse discrete cosine transform (IDCT), and motion compensation (MC)
- Turbo: input, permutation, interleave, encode, and multiplex
- G721en: input, input data linearization, prediction, quantization, and output
- G721de: input, prediction, signal reconstruction, tandem adjustment, and output

4.6.2 Mapping versions

For comparison, the result of the *single-process* execution, where applications are not partitioned but the whole application is mapped into 8 Tiles of the target architecture as only one thread of control, is provided in Table 4.2 with the line "Single" of column "Variant". Also, in this execution, column "#" shows the value of 1 to clearly specify that applications are mapped as one task for executing.

For comparison, different mapping versions of applications shown in Table 4.2 with variety number of Tiles assigned to tasks are also evaluated. They are not the whole possible mapping space, but typical representatives. For demonstrating efficiency of the proposed algorithm with practical applications, there exist two problems: (1) there is no standard algorithm to be compared, and, (2) with simple algorithms (for example, random mapping), it is difficult to successfully complete the place-and-route phase, so it is impossible to get the designs to evaluate their quality. In order to address this problem, I stretched the exploration space by relaxing the following conditions.

- *Task grouping:* Tasks are grouped without taking into account balancing their computation amount. Other conditions are still kept to get the reasonable design.
- *Adjusting:* At this step, there are two options related to whether unused Tiles are exploited to improve the execution time of TG_j. In a version, groups TG_j are adjusted according to the proposed step in Section 4.4.3.2 with unused Tiles in order to reduce the execution time of target groups. In the other version, groups TG_j are not adjusted with unused Tiles.
- *Topological mapping:* this step is performed exactly the same as proposed in Section 4.4.3.3.

These steps were performed almost automatically, and I tried to get as many results as possible. If the mapping results can successfully pass the place-and-route phase, it is recorded and assigned a tag number consisting of an application name and a number. On the other hand, if the version fails at the place-and-route phase, it is not taken into consideration. For some applications, I could get a lot of mapping results, and Table 4.2 only shows results with good achievement either in the performance metrics: critical path, clock cycle and throughput. However, other applications like MPEG, only one version other than the proposed method could pass the place-and-routing phase.

4.6.3 Implementation results

Table 4.2 presents the performance result of target applications. In column "Variant", *Single* denotes the result of applications implemented in the *single-process* execution. Columns "PEs/context" and "Memories/context" show the average number of required PEs and used memories (both VMEMs and HMEMs) in a context.

In some applications like Viterbi, not all Tiles are assigned because the parallelism of these applications is not high enough. Usually, the execution time could be reduced by increasing the number of Tiles; but, for applications having low parallelism, only small number of PEs is required, so it is not expected to reduce execution clock cycles even with a larger number of Tiles. Moreover, since using larger numbers of Tiles with low usage will increase the leakage and the clock distribution power, an optimum number of Tiles should be carefully selected for each implementation [82]. In many cases, this causes the actual number of Tiles is not the largest one as shown in Table 4.2.

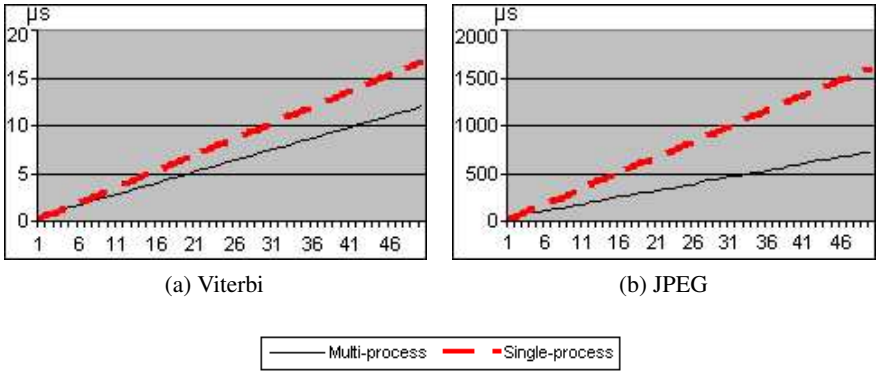
4.6.4 Throughput

The throughput of an implementation is represented with the amount of data processed in a second. Since all tasks are executed in a pipelined manner in the *multi-process* execution, the throughput of the total execution is limited by the stage with the largest execution time.

Table 4.2 shows that all implementations in the *multi-process* execution improve the throughput in a certain degree over the *single-process* execution up to nearly three time in JPEG implementation. In other words, by representing an application as a task network that are mapped into a DRPA as separate threads of control, the throughput could be improved. As the size of an input data block is the same, the throughput could be improved by either reducing the critical path or the number of execution clock cycles. In the *multi-process* execution, the critical path is usually shorter than that from the *single-process* execution. While a target application is mapped into the whole reconfigurable array in the *single-process* execution, in the *multi-process* execution, the critical path of an application is the longest one among child tasks, each of which is often mapped to one or several Tiles but not the whole reconfigurable array. For example, in DCT, the implementation in the *single-process* execution requires 8 Tiles of NEC's DRP, but in the *multi-process* execution, the largest task can only be mapped to 5 Tiles (version DCT1 in Table 4.2).

Moreover, the calculation of the throughput in the *multi-process* execution takes into account the largest number of execution clock cycles among tasks; and, by dividing an application into independent tasks, that number is often smaller than that of a big task executing in the *single-process* execution though the total number of clock cycles from all tasks is often larger due to task overhead. Since both the critical path and the number of execution clock cycles in the *multi-process* execution could be shorten, the throughput is likely to be improved.

The throughput could also be improved by taking advantages of the pipeline technique where multiple tasks are arranged to operate in a pipelined manner. In most cases, the output of a com-



(a) Viterbi (b) JPEG

Fig. 4.10: Execution time vs. Number of data blocks

putation step will be the input of the next step with no data or control hazard, so this is suitable for pipelining.

Among implementations in the *multi-process* execution mode, the one with my mapping algorithm achieves the best throughput. According to Hasegawa [82], each implementation has an optimum context size (the number of Tiles) where the performance becomes optimum. Other context sizes no matter whether they are smaller or larger than the optimum one result in performance degradation. The implementation according to the mapping algorithm is likely the one where constituent tasks are mapped with the optimum context sizes; therefore, the optimum throughput could be achieved. More importantly, in a pipelined environment, the throughput is greatly influenced by the balance of computation stages. The proposed mapping algorithm produces the most appropriate result in terms of task workload balancing. For example, balancing two tasks, row-direction computation and column-direction computation, in DCT is the most important factor for the throughput since they occupy the largest part of the total execution time.

Although the proposed mapping algorithm could improve the throughput in a certain degree, the main limitation is the number of available contexts. Since the number of required contexts becomes easily more than 16, the possibility of grouping tasks are strictly limited. This is the reason why the execution time of each task is still unbalanced.

4.6.5 Execution time

The execution time of an implementation for a given set of data can be computed as the product of the critical path and the number of execution clock cycles.

In the *single-process* execution, when multiple blocks of data are given, the number of execution clock cycles will be the multiple of that with one block of data and the number of data blocks. On the other hand, in the *multi-process* execution, because of the effect of pipelined processing, when more data are fetched, the number of execution clock cycles keeps increasing but less than the correspondence in the single-process execution. This is illustrated in Fig. 4.10, which shows the total

execution time for Viterbi and JPEG when processing N data blocks ($N = 1, 2..., n$). In the graph, the largest number of data blocks n equals to 50. Other target applications show similar behavior. When a number of data block is small, the execution time in the single-process execution is smaller because of the overhead in the multi-process execution. However, when there are enough input data, or in other words, when a number of data block provided is larger than a certain number, to keep the pipeline stages full, the execution time in the multi-process execution becomes smaller.

From Table 4.2, among implementations in the *multi-process* execution mode, the one with the proposed mapping algorithm achieves the smallest execution time since it has either the shortest critical path or the smallest execution clock cycle.

4.6.6 Area utilization

In order to investigate the mapping quality, I evaluate the area utilization in terms of the number of active PEs and on-chip memories. The overall area utilization could be presented by the number of consumed resources in every context. Since the number of contexts varies among applications, the average resource per a context will be used for examining the area utilization.

In Table 4.2, two columns *"PEs/context"* and *"Memories/context"* show the average number of active PEs and used memories in a context. They are calculated by dividing the total number of PEs and memories (both VMEMs and HMEMs) respectively for an implementation by the total number of contexts. From these two columns, it can be seen that, implementations in the *multi-process* execution demand more resources than correspondent implementations in the *single-process* execution. In terms of used PEs, the number of active PEs in the *multi-process* execution is often larger than that in the *single-process* execution. This is the result of having more than one task running at the same time.

Similarly, the requirement for active memories in the *multi-process* execution is also higher. One of the reasons is the use of FIFOs for inter-task communication. Even when requiring no memory for storing, an implementation still uses memories for communicating via FIFOs. This is illustrated in the implementation of a Viterbi decoder; while no memory for storing is demanded in the *single-process* execution, the *multi-process* execution still uses memories for FIFOs.

4.6.7 Two methods for topological mapping

As mentioned in Section 4.4.3.3, there are two methods for topological mapping in the proposed algorithm: all possible mapping exploration and dynamic programming approach. The difference between these two methods is the time they take to produce the final result. In order to find the difference, a program for topological mapping according to both methods is implemented, then the execution time is computed. The programs are compiled by gcc 3.3.6 with -O3 optimization option and executed on a Pentium 4 processor 3.2GHz with 512Mb internal RAM.

Table 4.3 shows the time needed for mapping each target application onto the target application

Table 4.3: Time for topological mapping

Application	All possible mapping exploration	Dynamic programming
DCT	65.4	2.3
IMDCT	65.3	2.3
Viterbi	65.3	2.3
JPEG	384.2	2.8
Turbo	384.2	2.8
MPEG	384.2	2.8

DRP-1. Columns "All" and "Dynamic" represent the methods of all possible mapping exploration and dynamic programming respectively.

Table 4.3 shows that the dynamic programming method greatly reduces time for topological mapping. Since target applications are not modeled with too many tasks, and the target architecture only has eight Tiles, the topological mapping step does not take too much time to produce the final result. Nonetheless, when the number of tasks an application is modeled increases, or when the target architecture has more number of Tiles, time for topological mapping will become a great concern. For example, taken time for applications modeled with four tasks (DCT, IMDCT and Viterbi) and applications modeled with five tasks (JPEG, Turbo and MPEG) on the same target architecture is considerably different when the method of all possible mapping exploration is applied.

Table 4.3 also shows that time for topological mapping depends on the number of tasks each application is divided into since applications with the same number of tasks take almost the same time. Moreover, the number of Tiles on the target DRPA influences time for mapping as well.

4.7 Conclusion

A systematic method for mapping an application modeled as a KPN onto a dynamically reconfigurable processing array is proposed. Using real applications and a real target architecture DRP-1, the impact of the proposed method on performance and area utilization is evaluated and analyzed. Evaluation results show that the throughput of the *multi-process* execution increases from two to three times compared with the *single-process* execution, while more area utilization is realized as a result of tasks being executed in parallel. In addition, my proposed mapping method results in the best throughput and execution time.

Chapter 5

Hardware Task Preemption

5.1 Problem

In order to further exploit the flexibility of reconfigurable devices, operating systems for managing task allocation, scheduling and configuration have been introduced. One of the focus areas of many studies is to build a multitasking environment to allow different tasks to efficiently share a piece of reconfigurable hardware. Among problems for realizing such an environment, a task preemption mechanism, where a higher priority task is allowed to interrupt an executing task, plays an important role. However, such a mechanism for tasks running on hardware circuits is not trivial involving the question of how to suspend and resume a hardware execution, and especially, how to capture the state data of a given task within a certain latency while guaranteeing a reasonable hardware overhead.

A considerable number of coarse-grained DRPAs such as DRP, DAPDNA-2, FE-GA and SAKE [14], which exploit a multicontext architecture to reduce configuration overhead, have been developed and commercialized. By providing storage for multiple configurations in each processing element, hardware configuration can be changed quickly often in a clock cycle. Compared with fine-grained FPGAs, since data and time for setting configuration from outside are small, task switching involving preemption in DRPAs seems to be more realistic than those for FPGAs. Nonetheless, while scheduling without preemption has been researched [97], and several methods to approach the problem of task preemption on FPGA-based devices have been proposed, only a few research efforts to implement such functions into DRPAs have been carried out.

In this chapter, I propose a method for preempting a hardware task and capturing its state data based on the analysis of resource usage at the design time. By modifying the state transition graph of applications implemented on dynamically reconfigurable systems at computation steps where the requirement of resources is small, the impact of preemption on performance and cost for task switching could be reduced.

5.2 Related Work and Research Contribution

5.2.1 Related work

Hardware multitasking on FPGAs have been a challenging subject for many researches. [29] deals with the support of concurrent applications in a multi-FPGA system by reconfiguring entire FPGAs. Storing the state information of a hardware task during interruption is discussed in [31]. The outline of a multitasking environment, which enables several tasks to run in parallel, is introduced in [98]. A hardware check-pointing approach for reconfigurable devices is proposed in [99] to divide a task into smaller modules to minimize the lost of computation when failures occur. In order to do so, the state of a task must be saved frequently at pre-defined checkpoints during execution.

One of the most important features to enable a preemptive multitasking environment is task preemption. In a such environment on reconfigurable devices, several solutions for saving and restoring the state data of a hardware task have been proposed. However, they mainly target fine-grained FPGAs.

- *Readback:* This solution is based on the configuration read-back capability, which allows to read the content of both registers and internal memory modules [31] [30] [100] [101]. Although demanding no extra hardware, the solution is slow due to a great amount of configuration data, and requires additional computation to filter out useless information. More importantly, the format of the configuration stream is crucial to extract useful data; so, this makes the solution depend on specific FPGAs.

- *Internal state supervision:* By adding extra interfaces to registers and internal memory modules, it is possible to access these elements when saving and restoring context data. This solution could be implemented as a scan chain, a memory-mapping structure, or a scan chain with shadow registers [101] [99] [102] [103]. Although achieving data efficiency as only needed information is saved, this approach demands extra resources and design efforts to implement interfaces to registers and memories. Additionally, hardware circuits should be modified, which is almost impossible for already available devices.

[104] proposes a systematic methodology for incorporating preemption constraints in application specific multi-task VLSI systems. By considering a predetermined set of applications, the method tries to insert preemption points taking into consideration both dedicated and shared registers in order to minimize the task switching overhead. This approach is suitable for fixed hardware platform like ASICs in which registers to be saved can be determined in advance.

5.2.2 Research contribution

The method proposed in this research differs from above-mentioned solutions in the following aspects.

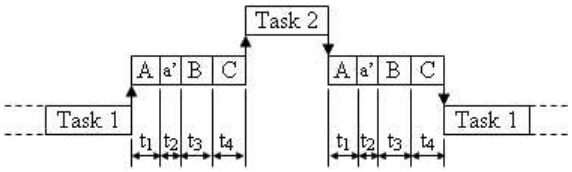


Fig. 5.1: Task switching

- By modifying the source program at high level languages (C and Verilog) during the design time to insert special states for saving and restoring state data, the method is independent from the details of the underlying hardware architecture such as which registers and memories are assigned to variables at the run time.
- The proposal targets coarse-grained DRPA devices, whose system design flow is tightly and automatically integrated with the necessary steps of the method in order for designers to examine and justify how the method affects implemented applications.
- The impact of the method on performance, hardware overhead and preemption latency can be measured at the design time.

The main contribution of this research includes:

- Algorithms to select and optimize the preemption points of a target application subject to given preemption latency constraints.
- Important steps to integrate the proposed method into the system design flow in order to automate the process of identifying and optimizing the list of preemption points for a target application during the design time.

5.3 Preemption Analysis

5.3.1 Task switching

In a preemptive multitasking environment, a typical task switching process can be illustrated on Fig. 5.1. While Task 1 is running, an interrupt signal, often caused by a system timer, indicating a possible task switch is generated. Before a new task (Task 2) can be executed, several preparing stages have to be done. First, an interrupt service is called to decide whether a task switch is necessary (Stage A). If it is, it might take some more times to wait because Task 1 may not be suspended immediately (Stage a'). Then, when Task 1 is ready to be stopped, the state data of Task 1 is saved in Stage B; and, that of Task 2, which had been preserved before, is loaded in Stage C.

(t_1) can be considered as interrupt latency; and, $\sum_{i=2}^{4} t_i$ is context switch latency. In many systems and depending on certain tasks, there is no Stage a', or in other words, tasks can be usually suspended immediately when receiving a preemption request. In such cases, time according to Stage a' (t_2) is

zero. On the other hand, some systems or certain tasks may not be interrupted right after receiving a request, for example, when the current running task is in a critical area, or when tasks only allow to be interrupted at certain points during execution. In this case, t_2 has a certain value. Generally speaking, the sum $\tau = \sum_{i=1}^{4} t_i$ can be considered as preemption latency. Usually, Stage A does not take a long time for modern processors and operating systems. In this study, I do not take Stage A or interrupt latency (t_1) into consideration, so preemption latency can be computed as follows:

$$\tau = \sum_{i=2}^{4} t_i \tag{5.1}$$

However, Stages B and C may take a considerably amount of time since the information representing the context of of a hardware task is specific for a given task implementation and may scatter on different state-holding elements. The amount of data captured when preempting a hardware task is often considerably large; so, the cost for a hardware task preemption mechanism should be minimized.

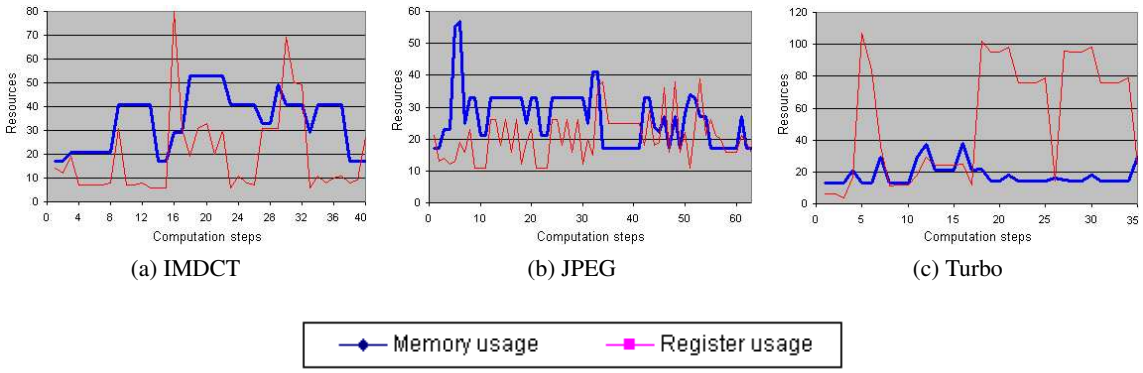
5.3.2 Approach

During execution, the amount of variables of a task mapped to internal registers and memories for storing intermediate results is considerably varying over time. Most target applications for dynamically reconfigurable processors are stream processing, that is, data blocks to be processed are iteratively received in a certain interval. Normally, between the processing of two data blocks, the amount of state data is relatively small. This fact can be applied to build a preemption mechanism that allows preemption only at predefined steps called *preemption points* [105] or *switchpoints* [106] during execution.

5.3.2.1 Preliminary evaluation

Fig. 5.2 shows requirement for resources in terms of memories and registers in each computation step when an IMDCT, a JPEG encoder and a Turbo encoder are implemented on NEC's DRP-1. X-axis of the figures shows the computation steps, and Y-axis shows the number of registers and memories. As shown in Fig. 5.2, the number of registers and memories for storing intermediate results and switching contexts greatly varies from step to step. For example, in IMDCT, steps 0, 1, 2, 3, 4, 6, 13, 19, 22, 29 and 34 do not require too many memories and registers; in addition, steps 10, 11, 23, 24 and 29 do not use a lot of memories though the number of registers is remarkable. Accordingly, by only allowing preemption at steps corresponding to steps where the requirement for memories and registers is minimized, the amount of data necessary to be saved can be dramatically reduced.

For a given task, in order to identify which data should be the target for capturing if the task is preempted at a certain computation step, it is critical to examine what resources are required and how they are used. The former can be done by source program analysis to identify variables and their life-time; and, the latter can be determined by an appropriate simulation. In a typical program,



(a) IMDCT (b) JPEG (c) Turbo

Fig. 5.2: Memory and Register usages vs. Computation steps

there are two types of variables: global and local, which specify the scope of variables. When a task is preempted, basically, all global variables should be saved; furthermore, local variables relating to the preempted place should also be captured. In addition, during execution, the use of variables is varying based on execution time and computation step.

5.3.2.2 Solution

In my approach, it is critical to evaluate the usage of variables in order to specify potential locations to be preemption points. Specifically, the proposed method is based on the state transition graph and the resource examination of a target application. This kind of resource evaluation can be done at the early stage of design by synthesis tools. The solution should be automatically done with a certain algorithm, since it must be combined into the design tool in the future. That is, the steps of the proposed method are automatically attached while an application is designed. The following policies are adopted.

- Steps where preemption is allowed are limited at predetermined points called *preemption points* where the demand of registers and memory modules is smaller than a certain limitation.
- At preemption points, special states tailored to current contexts are added for flushing and restoring state data.
- Performance degradation resulting from preemption is evaluated in order to optimize preemption points.
- Proposed algorithm is integrated into the system design flow of a target device.
- Algorithms for selecting and inserting preemption points are based on the design tools of a target architecture.

5.3.3 State transition graph

The homogeneous synchronous data flow [107] is used as the computational model for applications. The model iteratively processes semi-finite data blocks arriving in a certain interval. Stream applications, which are the main target of DRPAs, are suitable to be represented by this model. The behavior of a hardware task could be represented in the form of a State Transition Graph (STG) $G(N, E, START, END)$, where N is the set of nodes representing computation states; E is the set of edges showing the transition and data dependence between states; $START \in N$ is a distinguished start node without incoming edges; and $END \in N$ is a distinguished end node without outgoing edges. Fig. 5.5 shows the STG of two tasks, where (a_i, b_i) $(i = 0...n)$ shows numbered states, arrows represent possible transitions from states to states, (a_0, b_0) and (a_9, b_6) are start and end states. Transition can be switched conditionally as in state a_2.

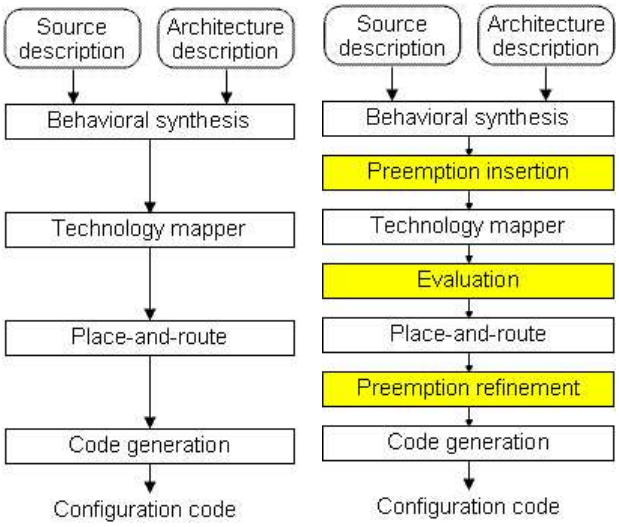
5.4 Preemption Algorithms

5.4.1 System design flow

Fig. 5.3(a) shows a typical design flow for DRPAs, in which the compiler or the behavioral synthesis, the technology mapper, the place-and-route and the code generation tool are assumed to be design tools available for the target DRPA. First, a source C-based program and an architecture description are taken as inputs for the behavioral synthesis, which extracts control flows as well as data flows, allocates operation resources, and produces reports about required resources. Then, the technology mapper actually produces the code in the form of hardware description language (HDL) for processing elements, and, a functional simulation at the register transfer level (RTL) can be executed. The place-and-route tool compiles the HDL code into a netlist. Exact reports about resource usage and critical path can be obtained at this step. Finally, the code generator produces configuration code for the underlying reconfigurable hardware.

Fig. 5.3(b) presents a modified design flow with the proposed preemption algorithm. Since most current DRPAs do not support dynamic memory allocation, the resource report produced by the compiler could describe quite exactly how variables are allocated and which resources are necessary for task switch. This is the basic for preliminarily analyzing preemption points and inserting preemption states at the step *preemption insertion*. The *evaluation* step is based on the RTL simulation for evaluating how added preemption states affect the implementation. The last step is the *preemption refinement* where preemption points are modified according to the exact report of resource usages and the evaluation result. A modified source program will be fed back to the technology mapper for re-compiling and re-evaluating.

According to the modified design flow, three additional steps are inserted to support the proposed preemption mechanism. Each step requires different input files and produces correspondent results as shown on Fig. 5.4.



(a) Origin design flow (b) Modified design flow

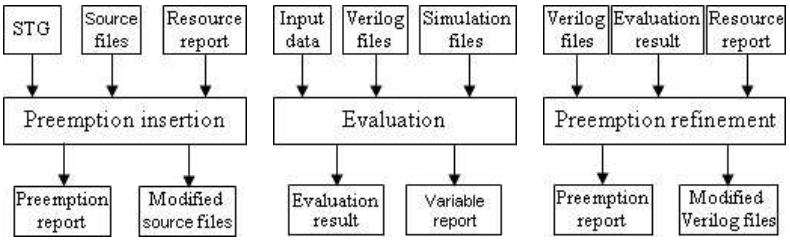
Fig. 5.3: System design flow

At first (Fig. 5.4(a)), the source files, the STG and the resource report of a target application are the basis for identifying possible preemption points. While the resource report is produced by the behavioral synthesis, the STG can be automatically generated by analyzing Verilog source files to identify computation states and their relationship. The generation of an STG could be integrated into the compiler/synthesis when the proposed method is applied in the future. The life time and scope of variables in source programs are analyzed to determine state data necessary to be saved at each computation state according to the STG. Computation states where resource demand is less than a certain threshold are marked as potential preemption points, then extra states for capturing and restoring state data are inserted into source programs at such preemption points.

Next (Fig. 5.4(b)), after the technology mapper step, produced Verilog files, simulation test bench and input data could be provided to an RTL simulator to evaluate how added states at preemption points affect the implementation. It is also possible to simulate and analyze how variables are initialized, used and discarded over each execution clock cycle. Accordingly, a variable report and an evaluation result report may be produced after the evaluation step.

Finally (Fig. 5.4(c)), based on the user-specified latency τ_{input}, the evaluation result and the place-and-route resource report, preemption points generated previously are modified in a way that the maximum preemption latency is satisfied. If the preemption latency cannot be achieved or hardware overhead is too large, another τ_{input} should be given. This process can be repeated until the list of generated preemption points can satisfy the following equation:

$$\tau_{max} \leq \tau_{input}$$



(a) Insertion (b) Evaluation (c) Refinement

Fig. 5.4: Input and output information for proposed method

where: τ_{max} denotes the maximum preemption latency, and τ_{input} specifies a constraint value of maximum preemption latency.

5.4.2 Preemption algorithm

The proposed algorithm achieves the target of minimizing context switch overhead by:

1. allowing preemption only at computation states where used resources are small, and
2. inserting special states for saving and restoring state data. Since including just input/output instructions, so these added states require a small number of resources, and no extra register files as well as memory modules.

The algorithm proposed here is consisting of three stages according to modified steps in Fig. 5.3: inserting preemption states into the original STG (Algorithm 1), evaluating, and refining (Algorithm 2).

5.4.2.1 Inserting preemption states (Algorithm 1)

Using the resource report generated by the behavioral synthesis as an estimation, the insertion algorithm (Algorithm 1) tries to find out potential preemption points.

Variable analysis (lines 2-8): It is necessary to analyze the variables of the target application from the source program in order to find all global and local variables (both register and memory variables) to all computation states. Global and static variables are often saved when the task is preempted unless they are not yet initialized and used. Local variables to a given computation state will only be saved if the associated state becomes a preemption point. In Algorithm 1, P holds a set of preemption points; PS is a set of special states where preemption is allowed, but state data do not need to be saved, as mentioned in the next step; S is a set of computation states, or in other words, it is a set of nodes in the STG of a target application. After variables are analyzed (line 5), each computation state is associated with correspondent variables (line 6), that means if state s_i becomes a preemption point, the correspondent set of variables $v_i(k_i)$ needs to be captured.

input : None
output: List of preemption points P

```
1  begin
2      P ← ∅;
3      PS ← ∅;
4      V ← ∅;
5      L ← ∅;
6      S ← {s_1, s_2, .., s_n};
7      V ← Variable_Analysis(S);
8      (s_1, s_2, ..., s_n) = (v_1(k_1), v_2(k_2), ..., v_n(k_n));
       /* Look for potential preemption points PS where no data is saved */
9      foreach s_i ∈ S do
10         if Initial_State(s_i) ! = 0 then
11             PS ← PS ∪ s_i;
12         end
13     end
       /* Detect computation loops and sort incrementally                 */
14     L ← Loop_Detection();
15     L ← Incremental_Sort(L);
       /* Look for preemption points inside and outside loops             */
16     foreach l_i ∈ L do
17         foreach s_i ∈ l_i do
18             if s_i ∉ PS then
19                 if Required_Resource(s_i) ≤ θ then
20                     P ← P ∪ s_i;
21                 end
22             end
23         end
24     end
25     foreach s_i ∈ (S \ L) do
26         if Required_Resource(s_i) ≤ θ then
27             P ← P ∪ s_i;
28         end
29     end
       /* Insert states for capturing and restoring state data at p_i ∈ P  */
30     P ← P ∪ PS;
31     foreach p_i ∈ P do
32         Insert a new state for capturing state data;
33         Insert a new state for restoring state data;
34         C ← ∅;
35         R ← {r_1, r_2, ..., r_p};
36         while R ≠ ∅ do
37             C ← {r_k, r_{k+1}, ...};
38             R ← R − C;
39         end
40     end
41 end
```

Algorithm 1: Insertion Algorithm

Potential preemption points analysis (lines 9-13): Certain computation states where variables are initialized may become potential preemption points since they can be easily re-executed without saving variables. These states can become preemption points with a special handle: instead of saving variables in states, these states will be simply re-executed when being restores. Such states are often found at the beginning of programs and at the end of loops. According to Algorithm 1, all states $(s_1, s_2, ..., s_n)$ are searched to find potential states, which are kept in PS. These states will become preemption points by being re-executed when the task is resumed.

Loop detection (line 14) At the beginning, Algorithm 1 detects all computation loops $L = l_1, l_2, ..., l_n$ using the given STG. Each loop l_i contains a number of states $l_i = s_{ij}, s_{ij+1}, ...$ The detection of computation loops is important since they are likely to take a considerable amount of time. Taking into account preemption latency, a loop without any preemption points inserted could violate a required preemption latency. For instance, a simple loop for resetting an array variable to a certain value is common in programs. Regardless of being unrolled manually or by the compiler, this kind of loop often contains only one or two states being executed repeatedly for a number of times. If no preemption is allowed in the loop, the given preemption latency might not be satisfied. Fortunately, instead of analyzing a complicated source program, it is more convenient to deal with the STG represented in the form of a flowgraph. Applying a loop detection algorithm [108] [109] on the STG of an implementation, all loops are identified and marked in order that the insertion algorithm will analyze and insert at least one preemption point among states constituting a loop.

Sorting (line 15): Using a suitable sort algorithm, loops l_i are sorted incrementally according to numbered states, i.e. $\forall s_u \in l_i \; and \; s_v \in l_k : \; s_u \leq s_v \; (i < k)$.

Preemption point finding (lines 16-29): Based on the resource report generated by the behavioral synthesis, loops are searched for possible preemption points where used resources are within a given threshold θ (lines 14-22). States that do not belong to any computation loops are also investigated to find out preemption points using the threshold θ (lines 23-27).

New states insertion (lines 30-40): At preemption points, new states are inserted for transferring necessary resources to an outside memory. Since the input/output interface of DRPAs often consists of a certain number of bits, resources are grouped into packets of those bits for output. Depending on the amount of resources and how memories are allocated, it would take a number of clock cycles for transferring. This contributes to reduce the preemption latency and affects the overall performance of the task.

5.4.2.2 Refining preemption points (Algorithm 2)

In order to prepare for refining the list of preemption points generated by 1, it is critical to quantitatively evaluate how preemption states inserted affect the implemented application. This can be done by executing simulations on the original and modified implementations of the application. Using design tools of most DRPAs, the RTL simulation can be performed in order to obtain the critical path, the execution clock cycles, and the used resources at the technology mapper level. With suitable computations, other parameters such as the operating frequency, the throughput and the preemption latency at a given state can be determined.

input : Input preemption latency ψ_{given}
output: None

1 **begin**
2 $S \leftarrow \{s_1, s_2, .., s_n\}$;
3 $k_1 \leftarrow 0$;
4 $M \leftarrow \emptyset$;
5 **for** $k_2 \leftarrow 1$ **to** P **do**
6 **while** $Preemption\_Latency(P_{k2}) > \tau_{input}$ **do**
7 $M = M \cup \{(s_{k_1} < s_i < s_{k_2})$ where required resource is minimized
8 **end**
9 $P \leftarrow P \cup M$;
10 $k_1 \leftarrow k_2$;
11 **end**
12 $B \leftarrow \emptyset$;
13 **for** $i \leftarrow 0$ **to** P **do**
14 **if** $Preemption\_Latency(P_i) \leq \tau_{input}$ **then**
15 $B \leftarrow B \cup P_k$;
16 **else**
17 **if** $n(B) \geq 2$ **then** $P \leftarrow P - B$;
18 $B \leftarrow \emptyset$;
19 **end**
20 **end**
21 **foreach** $p_i \in P$ **do**
22 $R \leftarrow \{r_1, r_2, ..., r_p\}$;
23 **while** $R \neq \emptyset$ **do**
24 $C \leftarrow \{r_k, r_{k+1}, ...\}$;
25 $R \leftarrow R - C$;
26 **end**
27 **end**
28 **end**

Algorithm 2: Refinement Algorithm

Preemption points generated by the previous step could be improved since Algorithm 1 often generates more than necessary. Moreover, the estimation of the critical path at the early stage of the design flow, which is basic for computing the preemption latency, is usually larger than the real one.

Therefore, based on the reports of the place-and-route phase and the evaluation results, redundant preemption points can be eliminated. Different requirements for preemption could become criteria for removing preemption points from the list generated by Algorithm 1. In the simplest case when the preemption latency can be tolerated, the refining algorithm just tries to eliminate preemption states consuming a larger number of clock cycles. In many cases, a user-specified constraint on preemption latency may be given. The following refining algorithm applies an input preemption latency τ_{input} as a criterion for optimize preemption points. In other words, τ_{input} is a constraint value for preemption latency. As a result, Algorithm 2 should generate such a list of preemption points that guarantees following condition.

$$\tau_{max} \leq \tau_{input}$$

where: τ_{max} denotes the maximum preemption latency or the maximum time to switch to a new task once a preemption is requested to the current running task. If such a list of preemption points, which guarantees above condition, could not be generated, the algorithm will report to designers and let them select another τ_{input}. This process can be repeated several times until a given τ_{input} is satisfied.

Preemption point scanning (lines 2-12): First of all, the algorithm scans the list of preemption points P generated at the previous stage to find out if the condition $\forall i : \tau_i \leq \tau_{input}$ $(i = 0, ..., n)$ is satisfied. If not, extra preemption points are inserted at states where required resources are small.

Unnecessary preemption point elimination (lines 13-21): The algorithm tries to remove unnecessary preemption points using the given preemption latency τ_{input}. Any preemption points in the list P between two preemption points t_1 and t_2 that satisfy $\forall p_i \in P : \tau_{pi} \leq \tau_{input}$, will be eliminated.

Preemption state modification (lines 20-26): Next, preemption states for saving and restoring context data are inserted or modified based on the accurate resource report generated by the place-and-route tool.

Management information: Necessary information should be defined for correctly managing data when saving and restoring, and for the outside operating system to control and schedule tasks. Therefore, while looking for preemption points and inserting preemption states, a data structure containing information such as the amount and the order of data saved and restored must be defined. By modifying the design at high levels (C-based and Verilog files), there is no need to know exactly very detailed information about the hardware architecture like in which PE a specific register variable is assigned to, or which memory modules hold a memory array variable.

5.4.3 Illustrative example

The example code in Fig. 5.6 is implemented on the NEC‘s DRP architecture to show how the proposed method works with the STGs of Task 1 and Task 2 (Fig. 5.5). The algorithm traverses

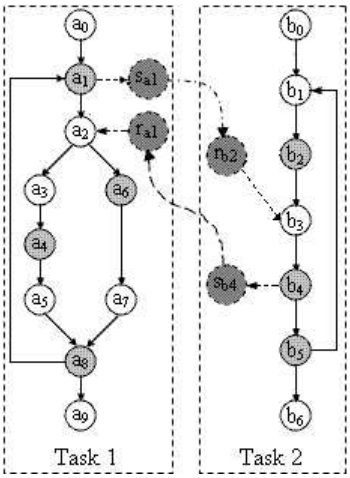


Fig. 5.5: Proposed solution

Saving states	Restoring states

reg(0:16) r[4];
mem(0:16) m0[8], m1[8];
mem(0:16) m2[8], m3[8];
…
dout = r[3]::r[2]::r[1]::r[0];
DataOut(dout);
m3[0]; m2[0]; m1[0]; m0[0];
$
dout = m3[0]::m2[0]::m1[0]::m0[0];
DataOut(dout);
m3[1]; m2[1]; m1[1]; m0[1];
$
…
	reg(0:16) r[4];
	mem(0:16) m0[8], m1[8];
	mem(0:16) m2[8], m3[8];
	…
	din = DataIn();
	r[3]::r[2]::r[1]::r[0] = din;
	
	$
	din = DataIn();
	m3[0]::m2[0]::m1[0]::m0[0] = din;
	
	$
	…

Fig. 5.6: Example code

STGs to find out the states where used resources are smaller than a certain limit; for example, states (a_1, a_4, a_6, a_8) of Task 1 and (b_2, b_4, b_5) of Task 2 with gray circles are such states. These states are marked as preemption points, and preemption states for saving (states s_{ai}, s_{bj}) and restoring (states r_{ai}, r_{bj}) data are inserted, and correspondent STGs are modified. These inserted states are assumed to be executed only when their coupled states are preempted. For example, when Task 1 is executing at state a_0, a preemption request occurs. Since a_0 is not a preemption point, Task 1 is not interrupted but continues to run until a_1, a nearest preemption point. Task 1 is stopped after a_1, and the execution is transferred to the correspondent state for saving the state information of a_1 (s_{a1}). After r_{b2}, which restores the state data of b_2, is executed, the execution is moved to Task 2, and Task 2 starts to run from b_3 (assume that Task 2 was preempted before at b_2). Switching from Task 2 back to Task 1 is handled in the similar way, for instance, when Task 2 is preempted at b_4.

The example code also illustrates how state data can be saved and restored by states s_k and r_k. The example is described in NEC's BDL [59], a C-based language. r, l_i $(i = 0,\ 1,\ 2,\ 3)$ are arrays of *register* and *memory* types with 16-bit width. Symbol :: shows a concatenation operator, which links variables together to form a larger bit width result. For example, the statement *r[3]::r[2]::r[1]::r[0]* concatenates four 16-bit elements of array r to create a 64-bit output. *DataOut* and *DataIn* are output and input functions working with 64-bit output/input interfaces via 64-bit variables *dout* and *din*. Symbol $ presents a timing descriptor to manually divide the codes into different states.

In the example, when all register and memory variables r, l_i $(i = 0,\ 1,\ 2,\ 3)$ need to be saved, it takes 17 clock cycles to transfer outside. However, the outside controller does not need to know about where saved data come from (registers or memories), and if saved values are 8, 16 or 32 bits. What the controller has to do is to allocate 64-bit buffers to hold data, to maintain the order of saving data, and to send in exactly the same amount in the type of 64-bit packets and order of data when data are restored.

The example shows that by modifying source programs at high level to insert preemption states,

it is possible to avoid the details of the underlying hardware architecture like the exact place of a register variable.

5.5 Target Device

Although the preemption algorithms proposed here can be extended to apply on other reconfigurable devices, in this research, I focus on DRP-1 from NEC Electronics as the target model. This device and the architecture of DRP-1 have been mentioned in Section 3.2.3. As mentioned before, applications designed for DRP platform can be described in the C-based hardware description language called BDL. Although BDL supports pointers, dynamically memory allocation is not allowed. All memory and register assignment are done at the compile time. Also, state registers allowing the DRP to transition from one state to another are determined at that time. The input/output interface of the DRP-1 is performed via two 64-bit separated channels. One input and one output operation can be executed concurrently in a clock cycle.

Currently, the DRP has no multitasking capability. At one time, only one application can be configured and executed on the whole 8-tile reconfigurable array. The basic operation model on the DRP is *time-division execution*, where an application is divided into multiple contexts, and one context at a time is activated and executed. The other operation model the DRP supports is *multi-process execution*, where an application is divided into several processes, each of which could be mapped into a group of Tiles and executed in parallel. Although the multi-process execution allows several threads of control (processes) to be present at the same time and run concurrently, it is not a true multitasking execution since processes belong to one application and no preemption is allowed.

5.6 Evaluation

5.6.1 Target applications

The proposed algorithm in this paper was evaluated on a number of real applications shown in Table 5.1. For each application, different cases are implemented and shown in column $Version$. Cases include implementations when no preemption points are inserted A_0; when preemption points are inserted without any constraint on preemption latency A_1 (only Algorithm 1 (Section 5.4.2.1) is applied); when the latency is specified $A_{\tau 1}$ and $A_{\tau 2}$ (both Algorithm 1 (Section 5.4.2.1) and Algorithm 2 (Section 5.4.2.2) with different input preemption latencies are applied); and when preemption is allowed at every state (A_n). (A represents the name of target applications). Columns 4 and 5 show the number of nodes and edges in the STG of each application. Columns 6-9 denote the input preemption latency (τ_{input}) (input parameter for Algorithm 2, Section 5.4.2.2), the critical path ($Delay$), the maximum preemption latency of each implementation (τ_{max}), and the total number of required PEs ($Used\ PEs$) respectively.

In Table 5.1, *Used PEs* shows the total number of required PEs in every state. τ_{input} specifies a constraint value of maximum preemption latency. If the proposed algorithm cannot generate a list of preemption points satisfying a certain τ_{input}, another value of τ should be given. It is repeated until the given τ_{input} is satisfied. τ_{max} specifies the longest response capability to switch to a new task from a currently running task. In the case of A_0, τ_{max} can be considered to be equal to execution time since when an application cannot be interrupted while running, switching to another application may only be possible when the application terminates.

Taking into account the delay from a moment a preemption request arriving to the moment the execution reaching the nearest preemption point, the calculation of preemption latency according to Fig. 5.1 and equation 5.1 becomes as follows:

$\tau = T_p + T_s + T_r$

where: T_p, T_s and T_r are time to reach the closest preemption point from the moment a preemption request is issued, time to save the state data of the preempted task, and time to restore the previously captured data of the preempting task respectively. T_p, T_s and T_r correspond to t_2, t_3 and t_4 in equation 5.1.

In a multitasking environment, the calculation of preemption latency depends on the combination of applications, and the combination of saving/restoring states of preempted and preempting applications. The former is difficult to determine and depends on specific scenarios; and, the latter causes preemption latency to vary even if there are only two applications executing in a system. In this paper, the calculation of preemption latency is performed at every state on the STG of a single application with the assumption that the same set of resources is applied for both saving and restoring. Though not being the exact situation in a real system, this gives us a relative overview on how preemption latency may vary.

All implementations do not pack multiple states into a single context (in the current DRP-1, maximum four states can be assigned to a context) in order to see the impact of inserted states on the performance. Therefore, the number of states is also the number of contexts after synthesizing. Although this prevents implementations with more than 16 contexts from executing on the real chip, it is still possible to complete the place-and-route phase, to execute simulations and to achieve suitable reports. The delay shown in Table 5.1 is the critical path of implementations. Since the option to pack multiple states into a context is not used, added states containing only input/output instructions for capturing and restoring state data do not have any influence on the critical path mainly formed by other main computation states.

5.6.2 Hardware overhead

Hardware overhead or context switch overhead H specifies the amount of additional resources required by added preemption states. If H_{PE} and H^*_{PE} denote the number of required PEs in the original and modified implementations respectively, H can be represented by: $H = H^*_{PE} - H_{PE}$.

Table 5.1: Target applications and evaluation results

Applications	Abbr.	Version	Number of nodes	Number of edges	τ_{input} [ns]	Delay [ns]	Max. latency τ_{max}[ns]	Used PEs
Discrete Cosine Transform [20]	DCT	DCT_0	28	32	-	67.3	6393.5	1105
		DCT_1	-	-	-	-	431.2	1197
		$DCT_{\tau 1}$	-	-	250	-	247.3	1274
		$DCT_{\tau 2}$	-	-	300	-	296.6	1218
		DCT_n	-	-	-	-	180.0	1459
Inverse Modified DCT [20]	IMDCT	$IMDCT_0$	40	48	-	129.5	174695.5	2582
		$IMDCT_1$	-	-	-	-	827.5	2854
		$IMDCT_{\tau 1}$	-	-	750	-	748.5	2820
		$IMDCT_{\tau 2}$	-	-	820	-	815.0	2943
		$IMDCT_n$	-	-	-	-	810.0	3714
Viterbi decoder [20]	Viterbi	$Viterbi_0$	10	10	-	30.4	334.4	843
		$Viterbi_1$	-	-	-	-	273.6	903
		$Viterbi_{\tau 1}$	-	-	200	-	194.8	959
		$Viterbi_{\tau 2}$	-	-	230	-	217.2	938
		$Viterbi_n$	-	-	-	-	828.0	1431
JPEG encoder [94]	JPEG	$JPEG_0$	62	76	-	54.7	34689.2	1851
		$JPEG_1$	-	-	-	-	436.9	2078
		$JPEG_{\tau 1}$	-	-	300	-	289.4	2113
		$JPEG_{\tau 2}$	-	-	350	-	344.1	2052
		$JPEG_n$	-	-	-	-	378.0	2742
Turbo encoder [95]	Turbo	$Turbo_0$	35	41	-	77.8	1249857.0	3008
		$Turbo_1$	-	-	-	-	712.4	3169
		$Turbo_{\tau 1}$	-	-	400	-	389.6	3387
		$Turbo_{\tau 2}$	-	-	450	-	413.4	3287
		$Turbo_n$	-	-	-	-	324.0	4232
MPEG-2 decoder [96]	MPEG	$MPEG_0$	89	101	-	67.9	106195.6	2787
		$MPEG_1$	-	-	-	-	751.0	3200
		$MPEG_{\tau 1}$	-	-	380	-	351.8	3311
		$MPEG_{\tau 2}$	-	-	450	-	419.7	3141
		$MPEG_n$	-	-	-	-	288.0	4445
G721 encoder	G721	GSM_0	12	17	-	93.6	50169.6	1085
		GSM_1	-	-	-	-	392.4	1126
		$GSM_{\tau 1}$	-	-	250	-	237.6	1206
		$GSM_{\tau 2}$	-	-	300	-	295.2	1150
		GSM_n	-	-	-	-	252.0	1376

Although containing only input/output instructions, additional states inserted into the STG of an application for capturing state data still require a number of PEs for concatenating data into *n-bit* packets. This causes a certain hardware overhead. Column $Used\ PEs$ in Table 5.1 shows the required hardware resource in term of PEs for my implementations. Using implementations without preemption points (A_0) as the basic, Fig. 5.7 presents how the hardware overhead varies when preemption points are inserted. In Fig. 5.7, symbols A_1, $A_{\tau 1}$, $A_{\tau 2}$ and A_n denote implementations corresponding to Table 5.1 where A is the name of applications. The hardware overhead varies from 4% to 15% for the A_1 case, from 11% to 15% for the $A_{\tau 1}$ case, from 6% to 13% for the $A_{\tau 2}$ case, and from 27% to

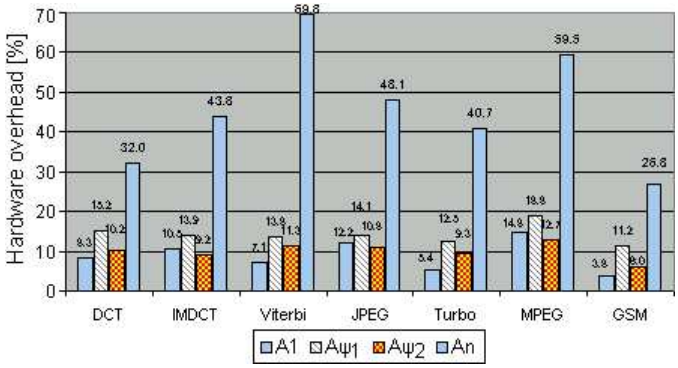


Fig. 5.7: Hardware overhead

70% for the A_n case.

The hardware overhead of implementations according to the proposed algorithm is not large. More importantly, they are even smaller than the A_1 implementation in some cases ($A_{\tau 2}$ vs. A_1 for IMDCT, JPEG and MPEG). This results from the optimization performed by the refining algorithm to eliminate redundant preemption points using a given preemption latency as a criteria. Although some additional preemption points need to be inserted in order to satisfy the given preemption latency, other unnecessary preemption points could be removed. As a result, the hardware overhead could be reduced.

5.6.3 Preemption latency

Preemption latency τ can be defined as the time from a preemption request until a preempting task is ready to run, and it can be computed according to equation 5.6.1.

Fig. 5.8 shows the maximum preemption latency for each implementation. Basically, τ_{input} is used as a constraint for optimizing generated preemption points in Algorithm 2 (Section 5.4.2.2). Such implementations ($A_{\tau 1}$ and $A_{\tau 2}$) have better preemption latency over correspondent versions without such constraints (A_0). In some implementations (IMDCT, Viterbi, JPEC and GSM) the preemption latency of the A_τ versions is even smaller than that of the correspondent A_n although the latter has no delay for reaching a preemption point (A_n versions can be preempted at any state). This means in those cases, time to save and restore state data at some points dominates the total preemption latency.

5.6.4 Hardware overhead vs. preemption latency

At the first sight, hardware overhead seems to be reduced when the number of preemption points is eliminated; or, in other words, preemption latency is increased. In order to see the trade-off between these two parameters, different preemption latencies are provided as the input parameter to the refinement algorithm (Algorithm 2, Section 5.4.2.2) and results are presented on Fig. 5.6.4, where preemption latency is on X-axis and hardware overhead in the number of PEs is on Y-axis.

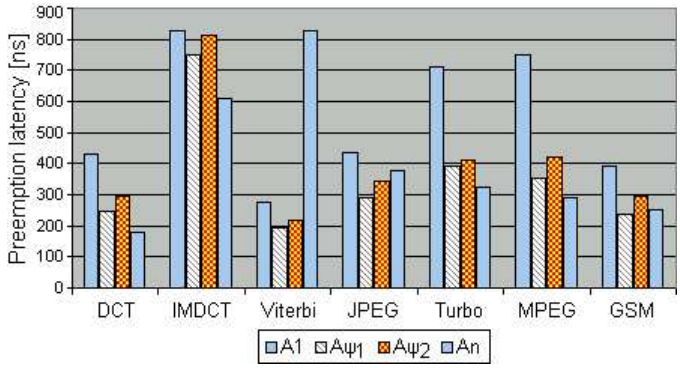


Fig. 5.8: Maximum preemption latency

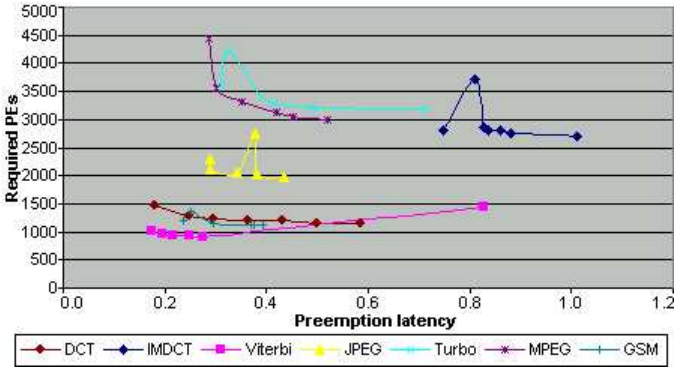


Fig. 5.9: Hardware overhead vs. Preemption latency

When preemption latency becomes larger, hardware overhead tends to reduced. Nonetheless, some implementations show more complicated relationship when preemption latency increases, hardware overhead also increases at some points. Looking into more details, these points correspond to the situation where preemption is allowed at every state (A_n version). In this case, both preemption latency and hardware overhead are influenced by the amount of state data necessary to save. When this amount is large, both these parameters also grow. Therefore, it may not be a good solution to allow preemption at every state, and my proposed method achieves its merit, which satisfies a constraint on preemption latency with reasonable hardware overhead.

5.7 Conclusion

A method for identifying preemption points and inserting extra states to capture and restore state data of applications implemented on coarse-grained dynamically reconfigurable devices based on resource requirements is proposed to enable a preemptive multitasking environment where a running task can be preempted. Evaluation results on the DRP architecture show that the proposed method may satisfy a user-specified preemption latency within a reasonable amount of hardware overhead. Moreover, the steps of the proposed method are integrated into the system design flow to assist

designers in developing applications on dynamically reconfigurable devices. Also, the trade-off between preemption latency and hardware overhead is also presented and discussed.

Chapter 6

Multicore Reconfigurable Architecture

6.1 Problem

Recently, multicore processors have emerged as a dominant trend in the chip making industry due to their advantages over the single-core architecture [15] [110] including: (1) to overcome the limitation of single-core architecture relating to CMOS process, wiring delay and power consumption; (2) to improve performance based on thread-level parallelism in addition to ILP, which has been exploited for improving the performance of the single-core architecture for the last two decades; (3) and, to bridge the gap of memory wall. In embedded devices, in which low power coupled with high performance is quite critical, a trend to shift to multicore can be observed as well [111].

Recently, dynamically reconfigurable accelerators [14] that use an array of simple coarse-grained PEs, have been popularly used, and some of them are embedded in SoCs toward electronic appliances [112], [113] and [114]. A PE is often composed of a simple shift and mask unit, an arithmetic and logic unit, and a register file. Each PE has a memory component to hold multiple circuit configuration data referred as contexts, each of which is constructed by operational instructions for the PE and connection instructions for routing resources. By rapidly switching contexts, the PE operations and intra/inter-PE connections can be dynamically reconfigured every one or few clock cycles.

For efficient use of such accelerators, since a single hardware task often lacks enough degree of parallelism, executing multiple tasks is advantageous. In Chapter 4, a task-level parallelism mechanism has been investigated by mapping the tasks of an application into a reconfigurable array that is partitioned into clusters of hardware execution units (tiles or tile groups). However, because of the limitation of the number of tiles and tile groups in a currently available DRPA, only a limited number of tasks can be implemented. Big applications composed of many tasks, or tasks whose sizes are large, or multiple applications each of which consists of a number of tasks are difficult and almost impossible to be implemented. Furthermore, it is hard to enlarge the reconfigurable array of a chip because of matters such as the size of the chip, power consumption, clock control and chip complexity. To further exploit a task-level parallelism, toward an efficient multitasking environment on DRPAs, it is necessary to investigate a reconfigurable multicore structure.

In Chapter 5, a mechanism for preempting a task executing on a DRPA has been proposed and examined. However, evaluation results obtained come from a simulation environment that allows to execute one task at a time. As being mentioned in such a situation, preemption latency is not the one in real systems but the worst case. In order to provide an environment where multiple applications can be assigned and executed in parallel, a multicore architecture is a good platform to experience and to further extend the proposed preemption algorithm.

In conventional FPGAs, multi-task systems with their partial reconfigurability have been proposed. The architecture of FPGAs allows to divide the reconfigurable array into one-dimensional (column oriented) slots, each of which can hold and execute a task; more importantly, that helps the partial reconfiguration mechanism be implemented [115] [116]. Using partial reconfiguration capability on such FPGAs, while many tasks are executing, it is possible to stop a certain task, free the slot occupied by the task, load a new task into this slot, and execute the new task [115] [117] [118]. Slots may either be equal in size, as used in [119,103], or have variable sizes, as proposed in [120,121,122].

Another example comes from the architecture of NEC Electronics' DRP-1, in which the reconfigurable array is divided into eight tiles, which can work independently and communicate with each other [63]. Multiple hardware tasks can be implemented on one or several numbers of tiles, and executed in parallel or pipelined manner.

Using a Network-on-Chip (NoC) for inter-task communications in reconfigurable platforms has been proposed. In an NoC, source nodes generate packets that consist of a header and a payload data, which are transferred by on-chip routers through connected links, and decomposed by destination nodes. Since different packets can be simultaneously transferred on multiple links, the bandwidth of NoCs is much larger than that of buses. In addition, the wire-delay problem is resolved, since each flit of a packet is transferred on limited length point-to-point links, and buffered in every router along the routing path. By introducing error detection and re-transmission protocols, dynamic transmission errors caused by crosstalk, which will come up in future smaller process technologies, can be also solved [123].

In this work, I propose a mutlicore reconfigurable architecture in which each core can be assigned a task for execution. Many multiprocessor systems equip an on-chip bus for inter-core communication because of the relatively small number of cores. Nonetheless, in the proposed architecture, a regular NoC is introduced in order to ensure each core scalability and modularity. Different from the tile-based architecture introduced in Chapter 4 in which tiles are directly connected and hardware tasks assigned to tiles for execution can communicate to one another using embedded memory modules arranged as FIFOs, the proposed multicore architecture is equipped an NoC to connect computational cores. These two architectures have not been well evaluated and compared with real application programs. In this section, I build correspondent architectures based on NEC's DRP-1 and implement applications on each architecture in order to see how these architectures affect performance and resource usage.

6.2 Related Work

Techniques on mapping multiple tasks into a single FPGA has been widely researched. The architecture of many FPGAs permits to execute multiple tasks based on the one-dimensional partitioning model where an FPGA's reconfigurable area is partitioned into the stripes of full array height or slots. Slots are placed side by side and may be equal in size [103] [119] or have variable widths [121] [120]. By exploiting the partial reconfiguration capability, slots can be individually reconfigured. In these systems, however, tasks communicate by means of programmable routing resources of the FPGA. Concerning resource management, the one-dimensional partitioning model is equivalent to a homogeneous multi-processor model.

The reconfigurable surface can also be divided into predefined slots, which are allocated to tasks for execution according to a scheduling algorithm. Due to the fragmentation problem, many solutions have been proposed. [21] tries to rearrange tasks executing on a partially reconfigurable FPGA by local repacking and ordered compaction. Exploiting the technique of task relocations and transforms to reduce fragmentation is discussed in [40]. [121] investigates different placement techniques for coarse-grained, non-rectangular tasks on partially reconfigurable FPGAs.

Unlike FPGAs, a DRPA supports a high speed dynamic reconfiguration mechanism during execution. In order to map multiple tasks into such devices, special architectural support is required. NEC's DRP-1 [63] is consisting of eight tiles each of which can control dynamic reconfiguration independently. Several tasks can be assigned into one or a number of tiles, and executed in parallel communicating with each other. Hitach's FE-GA [67], SANYO's car turner DRPA [114], and Toshiba's SAKE [113] uses a multi-core structure consisting of relatively small DRPAs for special mapping of multiple tasks. Such types of DRPAs are designed based on an observation that a group of relatively small sized arrays is efficient in various applications [82].

Using an NoC in a reconfigurable fabric has been proposed by [28] in order to improve the interconnection network in terms of structure, performance and modularity. [118] [124] [106] [125] have proposed an FPGA-based multi-task system in which an NoC is used for inter-task communications. In [118], a two-dimensional torus is introduced as a network topology. Each router for inter-task communications has two virtual channels, and provides wormhole packet switching and dimension-order routing, which is one of the deterministic routing algorithms. This system has been implemented on the Xilinx Virtex-II FPGA (XC2V6000).

6.3 Evaluated Architectures

6.3.1 Target Device

In order to evaluate and compare both architectures, DRP architecture from NEC Electronics is selected as the target device for this study. The detail of DRP can be referred from Section 3.2.3.

In this research, a *tile* of DRP is used as a basic unit to compute the size of computational cores.

Therefore, one DRP tile equivalent to 64 PEs is the smallest size a core could be. To describe different core sizes, different number of tiles arranged in a certain shape is used. Generally speaking, the size of a core is expressed as follows:

$$S_{core} = n \times \text{PEs/tile} \tag{6.1}$$

$$1 \leq n \leq 8 \tag{6.2}$$

where: S_{core} denotes the size of a core. n shows the number of tiles, and $PEs/tile$ is the number of PEs in a DRP tile. Since there are eight tiles in a DRP-1 chip, so the maximum core size is eight tiles. Fig. 6.6(a) - Fig. 6.6(d) show examples for the proposed architecture with different core sizes from one to four tiles. In the multi-process execution, there is a way to specify how tiles are joined together to form a certain shape, which is assigned to a task. I adopt this method to correctly specify the size of cores in implemented variants for a target application as show on Fig. 6.6. A target application is partitioned into tasks, whose sizes are small enough to be able to fit into cores. As a result, the smaller the core size is, the more tasks an application should be divided into.

6.3.2 Tile-based architecture

The tile-based architecture studied in this section is assumed to be a two-dimensional (2D) multicontext coarse-grained DRPA consisting of $M \times N$ hardware execution units, each of which is called a tile. A tile consists of a certain number of PEs and may also have other components such as memory modules, multipliers, register files and flip-flops. The size of a tile is computed as the number of PEs it contains. Generally speaking, tiles could have different sizes, but in order to simplify the mapping process, which is not the object of this study, in the thesis, tiles are supposed to be identical, or have equal sizes. This assumption creates a homogeneous architecture, which allows more flexible in mapping.

A hardware tasks can be mapped to a tile for execution providing that the size of the task is equal or smaller than that of a tile. Furthermore, several neighboring tiles can be joined together to form a tile group, where a task whose size is larger that the size of a tile can be mapped into. Embedded memory modules within the reconfigurable array are used to form a communication mechanism between tasks assigned to two tiles, a tile and a tile group, or two tile groups. When two tasks implemented on two different tiles want to exchange data, they can declare a shared memory module arranged as a FIFO to use as an inter-task communication method. FIFOs use a simple handshake mechanism in order for two tasks involving in communication to determine if an 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.

Fig. 6.1 shows an example of a tile-based architecture using the target device of NEC's DRP-1 with 4×2 tiles. Tile groups can be formed by combining close tiles together; so, they may have different shapes. TG_1 and TG_2 are created by the different number of tiles with different shapes.

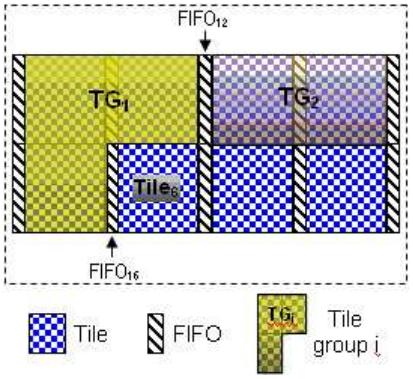


Fig. 6.1: Tile-based architecture

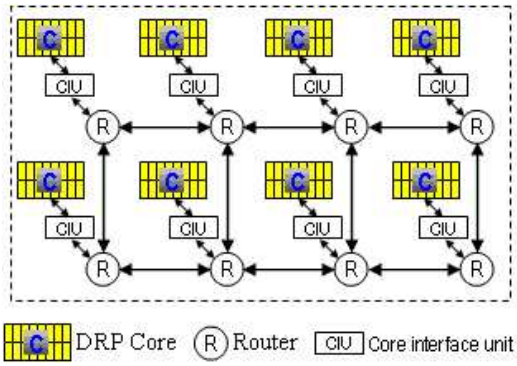


Fig. 6.2: Multicore architecture

$FIFO_{12}$ can be used to exchange data between tasks mapped to TG_1 and TG_2; and, $FIFO_{16}$ is for communicating between TG_1 and $Tile_6$;

For DRP-1, applications are manually partitioned and assigned to tiles or tile groups. Since having eight tiles, the maximum number of tasks that can be concurrently executed is eight. However, in most cases, the number of tasks a target application is partitioned into is less than eight since tile groups are often formed to accommodate tasks that are larger than the size of a tile. A FIFO mechanism, which employs VMEMs between tiles, is used as a inter-task communication method. A FIFO is for one-way communication and acts like a pipe. Writing to and reading from a FIFO are blocking, that is, a task needs to be stalled because of the data shortage.

6.3.3 Multicore architecture

NEC's DRP-1 is used as computational cores for the multicore architecture proposed in this section. Cores are connected by an NoC composing of routers. In order to compare with the tile-based architecture, only 4×2 cores with a two-dimensional mesh topology are used in the study as shown on Fig. 6.2.

The network uses the wormhole switching technique with dimension-order routing. A wormhole routing allows data packets to be pipelined through the network and requires only a small buffer in a router to store a part of a packet (flits). Dimension-order routing, which uses Y-dimension channels after X-dimension channels in a 2D mesh and torus, can be implemented with simple combination logic on a router and does not demand to store routing information in a routing table. In the network, a data packet is broken into flits, which belong to one of three types: header flits, body flits and tail flits. In order to avoid deadlocks, virtual channels are employed.

Fig. 6.3 shows the router architecture used in this study. A router consists of a crossbar switch, an arbitration unit (ARB), input and output physical channels. Each physical channel has two virtual channels, each of which has a FIFO buffer for storing four flits. The router architecture is fully pipelined, and allows to transfer a header flit through three pipeline stages that include routing computation, virtual channel and switch allocation, and switch traversal.

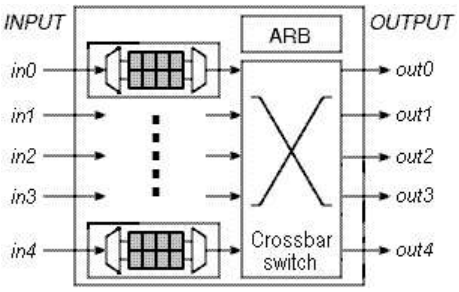


Fig. 6.3: Router architecture

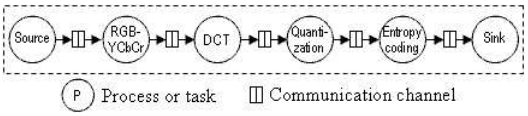


Fig. 6.4: Representation of JPEG encoder

Table 6.1 shows the implementation of a router. A router is synthesized, placed, and routed with a 90nm standard cell library.

Table 6.1: Router implementation

Parameters	Value
Process	ASPLA 90nm
Operating frequency	Maximum up to 500 MHz
Flit size	64/128 bits
Number of ports	5
Number of virtual channels	2
Buffer size	4-flit for each virtual channel
Packet length	4-flit data + 1-flit header

In order to connect cores to routers, a fixed interface, which is referred as a core interface unit (CIU), is used (Fig. 6.2). In this study, the input/output interface of DRP-1, which consist of two 64-bit separated channels, one for input and another for output, is exploited as a CIU. CIUs serve two purposes. First, a CIU can convert data exchanging between a core and a router. The channel width of the network W can be expressed as: $W(bit)$ = flit size + 2 , where *flit size* is the size of a flit or the width of a physical channel, and *2* represents two bits for flit header information. For example, since the input/output interface of DRP-1 with 64 bits is employed to connect to a router, the actual channel bit width between routers is 66 bits. A CIU is used to convert from core's 64-bit channels to router's 66-bit channels. When data are transferred from a DRP core to a router, the correspondent CIU adds two bits containing a flit type into 64 bits; on the way back, when data are sent to a DRP from a router, those two bits of a flit type are removed.

Second, another important reason for using CIUs is to allow tasks to be independent from physical locations. In a multitasking environment where tasks are dynamically assigned into and removed from cores, the exact place of a certain task can only be determined at the run time by the operating system, and it is often changed from time to time either when the system is defragmented or when the task is removed and later resumed. In a general case, for an application made of multiple tasks implemented on a such environment, the physical place of tasks are not known at the design time.

This raises the question of how tasks can communicate to one another if their places are not determined in advanced. My solution is to use CIUs. At the run time, when a task is assigned into a core, the operating system also updates the routing table of the correspondent CIU to clearly show where other related tasks are placed. When task A needs to communicate to task B, for example, the routing information of task B will be added into header flit coming out from task A. Therefore, by using CIUs, it is possible for tasks to communicate independently from their physical places.

6.3.4 Application model

A target application is represented with multiple tasks that can be arranged to execute in a pipelined manner. That is, computations can be specified as a data flow graph with streams of data items (edges) flowing between computation stages (nodes) (Fig. 2.2.1). Basically, a task gets a data stream from its input communication channel, executes its own computation, and produces the correspondent result to the output communication channel. Fig. 6.4 shows an example of a representation for a JPEG encoder.

With the tile-based architecture, each task of a target application is mapped onto a tile or a tile groups depending on their size. With the multicore architecture, tasks are mapped onto cores. In both architectures, all tasks belong to a target application are arranged for executing in a pipelined manner.

6.4 Evaluation

6.4.1 Simulation environment

Fig. 6.5 shows the simulation environment used in this study. A target application is partitioned into multiple tasks *(Task 1, Task 2, ..., Task i, ..., Task n)*, which are described in BDL. Configuration data for DRP-1 generated by the compiler/synthesizer can be loaded onto a DRP-1 chip for execution or for online debug. Apart from configuration data, the compiler also generates Verilog description files corresponding to BDL source files. These Verilog descriptions with Verilog files implementing router architecture, network architecture, core interface and test bench become the input for an RTL simulation. With relevant input data, the RTL simulator can run a simulation and produce correspondent results and reports such as critical path, resource usage, execution clock cycle for evaluation.

6.4.2 Two architectures

6.4.2.1 Target applications

Several real applications shown in Table 6.2 have been implemented in order to compare between the two mentioned architectures shown in Fig. 6.1 and Fig. 6.2.

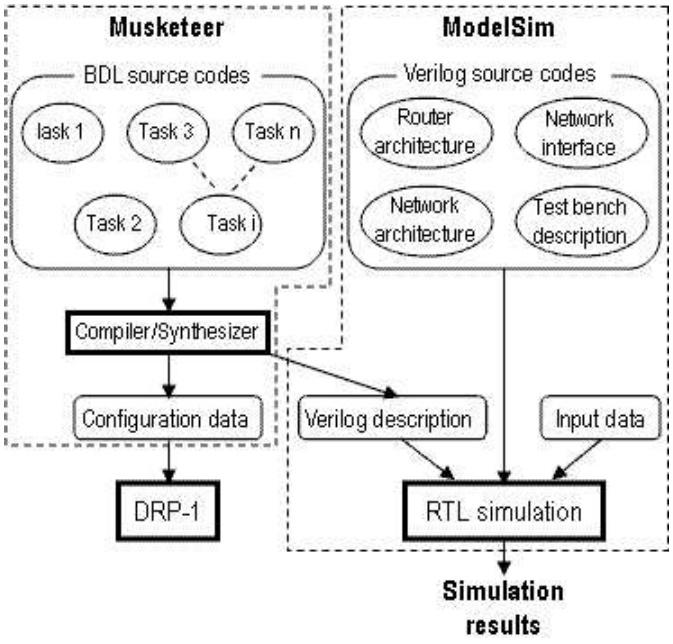


Fig. 6.5: Simulation environment

Target applications are manually partitioned into tasks in such a way that suits specific implementations. In the tile-based architecture, applications are modeled, partitioned into tasks and mapped onto the reconfigurable array of DRP-1 according to the optimized algorithm presented in [16]. Since data are exchanged using FIFOs, the communication channel bit width can be varied from 8 to the maximum of 64 bits. The same configuration is applied to the multicore architecture in order to evaluate the effect of using an NoC for inter-task communication instead of FIFOs. More importantly, the channel bit width of the NoC in the multicore architecture is evaluated with 64 bits and 128 bits in order to see how the width of communication channels influences on performance.

6.4.2.2 Results

Table 6.2 shows the throughput of target applications implemented in two mentioned architectures. *Tile-based* column specifies the result obtained with the tile-based architecture. *Multicore (64 bits)* and *Multicore (128 bits)* present implemented results in the multicore architecture with the channel bit widths of 64 bits and 128 bits respectively.

It is easy to see that the bit width of communication channels between tasks plays an important role in improving throughput. While the implementation in the multicore architecture with the 64-bit channel width, which is the same as the communication channel width of the tile-based architecture, results in lower throughput, the same configuration with the 128-bit channel width achieves better throughput in many cases except IMDCT and MPEG implementations. This shows that communication significantly affects throughput. While it takes only one clock cycle to transfer a data packet ($1bit \leq$ size of a data packet $\leq 64bits$) from one task to another in the tile-based architecture using a

Table 6.2: Throughput of two evaluated architecture

Application	Abbr.	Tile-based [Mbps]	Multicore (64 bits) [Mbps]	Multicore (128 bits) [Mbps]
Discrete Cosine Transform	DCT	341.9	297.1	345.0
Inverse Modified DCT	IMDCT	182.3	156.2	178.8
Viterbi decoder	Viterbi	4.32	3.9	4.34
JPEG encoder	JPEG	106.9	89.2	107.3
Turbo encoder	Turbo	4.0	3.4	4.1
G712 encoder	G721en	45.1	33.1	46.0
G712 decoder	G721de	10.3	9.2	10.5
MPEG-2 decoder	MPEG	10.4	8.7	9.7

FIFO, several clock cycles are needed to transfer a packet from one router to another in the multicore architecture depending on the distance between two routers and the amount of data. The further a target router is, the more number of clock cycle is required for communication. Moreover, while it takes at least four clock cycles to transfer one data packet, which corresponds to a flit, between two adjoining routers, time for communication will reduce if the amount of transferring data is increased since it takes a fixed number of clock cycles to set up a communication path between two routers in an NoC.

6.4.3 Evaluation with different core sizes

6.4.3.1 Implementation variants

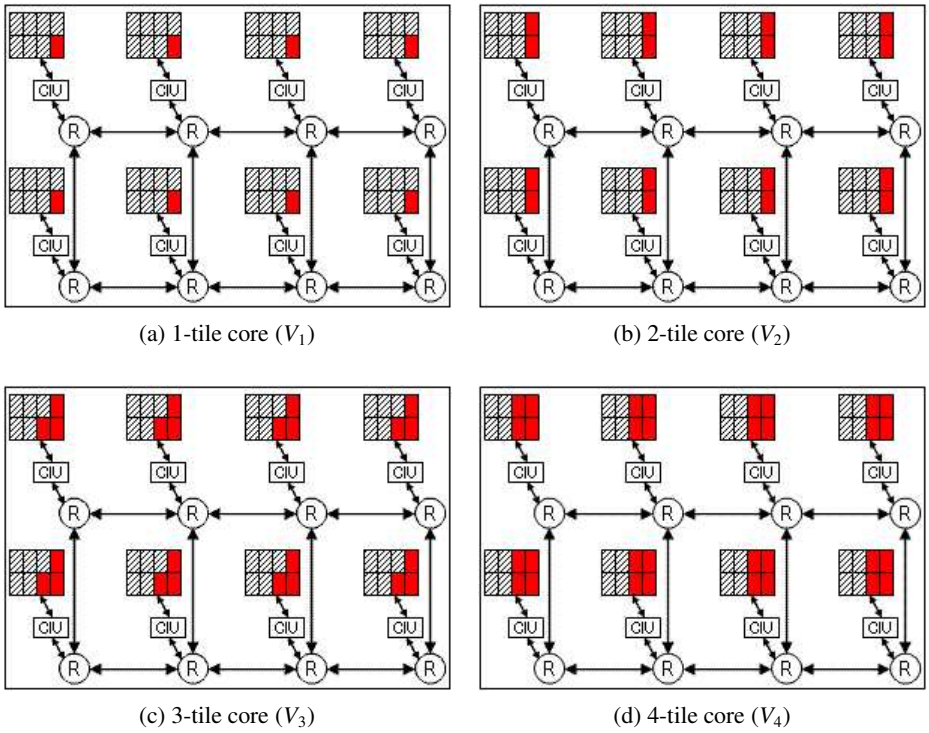
In order to see how the core size influences on the performance of target applications, this evaluation uses the same multicore architecture shown in Fig. 6.2 with variable core sizes. In Fig. 6.2, each core is an 8-tile DRP-1, but Fig. 6.6 show the same architecture with each core has the size of one (Fig. 6.6(a)) to four tiles (Fig. 6.6(d)). For example, Fig. 6.6(b) shows an architecture where each core has the size that is equal to two DRP tiles, or in other words, each core can be considered to contain 128 PEs. Fig. 6.6(a), Fig. 6.6(c) and Fig. 6.6(d) illustrate similar architectures with the core sizes of one, three and four DRP tiles, respectively.

Since the purpose of this evaluation is not to get the maximum performance, but to show the effect of the core size on performance, only 64-bit communication channels are applied for the interconnection network.

6.4.3.2 Application partitioning and mapping

Following rules are adopted when partitioning and mapping target applications onto evaluated architectures.

- Since all evaluated variants only use eight cores (Fig. 6.6(a)-Fig. 6.6(d)), the number of tasks each application is partitioned into must be equal or less than eight.



(a) 1-tile core (V_1)

(b) 2-tile core (V_2)

(c) 3-tile core (V_3)

(d) 4-tile core (V_4)

Fig. 6.6: Implementation variants

- For a specific variant, each application is partitioned into tasks whose sizes are suitable to the core size of the implementation. For example, in V_1 variant where the size of each core is equal to the size of one tile in DRP architecture (Fig. 6.6(a)), a certain target application should be divided into tasks in a way that each of which can be implemented using only 64 PEs. If a task cannot fit within one DRP tile, it must be further partitioned into two or more tasks, each of which is small enough to fit in one tile.

- For variants with large core sizes (V_3 and V_4 variants), the number of tasks each target application is partitioned into may be less than eight; or in other words, in such cases, not all cores are used. The decision of how many tasks an application should be partitioned into depends on performance. For each variant, different cases, which have different number of tasks, have been implemented and evaluated. And, the one with the best performance is chosen and presented on Table 6.3.

- The tasks of an application are mapped in such a way that two tasks that need to exchange data are assigned to two cores close to each other to reduce network delay as the number of routers between two cores decreases.

Table 6.3: Implementation results of target applications

Application	Abbr.	Variant	Symbol	Tasks	Throughput [Mbps]	Fragmentation [%]
Discrete Cosine Transform	DCT	1-tile core	V_1	8	142.4	11.9
		2-tile core	V_2	5	267.8	26.7
		3-tile core	V_3	4	297.1	43.7
		4-tile core	V_4	3	291.4	65.3
Viterbi decoder	Viterbi	1-tile core	V_1	6	2.5	26.7
		2-tile core	V_2	4	3.9	33.6
		3-tile core	V_3	4	3.9	57.1
		4-tile core	V_4	3	3.5	66.0
JPEG encoder	JPEG	1-tile core	V_1	8	39.9	17.1
		2-tile core	V_2	6	66.2	29.1
		3-tile core	V_3	5	89.2	30.2
		4-tile core	V_4	4	92.3	51.1
Turbo encoder	Turbo	1-tile core	V_1	8	1.7	18.1
		2-tile core	V_2	5	3.4	31.2
		3-tile core	V_3	4	3.2	37.7
		4-tile core	V_4	4	3.2	49.7
G712 encoder	G721	1-tile core	V_1	8	3.5	19.8
		2-tile core	V_2	6	5.2	30.2
		3-tile core	V_3	5	9.2	40.3
		4-tile core	V_4	4	8.7	51.2

6.4.3.3 Implementation results

Table 6.3 shows the evaluation results for every implemented variants of target applications. Column "Tasks" denotes the number of tasks each application is partitioned into. As mentioned earlier, for variants V_1-V_4, the number of tasks on column "Tasks" is the best implementation case I can achieve in terms of throughput.

6.4.3.4 Throughput

It is easy to see that the throughput of implementations with different core sizes (V_i variants) are considerably varying. The best throughput when the core size is equal to two tiles (V_2) is the implementation of Viterbi and Turbo; in the case of three tiles (V_3) this corresponds to DCT, Viterbi and G721; and when implementing with four tiles (V_4), JPEG implementation achieves the highest throughput. Apparently, the size of cores influences the throughput of applications implemented in a multicore architecture.

- When the core size is small, for example, (V_1) variants, a target application must be partitioned into multiple small tasks in order to fit small cores. This creates a great amount of communication, which causes throughput to drop significantly. With larger core sizes, the number of tasks each application needs to be divided into is reduced; so, less communication is required, and throughput is improved.

- Nonetheless, if the core size becomes large like (V_4), throughput might not be as good as expected. Among the target applications in this study, only JPEG has the best throughput with V_4 variant. A large core can hold a bigger task, so the partition of an application is likely to generate fewer tasks. This reduces the amount of communication among tasks, and the throughput could become better. However, a fewer number of tasks means a smaller number of pipelining stages, and, the workload balance between computation stages is hard to maintain. Therefore, throughput could be reduced. In addition, large cores do not use resources effectively as discussed in the next section.

6.4.4 Internal fragmentation

Internal fragmentation is the fragmentation inside the boundary of a task. This type of fragmentation comes from the rectangular shape of a task. In the proposed multicore system, each task is mapped onto a separated core with a fixed size; and, since the size of task is often smaller than that of a core, some resources may not be used, or they get wasted. There are several types of resources contained in a tile such as PEs, VMEMs, HMEMs, register files and flip-flops, but in this study, only PEs are taken into account for calculating internal fragmentation. For a multicontext DRPA, the total internal fragmentation of an application (F_{app}) can be computed as follows.

$$F_{context\_i}(\%) = \frac{PE_{not\_used\_i}}{N} * 100$$

$$F_{task\_j}(\%) = \frac{\sum_{i=1}^{k} F_{context\_i}}{k}$$

$$F_{app}(\%) = \frac{\sum_{j=1}^{T} F_{task\_j}}{T}$$

In this equation, $F_{context\_i}$ is the fragmentation in i^{th} context. It is the ratio of the number of unused PEs ($PE_{\text{not\_used\_i}}$) in the context to the size of a core (N). F_{task_j} denotes the fragmentation of j^{th} task; N is the size of a core in terms of the number of PEs; k is the total number of context a task requires to implement; and T is the number of tasks the application is partitioned into. According to the above equation, the internal fragmentation of an application is the average of the fragmentation of its tasks.

From the internal fragmentation of implementations in Table 6.3, It is easy to see that the fragmentation is growing as the core size increases, or in other words, the larger a core size is, the less effective resource usage becomes. As a result, the size of cores in a multicore system plays an important role in balancing between throughput and resource usage.

6.5 Conclusion

This section examines two architectures, a tile-based architecture using embedded memory modules arranged as FIFOs for inter-task communication, and a multicore architecture employing an NoC to connect computational cores. Target applications are partitioned into a number of tasks, each of which is mapped onto a tile or a tile group in the tile-based architecture, and onto a core in the multicore architecture. All tasks of an application are arranged to execute in a pipelined manner. Based on NEC's DRP-1, the evaluation result shows that the width of communication channels largely affects on performance. Also, using an NoC to form a multicore architecture proves to be an effective method toward improving the performance of implemented applications, which can be modeled using a pipelined processing manner. Moreover, this section tries to investigate how the size of cores influences the performance of target applications. Evaluation results received from a number of real applications implemented on the system with different core sizes show that the size of core plays an important role in balancing between throughput and resource usage. The most suitable core size in many cases is about the size of two or three tiles of DRP architecture.

Chapter 7

Conclusion and Future Work

7.1 Thesis Summary

The focus of this thesis is to propose, implement and evaluate feasible mechanisms that support to create a multitasking environment for coarse-grained dynamically reconfigurable processors. This is achieved by firstly introducing the methodology to guide research and to organize the thesis. Several concepts including the definitions of applications, hardware tasks and inter-process communications, which are somehow different from that of the microprocessor domain, are described. An outline of a system using a coarse-grained DRPA supporting a multitasking environment is also explained. Moreover, three multitasking models based on resource sharing are introduced.

Secondly the background of coarse-grain DRPAs is examined based on general architectural aspects available on currently existing devices. These aspects can be used as criteria to categorize coarse-grained DRPAs both currently available and future released devices into different classes. Then a number of DRPAs released by consumer-electronics makers in the last decade including not only commercially available products but also new devices under research and development is introduced. Although the target device of research in this thesis is NEC Electronics DRP-1, others described in Table 3.2 could become potential target hardware to apply proposed solutions with relevant changes. Specifically, the proposed preemption algorithms (Chapter 5 and the multicore reconfigurable system (Chapter 6) can be implemented with other DRPAs with minor modifications; the proposed mapping method (Chapter 4) needs to be modified to adapt to other platforms because the method assumes the two features specific to NEC's DRP architecture: using embedded memories as FIFO to exchange data between hardware execution units (tiles), and forming tile groups from separated tiles. As a result, the method can directly apply on platforms that support these two features; and for other devices without either such two characteristics, the mapping algorithm has to be changed.

Thirdly, 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 is proposed and evaluated. By exploring the multi-process execution in dy-

namically reconfigurable processors, which is a technique to enhance throughput through exploiting more inherent parallelism of applications. Basically, a total process for an application is divided into small processes, assigned into limited areas of a reconfigurable array, and concurrently executed in a pipelined manner. In order to do that, the size of tiles, which is a unit area of dynamically reconfigurable array, and the grouping of processes are adjusted. Using real applications such as DCT, JPEG encoder and Turbo encoder, the impact of different versions mapped onto the NEC Dynamically Reconfigurable Processor on performance is evaluated. Evaluation results show that the proposed mapping algorithm achieves the best performance in terms of the throughput and the execution time.

Fourthly, a research for a suitable mechanism to preempt tasks executing on dynamically reconfigurable processors is performed. Basically, when a task is preempted, its necessary state information must be correctly preserved in order for the task to be resumed later. Not only do coarse-grained DRPAs have different architectures using a variety of development tools, but the great amount of state data of hardware tasks executing on such devices are usually distributed on many different storage elements. To address these difficulties, a general method for capturing the state data of hardware tasks targeting coarse-grained DRPAs is proposed and evaluated. Based on resource usage, algorithms for identifying preemption points and inserting preemption states subject to user-specified preemption latency are proposed. Moreover, a modification to automatically incorporate proposed steps into the system design flow is also discussed. The performance degradation caused by additional preemption states is minimized by allowing preemption only at predefined points where demanded resources are small. The evaluation result using a model based on NEC ElectronicsĄf DRP-1 shows that the proposed method can produce preemption points satisfying a given preemption latency with reasonable hardware overhead (from 6% to 15%).

Last, a new multicore reconfigurable system, which is a potential architecture candidate for realizing a multitasking environment, is proposed. As an extension of Chapter 4, a comparision between application impementations using a multi-thread execution model on a tile-based architecture and a multicore architecture is performed. Based on such a multicore reconfigurable architecture, a research on how the size of cores in a multicore reconfigurable system influences on the performance and the resource utilization of target applications is examined and discussed. Recently, an efficient multicore architecture for processors toward multitasking design has emerged as a dominant trend in the chip making industry. As reconfigurable devices gradually prove their capability in improving computation power while preserving flexibility, I examine a multicore reconfigurable architecture composed of multiple hardware execution units or cores connected by an interconnection network. An application is partitioned into a set of tasks, each of which is mapped onto a core for execution in a pipelined manner. Using real applications implemented on the proposed architecture in which cores are based on the tile structure of NEC Electronics' DRP-1, the evaluation result shows that the size of core is a trade-off between throughput and resource usage, and the size equals to two or three DRP tiles is an appropriate choice for many cases.

7.2 Suggestions for Future Research

This thesis presents several key methods toward building a multitasking environment on coarse-grained DRPAs. Although this topic has been the main research subject for GPPs for a long time, and a lot of research results have come into real life through implementations in processor architectures, operating systems, software developments and user realization, related studies in the field of reconfigurable computing are still immature. To date, an effective reconfigurable architecture, an operating system and a software framework that can support a true multitasking environment have not yet appeared. Apart from having a few researches in this field, it is necessary to take into consideration a lot of difficulties to transform equivalent concepts and implementations from software world based on GPPs to hardware world based on reconfigurable devices. In this thesis, I developed only some basic methods for a simple multitasking computational model. As a result, the work opens up a great field for future research. At this point, I recommend following directions for future work

- **Mapping method for hardware tasks on multicore reconfigurable architecture:** When proposing the mapping algorithm in Chapter 4, I assume the two following conditions: (1) Embedded memories are used as FIFOs to exchange data between tiles since a target application is modeled as a KPN. (2) Neighboring tiles can be joined together to form tiles groups. However, it is not always the case for many DRPAs, some of which might not allow embedded memories to work as FIFOs for data communication between hardware execution units (HEUs), others may not have capability to join HEUs and allows them to work independently and in parallel. Moreover, in the proposed multicore reconfigurable system (Chapter 6), cores are connected by an international network not FIFOs; so, an original KPN cannot be directly applied. Besides, it is often difficult to join several cores together to create a core group, a concept similar to a tile group described in Chapter 4. As a result, a new mapping method is necessary for a multicore reconfigurable architecture, where cores cannot communicate using the FIFO mechanism.

- **Simulation environment for improving the proposed preemption algorithm under real situation:** the preemption algorithm introduced in Chapter 5 was evaluated in a limited scenario where only one application occupies and executes on the whole target reconfigurable array. That is why in the evaluation part of Chapter 5, only the maximum preemption latency was provided, not the real preemption latency. In a real situation, multiple applications can be assigned to different execution units for execution in parallel. Then a certain running application may be preempted; the resources occupied by the application are freed; and a new application could be brought in for execution. In such a circumstance, the actual preemption latency of a target application will vary depending on the number of applications in the system at a given time, the moment when a preemption request is issued, and the condition of the system. In addition, the result given in Table 5.1 does not take into account time for transferring

data between the target reconfigurable device and the outside memory. Since the input/output channel of the target device is shared with other applications, sometimes, it could be blocked. All of this happen in a real system and make the actual preemption latency increase. In order to evaluate preemption latency more elaborately, it is necessary to build a simulation environment that can capture all aspects of a real system.

- **Design issues for multicore reconfigurable architectures:** The multicore reconfigurable system proposed in Chapter 6 is just a first step. The next step is to evaluate different structures and configurations such as how different topologies of an interconnection network influence on performance, area overhead and the structure of the system; how large an interconnection network occupies or how much area overhead caused by a network is; how different channel bit widths affects area overhead and throughput; how to map an application consisting of multiple tasks onto a multicore architecture to achieve certain goals; and, how to defragment the system when tasks are dynamically allocated and freed.

- **Design issues for interconnection networks:** As interconnection networks may have great impact on performance, area overhead, wire delay and power of a multicore reconfigurable architecture, they need more attention [28, 118, 124, 123]. One of the solutions to the wire delay problem is three-dimensional IC technology that stacks multiple wafers or dice using vertical interconnects [126, 127, 128, 129]. Stacked dynamically reconfigurable processors are an innovative approach to solve the issue of stretched programmable wires.

- **Prototype of a reconfigurable architecture that supports a multitasking and multi-thread model:** In spite of introducing several mechanisms toward a multitasking environment on DRPAs, a complete system has not yet been build and evaluated. It is crucial to design such a system with proposed solutions in order to examine how well proposed methods work in a general architecture and in a real situation.

Abbreviation and Acronyms

2D	Two-dimensional
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BDL	Behavioral Design Language
CIU	Core Interface Unit
CPU	Central Processing Unit
DCT	Two-dimensional Discrete Cosine Transform
DMU	Data Manipulation Unit
DPU	Datapath Unit (CS2112)
DRP	Dynamically Reconfigurable Processor
DRPA	Dynamically Reconfigurable Processing Array
DSP	Digital Signal Processor, Digital Signal Processing
FFU	Flip-Flop Unit
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GPP	General Purpose Processor
HDL	Hardware Description Language
HEU	Hardware Execution Unit
HMEM	Horizontal Memory
I/O	Input/Output
ILP	Instruction-Level Parallelism
IMDCT	Inverse Modified Discrete Cosine Transform
IP	Intellectual Property
IRAM	Internal Random Access Memory
JPEG	JPEG encoder
KPN	Kahn Process Network
LUT	Look-up-Table
MAC	Multiply-Accumulate

MEM	Memory module
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instructions Per Second
MLT	Multiply and Accumulate Cell (Hitachi FE-GA)
MPEG	MPEG-2 decoder
MuCCRA	Multi-Core Configurable Reconfigurable Architecture
MULT	Multiplier
NoC	Network-on-Chip
OS	Operating System
PAE	Processing Array Element
PC	Program Counter
PCI	Peripheral Component Interconnect
PE	Processing Element
RC	Reconfigurable Cell (Mophosys)
RCFU	Reconfigurable Functional Unit
RFU	Register File Unit
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SE	Switching Element
SIMD	Single Instruction Multiple Data
SMU	Shift and Mask Unit
SoC	System-on-Chip
STC	State Transition Controller
STG	State Transition Graph
Turbo	Turbo encoder
Viterbi	Viterbi decoder
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VMEM	Vertical Memory
XPP	eXtreme Processing Platform

Bibliography

[1] R.W. Hartenstein and H. Grunbacher. The Roadmap to Reconfigurable computing. In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pages 27–30, August 2000. 1

[2] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992. 1

[3] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999. 1, 48

[4] S.M. Arnold D. Buell and W.J. Kleinfelder. SPLASH 2: FPGAs in a Custom Computing Machine. *IEEE Computer Society Press*, 1996. 2

[5] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. SPACE 2 as a Reconfigurable Stream Processor. In *Proceedings of 4th Australasian Conference on Parallel and Real-time Systems (PART'97)*, pages 286–297, September 1997. 2

[6] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 12–21, April 1997. 2, 26

[7] ITRS. *International Technology Roadmap for Semiconductors 2007 Edition*, 2007. available at `http://www.itrs.net/`. 2

[8] Timothy Mark Pinkston and Jeonghee Shin. Trends Toward On-Chip Networked Microsystems. *International Journal of High Performance Computing and Networking*, 3(1):3–18, September 2005. 2

[9] J. Patterson and H. Agah. Synopsys and Xilinx Unveil Next Generation Flow for Platform FPGAs. *Xcell Journal Online*, 41, September 2001. 2

[10] Xilinx. *Spartan-3A FPGA Family: Data Sheet*, 2005. 2

[11] Xilinx. *Virtex-4 Family Overview*, 2005. 2

[12] Xilinx. *Virtex-5 Family Overview*, 2007. 2

[13] Altera. *Stratix IV Device handbook, Volume 1*, 2008. 2

[14] Hideharu Amano. A Survey on Dynamically Reconfigurable Processors. *IEICE Transactions on Communications*, E89-B(12):3179–3187, December 2006. 2, 48, 67, 86

[15] David Geer. Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13, May 2005. 3, 86

[16] V.M. Tuan and H. Amano. A Mapping Method for Multi-Process Execution on Dynamically Reconfigurable Processors. *IEICE Transactions on Information & Systems*, E91-D(9):2312–2322, September 2008. 5, 93

[17] V.M. Tuan and H. Amano. A Preemption Algorithm for a Multitasking Environment on Dynamically Reconfigurable Processors. *IEICE Transactions on Information & Systems*, E91-D(12), December 2008. 5

[18] Ivica Crnkovic. Software Engineering and Science. Presentation on Research Methodology for Natural Sciences and Technology, 2006. `http://www.idt.mdh.se/kurser/ct3340/ht07/se-methods2003.pdf`. 7

[19] S. Hauck and A. Dehon. *Reconfigurable Computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann Publishers, 2008. 9

[20] M. Suzuki, Y. Hasegawa, Y. Yamada, N. Kaneko, K. Deguchi, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi, and T. Awashima. Stream Applications on the Dynamically Reconfigurable Processor. In *Proceedings of International Conference on Field Programmable Technology (FPT 2004)*, pages 137–144, December 2004. 9, 61, 82

[21] O. Diessel, H. Elgindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proceedings on Computers and Digital Techniques*, 147(3):181–188, May 2000. 11, 88

[22] Grant Wigley and David Kearney. The management of applications for reconfigurable computing using an operating system. *Australian computer Science Communications*, 24:73–81, January 2002. 11

[23] H. Walder, C. Steiger, and M. Plazner. Fast online task placement on FPGAs: Free space partitioning and 2D-Hashing. In *Proceedings of Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 178–185, April 2003. 11

[24] M. Handa and R. Vemuri. Area fragmentation in reconfigurable operating systems. In *Proceedings of the International Conference Engineering of Reconfigurable Systems and Algorithms (ERSA'2004)*, pages 77–83, June 2004. 11

[25] Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–639, April 1998. 12

[26] H. Krupnova and G. Saucier. FPGA-based emulation: Industrial and custom prototyping silutions. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 68–77, August 2000. 12

[27] Altera Corp. Altera Stratix Device Handbook, July 2005. `http://www.altera.com`. 13

[28] W.J. Dally and B. Towles. Route Packets not Wires: On-Chip Interconnection Networks. In *Proceedings of Design Automation Conference*, pages 684–689, June 2001. 14, 88, 102

[29] J. S. N. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook. Dynamic reconfiguration to support concurrent applications. *IEEE Transactions on Computers*, 48:591–602, June 1999. 17, 68

[30] L. Levinson, R. Manner, M.Sesler, and H. Simmler. Preemptive Multitasking on FPGAs. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 301–302, April 2000. 17, 68

[31] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA Coprocessors. In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pages 121–130, January 2000. 17, 68

[32] H.K.H. So and R.W. Brodersen. Improving usability of fpga-based reconfigurable computers through operating system support. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL'06)*, pages 1–6, August 2006. 17

[33] M. Middendorf, B. Scheuermann, H. Schmeck, and H. Elgindy. An evolutionary approach to dynamic task scheduling on FPGAs with restricted buffer. *Journal of Parallel and Distributed Computing*, 62:1407–1420, September 2002. 19

[34] J. Teich, S. Fekete, and J. Schepers. Compile-time optimization of dynamic hardware reconfigurations. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTAĄE9)*, pages 1097–1103, June 1999. 19

[35] J. Teich, S. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The Journal of Supercomputing*, 19:57–75, May 2001. 19

[36] PACT Inc. *XPP-III Processor Overview*, version 2.0 edition, July 2006. 21, 29, 48

[37] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000. 22

[38] Guy Lemieux and David Lewis. *Design of interconnection networks for programmable logic*. Kluwer Academic Publishers, 2004. 22

[39] M. Kato, Y. Hasegawa, and H. Amano. Evaluation of MuCCRA-D: A Dynamically Reconfigurable Processor with Directly Interconnected PEs. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2008)*, pages 353–359, July 2008. 23, 44, 47, 48

[40] K. Compton, J. Cooley, S. Knol, and S. Hauck. Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):209–220, June 2002. 23, 88

[41] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM1997)*, pages 22–28, April 1997. 23

[42] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A First Generation DPGA Implementation. In *Proceedings of the 3rd Canadian Workshop on Field-Programmable Devices*, May 1995. 23

[43] X. P. Ling and H. Amano. WASMII: A Data Driven Computer on a Virtual Hardware. In *Proceedings of the 1st IEEE Symposium on FPGAs for Custom Computing Machines (FCCM1993)*, pages 33–42, April 1993. 25

[44] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Motomura. A Virtual Hardware System on a Dynamically Reconfigurable Logic Device. In *Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM2000)*, pages 295–296, April 2000. 25

[45] S. Goldstein, H. Schmit, M. Moe, M. Budiuy, S. Cadambi, R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA1999)*, pages 28–39, May 1999. 25

[46] J. Arnold, D. Buell, and E. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322, June 1992. 25

[47] J. Arnold and W. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society, 1996. 25

[48] K. Nakajima, H. Sato, H. Asami, M. Iida, K. Shindome, H. Mori, K. Takahashi, and Y. Mizukami. FPGA-based Parallel Machine : RASH. In *Proceedings of the 20th International Conference on Applied Informatics (AI2002)*, pages 269–273, Feburary 2002. 25

[49] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – Configurable Custom Computing. In *Proceedings of the 3rd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM1995)*, pages 32–38, April 1995. 25

[50] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, 33(4):62–69, April 2000. 26

[51] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *Proceedings 6th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM1998)*, pages 28–37, April 1998. 26

[52] P. M. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993. 26

[53] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, , and S. Ghosh. PRISM-II Compiler and Architecture. In *Proceedings of the 1st IEEE Workshop on FPGAs for Custom Computing Machines (FCCM1993)*, pages 9–16, April 1993. 26

[54] F. Raimbault, D. Lavenier, S. Rubini, and B. Pottier. Fine Grain Parallelism on a MIMD Machine Using FPGAs. In *Proceedings of the 1st IEEE Workshop on FPGAs for Custom Computing Machines (FCCM1993)*, pages 2–8, April 1993. 26

[55] R. Razdan and M. Smith. A High Performance Microarchitecture with Hardware Programmable Functional Units. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO-27)*, pages 172–180, 1994. 26

[56] R. Witting and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In *Proceedings of the 4th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM1996)*, pages 126–135, 1996. 26

[57] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimaera Reconfigurable Functional Unit. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM1997)*, pages 87–96, April 1997. 26

[58] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA2000)*, pages 225–235, 2000. 26

[59] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing. High-level Synthesis Challenges and Solutions for a Dynamically Reconfigurable Processor. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design (ICCAD2006)*, pages 702–708, November 2006. 26, 31, 55, 79

[60] K. Wakabayashi and T. Okamoto. C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1507–1522, December 2000. 26

[61] B. Salefski and L. Caglar. Re-Configurable Computing in Wireless. In *Proceedings of the 38th Design Automation Conference (DAC2001)*, pages 178–183, June 2001. 27, 48

[62] PACT Inc. *Video Decoding on XPP-III*, revision 1.1 edition, July 2006. 30

[63] M. Motomura. A Dynamically Reconfigurable Processor Architecture. In *Proceedings of Microprocessor Forum*, October 2002. 30, 48, 50, 53, 87, 88

[64] T. Sugawara, K. Ide, and T. Sato. Dynamically Reconfigurable Processor Implemented with IPFlex's DAPDNA Technology. *IEICE Transactions on Information & System*, E87-D(8):1997–2003, August 2004. 32, 48

[65] T. Sato. Dynamic Reconfiguration and Its Granularity inside Future DAPDNA Architecture. In *Proceedings of International Symposium on Advanced Reconfigurable Systems*, pages 114–127, December 2005. 34

[66] T. Sato. A Dual-Core Dynamically Reconfigurable Engine Employs 955 Parallel Processing Elements. In *Proceedings of Microprocessor Forum*, May 2007. 34, 48

[67] T. Komada, T. Tsunoda, M. Takeda, H. Tanaka, Y. Akita, M. Sato, and M. Ito. Flexibile Engine: A Dynamic Reconfigurable Accelerator with High Performance and Low Power Consumption. In *Proceedings of the 9th IEEE Symposium on Low-Power and High Speed Chips (COOL Chips IX)*, pages 393–408, April 2006. 34, 48, 88

[68] T. Stansfield. Using Multiplexers for Control and Data in D-Fabrix. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, pages 416–425, August 2003. 36, 48

[69] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A Reconfigurable Arithmeric Array for Multimedia Applications. In *Proceedings of the ACM/SIGDA 7th International Symposium on Field-Programmable Gate Arrays (FPGA1999)*, pages 135–143, Feburary 1999. 36

[70] T. Matsumoto, K. Kimura, H. Takano, T. Amatsubo, K. Mori, K. Senda, S. Inoue, and M. Matsui. Performance Evaluation of Reconfigurable Processing Array in Area Efficiency and Operating Frequency. In *Proceedings of the 9th IEEE Symposium on Low-Power and High Speed Chips (COOL Chips IX)*, pages 423–434, April 2006. 37

[71] T. Kiyohara. Multimedia Processor-based Platform for a Wide Range of Digital Consumer Electronics. In *Proceedings of the 8th IEEE Symposium on Low-Power and High Speed Chips (COOL Chips XIII)*, pages 133–141, April 2005. 37

[72] M. Nakajima, T. Yamamoto, M. Yamasaki, T. Hosoki, and M. Sumital. Low Power Techniques for Mobile Application SoCs Based on Integrated Platform UniPhier. In *Proceedings of the 12th Asia and South Pacific Design Automation Conference (ASPDAC2007)*, pages 649–653, January 2007. 37

[73] B. Levine. Kilocore: Scalable, High-Performance, and Power Efficient Coarse-grained Reconfigurable Fabrics. In *Proceedings of International Symposium on Advanced Reconfigurable Systems*, pages 129–158, December 2005. 38, 48

[74] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proceedings of the 24th IEEE Custom Integrated Circuits Conference (CICC2002)*, pages 63–66, May 2002. 38

[75] F. Veredas, M. Scheppler, W. Moffat, and B. Mei. Custom Implementation of the Coarse-Grained Reconfigurable ADRES Architecture for Multimedia Purposes. In *Proceedings of International Conference on Field Programmable Logic and Application (FPL'05)*, pages 106–111, August 2005. 39, 48, 53

[76] Stretch Inc. `http://www.stretchinc.com/_files/s6ArchitectureOverview.pdf`. 41, 48

[77] J. Arnold. S5: The Architecture and Development Flow of a Software Configurable Processor. In *Proceedings of the 4th IEEE International Conference on Field Programmable Technology (FPT2005)*, pages 120–128, December 2005. 42

[78] M. Saito, H. Fujisawa, N. Ujiie, and H. Yoshizawa. Cluster Architecture for Reconfigurable Signal Processing Engine for Wireless Communication. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'05)*, pages 353–359, August 2005. 43, 48

[79] H. Amano, Y. Hasegawa, S. Tsutsumi, T. Nakamura, T. Nishimura, V. Tanbunheng, A. Parimala, T. Sano, , and M. Kato. MuCCRA chips: Configurable dynamically-reconfigurable processors. In *Proceedings of Solid-State Circuits Conference (ASSCC'07)*, pages 384–387, November 2007. 44, 45, 48, 59

[80] Y. Saito, M. Kato, S. Saito, T. Sano, Hirai K, T. Nishimura, T. Nakamura, S. Tsutsumi, Y. Hasegawa, and H. Amano. Practice Evaluation Dynamically Reconfigurable Processor MuCCRA-2. In *IEICE Technical Reports, RECONF2008-34*, pages 69–74, September 2008. (In Japanese). 45, 48

[81] E. Ahmed and J. Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12, 2004. 48

[82] Y.Hasegawa, S.Abe, S.Kurotaki, V.Tuan, N.Katsura, T.Nakamura, T.Nishimura, and H.Amano. Performance and Power Analysis of Time-multiplexed Execution on Dynamically Reconfigurable Processor. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium/ Reconfigurable Architecture Workshop (IPDPS2006)*, April 2006. 50, 63, 64, 88

[83] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of IEEE*, 75(9):1235–1245, September 1987. 51

[84] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of IEEE*, 83(5):773–799, May 1995. 51

[85] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475. North-Holland Publishing Co., August 1974. 51

[86] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal in Computer Simulation, special issue on "Simulation Software Development"*, 4(2):155–182, April 1994. 51

[87] Ralf Niemann. *Hardware/Software co-design for data flow dominated embedded systems*. Kluwer Academic Publishers, 1998. 51

[88] M.D. Galanis, A. Milidonis, G. Theodoridis, D. Soudris, and C.E. Goutis. A partitioning methodology for accelerating applications in hybrid reconfigurable platforms. In *Proceedings of the conference on Design, Automation and Test in Europe*, volume 3, pages 247–252, March 2005. 51

[89] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. *IEEE Design and Test of Computers*, 17(1):68–83, March 2000. 51

[90] S.P.Fekete, E.Koehler, and J.Tech. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 658–665, March 2001. 51

[91] P-H.Yuh, C-L Yang, Y-W. Chang, and H-L Chen. Temporal Floorplanning Using 3D-subTCG. In *Proceedings of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair*, pages 725–730, January 2004. 51

[92] T. Sugawara, K. Ide, and T. Sato. Dynamically Reconfigurable Processor Implemented with IPFlex's DAPDNA Technology. *IEICE Transactions on Information & Systems*, E87-D(8):1997–2003, May 2004. 53

[93] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21:35–46, March/April 2001. 53

[94] Independent JPEG Group. http://www.ijg.org/. 61, 82

[95] C. Berrou and A. Glavieux. Near optimum error correcting coding and decoding: Turbo codes. *IEEE Transactions Communications*, 44:1261–1271, October 1996. 61, 82

[96] MPEG Software Simulation Group (MSSG). http://www.mpeg.org/MPEG/MSSG. 61, 82

[97] J. Noguera and R.M. Badia. Multitasking on reconfigurable architectures:Microarchitecture support and dynamic scheduling. *ACM Transactions on Embedded Computing Systems*, 3(2):385–406, May 2004. 67

[98] G. Brebner. The Swappable Logic Unit: A Paradigm for Virtual Hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 77–86, April 1997. 68

[99] D. Koch, C. Haubelt, and J. Teich. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *Proceedings of International Symposium on Field-Programmable Gate Arrays*, pages 188–196, February 2007. 68

[100] S. A. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. In *Proceedings of Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, pages 301–302, September 1999. 68

[101] H. Kalte and M. Porrmann. Context Saving and Restoring for Multitasking in Reconfigurable Systems. In *Proceedings of 15th International Conference on Field-Programmable Logic and Applications*, pages 223–228, August 2005. 68

[102] S. Jovanovic, C. Tanougast, and S. Weber. A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 358–364, August 2007. 68

[103] M. Ullmann, M. Hubner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Proceedings of the18th International Parallel and Distributed Processing Symposium (IPDPSAf04) - Reconfigurable Architectures Workshop*, pages 135–142, April 2004. 68, 87, 88

[104] K. Kim, R. Karri, and M. Potkonjak. Micropreemption Synthesis: An Enabling Mechanism for Multi-task VLSI Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:19–30, January 2006. 68

[105] J. Simonson and J.H. Patel. Use of Preferred Preemption Points in Cache-Based Real-Time Systems. In *Proceedings of 15th International Computer Performance and Dependability Symposium*, pages 316–325, April 1995. 70

[106] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, April 2003. 70, 88

[107] E. A. Lee and D. C. Messerschmitt. Static Scheduling of Synchronous Data flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–36, January 1987. 72

[108] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler: Principles, Techniques, and Tools*. Addison Wesley, 1986. 76

[109] V.Sreedhar, G.R. Gao, and Y. Lee. Identifying Loops Using DJ Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6):649–658, November 1996. 76

[110] Steven A. Guccione. Multicore Devices: A New Generation of Reconfigurable Architectures. In *Proceedings of the 8rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'2008)*, pages 3–11, July 2008. 86

[111] Freescale Semiconductor. *Multi-Core Microprocessors in Embedded Applications*, 2005. White Paper. 86

[112] Y. Kurose, I. Kumata, M. Okabe, H. Hanaki, K. Seno, K. Hasegawa, H. Ozawa, S. Horiike, T. Wada, S. Arima, K. Taniguchi, K. Ono, H. Hokazono, T. Hiroi, T. Hirano, and S. Takashima. A 90nm Embedded DRAM Single Chip LSI with a 3D Graphics, H.264 Codec Engine, and a Reconfigurable Processor. In *Proc. of the 16th Symp. on High Performance Chips (Hot Chips 16)*, August 2004. 86

[113] Toshiba. *A Dynamically Reconfigurable Architecture for Stream Processing*, 2007. Tutorial at International Conference on Field-Programmable Technology 2007 (ICFPT'07). 86, 88

[114] Sanyo. *Reconfigurable Architecture for Car Tuners*, 2007. Tutorial at International Conference on Field-Programmable Technology 2007 (ICFPT'07). 86, 88

[115] P. Merino, J.C. Lopez, and M. Jacome. A Hardware Operating System for Dynamic Reconfiguration of FPGAs. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL'98)*, pages 431–435, August 1998. 87

[116] P. Merino, M. Jacome, and J.C. Lopez. A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 324–325, April 1998. 87

[117] A. Ahmadinia, C. Bobda, and J. Teich. Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware. In *Architecture of Computing Systems (ARCS)*, pages 125–139, April 2004. 87

[118] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs. In *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications (FPL'02)*, pages 795–805, September 2002. 87, 88, 102

[119] Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'2003)*, pages 284–287, June 2003. 87, 88

[120] Gordon Brebner and Oliver Diessel. Chip-based Reconfigurable Task Management. In *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL'01)*, pages 182–191, August 2001. 87, 88

[121] H. Walder and M. Platzner. Non-preemptive Multitasking on FPGA: Task Placement and Footprint Transform. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2002)*, pages 24–30, June 2002. 87, 88

[122] H. Kalte, M. Koester, B. Kettelhoit, M. Porrmann, and U. Ruckert. A comparative study on system approaches for partially reconfigurable architectures. In *Proceedings of the 4rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'2004)*, pages 70–76, June 2004. 87

[123] Luca Benini and Giovanni De Micheli. *Networks on chips: technology and tools*. Morgan Kaufmann Publishers, 2006. 87, 102

[124] T. Marescaux, J-Y. Migmolet, A. Bartic, W. Moffa1, D. Verkest, S. Vemalde, and RLauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable System. In *Proceedings of the 13th International Workshop on Field-Programmable Logic and Applications (FPL'03)*, pages 256–259, September 2003. 88, 102

[125] Y. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins. Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2002)*, pages 116–122, June 2002. 88

[126] B. Black, D.W. Nelson, C. Webb, and N. Samra. 3D Processing Technology and Its Impact on iA32 Microprocessors. In *Proceedings of the International Conference on Computer Design*, pages 316–318, October 2004. 102

[127] S. Das, A. Fan, K.N. Chen, C.S. Tan, N. Checka, and R. Reif. Technology, Performance, and Computer-Aided Design of Three-Dimensional Integrated Circuits. In *Proceedings of the International Symposium on Physical Design*, pages 108–115, April 2004. 102

[128] W.R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A.M. Sule, M. Steer, and P.D. Franzon. Demystifying 3D ICs: The Pros and Cons of Going Vertical. *IEEE Design and Test of Computers*, 22(6):498–510, November 2005. 102

[129] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir. Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. In *Proceedings of the International Symposium on Computer Architecture*, pages 130–141, June 2006. 102

Publications

Journal Papers

[1] Vu Manh Tuan, and Hideharu Amano, "A Mapping Method for Multi-process Execution on Dynamically Reconfigurable Processors", *IEICE Transaction on Information & Systems*, Vol. E91-D, No.9, pages 2312-2322, September 2008.

[2] Vu Manh Tuan, and Hideharu Amano, "A Preemption Algorithm for a Multitasking Environment on Dynamically Reconfigurable Processors", *IEICE Transaction on Information & Systems*, Vol. E91-D, No.12, pages 2793-2803, December 2008.

[3] Yohei Hasegawa, Shohei Abe, Syunsuke Kurotaki, Vu Manh Tuan, and Hideharu Amano, "Evaluation of Time-multiplexed Execution on the Dynamically Reconfigurable Processor", *IPSJ Transaction on Advanced Computing Systems (ACS)*, Vol. 47, No. SIG12 (ACS15), pages 171-181, September 2006. (In Japanese)

[4] Masayasu Suzuki, Yohei Hasegawa, Shohei Abe, Vu Manh Tuan, and Hideharu Amano, "A Novel Cost-Effective Context Memory Structure for Dynamically Reconfigurable Processors", *IEICE Transaction on Information & Systems*, Vol. J89-D, No. 6, pages 1101-1109, June 2006. (In Japanese)

International Conference Papers

[5] Vu Manh Tuan, Hiroki Matsutani, Naohiro Katsura, and Hideharu Amano, "Evaluation of a Multicore Reconfigurable Architecture with Variable Core Sizes", *In Proceeding of the 23th International Parallel and Distributed Processing Symposium (IPDPS 2009) / Reconfigurable Architectures Workshop (RAW2009)*, May 2009 (accepted).

[6] Vu Manh Tuan, and Hideharu Amano, "A Method for Capturing State Data on Dynamically Reconfigurable Processors", *In Proceeding of 2008 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 208-214, Las Vegas, July 2008.

[7] Vu Manh Tuan, and Hideharu Amano, "A Mapping Method for Multi-Processing Execution on Dynamically Reconfigurable Processors", *In Proceeding of 2007 International Conference on Field-Programmable Technology (ICFPT2007)*, 357-360, Kitakyushu, December 2007.

[8] Vu Manh Tuan, Yohei Hasegawa, and Hideharu Amano, "Performance Analysis of Multi-Process Execution Model on Dynamically Reconfigurable Processor", *In Proceeding of the 7th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA2007)*, pages 203-206, Las Vegas, NV, June 2007

[9] Yohei Hasegawa, Shohei Abe, Shunsuke Kurotaki, Vu Manh Tuan, Naohiro Katsura, Takuro Nakamura, Takashi Nishimura, and Hideharu Amano, "Performance and Power Analysis of Time-multiplexed Execution on Dynamically Reconfigurable Processor", *In Proceeding of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006) / Reconfigurable Architectures Workshop (RAW2006)*, Rhodes Island, Greece, April 2006.

[10] Masayasu Suzuki, Yohei Hasegawa, Vu Manh Tuan, Shohei Abe, and Hideharu Amano, "A Cost Effective Context Memory Structure for Dynamically Reconfigurable Processors", *In Proceeding of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006) / Reconfigurable Architectures Workshop (RAW2006)*, Rhodes Island, Greece, April 2006.

[11] Vu Manh Tuan, Yohei Hasegawa, Naohiro Katsura, and Hideharu Amano, "Performance-Cost Trade-off Evaluation for the DCT Implementation on the Dynamically Reconfigurable Processor", *In Proceeding of International Workshop on Applied Reconfigurable Computing (ARC2006)*, pages 115-121, Delft, Netherlands, March 2006.

[12] Yohei Hasegawa, Shohei Abe, Shunsuke Kurotaki, Vu Manh Tuan, Naohiro Katsura, Takuro Nakamura, Takashi Nishimura, and Hideharu Amano, "Application-Based Performance and Power Analysis of Dynamically Reconfigurable Processor", *International Symposium on Advanced Reconfigurable Systems*, Kyoto, Japan, December 2005.

Domestic Conference Papers and Technical Reports

[13] Vu Manh Tuan, Hiroki Matsutani, Naohiro Katsura, and Hideharu Amano, "Evaluation of a Multicore Reconfigurable Architecture", *IEICE Technical Reports, VLD2008-92*, pages 7-12, January 2009.

[14] Vu Manh Tuan, and Hideharu Amano, "A Method for Saving and Restoring Context Data of Hardware Tasks on the Dynamically Reconfigurable Processor", *IEICE Technical Reports, RECONF2008*, pages 71-76, January 2008.

[15] Yohei Hasegawa, Shohei Abe, Syunsuke Kurotaki, Vu Manh Tuan, and Hideharu Amano, "Evaluation of Time-multiplexed Execution on the Dynamically Reconfigurable Processor", *The 4th Symp. on Advanced Computing Systems and Infrastructures (SACSIS2006)*, pages 135-142, May 2006. (In Japanese)

[16] Yohei Hasegawa, Takashi Nishimura, Shohei Abe, Syunsuke Kurotaki, Vu Manh Tuan, and

Hideharu Amano, "Application-Based Performance and Power Analysis on Dynamically Reconfigurable Processor", *The 27th Parthenon Symposium*, pages 3-10, December 2005. (In Japanese)

[17] Yohei Hasegawa, Hideharu Amano, Shohei Abe, Syunsuke Kurotaki, and Vu Manh Tuan, "Performance and Power Analysis of Time-multiplexed Execution on Dynamically Reconfigurable Processor", *IEICE Technical Reports, RECONF2005-35*, pages 31-36, September 2005. (In Japanese)

[18] Naohiro Katsura, Yohei Hasegawa, Vu Manh Tuan, Hiroki Matsutani, and Hideharu Amano, "Performance Evaluation of Multi-core DRP for Stream Application", *IEICE Technical Reports, RECONF2006-52*, pages 49-54, November 2006. (In Japanese)

[19] Vu Manh Tuan, Yohei Hasegawa, Naohiro Katsura, and Hideharu Amano. Performance Evaluation of Hardware Multi-process Execution on the Dynamically Reconfigurable Processor, *IEICE Technical Reports, RECONF2006-31*, pages 25-30, September 2006.

[20] Naohiro Katsura, Yohei Hasegawa, Vu Manh Tuan, and Hideharu Amano, "Implementation of Stream Application on Programmable Devices by C Level Design", *IEICE Technical Reports, RECONF2005-82*, pages 31-36, January 2006. (In Japanese)