Reconfigurable Architectures: from Physical Implementation to Dynamic Behaviour Modelling

Kehuai Wu

Kongens Lyngby 2007
IMM-PHD-2007-180

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

This dissertation focuses on the dynamic behavior of the reconfigurable architectures. We start with a survey of the existing work with the aim of categorizing the current research and identifying the future trends. The survey discusses the design issues of the reconfigurable architectures, the run-time management strategies and the design methodologies.

The second part of our work focuses on the study of commercial FPGAs that support the dynamic partial reconfiguration. This work grants us a better understanding of the limit and the potential of the main-stream commercial FPGA, and justifies the necessity of employing more advanced technologies in order to enable the realization of highly efficient reconfigurable architectures.

The third part of our study is carried out on ADRES, a coarse-grained datapath-coupled reconfigurable architecture. The study on ADRES shows that Multi-threading not only is feasible for reconfigurable architectures, but greatly improves the architecture scalability as well.

Our concluding study proposes a simulation framework for coprocessor-coupled reconfigurable architectures, namely COSMOS. The COSMOS simulation framework comprises a generic application model and an architecture model, the combination of which captures the dynamic behavior of the reconfigurable architectures. Our framework is a tool for studying the run-time management strategies and for experimenting the design space exploration of the reconfigurable architectures, and offers a means of evaluating various other works on a common ground.

Resumé

Denne afhandling omhandler de dynamiske egenskaber ved rekongifurerbare arkitekturer. Først er der foretaget en undersøgelse af eksisterende forskning for at kategorisere denne og identificere tendenser for fremtiden. Undersøgelsen diskuterer designforhold for rekongifurerbare arkitekturer, run-time håndteringsstrategier og designmetoder.

Anden del af afhandlingen fokuserer på studier af kommercielle FPGAer, der understøtter dynamisk partiel rekongifurering. Dette danner baggrund for dybere forståelse af begrænsninger og muligheder ved generelle kommercielle FPGAer og understreger samtidig nødvendigheden for at anvende mere avancerede teknologier for at gøre realisering af effektive rekongifurerbare arkitekturer mulig.

Den tredje del omhandler arbejdet med ADRES, en grovmasket datavejskoblet rekongifurerbar arkitektur. Studierne af ADRES viser at fler-trådet ikke blot er muligt for rekongifurerbare arkitekturer, men også forbedrer arkitekturens skalerbarhed markant.

De konkluderende studier fremviser COSMOS, et simuleringsmiljø for coprocessorkoblede rekongifurerbare arkitekturer. COSMOS simuleringsmiljøet omfatter en generel applikationsmodel og en arkitekturmodel, som tilsammen modellerer de dynamiske egenskaber ved rekongifurerbare arkitekturer. Miljøet er et værktøj til studier af run-time håndteringsstrategier og eksperimentering med udforskning af de mulige designløsninger af rekongifurerbare arkitekturer, og muliggør sammenligning af forskellige løsninger under de samme forudsætninger.

Preface

This thesis was prepared at the institute of Informatics and Mathematical Modelling, at the Technical University of Denmark, in partial fulfillment of the requirements for acquiring the Ph.D. degree. The Ph.D. study was supervised by Professor Jan Madsen.

The study of the reconfigurable architecture has been a puzzling journey. One can immerse himself in the world of infinite possibilities and keep wondering what message is crucial to deliver to the others. My belief is that, at this moment, we should focus more on the run-time management study. With the goal of easing other's work, the COSMOS framework is presented. As fundamental as it is generic, the COSMOS model captures the essence of dynamic reconfiguration and suggests a realistic view on the all-too-complicated reconfigurable architectures.

The thesis consists of a summary report and a collection of chapters based on 3 research papers written during the period 2005–2007, and elsewhere published.

Lyngby, May 2007

Kehuai Wu

Papers contributed to the thesis

1 Kehuai Wu and Jan Madsen. ***Run-time Dynamic Reconfiguration: A Reality Check Based on FPGA Architectures from Xilinx*** Norchip Conference 2005. Published

2 Kehuai Wu, Andreas Kanstein, Jan Madsen and Mladen Berekovic ***MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture*** International Workshop on Applied Reconfigurable Computing 2007. Published

3 Kehuai Wu and Jan Madsen. ***COSMOS: A System-Level Modelling and Simulation Framework for Coprocessor-Coupled Reconfigurable Systems*** SAMOS VII: International Symposium on Systems, Architectures, Modelling and Simulation 2007. Published.

4 Kehuai Wu, Esben Rosenlund, and Jan Madsen. ***Towards Understanding the Emerging Critical Issues from the Dynamic Behavior of Run-Time Reconfigurable Architectures*** International Conference on Codesign and System Synthesis 2007. Submitted.

5 Kehuai Wu, Andreas Kanstein, Jan Madsen and Mladen Berekovic ***MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture*-extended version** International Journal of Electronics 2007. Accepted

Acknowledgements

Professor Jan Madsen has been a continuous inspiration throughout my Ph.D. study. He sets a great model for me with his generosity, commitment and enthusiasm.

Knowing Andreas Kanstein from Freescale is an important event in my life. I appreciate him being there for me during the hard time.

Mladen Berekovic and Frank Bouwens made my stay in Belgium a wonderful experience. There had never been a dull moment at IMEC, Leuven, thanks to them.

Flemming Stassen has done me so many kindnesses. Living in a foreign country has been made so much easier with his help.

I own my gratitude to so many people, especially the ones working in the SoC group as I have been. ARTIST2 gave me the financial support during my more productive time, and made some of my publication possible.

Knowing that my parents love me gives me strength. They helped me countless times, and gave me more than anyone can believe.

My wife, Xia, is my home, my shepherd and my hope. Without her, nothing will be the same.

Kehuai Wu
May 2007

Contents

Chapter 1

Introduction

1.1 Reconfigurable architectures in a nutshell

Traditional computer architectures mainly takes two approaches to execute an application. The first one is to employ a programmable microprocessor. The processor-based architectures usually support an instruction-set that covers a wide range of logic and memory operations, thus can execute various applications when the application is proper compiled. However, due to the usually limited amount of parallel computing resource, the large execution overhead and the memory bottleneck, these architectures are inefficient in terms of performance and energy.

The second approach is to tailor a hardware for a specific application, so that the application can be executed at the highest affordable speed. The resulting Application-Specific Integrated Circuit (ASIC) is usually dedicated to one application, or even only one configuration of a certain application, thus exceedingly lacks the flexibility. ASIC design also has a long development cycle, thus the consequence of having fabrication fault or design errors is more severe than the software-based application design. The lack of flexibility substantially increase the verification and test phase of the implementation.

In hopes of closing the gap between the processor-based architectures and the

ASIC, reconfigurable architectures come into play. The mainstream of the reconfigurable architectures is composed of a fixed logic part and a reconfigurable part. The fixed logic usually includes a programmable processor that executes the non-crucial parts of the application and controls the reconfigurable unit. The reconfigurable unit is a high-performance field-programmable logic frequently used to accelerate the execution of the application kernels.

From the architecture composition, we see that both the fixed part and the reconfigurable part are programmable, thus the programmability of the processor-based architecture is retained. The reconfigurable unit can even extend the instruction set of its fixed counterpart, therefore makes the programmability of the architecture even stronger. The reconfigurable unit usually is a scalable gate array with high amount of parallel computation resource. This gives the reconfigurable unit a potential performance advantage over the processors. Due to the nature of the field-programmable logic, the reconfigurable unit is usually not as efficient as the ASIC implementation in terms of power and speed, but the flexibility compensates for it.

1.2 The origin, and the revival

In year 1960, Gerald Estrin at the University of California at Los Angeles proposed a computer architecture that is very different from the main stream research [33] at that time. As shown in his original figure (figure 1.1), this computing machine is a combination of a Fixed (F) computing unit and a Variable (V) unit. The fixed part of the machine offers the user a consistent and friendly interface, while the variable unit of the system performs specific task as user requests. The variable part of the system offers a performance that is as high as dedicated hardware, and it can reconfigure itself to fit the user's application. This is the first time that the reconfigurable computing concept has been openly discussed. However, due to the lack of the technology support, this concept was not adopted well during 60's, and the microprocessor-ASIC combo have dominated both the industry and the academic research in the next few decades.

However, the advance of the silicon technology leads to many new digital system design strategies and trends. The appearance of the Complex Programmable Logic Device (CPLD) and the Field-Programmable Gate Array (FPGA), which are mostly used for implementing simple digital circuit and prototyping larger digital systems, respectively, gives very solid technical backbones to the reconfigurable computing. Through these devices, we have acquired a preliminary understanding of the configurability, the performance, the programmability and the application domain of the reconfigurable architectures.

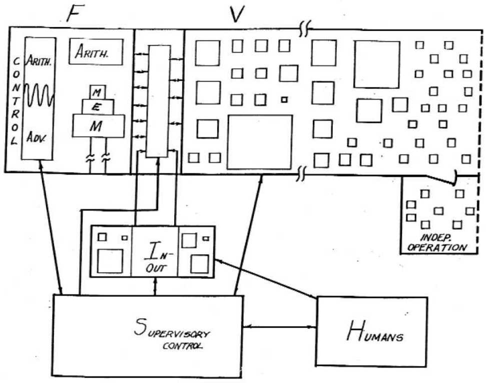


Figure 1.1: One of the earliest proposal of reconfigurable computing architecture [33]

In recent years, the chip fabrication cost and the non-recurrent engineering cost has increased to a level where the non-reusable custom ASIC design is hardly affordable for smaller business. Since the chip reusability becomes an important issue, FPGA is not only used as a prototyping device, but also a solution as part of the final product. Even though there is always a performance margin between the ASIC and the FPGA, thanks to the demand of the digital design community, this margin has been shrinking in the last few years, and even battery-powered FPGA-based designs are emerging. This trend suggests that the FPGA is the most appealing base for reconfigurable architecture study, and people shouldn't expect anything less from the reconfigurable architectures than from the FPGA in terms of performance.

Some other recent technologies contributed to the study of the reconfigurable architectures. The development of the intellectual property (IP) core is a promising strategy for increasing the design reusability. One of the byproducts of the IP core reusability study is the hardware run-time adaptivity, which is one of the enabling technologies of the dynamically reconfigurable architecture. Also, the high-level synthesis, especially the finite-state-machine based and loop-level optimization based synthesis, fits naturally to the programmability study of the reconfigurable architectures.

The technology scaling is driving the computer architecture research into the era of Multi-Processor System-on-Chip (MPSoC). Multiple cores interconnected with an on-chip network (NoC) is one of the most interesting on-going architecture research due to its potential for increasing design scalability and performance. When the MPSoC paradigm is applied on the reconfigurable architectures, the reconfigurable architecture is benefited from higher flexibility, better scalability and the better usage of coarse-grained parallelism.

Future embedded systems will be based on platforms which allow the system to be extended and incrementally updated while running in the field. This will not only extend the life time of the system, but also allow the system to adapt to the physical environment as well as performing self-repair, hence increasing the reliability and robustness of the system. Dynamically reconfigurable architectures is the most suitable technology for facilitating this, and is being studied in the research of self-evolving embedded system that can adapt to the environment and fault tolerable system that needs long lifecycle in the field.

To conclude, the reconfigurable architecture research at the current stage is of utmost importance and relevance. The study in this area has solid technology foundation from the previous work, and the issues to be addressed are tightly intertwined with many other crucial research areas. Investigating and understanding the reconfigurable architecture not only push the reconfigurable architecture study forward, but encourage the related fields of research as well.

1.3 Industry practice

Being the flagship of commercial FPGA developers, Xilinx [9] contributed significantly to the physical implementation of the reconfigurable device. Their VIRTEX FPGAs can partially reprogram themselves during run-time, hence are realistic platforms for studying the online adaptive systems. Their existing configuration development tool chain has been augmented for supporting the generation of partial configuration, and the on-chip configuration device is given a higher bandwidth to achieve faster reconfiguration. At the current stage, the killer application for such a system is yet to be found, and their partial reconfiguration approach is mostly used for academic experimentations.

There exists some off-the-shelf reconfigurable architectures. The XD1 [5] supercomputers from Cray, the MAP [8] processor from SRC and various systems from Nallatech [6] etc. tries to get the most out of the commercial FPGA by coupling them to other control modules. These systems are more focused on offering parallel computation power than frequent reconfiguration, and mainly attempt to offer user-friendly interface to the programmers. Many of them also focus on employing an array of FPGAs in order to improve the system performance even further.

Other in-the-lab commercial examples are known. The Chameleon System

Inc[96][1] proposed a single-chip solution, cs2000 architecture, to push the use of much more advanced reconfiguration strategies on commercial FPGAs. Their architecture development is discontinued from its infancy, but it still inspired many academic researches. The Silicon Hive [7] approached the reconfigurable system development from the IP development and programmability study. Celoxica [4] proposed to use a c-like programming language, Handle-C, to address the programmability issue of the reconfigurable architectures. There are many other new technologies used in some context, but a highly integrated tool flow or a dedicated architecture is yet to be seen.

In general, the commercial reconfigurable architecture is centered on the off-the-shelf FPGAs. The technologies being studied and put to practice is mostly computation-oriented rather than reconfiguration-oriented. The study on programmability and architecture is still premature, and the run-time management is not being recognized as a critical issue. The lack of highly automated tool support and the lack of better understanding of the application domain is hindering the acknowledgement of the reconfigurable architecture, and in turn, results in the miscarriage of many great technologies' commercial breakthrough.

1.4 State-of-the-art academic research

On the contrary, academic researches on reconfigurable architecture has witnessed an outburst of new ideas. The architecture study leads to the demand of the better understanding of the logic granularity issue, the architecture coupling issue and the configuration strategy issue etc. The programmability study leads to the demand of highly automated and efficient high-level synthesis, mapping, partition tools etc. The behavior study of the reconfigurable system leads to the need of highly complicated run-time management system design, which is closely related to the architecture design and application mapping strategy.

In the last couple of decades, logic units of various granularities have been proposed and evaluated. The logic granularity is the measure of how precise the configuration data can describe the function of the logic unit. The impact of the granularity variance has been studied to a great extend in terms of memory requirements, performance and programmability etc.

The coupling between the reconfigurable unit and the fixed part is also a complicated issue. Commercial solutions are usually multi-chip systems, where the fixed part and the reconfigurable part are not on the same chip, thus the coupling between these two parts is always very loose. Tighter coupling enables

[1] Not in business since 2001

much faster communication between the fixed logic and the reconfigurable unit, therefore results in much more interesting system behaviors and increases the occurrence of reconfiguration. The coupling has great impact on the architecture scalability, the data communication efficiency and the reconfigurability.

The online configuration strategy has also lead to many discussions. Whether a reconfigurable unit should be shared by multiple tasks in time or in space, and how to share the reconfigurable unit between tasks are open for further experimentation. These topics further lead to the study of the multi-context FPGA, inter-task communication and configuration memory hierarchy.

Programmability of reconfigurable architectures is another interesting topic. The reconfigurable architectures need the application to be partitioned into two parts, one being executed on the reconfigurable units, and the other being executed on the fixed part. The part of the application being executed on the reconfigurable unit needs to optimally use the reconfigurable unit, which offers parallel computing resources. To program for the reconfigurable architectures, we either need a high-level synthesis tool that integrates the partitioning tool, synthesis tool and compilation tool in one environment, or we need a development kit to carry out the software programming, hardware modelling and the interfacing at the same time. No matter which approach we take, the performance of the architecture is determined by how well we can explore the parallelism in the application from data level to task level, which is not a trivial task.

The dynamic reconfiguration is a costly operation. How frequently should a reconfigurable unit be reconfigured, and how should it be reconfigured needs to be decided at run-time, thus the run-time management system is another challenging design issue. For a large-scaled reconfigurable system that supports multi-tasking, the reconfigurable part of the system is a critical computing resource, and efficiently sharing it among several tasks is another challenge.

At current stage, the architecture design of the reconfigurable architecture is studied by many. The issues in the physical design is rather well-understood, and moving towards realization of dedicated reconfigurable device is not posing any prohibiting technical difficulties. The programmability of the reconfigurable architectures is still being discussed, and many tools and methodologies have been proposed. Due to the variety of reconfigurable systems, the study on tool chains are hard to converge, and most solutions take ad hoc approaches based on the architectures. The run-time system design is in a similar status as the programmability study is in. The complexity and variety of reconfigurable systems make it difficult to capture and generalize the run-time behavior of the reconfigurable system, thus makes it hard to develop a run-time system and assess its efficiency. The design verification and testing has been discussed by a few, but moving into verification is still not a concern for most people.

1.5 Thesis Outline

In our study, we would like to acquire a thorough understanding of the reconfiguration before we decide what issues are important at the current stage, and what possible technical support is available to build future technologies upon. We took the bottom-up approach to understand the reconfigurable architectures, thus our study went through the following four phases.

A survey of the reconfigurable architecture has been carried out in the first phase, and the findings of our study are documented in chapter 2. We noticed that the coupling between the fixed logic and the reconfigurable logic has huge impact on the architecture scalability, configurability and programmability etc, and therefore dedicated most of the rest of the study to investigate this issue.

Then we moved on to the study of the commercial FPGA, and experimented on the partial reconfiguration design flow supported by the Xilinx Virtex FPGAs. The objective of this study is to get a general understanding of what state-of-the-art commercial reconfigurable devices can achieve, what technologies are mature and feasible in practice, what technologies not being put to practice are actually feasible and crucial, and most importantly, what unconventional physical characteristics reconfigurable systems have. During our experimentation, we noticed that several limitations exist in the current Xilinx tool flow as well as in the architecture, and documented them in chapter 3. We also described what urgent issues need to be addressed to make the current Virtex FPGA a more suitable platform for building more complicated reconfigurable architectures.

To acquire a better understanding of the datapath-coupled reconfigurable architectures, we carried out some study on the state-of-the-art ADRES architecture developed at IMEC, Belgium, and extended it to support the simultaneous multi-threading (SMT). Our approach and conclusions are documented in chapter 4 of this dissertation. From this exercise, we gained the knowledge of the design pitfalls of datapath-coupled architectures, and proved that the threading is a feasible and important solution for improving the performance and scalability of these architectures.

After carrying out many studies in various areas, we are convinced that the coprocessor-coupled architectures have great potential, but there hasn't been enough investigation on many critical issues of these architectures yet. We propose our general system-level simulation framework, COSMOS, for further study on coprocessor-coupled reconfigurable architectures. We demonstrate how the COSMOS model can be used for acquiring a better understanding of the reconfigurable architectures' dynamic behavior, and for evaluating the performance of a reconfigurable system. The result is documented in chapter 5.

Chapter 6 concludes our work and discusses the perspectives of the reconfigurable architecture research.

Chapter 2

Survey of the Dynamically Reconfigurable Systems

In the last two decades, large number of reconfigurable architectures have been proposed, along with many new technologies. In general, the state-of-the-art reconfigurable systems still resemble the F+V system proposed by Gerald Estrin decades ago. The variable part of the reconfigurable architecture does the arithmetic computation to speed up the execution of user programs, while the fixed part offers a consistent programming interface to the programmer and supervises the use of the variable part at run-time. Quite a few architectures have shown great potential in accelerating user applications and improving energy-efficiency.

The architecture design in this area is getting mature, and many new technologies have been proven feasible. However, people started to notice that the dynamic behavior of the reconfigurable system is very different from that of the traditional architectures. Improving the run-time system efficiency, the reconfigurability and the programmability of the reconfigurable architectures are bigger challenges than the architecture design. Recently, the lack of integrated high-level compilation tools and efficient run-time systems starts to restrain the dissemination of reconfigurable system, so the focus of the mainstream research is currently moving towards these areas to provide the missing pieces.

Before we can understand where the current research trends are going and pinpoint what critical challenges lie ahead, we want to understand what has been done, or proved, by others and what is the state-of-the-art. We start our study by surveying the research activity in the last couple of decades, and documenting our observation in the next four sections. The first section gives a general overview of the architecture proposed in recent research, and discusses the new technologies being used. The second section focuses on the reconfiguration strategies, and discusses their impact on system-level design. The third section discusses the known run-time system design issues and some proposed strategies to address them. The fourth section discusses the methodology design issue currently under study and some general direction being taken to approach them. In the final section of this chapter, we conclude how the state-of-the-art motivates us to continue, and what is most relevant in the near future for us.

2.1 Architecture

The deterministic architecture design issues of reconfigurable systems are the host-reconfigurable unit coupling and the reconfigurable logic block granularity. Also, several recently proposed FPGA technologies contribute to the architecture design. In the last few years, all these three areas have evolved rapidly, and a classification is necessary.

2.1.1 Reconfigurable unit coupling

The reconfigurable unit (RU) can be coupled into the host architecture in four ways, as shown in figure 2.1 [23]. It can be a reconfigurable functional unit (FU) built into the datapath, a coprocessor, an attached reconfigurable unit or an external stand-alone processing unit. The coupling method has deterministic impact on the operating system design and the methodologies, and is the most important decision a designer has to make.

2.1.1.1 Datapath-coupled reconfigurable architectures

The reconfigurable units can be embedded into the datapath of a processor as a special function unit. Most of the known architectures in this class are similar to RISC processors or Very-Long-Instruction-Word (VLIW) architectures. But unlike VLIW architectures, this class of Reconfigurable architectures demands complicated compiler design due to the flexible instruction set. During

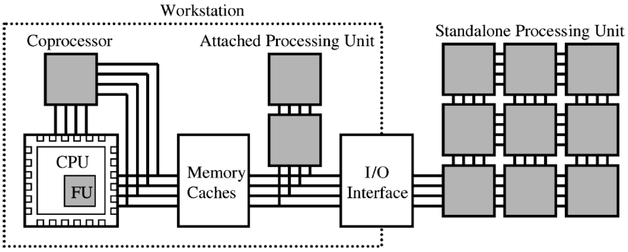
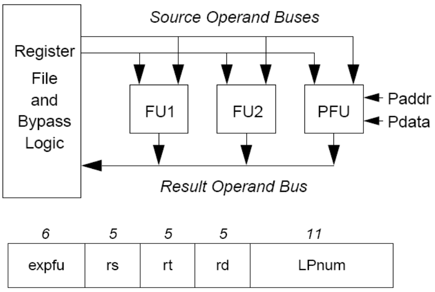


Figure 2.1: Different coupling methods of reconfigurable units [23]

compile time, the complicated and regular arithmetic operations are identified and extracted by the programmer's guidance[58] or by profiling[85]. If executing these operations on an RU is beneficial, a special RU-operation instruction with reconfiguration op-code extension will be generated from the compiler, and the extracted arithmetic operation will be synthesized into an RU configuration bitstream with dedicated synthesis tool. At run time, several RU operation bitstreams are stored on the RU, and the dedicated op-code extension bits selectively switch the bitstream enabled on the RU when an RU operation needs to be executed.

PRISC is an early example of such systems[85]. The datapath and the instruction format of PRISC are shown in figure 2.2. In the PRISC instruction format, the op-code *expfu* indicates whether the current instruction invokes RU, which is called PFU in the figure. The field *LPnum* indicates which pre-synthesized configuration should be loaded onto the PFU. The PRISC PFU is more flexible and efficient than a normal functional unit, but the size of it is still rather small (30.5K transistors).

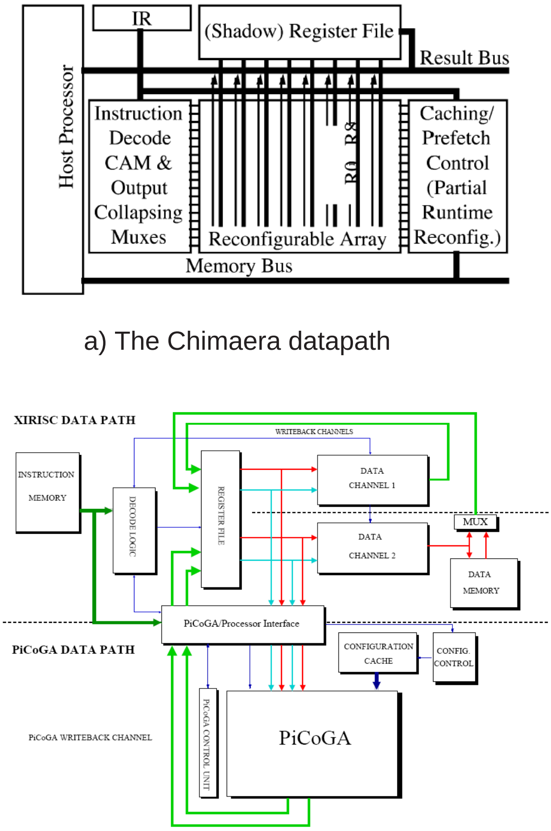


6	5	5	5	11
expfu	rs	rt	rd	LPnum

Figure 2.2: PRISC datapath and instruction[85]

The Chimaera architecture [47] and XIRISC[64][65] architecture shown in figure

2.3 are a couple of examples of more recent datapath-coupled architectures. The Chimaera system has an array of reconfigurable columns, several of which can be used to map one algorithm. It allows the configuration of several algorithms to co-exist in the RU, and programmers can use this feature to enable configuration caching. Similarly, the XIRISC system's reconfigurable unit, PicoGA, is partitioned into 3 blocks. Depending on the complexity of the algorithm mapped onto the PicoGA, blocks can be combined if necessary. PicoGA also uses redundant memory to achieve configuration prefetching and caching. The XIRISC prototype costs 12M transistors, and more than 1/3 of the total area is occupied by the PicoGA unit.



a) The Chimaera datapath

b) The XIRISC datapath

Figure 2.3: Other recent datapath-coupled architectures

In general, this class of reconfigurable systems has the tightest coupling between the host and the RU. The RU is easy to access by the host, and it is frequently reconfigured during function execution. The operating system support of such a system is simple, and the compilation is straight-forward. The drawback of such system is the regularity of the memory-to-RU interface and the scalability of the RU, and they limit the type and the size of the digital circuits that gain benefit from the RU.

2.1.1.2 Reconfigurable coprocessor

The RU coupled with the host as a coprocessor has direct access to the host's memory hierarchy. The host usually controls the coprocessors through message passing instead of instruction. Interaction between the host and the RU is much less frequent compared to the datapath-coupled systems, and the host and the RU can execute different applications concurrently and independently.

The GARP system[48][21] shown in figure 2.4 is a typical architecture of this class. The host MIPS II processor handles the configuration, task execution control and data transfer between the MIPS II and the RU. The Chameleon CS2112 chip[96] and the MorphoSys[90] have similar structures, but their RUs are further optimized for configuration data size reduction and configuration caching. The MaRS system[95] is an advanced version of the MorphoSys. The RU of MaRS is shared by a group of processors, and the memory modules are distributed among the processors in order to increase bandwidth.

The work from [70] experimented on the Xilinx Virtex-II device. The architecture used the existing on-chip instruction set processor PowerPC as the host, and divided the rest of the FPGA into several reconfigurable blocks. These reconfigurable blocks are connected with 3 on-chip networks (NoC), as shown in figure 2.5. These NoCs include a reconfiguration network (RN), a data network(DN) and control network(CN). The host is responsible for controlling all the networks and activities.

The Amalgam processor[55][38] shown in figure 2.6 is another NoC based architecture. In Amalgam, there are totally 4 reconfigurable units (RCluster) and 4 programmable units (PCluster). Managing so many computation resources at run-time is complicated, so the control of this system relies heavily on static analysis. The DART system[27][26][28] has 4 clusters of reconfigurable units, and the host processor is simply a task control unit. This architecture is suitable for linear and computationally demanding applications, but is not very flexible for general purpose computation.

The RU of this class of architectures is scalable, but when the RU is upscaled

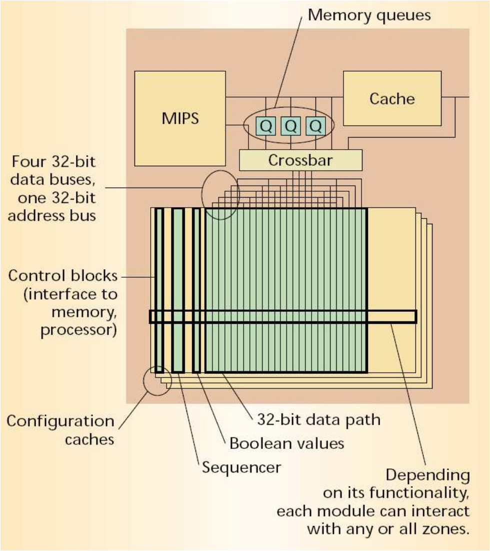

Memory queues
MIPS
Cache
Q Q Q
Crossbar
Four 32-bit data buses, one 32-bit address bus
Control blocks (interface to memory, processor)
Configuration caches
32-bit data path
Boolean values
Sequencer
Depending on its functionality, each module can interact with any or all zones.

Figure 2.4: The GARP processor architecture

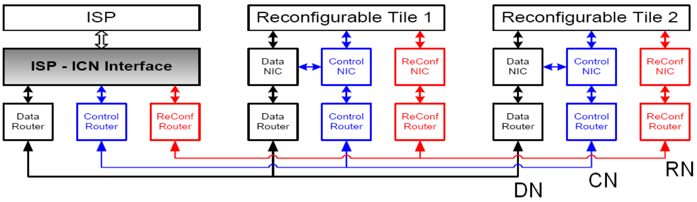


Figure 2.5: The NoC support for the reconfigurable system

to a certain extend, dividing an oversized RU into several smaller ones is more flexible and practical. Using complicated buses or NoCs to support a multi-RU system is often necessary, but the overhead and bandwidth requirements always pose design challenges. Also, having large RU enables designers to map complicated algorithms onto the RU, but it also increases the overhead of dynamic reconfiguration, e.g. long reconfiguration latency. What is also worth noticing is that the co-processors often can directly access the main memory or even the cache, thus creates two problems. First, the bus is sometimes overloaded, and it leads to stalling on both RU and the host. Second, if data consistency can not be guaranteed through static analysis, the cache consistency issue need to be addressed at run-time, thus the hardware and the performance overhead will increase.

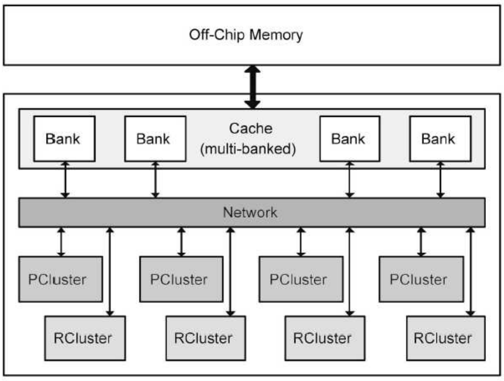


Figure 2.6: The Amalgam architecture.

2.1.1.3 Attached processing unit

The attached processing unit is coupled to the host through ports and external bus. The host system's memory is not directly accessible to the RU, and the RU functions are very independent. The RU is controlled by the host through device drivers or the operating system API call. Due to the inconvenience, the RU is rarely configured. The single chip solution of this class is much less efficient compared to the coprocessor architecture, and the RU is often built with several commercial FPGAs.

The GECKO system[101] is an experimental system that belongs to this category. This system's host is a complete COMPAQ iPAQ pocket PC, and the RU is a Xilinx Virtex-II device. These two parts have their own clock domains, power supplies and memory systems. One of the most interesting objectives of this project is to study the dynamic task migration, e.g. how a task can be moved to run on the host or the RU, and what the cost is for doing so.

Most of the commercial FPGA-based development systems belong to the category of attached reconfigurable processing unit. These systems are usually infrequently reconfigured by a host system, but they also have their own memory and peripheral devices. These architectures' coupling is very weak, and can usually be reconfigured through many standard interfaces.

2.1.1.4 Stand-alone processing unit

The stand-alone processing unit coupled reconfigurable system is the most loosely coupled architecture. The reconfigurable unit can even be a workstation accessed through the ethernet. The RU is usually wrapped up by many layers of software, from network protocol to device drivers, and the users of such system may have no knowledge of the RU. This type of system has high volume, and is scarcely reconfigured.

The Cam-E-Leon system is a typical stand-alone processing unit. Figure 2.7 shows the architecture of this system. User of this system accesses services by using a web browser and remotely gets the image processing service.

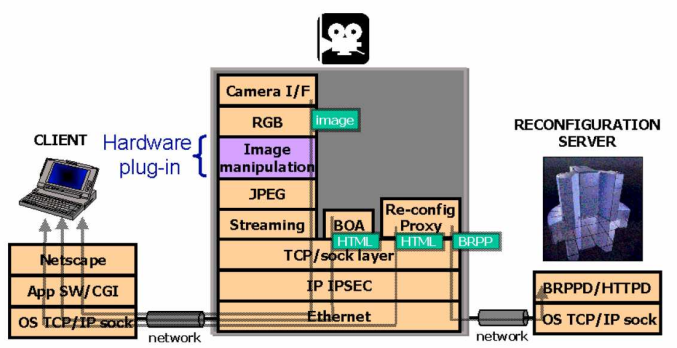


Figure 2.7: The Cam-E-Leon system layer

The attached processing unit and the Stand-alone processing unit have relaxed size constraints, but the communication cost between the host and the RU is very high. These systems fit for dedicated and highly complicated operations, and the dynamic reconfiguration means little to them. Both of these classes are well-understood and widely used, thus are not the focus of the current reconfigurable system research.

2.1.2 Logic block granularity

The logic block granularity is one of the primary factors that decides the system performance. The granularity of the reconfigurable logic is defined as the complexity of the atomic logic unit addressed during logic mapping. In general, finer-grained logic block is more flexible when being used to implement digital circuits, while coarser-grained logic requires less configuration memory and can achieve faster reconfiguration.

2.1.2.1 Fine-grained logic block

The most commonly used configurable logic blocks (CLB) are fine-grained. The input and output data of the fine-grained unit are single-bit wide, as shown in figure 2.8. The look-up table (LUT) is the most frequently used building-block of CLBs, especially in commercial FPGAs.

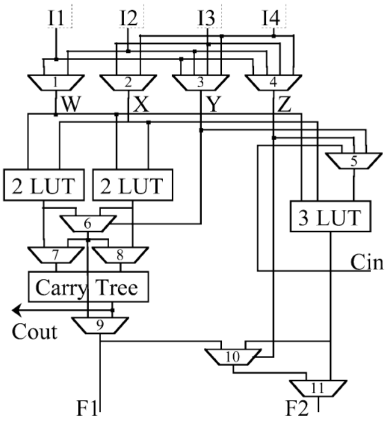


Figure 2.8: The fine-grained logic block from Chimaera system

The fine-grained reconfigurable unit suffers from the costly reconfigurable overhead. Due to the high volume of the configuration data, the storage requirement of these system is usually very high, e.g. approximately 30% of total chip area for commercial FPGAs. Also, the latency to load a configuration into an RU is proportional to the configuration data volume, thus fine-grained RU often suffers from slow reconfiguration. These shortages make the caching/prefetching of configuration difficult, but not prohibitive.

The greatest advantage of the fine-grained system is the flexibility. Any algorithm can be mapped onto the fine-grained logic device, and the bit-level operation-intensive applications use the fine-grained systems very efficiently. Due to the low granularity, these devices also have the highest utilization rate if compared to the coarser-grained devices.

2.1.2.2 Medium-grained logic block

The medium-grained logic block is a compromise between flexibility and reconfigurability. Figure 2.9 shows the medium-grained logic unit from PicoGA's reconfigurable unit[65]. As shown in the figure, the medium-grained logic block is very similar to the fine-grained version, except for the input/output data bit-width. Most of the medium grained systems are 2 or 4-bit wide.

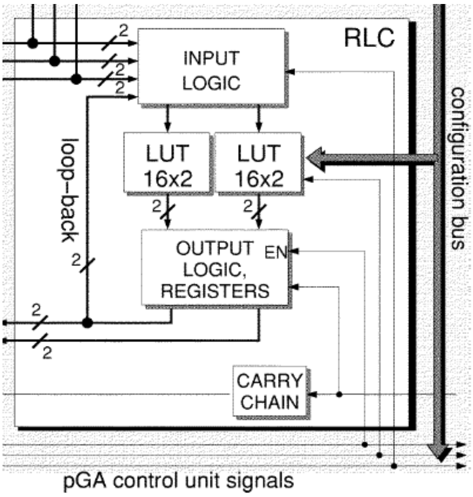


Figure 2.9: The medium-grained logic block from PicoGA system

The medium-grained systems are more friendly to reconfigure, but harder to map some application on. The utilization rate of the logic device is normally lower than that of the fine-grained systems, but for applications that do not perform many bit-level logic operation, the medium-grained systems are still practical and efficient.

2.1.2.3 Coarse-grained logic block

The coarse-grained logic has no regular form. The reconfigurable unit of the MorphoSys is an 8X8 array of 16-bit ALU, as shown in figure 2.10. The Montium tile processor[84] has a similar logic block, but extended with a butterfly-shaped MAC unit. The ADRES [75] architecture has up to 64 32-bit heterogeneous functional units as basic logic block.

The coarse-grained logic block could be more complicated than an instruction-

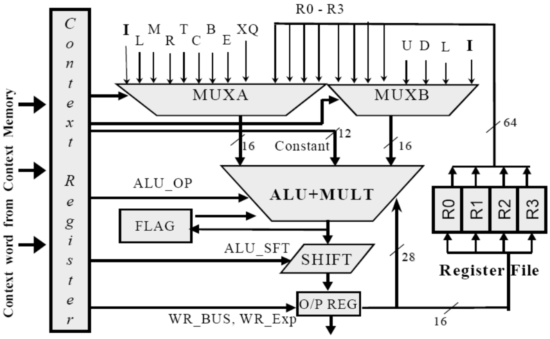


Figure 2.10: The coarse-grained logic block from the MorphoSys system

set processor. The RAW processor[97] is a compiler-directed reconfigurable system. As shown in figure 2.11, it is constructed with 16 tiles of independent processing units. In each tile, there is a MIPS-style processor interfaced with a programmable router. At compile time, single task will be partitioned and mapped onto one or more adjacent tiles. The RAW compiler is architecture-conscious, and orchestrates the routers statically. The programmable router layer of this system is one of the early NoC.

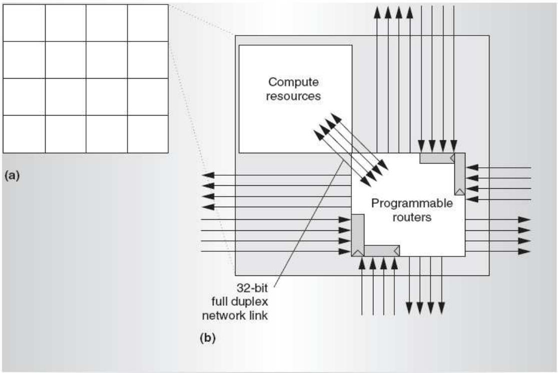


Figure 2.11: The RAW processor architecture

The coarse-grained logic block has the lowest reconfiguration overhead in terms of memory cost and reconfiguration latency. Many architectures employ ALU or similarly coarse-grained logic unit as the basic logic block, thus instruction-logic block mapping is more frequently used for these architectures instead of logic synthesis. This enables designers to program their applications in high-level language and apply advanced compilation technologies to use the architecture optimally.

2.1.2.4 Mix-grained unit

As mentioned earlier, architectures with lower logic block granularity fit narrower application domain. To improve the robustness and flexibility of these reconfigurable architectures, mix-grained architecture was introduced.

The technical proposal in [72] discussed a hierarchical architecture. The hierarchy of this reconfigurable unit is a quadric-tree. The lowest-level cluster is composed of an arithmetic node, a bit-manipulation node, a finite state machine (FSM) node and a scalar type operation node, since the operations of these four different algorithm domains are very incompatible. The four functional nodes are recursively clustered by a matrix interconnect network. The logic granularity of these different nodes is apparently different.

Another example is the DART architecture. The reconfigurable cluster of the DART has six coarse-grained logic unit (DPR) and an FPGA, as shown in figure 2.12. The DPR is used to execute most of the instructions, but the bit-level manipulation is handled by the FPGA. For an architecture like this, the task partitioning is another challenge.

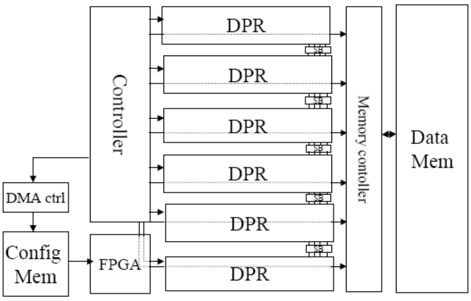


Figure 2.12: The mix-grained DART reconfigurable unit

The logic mapping of the mix-grained RU is more complicated than that of the mono-grained RU. If an RU is comprised of both the fine-grained and the ALU-grained logic, two of which require different compilation/synthesis tools, the integration of the tool will be difficult. Also, applications may need to be partitioned in an early stage to ease the compilation and the synthesis, and optimal partitioning will be a great challenge.

2.1.3 FPGA technology

The traditional FPGA suffers greatly from reconfigurable overhead. Without applying more advanced FPGA technologies, dynamic reconfiguration is only suitable for very small-scaled system. The most important technologies that increase the FPGA's reconfigurability are the run-time partial reconfiguration and the multi-context design. Routing issue of the commercial FPGA has always been a great challenge, and some other work proposed some means of simplification to this issue.

2.1.3.1 Partial reconfiguration

The partial reconfigurability allows part of the FPGA to be reconfigured, while the other part is running. This function is already supported by many commercial FPGAs, e.g. Xilinx Virtex family and ATMEL at6000 series[1].

The Virtex-II FPGA, as an example, can be divided into several separated blocks at very early design phase. The separated blocks, which are called PR logic in figure 2.13[11], are independent, and they communicate to the surroundings through dedicated ports. The boundary of each block cannot be changed once the algorithm starts running on the chip, but the algorithm mapped on the blocks can be reconfigured at run time. Since the reconfigurable block's boundary is rarely changed, the system is equivalent to a group of smaller FPGAs.

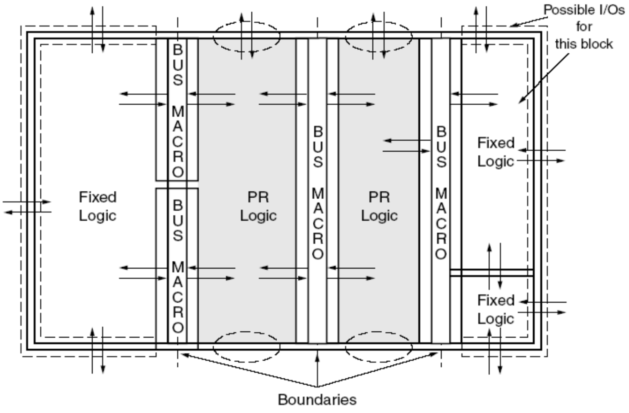


Figure 2.13: The partial reconfigurable logic of Virtex FPGA

The partial reconfiguration opens up many possibilities, e.g. it enables the hardware context switching and multi-tasking. If the system has a large amount of redundant reconfigurable blocks, the idle blocks can be used for configuration prefetching. However, the current FPGAs and their tool chains are not very friendly to use, and killer application is yet to be found.

2.1.3.2 Multi-context FPGA

Conceptually, the logic layer of an FPGA is an array of configurable logic blocks and interconnection nodes, and the configuration layer is physically a collection of distributed SRAMs or register files that store the configuration data of the logic layer. Unlike normal FPGA, which has one configuration layer and one logic layer, multi-context FPGA has multiple configuration layers but one logic layer, and all the configuration layers configure the same logic layer. Each configuration layer can store one set of complete configuration data and the intermediate data of the whole logic layer, thus is called one context. These configuration layers are connected to the logic layer through a multiplexing circuit, and the multiplexing circuit selects which configuration layer currently activates the logic layer. For those configuration layers that are inactive, they can be used as configuration caches. Most of the multi-context architectures can change their configuration in only one clock cycle, if the configuration is properly cached.

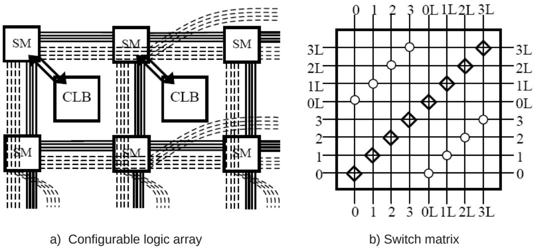
Xilinx has proposed a time-multiplexed FPGA[98] based on their XC4000E FPGA. This time-multiplexed FPGA has eight configuration layers and one logic layer. The reconfiguration loading time of the whole chip is only 5ns, which gives almost no reconfiguration penalty. Their proposal has not been commercialized, but their idea is adopted by many other research group. The DRLE system is also an 8-configuration system. In their work[35] the trade-off between the energy-latency product and the area has been studied. Their result shows that the 4 or 8-context FPGA is the most efficient for their architecture. The MorphoSys has coarse-grained logic block, and can store up to 32 configurations on-chip. The PicoGA FPGA has 4 configuration RAMs. As mentioned before, the PicoGA is partitioned into 3 blocks, thus one context switching can switch in up to 3 new functions.

The multi-context design provides configuration caching, which helps to hide the reconfiguration overhead. This technique also enables reusing the logic layer for executing different parts of a task, thus reduces the size of the logic layer. A main drawback of this technique is the high volume of the storage, thus is mostly applicable on coarser-grained systems.

2.1.3.3 Alternative FPGA design technologies

To make the reconfigurable architecture more flexible and user-friendly, many effort has been put into reducing the latency of creating a configuration from a netlist. Here is an example of how the architecture simplification can reduce the placement and routing latency.

The flexibility of fine-grained configurable logic block is not fully used by many applications, and the research described in [66][67] propose to simplify the fine-grained FPGA without significantly losing performance. As shown in figure 2.14a, their routers (SM) of each configurable logic block (CLB) link to their 1-hop neighbors and their 2-hop neighbors with solid and dashed lines, respectively. The internal structure of the router offers very limited connectivity: the wire from one side of router can only be connected to the wires of the other three sides with the same name, as shown in figure 2.14b. The result of the project shows that the WCLA FPGA, combined with the tool chain ROCPAR from the same group, are comparable to XILINX Virtex-E FPGA. Due to the simplified hardware structure, the execution time of the ROCPAR is on average 40 times faster than XILINX tool. One extra benefit is the whole tool chain ROCPAR, from logic synthesis to P&R, can be fit into the cache memory of the ARM processor.



a) Configurable logic array

b) Switch matrix

Figure 2.14: The WCLA FPGA routing[66]

2.2 Reconfiguration strategy

Depending on the coupling of the reconfigurable units, reconfiguration can be performed differently. Stand-alone processing units and attached reconfigurable units are usually less frequently reconfigured due to the device complexity and memory bottleneck, but they are scalable, reliable and simple to use. User of these devices are not very interested in the device flexibility, but mostly in the computation power, thus the reconfiguration of these devices are not interesting enough to study.

The coprocessor-coupled and the datapath-coupled architectures are more flexible and versatile, and the reconfiguration of these architectures are frequently discussed. They are both the main focus of the run-time reconfiguration (RTR) research, but due to their different characteristics and potential, they are reconfigured differently. The coprocessor-coupled architectures have great potential in scalability and performance, but are also the most complicated to reconfigure.

2.2.1 RTR of the datapath-coupled architectures

The datapath-coupled architectures are very frequently reconfigured. The PRISC system can reconfigure its PFU every clock cycle if the configuration is preloaded into the PFU. Once the reconfiguration occurs, the whole RU is updated. The Chimaera system and PicoGA system's RUs support partial reconfiguration, thus several configurations can co-exist on the RU. Comparing to the system that cannot be partially reconfigured, The Chimaera system and PicoGA system are not very frequently reconfigured, but it is very normal that their RUs are reconfigured several times when executing one task.

The datapath-coupled architectures are relatively small and regular. For systems like PicoGA, the reconfigurable array is homogeneous, has regular structure and is partitioned. Thanks to these characteristics, the dynamic reconfiguration overhead is manageable. The most frequently used strategy for these architectures is to statically explore the fine-grained parallelism, e.g. instruction-level parallelism (ILP) or loop-level parallelism (LLP), generate the configurations, and plan for the reconfiguration statically.

2.2.2 RTR of coprocessor-coupled architectures

Multi-tasking is one of the greatest potential of the coprocessor-coupled system. Depending on the size and the number of the co-processing RUs, the multitasking architecture varies, so is the RTR strategies. In general, the two main multi-tasking strategies are the single-coprocessor multi-tasking (SCMT) and the multi-coprocessor multi-tasking(MCMT).

2.2.2.1 RTR of SCMT system

The SCMT systems usually have a large non-partitioned reconfigurable unit that allows several tasks to run on it concurrently. Figure 2.15[45] shows the run-time multi-tasking strategy of such systems. Each shaded rectangular area on the FPGA models a task or a kernel of the task.

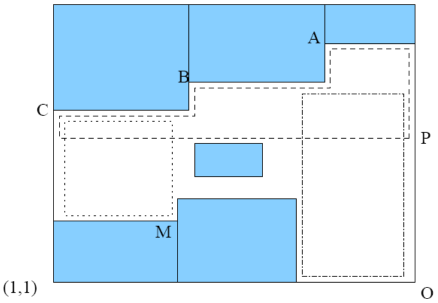


Figure 2.15: The SCMT system[45]

There are several design issues for SCMT systems. Firstly, the task allocation results in fragmentation on RU. Several research groups[24][45] have proposed algorithms to defrag the free space. Most of the defragmentation methods require reallocation of the issued tasks, which could be extremely time-consuming to do. Secondly, the rectangular model of the tasks is inadequate for many tasks, and more realistic models greatly increase the execution time of the task placement algorithms. Finally, the task communication interfaces must be persistent or at least traceable after reallocation, and run-time rerouting might be needed to handle the task communication channel.

The most critical performance bottleneck of the SCMT system design is the reallocation of the tasks. The issuing, allocation and reallocation of a task must occur at system run-time, which is not supported by the traditional FPGA design methodologies. Xilinx has contributed the JBits[40][39], a Java based program that can manipulate the FPGA bitstream at run-time, to support run-time reallocation. The JBits can access the logic and routing blocks when the FPGA is powered-on, reprogram any part of the circuit, and enable the updated part. The JBits operates at the logic level, which not only gives great flexibility but also results in many drawbacks. It is manual, and requires the programmer to have very good understanding of the FPGA. It also lacks a verification tool that can exam the modification and verify the timing of the final results.

JBits has stimulated many other research activities. In order to hide the low-level detail of the FPGA, Xilinx developed several other tools running on top of the JBits. Run-time parameterizable cores[41] are extended from the traditional static core models. Due to its dynamic parameterizable nature, the bitstream of an IP core can be dynamically synthesized and downloaded into the FPGA. The interconnects among cores are handled by using a stitcher class. User of the system only need to manually allocate the interface of the cores and stitchers, and the low level details will be automatically handled by the Java program. JRoute[54] is another automated routing program from Xilinx. JRoute supports more flexible routing/unrouting features and functional debugging. User of JRoute and the Run-time parameterizable core methodology needs very little knowledge of the FPGA, and their designs are portable. The software PARBIT[49] generates the partial bitfile of a given task and rearrange a running FPGA bitstream to fit the partial bitfile into the bitstream. This program extends the idea of JBits into task level. The CLB reallocation software introduced in [37] uses JBits as part of their reallocation flow. The proposed reallocation tool is capable of reallocating the circuit when it is running, thus hides the reallocation overhead.

The SCMT system's performance depends on the run-time RU management, and the task reallocation adds significant overhead to the reconfiguration latency. The recent FPGAs can support task reallocation, but the efficiency is rather low. The design methodology is currently under research, and there are only few architecture-OS combinations proposed.

Multi-context FPGA can eliminate the need of task reallocation. If we assume that each context of the FPGA stores the configuration of one task, then the tasks can share the reconfigurable unit in time rather than in space. This strategy has several disadvantages. Firstly, multi-context architectures normally have less computation resource due to the high memory cost, thus the size of the task that can be fit into the RU is more limited. Tasks will have a tighter area constraint when being synthesized, and larger tasks have to be partitioned. Secondly, tasks cannot be executed in parallel anymore, but have to be executed in turn. The overall performance of the system might be even lower than the systems suffering from task reallocation penalties. Finally, inter-task communication might have to go through special memory device, since communicating tasks can not be active at the same time. In general, Multi-context systems are also hard to design on SCMT system, and practical methodology and run-time system design is yet to be seem.

2.2.2.2 RTR of MCMT system

The MCMT systems have an array of reconfigurable unit tiles. A tile of RU could be a coprocessor, an FPGA or a partially reconfigurable module of an FPGA. The reconfigurable units are often small and not able to execute a complete application. Complicated tasks are accomplished by a group of interacting units that are connected with NoC or bus. Figure 2.16 shows the 16-tile RAW processor running 4 tasks concurrently as an example.

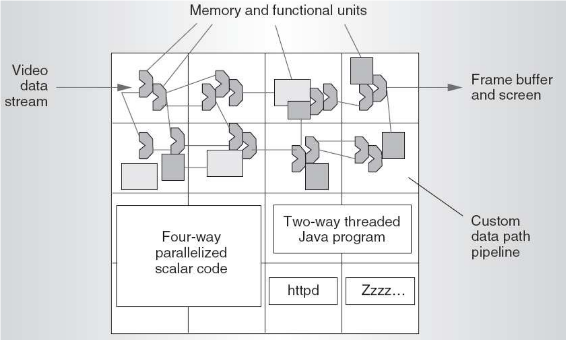

Memory and functional units
Video data stream
Frame buffer and screen
Custom data path pipeline
Four-way parallelized scalar code
Two-way threaded Java program
httpd
Zzzz...

Figure 2.16: The multi-tasking of the RAW processor[97]

The MCMT system is scalable, flexible and easy to control at run-time. The task model of the MCMT system is similar to that of the multi-processor system, but there are extra (re)configuration delay during task initiation and reconfiguration. The run-time support of the MCMT system only need to assign a task to a certain tile and setup the inter-task communication, hence is considerably simpler than that of the SCMT systems. The executed algorithm is partitioned and optimized at compile time, therefore the reconfiguration overhead is predictable and small. The reconfiguration of a tile is systematic, thus can be optimized by many existing technologies. The drawback of the MCMT design is the complicated compilation system, but many existing embedded system design technologies can be adopted.

2.3 Operating system design

The coupling of the reconfigurable units determine what operating system (OS) support is relevant. For datapath-coupled reconfigurable units, the RU is managed as a flexible datapath of the host processor and requires little OS support. For reconfigurable coprocessor, multi-processor OS design can be adopted. For attached processing unit, the OS manages the RU as a peripheral device. For stand-alone processing unit, the RU is usually a server with its own OS, and can not be accessed directly by other users. Due to the increasing complexity of the reconfigurable systems, traditional OS designs must be extended in many aspects, and some of the features need to be implemented in hardware to achieve higher efficiency.

2.3.1 Reconfigurable unit virtualization

From the programmer's point of view, the reconfigurable computing resource is always there to speed up the application execution, but in reality, the RU is a limited resource. If several tasks need to access the RU during a short period of time, and the total resource requirement exceeds the RU's capacity, the RU must be shared by tasks in time. In this case, which can be quite common, a virtulization mechanism must be built into the OS to facilitate this.

Such virtual reconfigurable resource management system is similar to the virtual memory management. But in contrast to the virtual memory management, RU has more complicated physical constraints, and the virtualization should partly be done at application compile time. For instance, larger tasks that can not be fit into the RU should be partitioned during compilation in order to reduce the run-time overhead. However, this research topic has not been recognized as an urgent issue to address. Even though it has been mentioned by many, solid solution are yet to be seen.

2.3.2 Virtual memory management

For systems that can support multi-tasking, the data allocation problem should be addressed. Reconfigurable systems that can support multi-tasking is often capable of running complicated algorithm. In case the local memory in a reconfigurable tile is not sufficient to hold the intermediate variable, the main memory access from the tile is necessary. The virtual memory (VM) management of the main memory access comes into the picture.

A simple method of managing VM in MCMT system is to maintain a table in the OS. Every entry of the table corresponds to a reconfigurable tile. When a task is issued to a reconfigurable tile, the corresponding table entry is updated with the virtual address of the task. When a tile tries to access the main memory, the data address is translated by the OS through the corresponding table entry. The work described in [102] is a hardware implementation of the concept. As shown in figure 2.17, the memory management unit (MMU) unit translates the physical address from the processor to the main memory. The window management unit (WMU), which is the MMU for the coprocessor, performs the same function at coprocessor side.

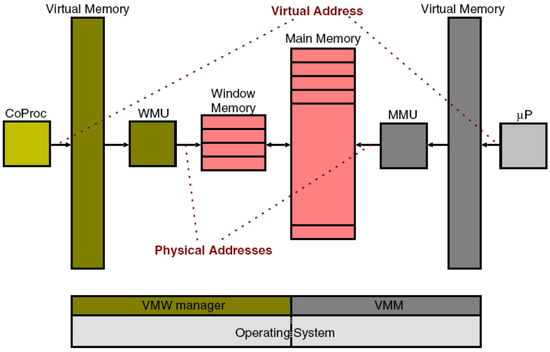


Figure 2.17: The Virtual Memory Management hardware[102]

2.3.3 Inter-task communication

The inter-task communication of the reconfigurable system differs greatly from that of the traditional OS. The tasks of the reconfigurable system could be located in the host processor as software or in the RU as hardware. In order to enable the inter-task communication between the host processor and the RU, an abstraction layer of RU should be built into the OS[81][77]. If the abstraction layer is well-designed, the traditional message passing is still appropriate for the reconfigurable system.

As shown in figure 2.18, the communication can be categorized into 3 types: the software-software, the software-hardware and the hardware-hardware communication. The software-software communication on the host processor is similar to the inter-task communication of the traditional system. The software-hardware communication passes through the hardware abstraction layer (HAL). In this case, the HAL is responsible for translating the OS specified task ID to the physical reconfigurable tile. The hardware-hardware communication can be handled by several manners. The straight-forward method is to pass the message to the OS and let OS transfer the data among different reconfigurable tiles through HAL. This method is easy to implement, but the data bus becomes the performance bottleneck. A more complicated but scalable communication method is achieved by the cooperation between the OS and the on-chip network that connects the reconfigurable tiles. The OS is only responsible for maintaining a routing table that keeps track of the location of the active hardware tasks. Once a message passing starts, the message source task fetches the physical location of the destination task from the routing table, packs the location information into the message and sends the message through the on-chip network. The MCMT system is very suitable for this communication scheme due to its multi-threading nature.

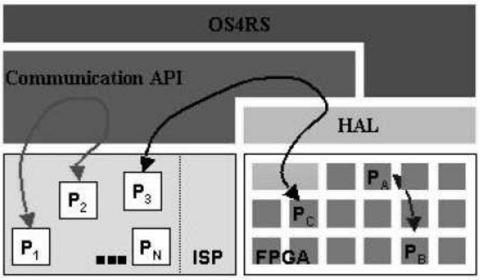


Figure 2.18: Three possible cases of message passing[81]

2.3.4 RU-OS interface

The choice of the interface method between the operating system and the reconfigurable units depends on the coupling between the host processor and the RU. For datapath-coupled systems, the compiler has a global view of the whole architecture and orchestrate the software execution at compile-time. At run time, software is directly executed on the reconfigurable hardware without any operating system interface support.

The architectures coupled in other methods usually need a device driver built into the operating system, unless the architecture is very simple. The driver can offer an abstract view of the underlying RU, buffer the input/output data and solve resource sharing problems. As shown in figure 2.18, the hardware abstraction layer hides the detail of the hardware implementation on the FPGA by offering a simple inter-task communication interface to the software. The HAL keeps track of the use of the reconfigurable tiles and location of the service, and if conflicts need to be solved, a message buffer can be implemented in the HAL.

The work described in [19] focuses on the single-thread applications. For each active procedure, no matter if it is implemented in hardware or software, there is a corresponding interfacing stub registered in the OS. Caller calls the callee with remote procedure call through the stubs and locate the required service without even knowing the location of the callee. Their device driver also supports the configuration readback function and partial configuration function.

2.3.5 Hardware context switching

Hardware context switching is difficult to handle for the following reasons. Firstly, context switch latency is normally high, and it adds to the task execution time. Secondly, loading a configuration into an RU costs memory bandwidth, and in turn lowers the overall system performance. Thirdly, storing a digital circuit's current state means storing all the data in its memory element, and it can be tricky and costly to do. Without proper optimization, hardware context switching causes huge performance penalty.

There are several methods to reduce the context switching overhead. The first one is to take the context switching overhead into account when assigning task priority. E.g. periodic tasks that demand high data bandwidth should be given a higher priority than the other tasks, thus be preempted less frequently. The second method is to use the RUs that can support bitstream readback. The readback should be able to access the status of all the internal registers and RAMs[60]. The third method is to define certain context switching points[78] in the application program. Experienced programmers can choose the best place for the context switching to reduce overhead cost.

2.4 Design methodology

The reconfigurable system can speed up the application execution significantly, but the performance gain is not easy to obtain. The programmers must have ample knowledge of the underlying architecture and great deal of experience in parallel programming in order to fully utilize the architecture's computation power. As shown in figure 2.19, the design flow of the reconfigurable system's application is also much more complicated than usual software design flow. It usually requires several software engineers and hardware engineers working together to program the reconfigurable systems, and the application development cycle can be very long.

The design automation is recognized as the crucial issue if the reconfigurable system wants to be adopted by the mainstream software engineers. When designing an application, the manually optimized application is the most performance-

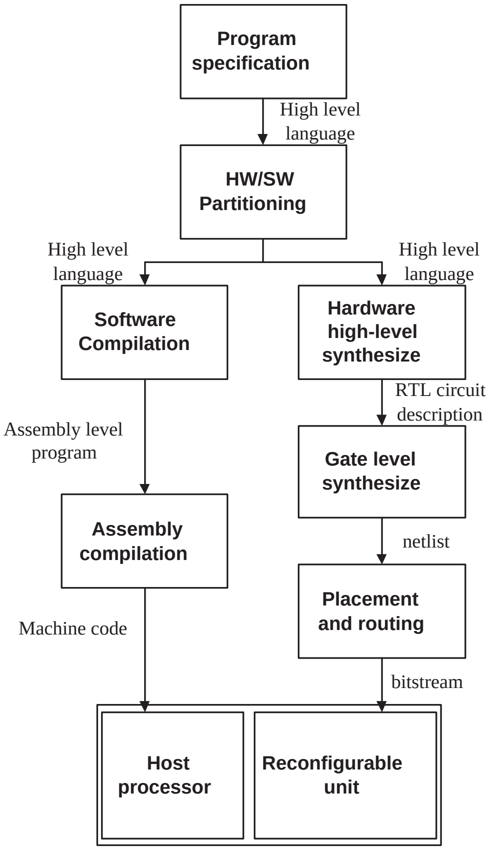


Figure 2.19: A typical design flow of application implementation for reconfigurable system[81]

optimal, but it takes too long time to design. Automated design tools still have only limited ability to explore the design space and take advantage of the RU, but are much faster than the completely manual approach. A compromise of the two extremes is to let programmer control part of the design flow. The early decisions in a design flow have the most significant influence on the performance, thus the partitioning and the high-level hardware design are often done under the supervision of programmers.

The objective of the design automation is to explore the application parallelism, take full advantage of the available on-chip resource and partition the algorithm efficiently into hardware and software without violating the resource and timing constraints. The programming language, the compiler and the synthesis tool play the most important rolls in the design flow. Many research of these areas have been done, and quite a few interesting results have been seem.

2.4.1 Programming

2.4.1.1 Register transfer level design

VHDL and Verilog are the most well-known Register Transfer Level (RTL) hardware design languages. Experienced reconfigurable system programmers can parallelize the application and manually map the algorithm on to a given RU with these languages. The timing of the design is manually constrained and optimized, and the use of the registers in the algorithm is pre-defined. The designer has control over all the details of the algorithm implementation, thus the development circle is very long.

SystemC extends the ANSI C with its own library. The SystemC-based FPGA design flow is very similar to VHDL/Verilog based design flow, although it offers a more friendly programming environment. The design is still at RT level, thus the clock signal and the circuit structure are explicitly defined by programmers. Since SystemC is based on C, it can be used for both software and hardware designers.

JHDL[15] is a Java based RTL design tool. Compare to SystemC, JHDL has explicit mechanism that supports reconfiguration. In JHDL, FPGA is represented as a *Reconfigurable* class, and the reconfiguration process is represented by the *Reconfigurable* object instantiation. When the hardware is realized, the interface between hardware and software is interpreted by device drivers. Programmers can manually partition the application into software parts programmed by usual Java semantics, and reconfigurable hardware parts encapsulated by reconfigurable class. The design can be easily co-simulated in Java environment.

2.4.1.2 High-level programming language

Cliff is an embedding of a network-domain specific language[56]. The fundamental unit of Cliff is an **element**. These elements have uniform interface, which is shown in figure 2.20. Communication among Cliff elements is based on three-way handshake protocol. When synthesized, all the elements are implemented as FSM with communication state and user state.

The work described in [79] generates RTL hardware description from DSP as-

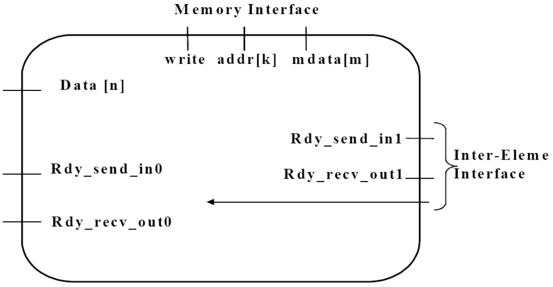


Figure 2.20: The interface of the cliff elements[56]

sembly code. Their compiler takes assembly language as input and decompiles it into Control and Data Flow Graph (CDFG). After the CDFG is optimized, it is translated into another intermediate abstract syntax tree that can be converted into RTL VHDL or RTL Verilog.

SA-C is a C-based single assignment language that targets image processing algorithms[80]. It supports integer and fixed-point number with user specified bit width[31]. The pointer and recursion are not supported due to the single assignment nature of the language. The body of a SA-C function is constructed with an input data window, a set of loops and a set of data return points. When VHDL code is generated from software function, a uniform hardware model that structurally represents each part of the software is used. As shown in figure 2.21, the loop generator, which is translated from the input data window, fetches data from the memory and sends them to the arithmetic unit called inner loop body (ILB). When ILB finishes the computation, the results will be stored in the memory by data collector, which is translated from the data return point. The ILB unit is synthesized from the optimized software loop. Because the ILB is fully combinatorial, its latency varies according to the software loop complexity. The timing and control of the computation process is handled by the loop generator and data collector. The SA-C also supports external VHDL component plug-ins and pragmas that mark the region of the code that need to be optimized[44]. The programmer can control function inlining, loop fusion, loop unrolling e.tc. through the use of the pragmas.

Handle-C is another C-based high level programming language[4]. Even if there is no explicit clock information specified by programmers, the compilation system assumes that certain instructions in the algorithm are clocked. E.g. all assignment statements and if/while statement execute in a single clock cycle, and a value assigned to a register can only be available in the next clock cycle.

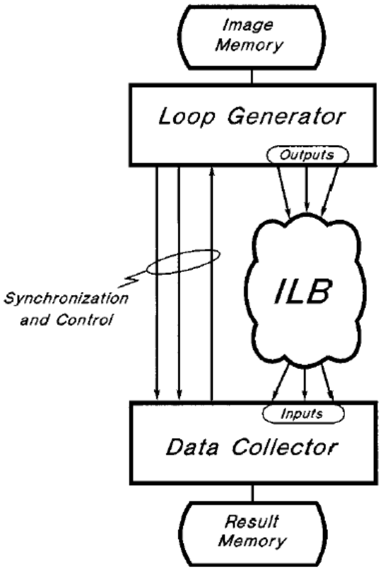


Figure 2.21: The hardware model of SA-C[86]

Haydn-C extends the idea of the handle-C by using two timing models[25]. The strict timing model of haydn-C is similar to the timing model of handle-C, and the flexible timing model is more advanced. Haydn-C compiler can break down the C description of an application into a dataflow graph (DFG). With the DFG and some user constraints, haydn-C compiler can reschedule the application in flexible timing model to find optimal implementations. Haydn-C also borrows the concept of entity and component from VHDL that make design more structured.

2.4.2 Software optimization

Software kernels are the most important candidates for intensive optimization analysis, since they are most frequently mapped to the reconfigurable units. The optimization objective is to exploit the data-level parallelism (DLP), the instruction-level parallelism (ILP) and the loop-level parallelism (LLP) of the high-level application description, and to efficiently map the parallelized kernels onto the RU. The optimization is mostly based on four different types of analysis: loop analysis, data analysis, communication analysis and pipeline analysis. Here we introduce some of the most frequently used technologies in these categories and discuss how they are different from traditional issues when applied to the reconfigurable systems.

2.4.2.1 Loop analysis

Loop unrolling: The most commonly used LLP optimization is the loop unrolling. For reconfigurable system, the unrolling factor is determined by several hardware constraints. For instance, the available memory bandwidth is a limiting factor of unrolling. If the available memory ports are fully utilized, the unrolling can hardly improve execution time of the kernel. As shown in [30], data producing rate and consuming rate of the reconfigurable tiles become the bottleneck of the system performance when the unrolling factor reaches certain threshold. The size of the reconfigurable tile can also be a physical limit of the unrolling factor. The number of CLB required to implement a kernel in hardware is proportional to the unrolling factor, thus the optimal unrolling factor should take the hardware space usage into account.

Loop merging: If two loops have the same index space, they can often be merged into one loop[103]. The benefit of the loop merging is the reduced data communication and loop control overhead. In case there exists data dependency between the merged loops, delay must be added into the merged loops.

Loop distribution: Quite opposite to the loop merging, loop distribution splits a loop into several loops. This technique is useful if the loop is too big to be mapped onto an reconfigurable tile. Loops sometimes cannot be distributed due to data dependency.

2.4.2.2 Data analysis

Scalar replacement: Arrays are usually stored in the data memory. A frequently reused array references can be replaced with temporary scalar variables, which are mapped to the RU's on-chip register by behavioral synthesis[107]. By doing scalar replacement, unnecessary memory fetching is reduced. Some high-level synthesis tools are able to exploit register reusability and decide which array reference should be replaced by scalar variables.

Tapped delay line: If consecutive and overlapping elements of a given array are accessed over consecutive iterations of a loop, it is beneficial to store the array in a linearly-connected set of shift-registers[30]. The registers can be concurrently accessed, thus data can be processed in parallel. The cost of achieving such optimization is the chip area for the registers and the multiplexing tree, hence the RU size limits how big an array can be implemented with registers.

Data layout and distribution: This technique lays out data in the memory in certain patterns so that multiple data being processed by an algorithm can be easily indexed and fetched. For reconfigurable systems that have memory units distributed on the RU, data layout and distribution are often intertwined with hardware/software partitioning, thus become very hard to analyze. This technique has been used by many in ad hoc manner, but the automation is still a challenge.

Data partitioning and packing: An array can be partitioned into a set of smaller arrays and distributed on the reconfigurable tile, so that various parts of it can be processed in parallel. This technique is frequently applied with data layout and distribution. Data packing is used to pack smaller data into one pack. It is often used to improve the data transmission efficiency , but it can also improve the data processing rate and reduce the reconfiguration rate when applied on reconfigurable systems.

2.4.2.3 Communication analysis

Static single assignment (SSA): SSA transformation is known to be able to reduce the data traffic by removing false data dependencies[76] at the cost of extra multiplexer. The work described in [52] shows that the placement of multiplexer in the software has significant influence on floorplan. If properly placed, the physical connection on chip can be optimized in terms of communication delay.

2.4.2.4 Pipeline analysis

Data context switching: If nested loops exist in the application, and the inner loops have data dependency that stalls the pipelined datapath, the data context switching can improve the performance[17]. Data context switching interleaves the outer loop execution by mixing different outer loop iterations in time. The cost of data context switching for reconfigurable system is the extra temporary storage and multiplexer tree in the pipelined datapath. E.g. if different outer loop iterations use different set of data as inputs, the data set should be stored into the pipeline with registers. As outer loop interleaves, the data registers are switched with multiplexing circuit.

2.4.3 Profiling

A common observation in hardware/software co-design is that the performance of the design depends on the partitioning. The quality of the partitioning relies on the estimation of the application, which is carried out by profiling. For reconfigurable system design, loop-accurate profiling is usually necessary, because a rougher estimation often results in the waste of reconfigurable resource. The profiling of an application could be done at program run-time or simulation time, depending on how it fits the methodology. Profiling should not only track the loop execution frequency, but also estimate the area cost of certain algorithm if implemented in hardware.

2.4.3.1 Run-time profiling

Run-time profiling usually requires compiler support. When an application is compiled for profiling, compiler adds counters into various parts of the application. At run-time, the corresponding counter is increased when certain part of the application is executed. The values of counters are reported to the user as feedback by the end of the profiling run. Run-time profiling only increases the execution time by a few percent, but it also only gives limited amount of feedback.

Profiling results can be case-dependent, thus a few profiling results may not be general enough to be the optimal partitioning guidance. Work presented in [43] proposes to generate both the software and the hardware implementation of the same application during design phase. When the application is compiled, both the complete software executable and the complete hardware implementation are generated. At application run time, application still generates profiling information, which is used to point out which part of the application should be executed with hardware.

2.4.3.2 Simulation-time profiling

Simulation-time profiling occurs at design and compile-time. It can give programmer more detailed feedback information, and the simulation can be very accurate. The costs of this profiling method are the effort of designing an accurate simulation model of the reconfigurable system and very long simulation time. The simulation time of an algorithm is frequently thousand to million times longer than the algorithm run time, depending on the accuracy of simulation and the complexity of the system being simulated.

Traditional simulation-time profiling runs the user application on a software-emulated processor, thus increases the simulation time greatly. The work described in [89] suggests to use a hardware-emulated processor instead of the software-emulated version. The idea is to synthesize the execution platform of the application, e.g. an instruction-set processor, onto an FPGA with supervising profiling counters. Users can run their applications on the synthesized processor and extract the profiling counter value from the FPGA when the simulation is finished. To facilitate this profiling method, an actual RTL model of the execution platform has to be built, and it can be a non-trivial task.

2.4.3.3 Profiling feedback

The main interest in profiling the reconfigurable system's application is to know the execution frequencies and latencies of loops. The most frequently executed loops, or kernels, are the critical candidates to be mapped to the RU. The FLAT profiler[94] and many others[61][50] focus on the loop-profiling and function-call-profiling. Their profiling results show the detailed timing break-down of the user applications.

In order to reveal more detailed hardware nature of an application before partitioning, many profiling tools quickly estimate the synthesized result of the applications in terms of area. As described in [57], the area-bitwidth relationship of arithmetic operations are categorize into 5 types. E.g. the area of addition and multiplication unit increases linearly and quadratically, respectively, if bitwidth increases. This estimation method is built into the SA-C compiler and shown to have 90% accuracy.

2.4.4 HW/SW partition

Partitioning greatly impacts how efficiently the RU can be used, thus has been discussed frequently. Traditional HW/SW partitioning methods can be used on many reconfigurable architectures, but they do not take full advantage of the flexibility of the reconfigurable systems. Many recent work proposed novel partitioning techniques to cope with some special characteristics of the reconfigurable system.

2.4.4.1 General partitioning strategy

The partitioning techniques for reconfigurable devices are very similar to traditional ones. Most frequently executed tasks that can be parallelized are mapped to hardware[61][94], and the size of the hardware constrains the partitioning. More advanced partitioning tools take communication and data locality into account, and apply heuristics to analyze various partitioning strategies.

Reconfigurable architecture makes the partitioning a bit more complicated. Besides increasing the system performance and reducing data communication[34], a good partitioning strategy should also increase the configuration reusability[53][18], reduce memory latency[46] and fully use the reconfigurable resource[61]. However, heuristics used in solving traditional partitioning problems still fit the reconfigurable architecture if their cost functions can cope with additional complexity.

2.4.4.2 Partitioning for reconfiguration

A hardware task may be too large to be mapped onto the on-chip reconfigurable unit completely. In case this happens, the oversized task can be partitioned into several smaller tasks and be executed separately in time. As shown in figure 2.22 [63], a task graph can be evenly and efficiently divided into several stages, each of which can fit into available reconfigurable unit. The staging determines the communication among tasks. If the communication can not be localized in the RU, minimizing the communication can be an interesting issue to address.

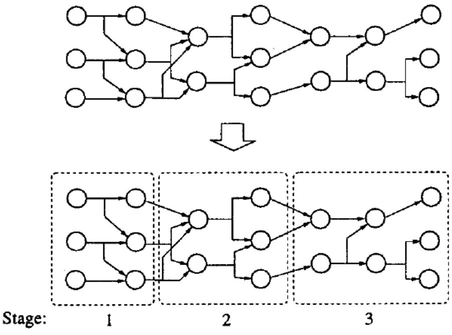


Figure 2.22: The temporal partitioning of oversized task[63]

Multi-context FPGAs can benefit greatly from such temporal partitioning[62][36][105]. Data communication among different time stages can simply be localized by using memory elements in the RU, thus results in almost no communication overhead. Since multi-context FPGAs can switch the configuration in one clock cycle, tasks being temporal-partitioned can be executed seamlessly. Also, the partitioning algorithm can spend less effort evaluating the communication cost, which greatly reduces the compilation time.

2.4.5 Scheduling

Instruction-level scheduling is frequently used for coarse-grained datapath-coupled architectures. Most of these schedulers explore fine-grained parallelism in the application and schedule the instructions on the reconfigurable unit statically. The scheduling issue often resembles an instruction-level placement and routing issue. For architectures that rely on multi-context support to execute kernels[75], their statical instruction-level scheduling issue is a 3D placement and routing issue[76], since time-axis also need to be considered. Modulo scheduling algorithm based algorithms[91] solve these problems quite efficiently.

2.4.6 High-level synthesis

Software and hardware designers have very different understanding about computation. Software designers view a computing system as a **sequence** machine that has virtually **infinite** amount of resources, but hardware designers try to explore **parallelism** on **limited** resources. For hardware designers, the flexibility that software designers desire is difficult to translate into hardware description. At the current stage, advanced software programming technique like polymorphism, use of pointers, multi-dimensional data structure or recursion are still difficult to convert into hardware[42]. But fortunately, most computation-hungry parts of user application are presented in loops, which can often be represented with arithmetic units, memories and counters. Even if several limitations exist, many application programmed in high-level language that can benefit from hardware acceleration can still be partially synthesized.

The most frequently used strategy of high-level logic synthesize is to convert the optimized and partitioned high-level software description into a hierarchy of finite state machines (FSM)[59][56]. The logic part of the FSM implements the core algorithm. Synthesis techniques like pipelining and multi-cycle data path are frequently used[50]. Retiming and clock-gating have also been discussed[91]. The control part of the FSM acts as a data access arbitrator. It keeps track of a set of counters to generate memory addresses if external memory needs to be accessed. The data exchange among tasks, operating system controls and resource sharing should be planned at synthesis-time and carried out by the FSM control part at run-time.

Some of the research is dedicated to compiling certain intermediate software presentation into an RTL hardware description. The Stanford SUIF2[2] and OCAPI-xl[100] can be used as frontend tools that generate parallel presentation of user application. The more recent achievement SPARK[42] takes C program as input and produces RTL VHDL code that is compatible to Synopsys Design Compiler and Xilinx XST/ISE tool chain. These approaches more or less follow the FSM based approach and extend it in various ways.

2.5 Conclusion

A reconfigurable system's coupling method has decisive influence on the system's scalability and reconfigurability, as shown in figure 2.23. Attached processing units and Stand-alone processing units are very loosely coupled to the host processors. They are usually off-chip, thus can grow very large in size. Due to the size, the tool chain, the control methods and the application domain, these systems scarcely need to be reconfigured dynamically. Much work has been done to improve these systems, but dynamic reconfiguration is not yet an interesting topic for them.

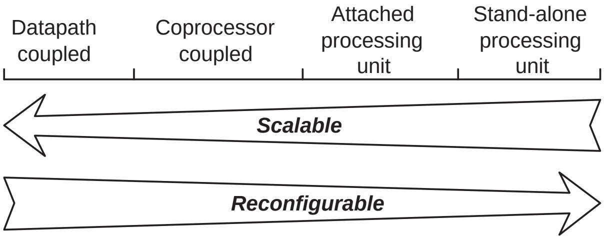


Figure 2.23: Scalability and reconfigurability contradiction[63]

The datapath-coupled system is another extreme. These reconfigurable units are used as special functional units, thus have limited data bandwidth and complexity. Many of these architectures have multi-context support to achieve single-cycle dynamic reconfiguration. These systems are relatively flexible to use and friendly to program, and demand little from compiler and operating system. But due to their limited scalability, the performance of these architectures is not much faster than a state-of-the-art Multi-Processor System-on-Chip system. Even if not yet recognized by many, we believe that the scalability is one of the critical issues to be addressed.

The coprocessor-coupled systems are the most complicated yet interesting ones. In theory, their RUs can be easily upscaled, no matter whether the RU is a single unit or composed of multiple cores. They are ideal platforms to support multi-threading, and have great potential to improve the performance of the host processor. However, due to the reconfiguration overhead, the extremely high design complexity, the unpredictable dynamic behavior, the limited programmability, the complicated run-time resource management, etc, these systems are still very far from being put to practice. From previous study, we observed that not only is such system hard to design, but the interplay among the architecture design, the methodology design and the run-time system design makes it extremely complicated to evaluate and optimize.

What coprocessor-coupled system research needs is not a few more new techniques to improve performance or help partitioning, but a user-friendly integrated system-level simulation framework. Such a framework can be used to gain better understanding of these systems' behavior and to evaluate various techniques. Due to the complexity of reconfigurable systems, building such a simulation framework is an ambitious and open-ended project. We believe that such an integrated environment can help people understand the coprocessor-coupled reconfigurable systems better, and intrigue people to create more practical ideas.

Before we can build our simulation framework, the reconfigurable system and its application has to be modelled. In order to make our model realistic, we carried out a few partial reconfiguration experiments on commercial FPGAs. Through the experiments, we also got to know how mature the commercial FPGA is, what potential these devices have, and what improvement is urgently needed.

To conclude, we would like to continue our study by experiencing the commercial FPGA's partial reconfiguration flow, studying the scalability of datapath-coupled architectures, and proposing a simulation framework that can capture the dynamic behavior of the coprocessor coupled architectures. Our discoveries are recorded in the next few chapters.

Chapter 3

A Reality Check Based on FPGA Architectures from Xilinx

FPGA technology has mainly been used as a means of rapid prototyping for ASIC designs. From a fabrication's point of view, FPGAs are general-purpose devices produced in extraordinarily high volumes, allowing FPGAs to take advantages of the latest semiconductor technologies and improve their densities and speed. These characteristics effectively reduce the performance gap between FPGAs and ASICs over generations.

Recently, several FPGA vendors have investigated the potential of the dynamic reconfiguration of their logic devices, and try to bring the device reusability to a new level. Xilinx proposed a technology called partial reconfiguration[11], which enables an FPGA to be partitioned into several logically non-committed modules that can be reconfigured and assembled at run-time. On its quest to support dynamic reconfiguration, Xilinx introduced an Internal Configuration Access Port (ICAP) which allows the device configuration to be accessed and modified internally. Having ICAP on-chip leads to an easy and efficient way of building a run-time self-reconfigurable system, in which a (soft-)processor can access the ICAP device through a user-friendly configuration controller.

The current research trend in dynamic reconfigurable systems is focused on tightly coupled architectures, such as coprocessors or datapath coupled architectures. For this purpose, the Xilinx Virtex family appears to be an ideal prototyping platform, hence many proposed architectures and design methodologies[104] have been prototyped and experimented on the Xilinx device.

In this chapter, we present a reality check on the state-of-the-art support for dynamic reconfiguration based upon the Xilinx Virtex family of FPGAs. Our main objective is to understand how the partial reconfiguration is carried out on Virtex FPGA, what design methodologies has been applied to support partial reconfiguration, what scenario a reconfiguration goes through and how friendly it is for a software programmers to reconfigure the Xilinx device. We also pinpoint the existing design pitfalls, and propose solutions to overcome some of these issues.

3.1 The Virtex configuration organization

Virtex family has a column-based organization that offers modest configuration flexibility[12]. As shown in figure 3.1, the configuration memory of the FPGA is organized as a one-dimensional array of heterogeneous columns. Depending on the type of the column, different numbers of storage frames are needed to store the column configuration. For instance, for any Virtex and Virtex-E FPGA, a CLB column always contains 48 frames while a memory column always contains 64 frames.

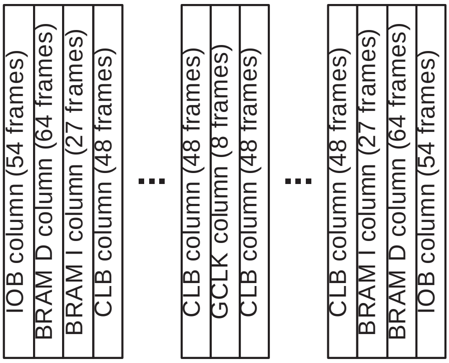


Figure 3.1: Xilinx Virtex's column based configuration memory structure

As shown in figure 3.2, columns are organized as a 1D array of frames. All frames are single-bit wide and span from top to bottom within their designated column. Data stored in a frame is hard to interpret. E.g. for the xcv1000 FPGA, each column contains the configuration of 64 CLBs. Each CLB needs approximately 864 bits to specify its configuration, including the local routing configuration. These 864 configuration bits for a single CLB are distributed into 48 frames so that each frame only contains 18 bits of the CLB configuration. I.e., each frame contains a small portion of these 64 CLB configurations, making the data stored in a frame almost unreadable.

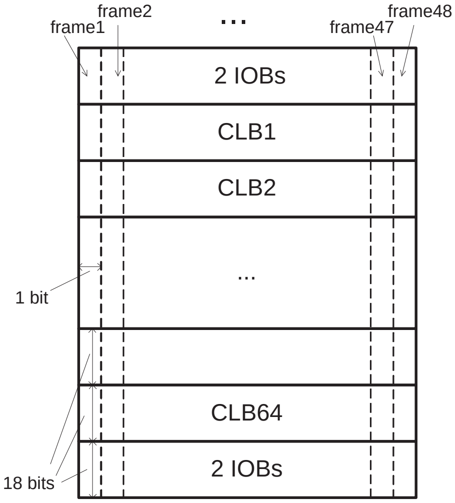


Figure 3.2: One column of a xcv1000 CLB

For each generation of the Virtex family, the column-frame organization varied slightly, but the column-frame based backbone has largely been the same. The understanding of the column-frame based architecture is essential in order to understand the mechanisms which enable and limit the Xilinx partial reconfiguration methodologies. Xilinx partial reconfiguration design flows, introduced in the next section, can be viewed as “frame content manipulation”.

3.2 Xilinx dynamic reconfiguration design flows

Xilinx has two documented dynamic reconfiguration flows: module-based design flow and difference-based design flow[11]. In general, the module-based design flow is suitable for systematic design of larger systems. In this flow, the partial reconfigurable units are modularized so that all the partial designs are implemented in the same manner. The cost of this flow is the inevitable high storage cost and long reconfiguration latency. The difference-based design flow takes advantage of the existing correlation between partial designs. This design flow is mostly suited for very small systems.

3.2.1 Module-based design flow

The module-based design flow is an area-constraint driven design methodology. It is extended from the modular design flow[3] that is commonly used in large-scale ASIC/FPGA design. Figure 3.3 shows the complete design/reconfiguration flow. The objective of this flow is to create an initial bitstream and a set of independent partial bitstreams. The initial bitstream is downloaded to the device as the default device setup, while the partial bitstream is downloaded when needed during run-time.

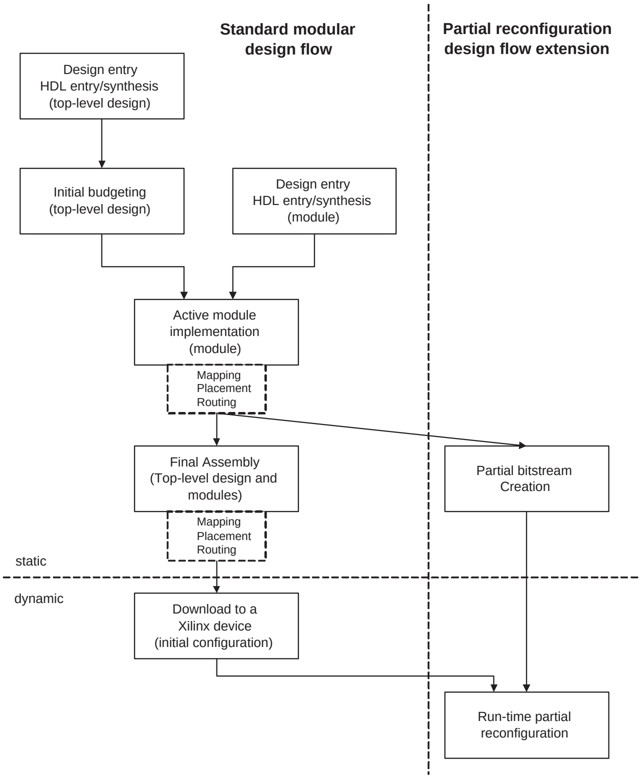
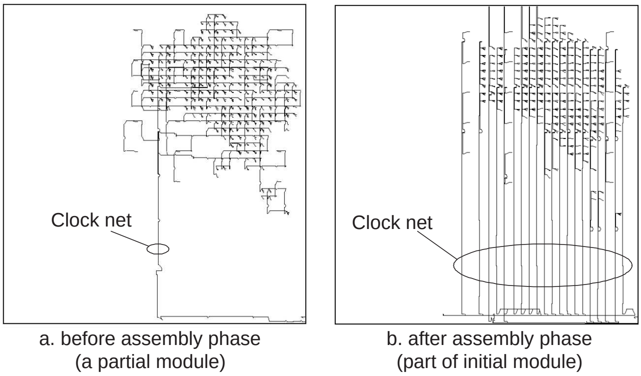

Standard modular design flow
Partial reconfiguration design flow extension
Design entry HDL entry/synthesis (top-level design)
Initial budgeting (top-level design)
Design entry HDL entry/synthesis (module)
Active module implementation (module)
Mapping Placement Routing
Final Assembly (Top-level design and modules)
Mapping Placement Routing
Partial bitstream Creation
static
dynamic
Download to a Xilinx device (initial configuration)
Run-time partial reconfiguration

Figure 3.3: Module-based design flow (based on figure 4-1 from [3])

During floor-planning, the FPGA is partitioned into several non-overlapping regions, which are conceptually viewed as reconfigurable units. During placement and routing, each partial design is completely fitted into a region. When a partial design is downloaded, a region is reconfigured without affecting the rest of the FPGA.

The communication among regions is implemented with on-chip tristate buffers or multiplexors. The communication interface is persistent even if the region can be reconfigured. This requires that all partial designs, which are allocated to the same region, have identical physical interface. The logic units used for communication are not to be reconfigured.

Pitfalls: The module-based design flow is scalable and systematic, and the designer does not need to know low level details about the device. The area, timing and storage cost for each reconfiguration is constant for each region. However, many problems exist in this flow. Our experiments discovered that the same submodule can have different clock trees before and after the assembly phase, as shown in figure 3.4. The Xilinx design environment may have tried to reduce the clock skew during the assembly phase, but it leads to the discrepancy between two layouts of the same circuit. This problem may lead to constant clock skew among different modules, and may require extra synchronization units to be added to the user design. A hypothetical solution to this problem is to use the assembled design as the guidance to create the partial bitstream so that the partial design can have optimized timing as the assembled design.



a. before assembly phase
(a partial module)

b. after assembly phase
(part of initial module)

Figure 3.4: Two different clock nets of the same logic unit

Another problem of the design flow is that the partial reconfigurable bitstream can only be placed to a fixed region. Since the partial bitstream is region-specific, if a partial design need to be placeable to two regions, two partial bitstreams must be generated. This increases the cost of storing the partial bitstream, and the design is not as flexible as expected.

Finally, the design methodology is fundamentally an area-constrained placement and routing (P&R) process. There is no hardware support to guarantee the successful partial P&R, thus the layout level design is subject to P&R errors. Many pre-synthesized cores are partially placed and routed during RTL design phase in order to satisfy specific timing/area constraint, and these cores may not even be able to fit into many user partitions without violating performance constraints or design rules. A highly automated implementation of a partial design is still not available, since constrain violation and P&R errors are hard to eliminate by the current software support.

3.2.2 Difference-based design flow

Difference-based design flow is currently very inefficient for large design. The principle of this flow is as the following: Suppose there are two placed and routed circuit descriptions, namely A and B. Instead of creating a full configuration bitstream for each of them, we only create the full configuration bitstream of A. For design B, we compare its circuit description frame-by-frame with that of design A, note down all the different frames between A and B, and then create a partial reconfiguration bitstream that only modifies the frames with differences. With this partial configuration bitstream, we can dynamically reconfigure design A, which must already be downloaded into the FPGA, into design B. Compared to the module-based design flow, the frames being reconfigured does not need to be consecutive, and the timing/storage cost of performing such a reconfiguration is proportional to the number of different frames, plus some overhead. Apparently the cost of this design flow can be low only if design A and B resemble each other on the layout level.

Pitfalls: At current stage, there is no automated ways of making two unrelated designs similar at the layout level, thus this design flow heavily relies on manual work. Usually the designer will start from a placed and routed design description and manually identify the possible modifications at layout level. Hence, only very simple applications can take advantage of this design flow and really get some benefits, even if this design flow provides fast frame-level reconfiguration.

Another issue is that a specific configuration must be on the device before certain partial configuration bitstreams can be used. If the reconfiguration from any n possible configurations to any other $n-1$ configurations has to be achieved by a single reconfiguration, $n(n-1)$ configuration bitstreams have to be created and stored in the system memory. A possible solution for reducing memory overhead is to perform a two-hop reconfiguration. If a common neutral configuration can be found for a reconfigurable unit, we can reconfigure any reconfigurable unit from the current configuration to the neutral configuration, and then from the neutral configuration to the target configuration. This solution can reduce the memory requirement from $n(n-1)$ to $2n$ by doubling the configuration time. However, this is only feasible when the neutral configuration is similar to most of the possible configurations, thus keeping all configuration bitstreams small.

3.3 ICAP

The ICAP is a simplified substitute of the Xilinx SelectMap reconfiguration solution, which is a byte-parallel external reconfiguration interface that can achieve the highest possible reconfiguration speed on Xilinx devices. It is assumed that the ICAP will only be used for partial reconfiguration, since the part of the FPGA that controls the ICAP must never be reconfigured through ICAP. Xilinx offers device drivers in the latest embedded system development kit (EDK) releases and offers an API that can easily control this device.

Theoretically, the ICAP device can load one byte of configuration data each system clock cycle. Xilinx has reported that the minimum reconfiguration time of each xc2v1000 frame is 13 μs[16]. However, experiments have shown that in practice the Xilinx device driver is much less efficient. The timing cost of loading a frame from memory and sending it to the ICAP device buffer dominates the reconfiguration process. According to our ModelSim simulation the actual cost to reconfigure a xc2v1000 frame is increased to 69 clock cycles per word, and it will be even slower if the memory hierarchy is more complicated. This results in a frame reconfiguration time of around $300\mu s$ at 66 MHz, if the overhead is included. As long as the software scheduling and the caching technique lags behind, the speed potential of ICAP will be greatly limited.

3.4 Conclusions

From our experience, we can hardly say that the Virtex device is dedicated and advanced enough to be the experimental platform for most of the recent researches. From the frame/column structure, we can see that the CLBs, which are the ideal atomic reconfiguration unit, are indexed in an awkward way in the configuration memory. From our analysis of bitstream composition, we discovered that loading a configuration to non-consecutive configuration memory location costs extra configuration latency, thus many unnecessary timing penalty is added when reconfiguring the Virtex devices. This latency can not be reduced unless the frame/column based structure is changed.

Various computation resources, e.g. multipliers, digital signal processing units and memory elements, are distributed on the FPGA in a non-uniform manner. This complicates the partitioning of an FPGA. Which part of an application can be mapped to which partition is a difficult issue to analyze, since the designer must have minute knowledge about both the application and the resource distribution of each partition. Even if the synthesis tool can assist the designers, such analysis is still a time-consuming task.

A digital circuit's implementation is device dependent, thus a partial bitstream can only be used on the same type of FPGA. The circuit implementation is also partition-specific, thus each partial design can only be run on one specific FPGA partition. If an algorithm needs to be executed in multiple partitions, each partition has to have its own copy of the same algorithm implemented. This issue makes the memory cost of the configuration very high, and the management of multiply algorithm can be challenging.

The partial configuration latency is another issue. Suppose that a Virtex 4 LX 25 device, which is a rather small FPGA, is partitioned into two modules. Depending on the reconfiguration methods, the reconfiguration latencies range from tens of milliseconds to seconds. If we assume that the reconfiguration latency costs one percent of the total application execution time, the reconfiguration is only allowed to occur on second or minute bases. It is hard to argue what kind of applications can benefit from such a low reconfiguration rate.

”To help minimize problems related to design complexity, the number of reconfigurable modules should be minimized (ideally, just a single reconfigurable module, if possible). This said, the number of slice columns divided by four is the only real limit to the number of defined reconfigurable module regions.” [1] No matter how pessimistic it sounds, it is an honest assessment for current state-of-the-art commercial FPGA. Virtex series FPGA have some potential in terms of partial reconfiguration, but are currently not suitable for implementing highly complicated reconfigurable system due to many shortcomings in their architectures and methodologies. It is feasible to built small-scaled systems with two reconfigurable units that can communicate with each other. But beyond that, there will be too many physical limiting factors which effectively reduce the system efficiency. We conclude that the current FPGA can only be used for demonstration purpose with toy applications, but it is not a satisfactory platform for building highly, or even moderately, complex and advanced reconfigurable systems, which is the aim of the reconfigurable research community.

[1] Quoted from Xilinx Application Note 290.

Chapter 4

MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture

Datapath-coupled architectures are generally hard to upscale but easy to reconfigure, thus how to increase the scalability of these architecture is an interesting yet challenging issue. To investigate the performance bottleneck and the scalability of the state-of-the-art datapath-coupled reconfigurable architectures, we studied the coarse-grained reconfigurable architecture ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) developed by IMEC, Belgium. Through our research, we understood the advantages and the limitations of datapath-coupled architectures, and proposed to explore task-level parallelism to improve the scalability of similar architectures.

In this chapter, first we give a short introduction to the ADRES architecture, point out its limitation, and propose to apply multi-threading on ADRES. Then we discuss how the ADRES architecture can be extended to support multi-threading, and how the ADRES compilation tool flow needs to be extended to cope with that. We continue our work by running a dual-threaded MPEG2 decoder on a customized ADRES architecture to demonstrate that multi-threading is feasible for ADRES. Through the MPEG2 experiment we discovered some design pitfalls that hinder the performance of the threaded ADRES, and discussed

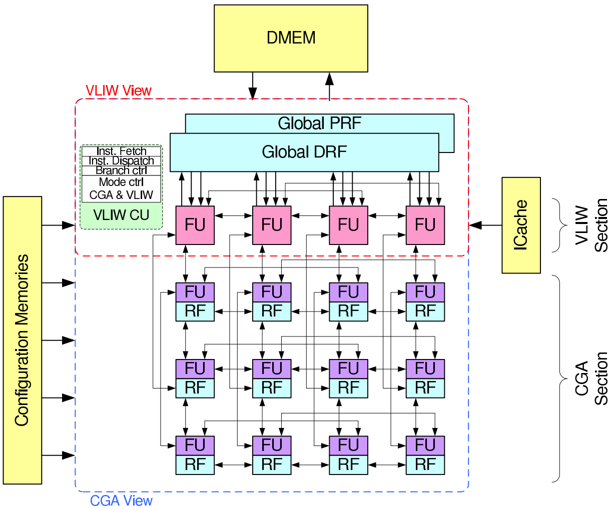


Figure 4.1: ADRES architecture template and its two operation modes

what technologies can further improve the performance of the multi-threaded ADRES. Finally, we conclude our study and discuss what issues need to be addressed in future work.

4.1 Introduction

The ADRES architecture template is a datapath-coupled coarse-grained reconfigurable architecture [75]. As shown in Figure 4.1, it resembles a very-long-instruction-word (VLIW) architecture coupled with a 2-D Coarse-Grained heterogeneous reconfigurable Array (CGA). The CGA is an extension of the VLIW rather than a reconfigurable unit attached to it, and the computation power of the VLIW can also be used by the CGA when necessary. The ADRES architecture brings the coarse-grained logic design to its extreme by employing the heterogeneous functional unit (FU) as the atomic logic unit of the CGA, hence reduces the requirements of the configuration memory size to the minimum and enables the multi-context storage support. Since the CGA is an array of FUs, the programmer can use a high-level language to program his/her application, therefore can focus more on exploiting the instruction-level parallelism of the application instead of the data-level parallelism.

As ADRES is defined using a template, many design factors can vary depending on the use cases. In principle, FUs that are connected to the global register files are usually defined as parts of the VLIW, and any FU can be defined as part of the CGA. But in practice, there is no constraint of how many FUs can be used as VLIW functional units, or how many FUs can be excluded from the CGA. As a result, the boundary between the VLIW and the CGA is usually blurred, thus ADRES offers more freedom for the designer to improve the architecture performance. The interconnection among FUs are also described in the template. The designer can freely experiment with the use of buses, on-chip network (NoC) of various topology, or even the mix of them to study the impact on the communication and the implementation cost. The designer can also tailor the instruction set supported by each FU, and investigate what combination or distribution of the instruction set is optimal for a specific application. The architecture described in the template can have arbitrary size and complexity, and the feasibility of the physical implementation does not impose any constraint in such an early development stage.

The processor operates either in the VLIW mode or in the CGA mode. When compiling an application for ADRES with the DRESC compiler [76], the application is partitioned into kernels and control codes. The kernels and the control codes are modulo-scheduled for the CGA or compiled for the VLIW, respectively, by the DRESC compiler. By defining several FUs that can access the global data register file and sharing them between the VLIW mode and the CGA mode, the global data register file serves as a data interface between two modes, enabling an integrated compilation flow. By seamlessly switching the architecture between the VLIW mode and the CGA mode at run-time, statically partitioned and scheduled applications can be run on the ADRES with a high number of instructions-per-clock (IPC).

As shown in figure 4.2, the DRESC tool chain is an integrated environment for both the application compilation flow and the architecture synthesis flow. The application compilation tool chain uses C programs as input. The C program is iteratively profiled and transformed to ease the exploitation of parallelism. When frequently-executed algorithm kernels are identified during profiling, the application is partitioned into two parts to be executed separately on the CGA and the VLIW. The generation of the code is dependent on a specific instance of the ADRES architecture, thus the architecture abstraction is used as a reference of mapping constraint during the compilation. The generated binary code is then verified with various architecture-dependent simulators.

The architecture design flow starts with an architecture generator programmed in PHP, which is a general-purpose scripting language. The PHP generator reads a highly abstract user-defined architecture specification and generates an XML architecture description, which includes detailed descriptions of each functional unit and communication device. The XML architecture description is then parsed into several simulation models by several tools, including a Register-

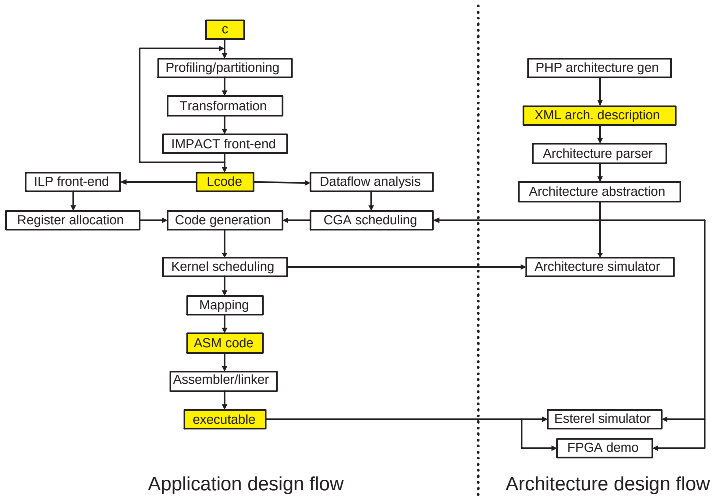


Figure 4.2: DRESC, the ADRES compilation flow

Transfer Level (RTL) simulation model generator. After the application compilation tool chain generates the program binary code, it is loaded into the memory modules of the simulation models for functional verification. The correctness of the design can be verified by downloading the synthesized RTL model into an FPGA.

Earlier study on single-threaded ADRES is based on the MPEG2 decoder. We have observed [73] that most MPEG2 decoder kernels can be scheduled on the CGA with the IPC ranging from 8 to 43. Some of the most aggressive architecture instances have the potential to execute 64 instructions per clock cycle, but few kernels can utilize this level of parallelism, resulting in a much lower average IPC. This is caused by two reasons: (1) The inherent ILP of the kernels are low and cannot be increased efficiently even with loop unrolling, or the code is too complex to be scheduled efficiently on so many heterogeneous units due to certain resource constraints, for example the number of memory ports. (2) The CGA is idle when executing sequential code in VLIW mode. The more sequential code is executed, the lower the achieved application's average IPC, and in turn, the lower the CGA utilization. This is commonly known as Amdahl's law [14]. In conclusion, even though the ADRES architecture is highly scalable, we are facing the challenge of getting more parallelism out of many applications, which fits better to be executed on smaller ADRES arrays.

Knowing that the instruction-level and the loop-level parallelism (LLP) explorations have their limitation, we need a new strategy to increase the utilization of the ADRES. The ADRES architecture has a large amount of FU, thus the most appealing approach for increasing the application parallelism would be to employ simultaneous multi-threading (SMT) [32] and exploit the application's task-level parallelism. Such architecture for the domain of supercomputing has been developed at UT Austin [20]. However, as the ADRES implementation applies static scheduling due to low-power requirements, such dynamic/speculative [13, 83] threading is not practical. Instead, our threading approach identifies an application's coarse-grained parallelism based on static analysis.

If properly reorganized and transformed at programming time, multiple kernels in the same application can be efficiently parallelized by the application designer. We can statically identify the low-LLP kernels through profiling, estimate the optimal choice of ADRES array size for each kernel, and partition a large ADRES array into several small-scaled ADRES sub-arrays that fits each kernel, which is parallelized into a thread if possible. When an application is executed, a large ADRES array can be split into several smaller sub-arrays for executing several low-LLP kernels in parallel. Similarly, when a high-LLP kernel needs to be executed, sub-arrays can be unified into a large ADRES array. Such a multi-threaded ADRES (MT-ADRES) is highly flexible, and can increase the overall utilization of large-scaled ADRES arrays when the LLP of application is hard to explore.

As discussed earlier, the DRESC tool chain is highly robust and complicated, thus having the complete DRESC updated for threading is not a trivial task. We propose how the threading can be augmented to the DRESC tool chain and presents a demonstrative dual-threading experiment on the MPEG2 decoder implemented on top of the current single-threaded architecture and its matching compilation tools. Through this experiment, we have proven that the multithreading is feasible for the ADRES architecture, and that the scalability of ADRES can be greatly improved.

Previously, a superscalar processor loosely coupled with a reconfigurable unit has been presented in [99]. Their approach enables the superscalar processor and the reconfigurable unit to execute different threads simultaneously. The CUSTARD architecture presented in [29] has a tighter coupling between the RISC and the reconfigurable unit. Their architecture and tools support the block/interleaved multithreading. To our knowledge, the parallelism supported by the reconfigurable unit has not been exploited by threading so far, and the MT-ADRES is the first attempt to support SMT both on the processor and the reconfigurable unit.

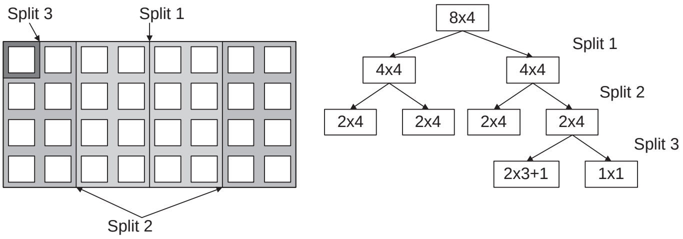


Figure 4.3: Scalable partitioning-based threading

4.2 ADRES Multithreading

We propose a scalable partitioning-based threading approach for ADRES, as shown in Figure 4.3. A large ADRES instance can be partitioned into two or more sub-arrays, each of which can be viewed as a down-scaled ADRES architecture and be partitioned further down hierarchically. Such a flexible threading strategy enables the designers to experiment on the combination of threading and partitioning in one compilation/synthesis environment, hence can be integrated into the DRESC tool chain. This technique can serve as a template for future FPGA-like platforms.

In practice, each thread has its own resource requirement. A thread that has high fine-grained parallelism requires more computation resources, thus executing it on a larger partition results in the optimal use of the ADRES array and vise versa. A globally optimal application design demands that the programmer knows the IPC of each part of the application, so that he can find an efficient array partition for each thread. A programmer starts from a single-threaded application and profiles it on a large single-threaded ADRES. From the profiling results, kernels that has low IPC and are less dependent to the other kernels are identified as the high-priority candidates for threading. Depending on the resource demand and the dependency of the threads, the programmer statically plans on how and when the ADRES should split into partitions during application execution.

4.2.1 Architecture Design Aspects

The FU array on the ADRES is heterogeneous. There exists dedicated memory units, special arithmetic units and control/branch units on the array that constrain the partitioning. When partitioning the array, we have to guarantee that the thread being executed on certain partition can be mapped. This requires that any instruction invoked in a thread to be supported by at least one of the functional unit in the array partition. The well-formed partitions usually have at least one VLIW FU that can perform branch operation, one FU that can perform memory operation, several arithmetic units if needed, and several FUs that can handle general operations. Optimally partitioning the ADRES also requires a good understanding about the kernel, so that the partition can offer enough parallel resources to match the kernel's demand.

On the ADRES architecture, the VLIW register file (RF) is a critical resource that can not be partitioned easily. The most recent ADRES architecture employs a clustered register file that has previously been adapted in VLIW architectures [106, 22]. If we prohibit the RF bank to be shared among several threads, the RF cluster can be partitioned with the VLIW/CGA, and the thread compilation can be greatly simplified. In case a single global RF is used, the register allocation scheme must be revised to support the constrained register allocation, as suggested in our proof-of-concept MPEG2 experiments.

The ADRES architecture requires ultra-wide memory bandwidth. Multi-bank memory has been adapted to recent ADRES architecture [74], and proven to cope nicely with the static data-allocation scheme used in DRESC. On ADRES, the memory and the algorithm core are interfaced with a crossbar with queues. Such a memory interface offers a scratchpad style of memory presentation to all the load/store units, thus the multi-bank memory can be used as a shared synchronization memory.

Besides the shared memory, other dedicated synchronization primitives like register-based semaphores or pipes can also be adapted to the ADRES template. These primitives can be connected between pairs of functional units that belongs to different thread partitions. Synchronization instruction can be added to certain functional units as intrinsics, which is supported by the current DRESC tools.

In the single-threaded ADRES architecture, the program counter and the dynamic reconfiguration counter is controlled by a finite-state-machine (FSM) type control unit. When implementing the multithreading ADRES, we need an extendable control mechanism to match our hierarchically partitioned array. As shown in Figure 4.4, we duplicate the FSM type controller and organized the controllers in a hierarchical manner. In this multithreading controller, each possible partition is still controlled by an FSM controller, but the control path is extended with two units called merger and bypasser. The merger and the bypasser form a hierarchical master-slave control that is easy to manage during program execution. The merger path is used to communicate change-of-flow information to the master controller of a partition, while the bypasser propagates the current PC or configuration memory address from the master to all slaves

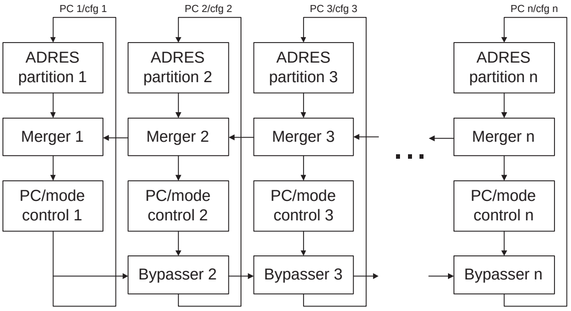


Figure 4.4: Hierarchical multithreading controller

within a partition.

The principle of having such a control mechanism is as follows. Suppose we have an ADRES architecture that can be split into two halves for dual threading, while each half has its own controller. In order to reuse the controllers as much as possible, we need each controller to control a partition of the ADRES when the program is running in dual threaded mode, but also prefer one of the controller to take full control of the whole ADRES when the program is running in the single-threaded mode. By assigning one of the controller to control the whole ADRES, we created the master. When the ADRES is running in the single-thread mode, the master controller also receives the signal from the slave partition and merge them with the master partition's signal for creating global control signal. At the same time, the slave partition should bypass any signal generated from the local controller and follow the global control signal generated from the master partition. When the ADRES is running in the dual-threaded mode, the master and slave controller completely ignores the control signals coming from the other partition and only responds to the local signals. This strategy can be easily extended to cope with further partitioning.

4.2.2 Multithreading Methodology

The multithreading compilation tool chain is extended from the existing DRESC compiler[76]. With several modifications and rearrangements, most parts of the original DRESC tool chain, e.g. the IMPACT frontend, DRESC modulo scheduler, assembler and linker can be reused in our multithreading tool chain. The most significant modifications are made on the application programming, the architecture description and the simulator.

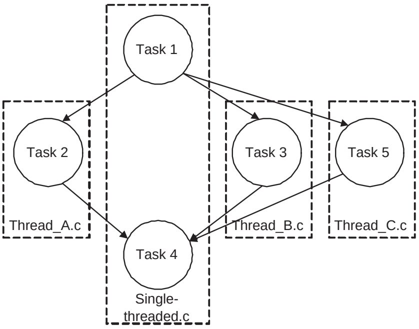


Figure 4.5: Source code reorganization

Before the threaded application can be compiled, the application needs to be reorganized. As shown in Figure 4.5, the application is split into several C-files, each of which describes a thread that is executed on a specific partition, assuming the application is programmed in C. The data shared among threads are defined in a global Header file that is included in all the thread C-files, and protected with synchronization mechanism. Such reorganization takes modest effort, but makes it easier for the programmer to experiment on different thread/partition combinations to find the optimal resource budget in the DRESC environment.

The multithreading ADRES architecture description is extended with the partition descriptions, as shown in Figure 4.6. The partition description is a list of functional units, interconnection nodes and memory elements. Similar to the area-constrained placement and routing on the commercial FPGA, when a thread is scheduled and mapped on an ADRES partition, the instruction placement and routing is constrained by the partition description. The generated assembly code of each thread goes though the assembling process separately, and gets linked in the final compilation step.

As in the original DRESC tool chain, the simulator reads the architecture description and generates an architecture simulation model before the application simulation starts. As shown in Figure 4.4, each partition has its own controller, thus the generation of the controller's simulation model depends on the partition description as well. Furthermore, the control signal distribution is also partition-dependent, thus requires the partition description to be consulted during the simulation model generation.

Some other minor practical issues needs to be addressed in our multithreading methodology. The most costly problem is that different partitions of the ADRES are conceptually different ADRES instances, thus a function compiled for a

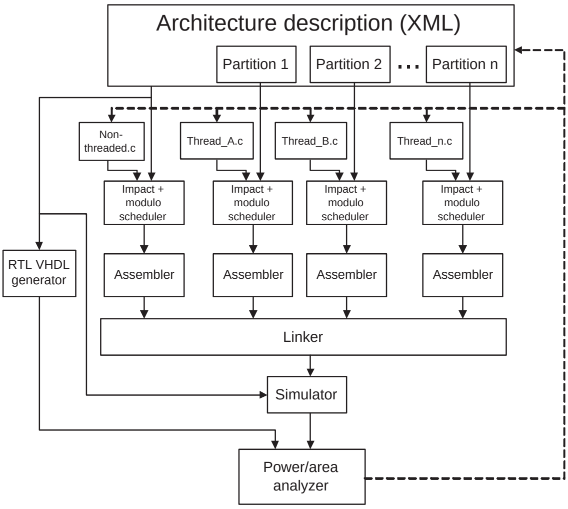


Figure 4.6: Multithreading compilation tool chain outline

specific partition cannot be executed on any other partitions. When a function is called by more than one thread, multiple partition-specific binaries of this function has to be stored in the instruction memory for different caller. Secondly, multiple stacks need to be allocated in the data memory. Each time the ADRES splits into smaller partitions due to the threading, a new stack need to be created to store the temporary data. Currently, the best solution to decide where the new stack should be created is based on the profiling, and the thread stacks are allocated at compile time. And finally, each time the new thread is created, a new set of special purpose registers needs to be initialized. Several clock cycles are needed to properly initial the stack points, the return register, etc. immediately after the thread starts running.

4.3 Experiment

As discussed earlier, the DRESC tool chain is a complicated co-design environment. In order to understand what feature is needed in future DRESC tool chain for supporting the multi-threaded methodology and prove its feasibility, we carried out an experiment based on the MPEG2 decoder, our well-understood benchmark. Our objective is to go through the whole process of generating the threaded application executable, partitioning the instruction/data memory for threads, upgrading the cycle-true architecture simulation model and successfully simulating the execution of MPEG2 decoder with our simulator. By

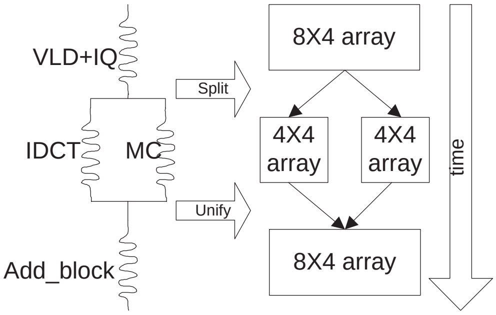


Figure 4.7: Threading scenario on MPEG2 decode

going through the whole process, we can acquire ample knowledge on how to automate the compilation for threads and simulation/RTL model generation of MT-ADRES.

4.3.1 Application and Methodology

Our proof-of-concept experiment achieves dual-threading on the MPEG2 decoder. The MPEG2 decoder can be parallelized on several granularities [51], thus is a suitable application to experiment on. We choose the Inverse Discrete Cosine Transform (IDCT) and Motion Compensation (MC) as two parallel threads, and reorganized the MPEG2 decoder as shown in Figure 4.7. The decoder starts its execution on an 8x4 ADRES, executes the Variable Length Decoding (VLD) and Inverse Quantization (IQ), then switches to the threading mode. When the thread execution starts, the 8x4 ADRES splits into two 4x4 ADRES arrays and continues on executing the threads. When both threads are finished, the two 4x4 arrays unify and continue on executing the add block function. We reorganized the MPEG2 program as described in Figure 4.5, and added “split” and “unify” instructions as intrinsics. These intrinsic instructions are only used to mark where the thread mode should change in the MPEG2’s binary code. These marks are used by the threading control unit at run time for enabling/disabling the thread-mode program execution.

The current dual-threading compilation flow is shown in Figure 4.8. The lack of partition-based scheduling forces us to use two architecture descriptions as the input to the scheduling. The 8x4 architecture is carefully designed so that the left and the right halves are exactly the same. This architecture is the execution platform of the whole MPEG2 binary. We also need a 4x4 architecture, which is a helping architecture that is compatible to either half of the 8x4 array. This

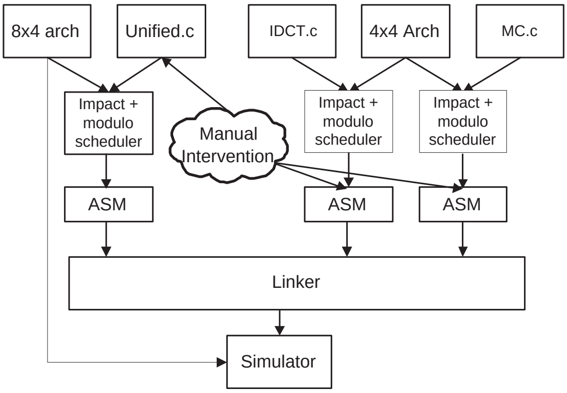


Figure 4.8: Experimental Dual-threading compilation flow

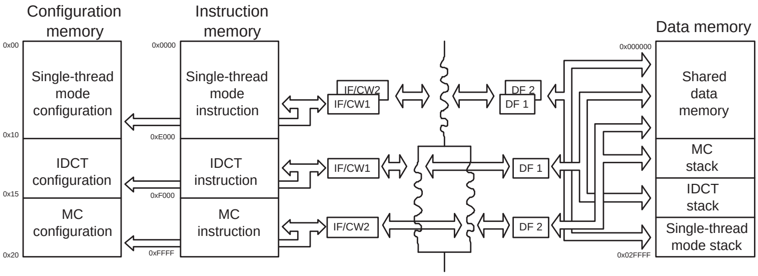


Figure 4.9: Dual-threading memory management

architecture is used as a half-array partition description of the 8x4 architecture. With these two architectures in place, we compile the single-threaded C-file and the threads on the 8x4 architecture and the 4x4 architecture, respectively. The later linking stitches the binaries from different parts of the program seamlessly.

4.3.2 Memory and Register File Design

The memory partitioning of the threaded MPEG2 is shown in Figure 4.9. The instruction fetching (IF) unit, data fetching (DF) unit and the configuration-word fetching (CW) unit have been duplicated for dual-threading. The fetching unit pairs are step-locked during single-threaded program execution. When the architecture goes into the dual-threading mode, the fetching unit pairs split up into two sets, each of which is controlled by the controller in a thread partition.

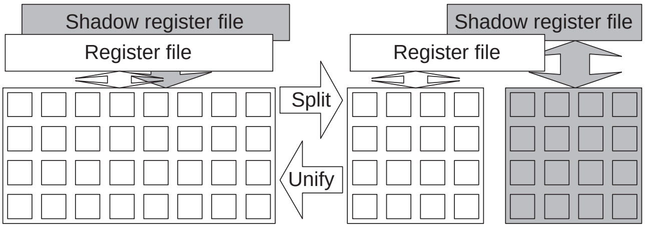


Figure 4.10: Shadow Register file setup

During the linking, the instruction memory and data memory are divided into partitions. Both the instruction and configuration memory are divided into three partitions. These three partition pairs store the instructions and configurations of single-threaded binaries, IDCT binaries and MC binaries, as shown on Figure 4.9. The data memory is divided into four partitions. The largest data memory partition is the shared global static data memory. Both the single-threaded and the dual-threaded program store their data into the same shared memory partition. The rest of the data memory is divided into three stacks. The IDCT thread's stack grows directly above the single-threaded program's stack, since they uses the same physical controller and stack pointer. The base stack address of the MC thread is offset to a free memory location at linking time. When the program execution goes into dual-threading mode, the MC stack pointer is properly initialized at the cost of several clock cycles.

In the future, we aim at partitioning the clustered register file among the array partitions so that each thread has its own register file(s). However, due to the lack of a partitioning-based register allocation algorithm at the current stage, the partitioning approach is not very feasible. We experiment on the ADRES architecture with a single global register file and go for the duplication based approach to temporary accommodate the register file issue. As shown in Figure 4.10, a shadow register file has been added into the architecture. When the single-threaded program is being executed, the shadow register file is step-locked with the primary register file. When the program initiate the dual-thread execution, the MC thread gets access to the shadow register file and continues the execution on the array partition and shadow register file. When the program resume to the single threaded execution, the shadow register file become hidden again. The MPEG2 program is slightly modified so that all the data being shared between threads and all the live-in and live-out variables are passed through the global data memory.

4.3.3 Control mechanism design

The scalable control concept in Figure 4.4 has been verified in our simulation model. By having our control scheme tested on the dual-threading, we are positive that this scheme can be extended to a certain scale, and the control unit simulation model generation can be automated.

During the program linking, we identify where the "split" and "unify" instructions are stored in the instruction memory. These instructions' physical addresses mark the beginning and the ending point of the dual-threading mode. During the simulation model generation, these instructions' addresses are stored in a set of special-purpose registers in a threading control unit. After the program starts executing, the program counter's (PC) values are checked by the the threading control unit in each clock cycle. When the program counter reach the split point, the threading control unit sends control signals to the merger and bypasser to enable the threading mode. After the program goes into the threaded mode, the threading controller waits for both threads to join in by reaching the PC value where the "unify" instructions are stored. The first thread that joins in will be halt till the other thread finish. When the second thread eventually joins in, the threading control unit switch the ADRES array back to single-threaded mode, and the architecture resumes to the 8x4 array mode. The overhead of performing split and unify operation mainly comes from executing several bookkeeping instructions on some special-purpose registers, and such overhead is negligible.

When an application gets more complicated and has multiple splitting/unifying point, the current approach will become more difficult to manage, thus the future architecture will only rely on the instruction decoding to detect the "split" and "unify" instructions. The threading control unit will be removed in the future, and part of its function will be moved into each partition's local controller.

4.3.4 Simulation result

The simulation result shows that the threaded MPEG2 produces the correct image frame at a slightly faster rate. Table 4.1 shows the clock count of the first 5 image frames decoded on the same 8x4 ADRES instance with and without threading. The **cc count** column shows the clock cycle count of the overall execution time when an image frame is decoded, while the **decoding time** column shows the clock cycle count between two frames are decoded. The dual-threaded MPEG2 is about 12-15% faster than the single-thread MPEG2 for the following reasons.

frame number	single-thread cc count	dual-thread cc count	single-thread decoding time	dual-thread decoding time	speed-up
1	1874009	1802277			
2	2453279	2293927	579270	491650	15.1%
3	3113150	2874078	659871	580151	12.1%
4	3702269	3374421	589119	500343	15.1%
5	4278995	3861978	576726	487557	15.5%

Table 4.1: Clock cycle count of single and dual threaded MPEG2 on the same architecture

Both IDCT and MC algorithm have high loop-level parallelism, thus can optimally utilize the single-threaded 8X4 architecture. When scheduled on the 4x4 architecture as threads, the IPCs of both algorithms are effectively reduced by half due to the halved array size, thus the overall IPCs of the non-threaded and the threaded MPEG2 are nearly the same. As mentioned earlier, when the ADRES' size is increased to certain extend, the scheduling algorithm has difficulty exploring parallelism in the applications and using the ADRES array optimally. It is clear that doubling/quadrupling the size of the ADRES array or choosing low-parallelism algorithm for threading will result in more speed-up.

As mentioned earlier, when one of the threads finishes its execution, it has to wait until the other thread to finish before the architecture can unify. This implies that during the waiting period, only half of the array is being used. This can significantly reduce the performance if not properly dealt with. The IDCT and the MC happens to have very close execution time when running on the 4x4 partition, so the penalty is not noticeable. In case the execution time of the two threads are not very close, one can balance the execution time by unevenly partitioning the ADRES instance and mapping the thread with the longer execution time onto the larger partition.

As we have observed, the marginal performance gain is mostly achieved from the ease of modulo-scheduling on the smaller architecture. When an application is scheduled on a larger CGA, many redundant instructions are added into the kernel for routing purpose. Now the IDCT and MC kernels are scheduled on a half-CGA partition instead of the whole ADRES, even if the overall IPC of the application is not significantly improved, the amount of redundant instructions added during scheduling for placement and routing purpose has been greatly reduced.

Kernel	Algorithm	Avg. execution time(%)	4x4 ADRES Execution time(kcc/frame)	8x4 ADRES Execution time(kcc/frame)
Fast IDCT Saturate	IDCT	16.3	139.7	87.6
Form component prediction	MC	12.6	133.3	69.2
Clear block Dequantize non-intra block Dequantize intra block	VLD+IQ	50.5	398.8	266.1
Add block	Add block	14.4	129.9	76.5
Total:	MPEG2_dec	93.8	801.1	499.4

Table 4.2: Execution time break-down of the MPEG2 decoder on 4x4 and 8x4 architectures

4.4 Discussion

Our threading experiment shows that the speedup is marginal, due to the parallel nature of the IDCT algorithm and the motion compensation algorithm. Whether the performance of the MPEG2 decoder can be further improved on an 8x4 architecture will not depend on these two algorithms, but somewhere else. We profiled the whole MPEG2 decoder on various configurations of 4x4 and 8x4 architectures to gain a better understanding of the MPEG2 decoder.

The statistics of our study is listed in table 4.2. The first column of the table lists all the kernels that contribute significantly to the overall execution time of the decoder. The second column points out in which part of the MPEG2 decoding algorithms each kernel is executed. The third column shows the average execution time of each algorithm in percentile. The last two columns lists the average execution time of each algorithm when the decoder is profiled with a image frame stream on the 4x4 or the 8x4 ADRES instances, respectively, in the unit of kilo clock cycle per frame. The bottom row is a summary of the overall MPEG2 decoder performance. The kernels take up 93.8% of the whole MPEG2 decoder's execution time, so the non-kernel part of the decoder only takes up 6.2% of the overall execution time, thus is ignored from our study.

Table 4.2 shows that the execution time of the VLD+IQ algorithm is 266.1kcc/frame on the 8x4 ADRES instance. When the same algorithm is executed on the 4x4 ADRES, we expect that the execution time to be doubled since the array size is reduced by half. But in practice, the VLD and the IQ algorithm are not very easy to parallelize on the 8x4 ADRES due to their sequential nature, thus the execution time of the VLD+IQ kernels is only $(\frac{398.8}{266.1}\% - 100\%) = 50\%$ longer when running on the 4x4 array. This indicates that the VLD+IQ part of the MPEG2 decoder uses the 4x4 ADRES array better than it uses the 8x4 array,

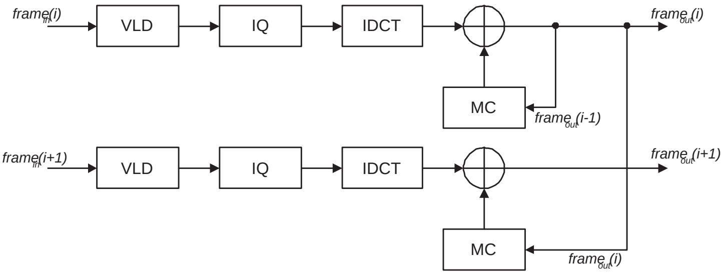


Figure 4.11: Source code transformation for alternative MPEG2 decoder threading startegy

thus is a better candidate for threading than the IDCT/MC algorithms. Also, the average execution time of the VLD+IQ part takes up 50% of the overall execution time of the whole MPEG2 decoder, thus has the greatest potential to make impact on the overall performance. Unfortunately, because the execution of the VLD and the IQ is sequential, this part of the MPEG2 algorithm is hard to parallelize.

To make use of the potential of the VLD and the IQ algorithm, the structure of the MPEG2 decoder has to be revised. An alternative way of restructuring the MPEG2 decoder is shown in figure 4.11. Instead of decoding the video stream on a frame-to-frame base, the frames are decoded in pairs. The two instances of VLD+IQ can be easily parallelized into threads, thus improves the overall decoding efficiency. The motion compensation of the second frame is dependent on the previous frame, thus the MC of the second frame can not be started before the MC of the first frame is finished. But since the IDCT and the MC of the first frame can be parallelized and use the dual-threaded 8x4 ADRES efficiently, the dependency between the two MC instances doesn't impose any performance penalty.

As shown in the profile results in table 4.2, decoding one frame with the 8x4 ADRES takes around 499.4kcc/frame, and decoding a frame with the 4x4 ADRES takes around 801.1kcc/frame. If the strategy in figure 4.11 is implemented, we anticipate that the overall performance of the decoder should be $100\% - \frac{801.1}{499.4 \times 2}\% = 20\%$ faster with no added hardware or energy cost. The latency to produce the first image frame will be slightly increased, but more importantly, the average decoding throughput will also increase.

4.5 Conclusions and future work

By carrying out the dual-threading experiment on MPEG2 decoding algorithm, we have gained ample knowledge on the MT-ADRES architecture. The simulation results show that the MPEG2 has gain 12-15% of speed up, and has potential to gain another 8% when more effort is put into source code transformation. We are convinced that more speedup can be gained in future benchmarking, when full tool support permits the optimization of the template and the software architecture. The results so far demonstrate that our threading approach is adequate for the ADRES architecture, is practically feasible, and can be scaled to a certain extend. So far, the only extra hardware cost added onto ADRES is a second control unit, the size of which can be neglected for an ADRES larger than 3X3. Based on the success of our proof-of-concept experiments, we have very positive view on the future of MT-ADRES.

However, even if threading can improve the scalability of the datapath-coupled reconfigurable architectures, it is not always an easy job to find out which parts of an application can be parallelized. Tasks being selected as threads need to have low instruction-level and loop-level parallelism, and are preferred to have no dependencies to the other tasks being chosen for threading at the same time. The selection of tasks is not always intuitive, and complicated source code transformation is needed. For complicated applications, it is not easy to investigate the characteristics of kernels, thus having an automated tool to assist in profiling and structuring the application is an urgent need for future work.

Chapter 5

COSMOS: A System-Level Modelling and Simulation Framework for Coprocessor-Coupled Reconfigurable Systems

One of the biggest challenges in reconfigurable system design is to improve the rate of reconfiguration at run-time by reducing the reconfiguration overhead. Such overhead comes from multiple sources, and without proper management, the flexibility of the reconfiguration can not justify the overhead cost. Many new technologies and designs for minimizing the reconfiguration overhead have been proposed. Logic granularity [75, 71], host coupling [23], resource management [92, 93] etc. have been studied in various contexts. These technologies substantially increase the practicality of the reconfigurable systems, but also often lead to highly complicated system behavior. There exists several highly efficient architectures, but many of them have significant drawbacks in terms of programmability, flexibility, scalability or utilization rate.

Even though low-level technologies have drawn a lot of attention, the study on system-level behavior and compilation is still in their infancies. It is known as a rule-of-thumb that high level design decisions made earlier in the design process can have higher impact on the system performance. But currently, the evaluation of applications executing on a reconfigurable system in the early development stages is a new challenge to be addressed.

In this context, a key issue is to understand the real-time dynamic behavior of the application when executed on the run-time reconfigurable platform. This is a very complicated task, due to the often very complicated interplay between the application, the application mapping, and the underlying hardware architecture. However, understanding the real-time dynamic behavior is critical in order to determine the right reconfigurable architecture and a matching optimal on-line resource management policy, given a specific application. Although architecture selection and application mapping have been studied intensively, they have not been thoroughly studied in the context of run-time reconfigurable system. Not only do we need to understand the real-time dynamic behavior of these systems, we also need to understand the importance of understanding this behavior, i.e. which aspects of the dynamic behavior should we capture in order to derive efficient solutions.

For datapath-coupled architectures [65, 76], reconfigurable unit (RU) is frequently designed as a special instruction-set functional unit or extended to a large-scale VLIW processor, thus the application can be efficiently evaluated with instruction-level simulation. However, coprocessor-coupled architectures, which are usually large-scaled and highly complicated, need advanced run-time resource management support. Hence, to improve the system efficiency, we need to be able to model and analyze such systems' architecture, run-time system and the applications running on them.

In this chapter we present COSMOS, a flexible framework to model and simulate coprocessor-coupled reconfigurable systems. First we propose a novel real-time task model that captures the characteristics of dynamically reconfigurable systems' task in terms of initialization, reconfiguration and reallocation. We also propose a general model of coprocessor-coupled reconfigurable systems. The task and architecture models are based on an existing MPSoC simulation model, ARTS [69], which has been extended by us to facilitate the study of run-time resource management strategies. We demonstrate how a simple "worst case" run-time system can be modelled in the COSMOS framework as a firmware to manage the application execution.

Then we use the COSMOS model to experiment on various combinations between the application and the architecture to gain a better understanding of the emerging critical issue in reconfigurable architecture design. We present the results of a set of experiments that are carried out on a MP3 task graph. We study how the numbers of RU, the sizes of RUs, the number of reconfiguration contexts and the granularity of RUs impact the run-time behavior of the system. We also address how more advanced run-time system design, especially the task allocation and reallocation, can impact the system performance. We propose several reallocation strategies, and study their effectiveness through several simulations.

Finally, we discuss how the COSMOS framework can be improved in the future and conclude our work.

5.1 Background

During a reconfiguration, reconfigurable architectures suffers from latencies due to the context switching (configuration and intermediate data) of an RU. The severity of this latency is determined by several physical factors, e.g. the scale of the RU, the logic granularity, the configuration memory bandwidth, the rate of reconfiguration or the buffering technics of reconfiguration memory fetching. In the following we will give an overview of the related research areas that can reduce such latencies, and discuss how they affect system behavior.

One research trend assume that the applications, or a collection of tasks, share the RU in time, as shown in Figure 5.2A. [98] proposed a multi-context FPGA that can significantly reduce the reconfiguration time, but the extra cost of chip area is hardly justifiable by the performance gain. A solution that can substantially reduce the area overhead is to increase the logic granularity of the RU to medium- or coarse-grained, as shown in Figure 5.1. Even if these higher-granularity architectures do not offer highly optimal solutions to applications that heavily exploit bit-level data manipulations, the concept of multi-context is proved feasible. But still, the number of contexts being cached on the RU is usually limited, and optimal utilization of the limited context resource at run-time is a difficult challenge for a multi-tasking system [63].

Another type of reconfigurable architecture assume that the RU is shared in space [11, 92], as shown in Figure 5.2B. The RU is partially reconfigurable and large-scaled, thus several tasks can be run on the same RU with no conflicts among each other. Besides reconfiguration latency, this class of architectures leads to complicated inter-task communication and resource management. Since a task can be allocated on an RU at any free location during run-time, data traffic between tasks go through multiple possible paths, maybe requiring dynamic routing. For a large programmable array, the complexity of performing the task placement and data routing at run-time can be very hard to handle. Also, it

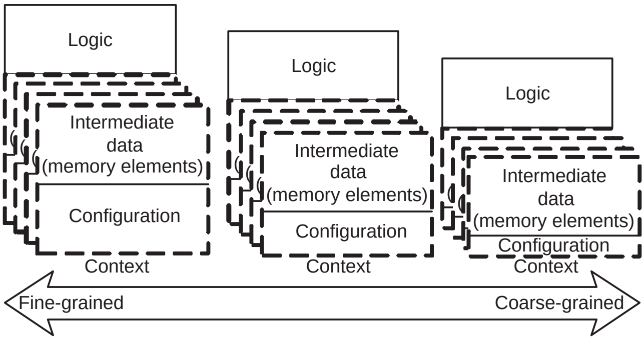

Logic
Intermediate data (memory elements)
Configuration
Context
Logic
Intermediate data (memory elements)
Configuration
Context
Logic
Intermediate data (memory elements)
Configuration
Context
Fine-grained
Coarse-grained

Figure 5.1: The impact of logic granularity on the chip area of reconfigurable architectures.

is clear that the fragmentation is a common issue for this kind of design, thus task (context) reallocation and rerouting is consistently required for defragmentation. In summary, the behavior and efficiency of such system can be very unpredictable, and understanding the system behavior in the early development stage is crucial.

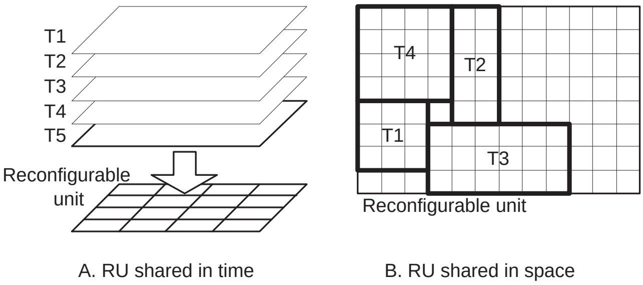


Figure 5.2: Reconfigurable unit design

A third type of RU is a hybrid of the two former families. This type of architectures is viewed as an array of networked multi-context RUs. Such system also requires efficient dynamic resource management, but the routing problem is greatly simplified compared to space-shared architectures. Nollet et al. [82] proposed an architecture that resembles several heterogeneous reconfigurable units (RU) being interconnected with an on-chip network (NoC). They use a hierarchical control scheme to efficiently manage the computation resources at run-time, so that the architecture can be extended to a large scale. Since the RUs are assumed to be heterogeneous, the resource management can still be very time-consuming to perform at run-time, resulting in large run-time overheads.

In general, we are facing the increasing complexity of the reconfigurable system's spatial and temporal behavior. New technologies that improve the system's efficiency also complicates the architecture, and the value of the tradeoff between performance and design complexity is not easily assessable. A system-level simulator is much needed for evaluating the performance of dynamically reconfigurable systems. Such a simulator should give the designer the opportunity to tune various design parameters and to study the consequences on system performance.

To build a system-level simulator, we need a thorough understanding on how to model the tasks which comprise the application. A task running on the reconfigurable architecture has a different execution behavior than the classical real-time task, thus the classical model does not fit our purpose. For the simulator design, we need i) a general and generic model of the RU which can represent various types of coprocessor-coupled RU designs, ii) the dynamic resource management issue of the RU to be addressed, and iii) the simulation to be parameterizable so that the consequences of changes in the physical design can be captured within the model.

The ARTS modelling Framework captures real-time behavior of heterogeneous multiprocessor systems, where each processor may run its own operating system. In our work, we adopt the underlying message-passing based mechanism of the ARTS model and some of its RTOS functionality. We extend it further to support the modelling of dynamically reconfigurable systems. In particular, our model, unlike ARTS, can handle task reconfiguration and reallocation during run-time, i.e. during simulation.

5.2 Task model

In the ARTS framework, an application is modelled as a set of task graphs, and each task is modelled as a finite state machine (FSM), as shown in Figure 5.3. The state transitions of a task are driven by the operating system control messages. Whether a task should run, or be preempted, depends on the resource allocation, scheduling and task dependency. But for reconfigurable system, such an FSM is not sufficient to capture the task execution scenario.

Firstly, to initialize the task execution, the initial configuration needs to be loaded from the configuration memory to the RU. Depending on the RU's granularity, size and memory interfacing, the timing cost for fetching the whole configuration can be a big overhead. To explicitly express this task execution phase, a new state **init_config** has been added to the task model, as shown in

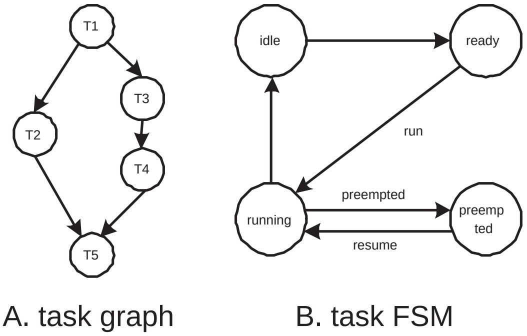


Figure 5.3: ARTS task model

Figure 5.4.

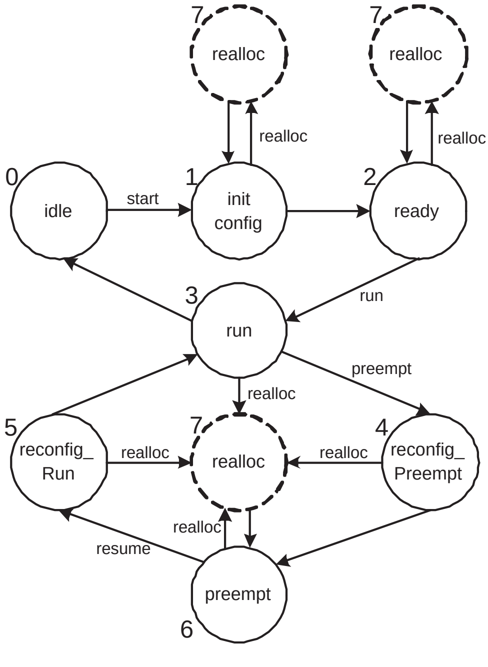


Figure 5.4: Real-time reconfigurable task model

Secondly, the preemption is not simply a process of task giving up an RU for other tasks, but is also a process of hardware context switching. Differ from the software context switching, which mostly involves backing up special-purpose registers and bookkeeping the operating system management entries, hardware context switching needs to back up the configuration and all the intermediate data stored in all the memory elements. The timing cost of the preemption can be extremely high if the context is stored in the external memory, or as low as a single clock cycle if the RU has multi-context support. Architecture designers would experiment on various combinations of different context storage design in order to find an optimal strategy, thus the reconfiguration latency may vary. In our model, we added two delay states, **reconfig_preempt** and **reconfig_run**, to represent the timing cost of the preemption.

Finally, the effect of process of task migrating among multiple RUs need to be modelled. As shown in Figure 5.4, we add the reallocation state **realloc** at three places and marked it with dashed circles, in order to emphasize that this single state has multiple entry points and exit points. This is modelled so because reallocation can happen anytime after a task leaves the idle state, and at different point of time, the reallocation has different effect on task execution. If the reallocation is started before a task is run for the first time, the task needs to be initialized on a different RU. In this case, the (partial) context of the reallocated task is moved to another RU, then it resumes the previous state for either continue initializing or waiting to get permission to run. If the task has been run before, then the allocation must be ended with the task going to the preempt state. The reason for such a setup is because the task doesn't know if it can continue executing after the reallocation, since the resource status of the reallocated RU is unknown. It is safe for a task to preempt itself and request resource management unit for permission to continue execution.

5.3 Coprocessor coupled architecture model

5.3.1 Architecture model outline

Our work takes the scalability of reconfigurable architectures as the most critical measure. As shown in Figure 5.2, the architecture design is heading to two directions. Besides the aforementioned resource management issue, the time-shared architectures also suffer from scalability issue, since parallelizing a task to use the full RU gets harder when RU's size increases. Similarly, the space-shared architecture's defragmentation gets harder when the RU upscales, and the rerouting becomes impossible to handle at run time. Unless the RU is partitioned and modularized, the space-shared architecture has too many practical issue to realize.

To solve the problem of both types of the reconfigurable architecture, we propose a hybrid architecture model. As shown in Figure 5.5, our coprocessor consists of an array of homogeneous multi-context RUs connected with on-chip networks (NoC). Similar to the time-shared design, the computation resource of our architecture is still the context of the RUs. By statically partitioning the applications into tasks, each of which is small enough to fit into one RU's context, or one resource, we can explore the application's parallelism at various levels and efficiently utilize the potential of the coprocessor. Such architectures are scalable not only as a single-chip solution, but multi-chip ones as well. If the off-chip communication protocols can be handled as on-chip ones, the chip boundary is indifferent for each RU, thus the coprocessor can be upscaled beyond the size of a single die.

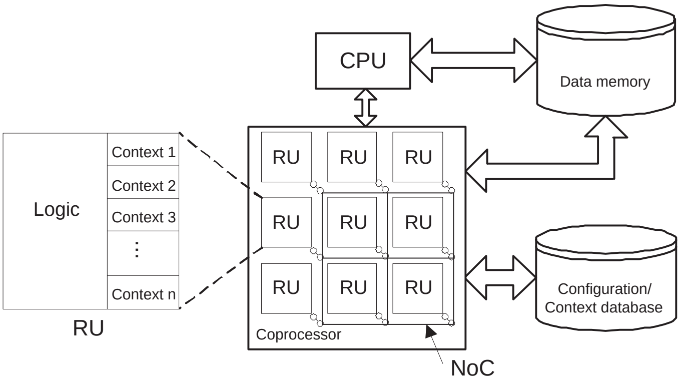


Figure 5.5: Hybrid coprocessor-coupled reconfigurable system

Each RU is a collection of memory elements and logic elements, and is the atom of the reconfiguration. The number of configuration contexts supported by the hardware is explicitly modelled as reconfiguration resources, the management of which is a critical issue to be addressed by the run-time management. The chip area composition breakdown is as following:

$$\begin{aligned} A_{total} &= A_{NoC} + \#RU * A_{RU} \\ A_{RU} &= A_{logic} + \#context * A_{context} \end{aligned}$$

Even if the architecture model appears quite simple, the dynamic system behavior is still very complicated, as discussed in our experiments.

As an architecture design, our architecture has several advantages compared to many previous designs. Our coprocessor is upscaled by increasing the number of RUs, thus has more flexibility to efficiently support applications of various complexity. With the support of the NoC, rerouting problem can be solved on the fly when the tasks are communicating. Since the coprocessor is modularized, defragmentation is not a prohibitively crucial issue as in previous space-shared design. When combined with our resource management strategy, which will be discussed later, our co-processor is highly scalable.

As a model, our model can easily be used for both time-shared and space-shared architecture PSE. To model the time-shared architectures, by assuming the number of RUs to be one, our model imitates a multi-context architecture. As to model a space-shared architecture, by assuming all the RUs to be single-context, our model can be viewed as a modularized space-shared RU. By employing a NoC and assuming that any task can be allocated to a randomly selected RU, the dynamic placement and routing issues of space-shared architecture becomes a much easier issue to address. However, the homogeneity of RUs adds an extra compile/synthesis resource constraint.

5.3.2 Assumption and simplification

Currently we assume that the RUs are homogeneous. From the compilation or synthesis point of view, homogeneous RUs demand more static analysis of the applications. For instance, when an application is partitioned into a task graph, each task has to be able to optimally utilize the computation power of the RU. The homogeneity may appear to be a cumbersome constraint for task partitioning, but the run-time management of resources can be greatly simplified. Homogeneous RUs also enable easy task reallocation, and have the great potential to support run-time fault-tolerance, which are greatly demanded by the future chip designs. We assume that a task can be reallocated to any RU with free context, as long as the lifecycle of the task is not over.

The RUs are assumed to be mono-grained. For extremely coarse-grained RUs that use the FU as logic blocks, we assume that the cost of configuration storage is neglectable, and that the number of context an RU can have is unlimited. However, we assume that the number of tasks that can be mapped onto one RU is still limited by a small number, since the run-time management of tasks is still an issue for these architectures.

We will not go into detail of the NoC model design in our work, since it has been addressed by the ARTS model described in previous work[68]. In our model, we simply assume that if two communicating tasks are k hops away on the coprocessor, the communication latency is kT, where T is the single-hop base communication latency between those tasks decided through static analysis. The overall communication latency of an application is greatly affected by the allocation strategy, which is one of the most interesting issue to be addressed by our model.

When a task is reallocated, the context of a task is transferred from one RU to another. This process results in a burst of data transfer on the NoC in a short period of time. Compared to the context transfer, inter-task communication happens much more frequently than reallocation, and data is often delivered in smaller packets. These two types of data transmission have very different requirements on the NoC design, thus we separate them into two NoCs. The reallocation NoC is assumed to be able to establish preset paths that can guarantee to finish the reallocation in a short period of time, thus the physical distance between the context transfer's source and destination should not play a significant role in the overall reallocation latency. In our model, we assume that any reallocation takes a constant period of time, and several reallocation can take place concurrently without blocking each other. Physically, the configuration data communication could share the inter-task communication NoC, but to demonstrate the concept, we choose not to do so at the moment.

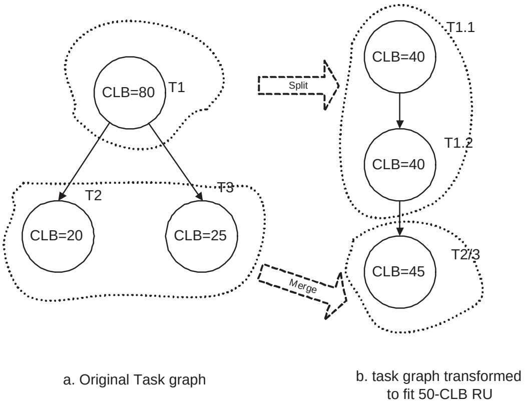


Figure 5.6: Task transformation

To ease the dynamic resource management, we assume that tasks never share one RU in space, even if several tasks can fit into one RU at the same time. This guarantees that each task can be reallocated without interfering the execution of the other tasks. Tasks that are unable to fit into one RU need to be split into smaller tasks. As shown in figure 5.6a, task T1 requires 80 CLBs to execute, but we assume that the RU can not fit a task that cost more than 50 CLBs. Task T1 has to be split into two smaller tasks that can fit into each RU, but the communication cost will increase if the partitioning is ill-performed, thus such splitting is not a trivial task. For the RUs that are underused by the tasks, e.g. task T2 and T3 in figure 5.6a, merging several tasks into one task simplify the task management. Task splitting and merging can significantly impact the overall execution time, and finding an optimal transformation is a critical static analysis issue.

5.3.3 Run-time management

The resource management is still a problem for our architecture, since the run-time system needs to manage tasks in both space and time. For a small-scaled coprocessor, the CPU/operating system can be used to manage the resource. But if the system reaches certain scale, it is foreseeable that taking a snapshot of the whole coprocessor's resource distribution, evaluating it and allocating/reallocating task by using the CPU can be a performance bottleneck. Here we introduce our alternative to address this issue.

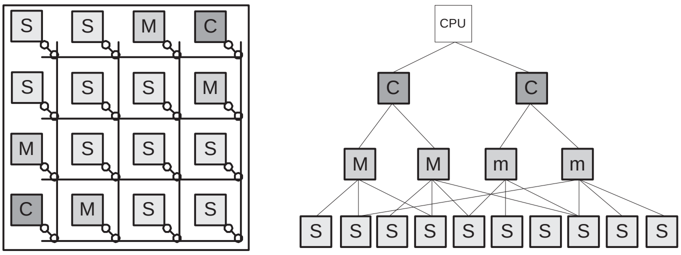


Figure 5.7: Hierarchical organization of reconfigurable units

As shown in Figure 5.7, some nodes in the coprocessor are selected as Coordinator nodes (C-nodes) or Master nodes (M-nodes), and the rest are Slave nodes (S-nodes). By structuring the whole design into hierarchies, the resource management is distributed into different roles each type of the nodes play.

C-nodes are the resource management nodes. Each time an application is started by CPU, all C-nodes send message to the lower hierarchy for resource check. Then M-nodes collect the weighted resource distribution status from S-nodes and pass it to the C-nodes. Then the C-nodes, all of which run the same decision-making protocol, select a resource-optimal M-nodes to initialize and synchronize the application's execution.

M-nodes are the task execution management nodes. After the C-nodes assign an application to an M-node, the M-node reallocates the currently running tasks to free up some resources if the new application has a higher priority. Then the M-node initializes the new application's tasks to free resources, and start its execution. During execution, depending on the task dependencies and priorities, the M-node can reallocate the tasks or preempt the task execution. C-nodes and M-nodes forms a cluster. M-nodes is only controlled by the C-nodes in its cluster, thus any message received from other C-node will be ignored.

S-nodes are the computation units. When a task is allocated to an S-node, the task can be blocked or selected for execution, depending on its priority or deadline. The node keeps track of how many resources is currently in use and how many is still available. The multi-context S-nodes is not bounded to one specific M-nodes. As long as it gives optimal results, contexts on a S-node can be shared among all M-nodes.

The tasks of a certain application are distributed on the S-nodes near one M-node selected by the C-nodes. The higher priority an application has, the more effort the M-nodes will put into to cluster its tasks, in order to lower the communication cost. Lower priority applications' clusters can be disrupted by the M-nodes when a new application with higher priority is started. Careful placement of clusters can help achieve overall system optimality, thus is crucial in our approach.

Our hierarchical design represents our general resource management strategy, but we don't enforce a physical bounding between the function of a resource/task management unit and a RU, except for the S-nodes. For instance, when the coprocessor is small, the function of the C-node and M-nodes can be realized by the operating system running on CPU, or be combined into one physical RU. It gives us the freedom to model the architecture on various scales. In our experiment, priority is based on the overall communication demand of an application, but we don't constrain how task priority is defined or what allocation strategy is used. Different designer may have different preference on specific parts of the system, and we leave them open for experimentation.

Even though it is not the focus of our work, the latency of off-coprocessor data communication can be easily modelled by our framework. Data IO ports connected to the main memory can be modelled as S-nodes with no context limitation, and off-coprocessor data communication can be modelled as special tasks that can only be allocated on the S-nodes that imitates the coprocessor's IO ports. Given a set of coordinates to the ports, which is preferably on the boundary of the coprocessor, the off-coprocessor communication latency can change when the task reallocation occurs, depending on the distance between the IO port and the tasks that need access to the main memory.

5.4 System-C simulation model

The general structure of our System-C model is shown in Figure 5.8. Various types of modules are organized as mentioned in Figure 5.7 and connected with communication links defined in the System-C master-slave communication library. The links in solid line are used to convey resource allocation control messages, while the links in dashed lines are for task execution control message passing. The critical design issue of our model is to support task allocation, task execution and task reallocation.

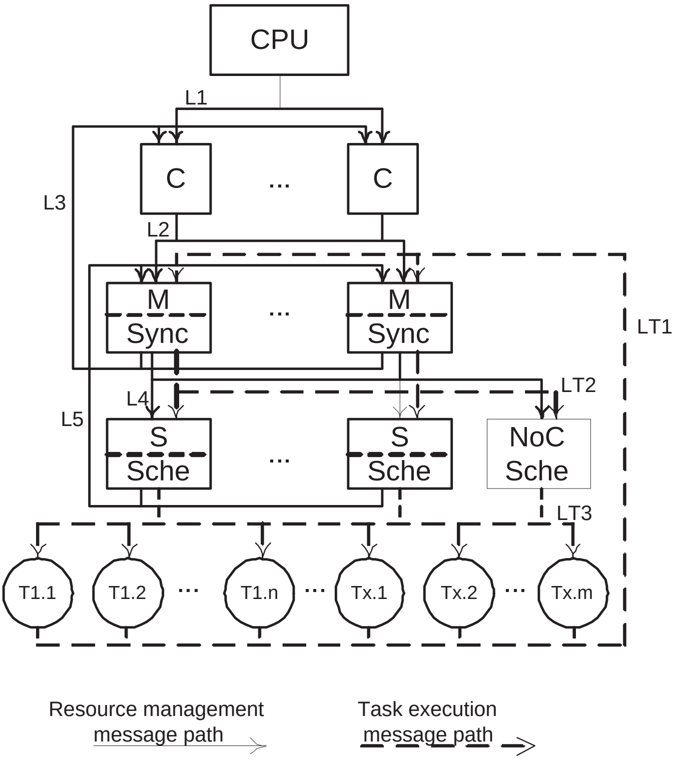


Figure 5.8: COSMOS model structure

5.4.1 Task allocation

When CPU requests to execute an application, it sends out a message that includes the header description of the application to all the C-nodes through link L1. The information contained in the header are the application's **allocation priority**, **default distribution requirement**, **distribution matrix** and the **application size**. Applications with higher **allocation priority** can force low **allocation priority** tasks to be reallocated and give up resources. The **default distribution requirement** is an integer that specifies the number of S-nodes needed for optimal allocation of an application. For instance, the task graph in Figure 5.3A will be optimally executed if it is allocated on 2 S-nodes, due to its task-level parallelism. The **distribution matrix** specifies how the tasks are divided into groups, each of which should be allocated on the same S-node. For example, the task graph in Figure 5.3A can have a **distribution matrix**= [[T1, T3, T4],[T2, T5]]. This indicates that, in order to optimally utilize the task level parallelism and minimize the communication cost, M-node should attempt to allocate task 1, 3 and 4 on one S-node, and allocate task 2 and 5 on the other one. **Application size** simply stands for how many tasks the application has been partitioned into.

Upon receiving the message that requests C-nodes to start up an application, each C-node will further send request through link L2 to the M-nodes in their clusters to exam the resource distribution. M-nodes send the request further down to S-nodes through link L4, and each S-node reports how many free context it has to M-nodes through link L5.

At this point, each M-node has an updated resource distribution map of the whole coprocessor, and needs to evaluate if the M-node itself is resource-abundant. Application is optimally allocated if its tasks are nested into cluster, thus having clustered free resources around an M-node ease the allocation for this M-node. Depending on the resource distribution around a specific M-node, resources are weighed for this M-node. Another factor that influence the allocation is the reallocation potential of each M-node. If there are many high-priority applications nested around and being controlled by an M-node, reallocation will be difficult to perform for this M-node. Thus, we sum up the priorities of all the running tasks being controlled by each M-node, and use it to downgrade the overall resource count. To summarize, each M-node weighs its resource distribution map and sums up all the weighed resource to get an overall weighed resource count, then the number is divided by the priority sum of all the running tasks.

After the M-nodes calculate their final resource count, the number is sent to the C-nodes through link L3. C-nodes then decides which M-node has the highest amount of resource available for the application. Together with the **application priority** and the **distribution matrix**, the decision is then passed through link L2 to the selected M-node for setting up the task execution.

The selected M-node first attempts to reallocate some running tasks, whose priorities are relatively lower than that of the new application, to free up some S-nodes till there is enough free clustered context to allocate the newly-started application. Then the new application's tasks are allocated to the free S-nodes with the guidance of **distribution matrix**. If the **distribution matrix** can not be strictly followed due to the resource availability, spanning to several more S-nodes is allowed. After each task is allocated onto an S-node, the M-node sends a "start" message to all these tasks through link L4, the corresponding S-node and link LT3 to signal the task execution. And finally, some bookkeeping is done in M-nodes and S-nodes.

5.4.2 Task execution control

The task execution in COSMOS is essentially the same as in ARTS multiprocessor model. Task execution is controlled through the synchronizer in the M-node and the scheduler in the S-node, as shown in Figure 5.8. In COSMOS, we adapt to the well-understood direct synchronization (DS) protocol and the earliest-deadline-first (EDF) scheduling for initial experimentation.

Task interacts with the run-time system in the similar way as ARTS tasks model. When a task is ready for execution, it sends a “ready” message to synchronizer through link LT1. When the synchronizer and scheduler permit the task execution to start, the task receives a “run” message through link LT3 and starts executing. When the task is in the “run” state, it can be preempted by the scheduler at any point of time. Similarly, when the task is in the “preempted” state, scheduler can issue “resume” message to let the task continue executing.

The synchronizer acts as a task dependency filter. Its purpose is to block the execution of those tasks that have unsolved dependencies to the preceding tasks. When a task is initialized and has requested for initial execution through link LT1, the synchronizer will immediately block the task if there are unsolved data dependencies. Every time a task finishes its execution, the synchronizer check if any blocked task’s dependency is completely solved and ready for execution. The M-node is selected to perform the synchronization since it has the control of the whole application.

When a task is released by the synchronizer, a message is sent from the synchronizer to the scheduler through link LT2. The EDF scheduler then decide if the task should start the execution on the S-node immediately or be blocked until the currently running task is finished, depending on which task’s deadline is arriving earlier. If the task released by the synchronizer has a tighter deadline compared to the currently running task, the currently running task is preempted and blocked in a task list. Once the running task has finished its execution, the blocked task that has the earliest deadline is selected for execution.

5.4.3 Task reallocation

As mentioned before, task reallocation can occur anytime between the time the task starts initialization and the end of execution. The reallocation basically involves putting the task into the reallocation state for a period of time and updating the task model's information about onto which S-node it is reallocated. The reallocation of a task goes through several different scenarios when the reallocation is initiated at different point of time.

The first possibility is the case that a task is reallocated during initialization. In this case, the task hasn't been blocked by either the synchronizer or scheduler, and the task goes back to initialization right after the reallocation is finished.

The second possible case is the situation that a task is reallocated when it is in the ready state. In this case, the task might be blocked by either the scheduler or the synchronizer. When the task goes into the reallocate state, the synchronizer or the scheduler that blocks the reallocated task need to clean up the record of blocking. When the task finishes the reallocation, the task sends the "ready" message again to the synchronizer to get processed again and goes into the "ready" state again. It is worth noting that, if a task is blocked by the synchronizer when the reallocation start, it is not necessarily true that the same task is still blocked by the synchronizer when the reallocation is finished, since the task dependency can be solved during the reallocation process.

The last possibility is the case that a task has been partially executed before reallocation. The task can only be blocked by the scheduler, or not be blocked at all. If the task is blocked by a scheduler, the scheduler also needs to clean up the record of task being blocked. After the task is reallocated, the task goes into the "preempted" state and send a "ready" message to synchronizer, which will directly pass the message to the scheduler where the task is reallocated since the task dependency has been solved before.

5.4.4 NoC model and communication tasks

The ARTS framework explicitly models the communication latency between tasks if the tasks are allocated on different processing elements. As shown in Figure 5.9, communication between tasks are treated in two different ways. A local communication inside of a processor, e.g. the dependency between T1 and T3, is assumed to have no timing cost, and the dependency is implicitly solved by the local synchronizer. But the communication between tasks allocated on different processors are transformed into communication tasks with explicit execution time, e.g. as task c1_2. A NoC scheduler is used as shown in Figure 5.9 and 5.8 to handle the communication latency and NoC scheduling strategy. The communication latencies are decided before simulation time.

In COSMOS model, since the tasks can be reallocated at simulation time, any

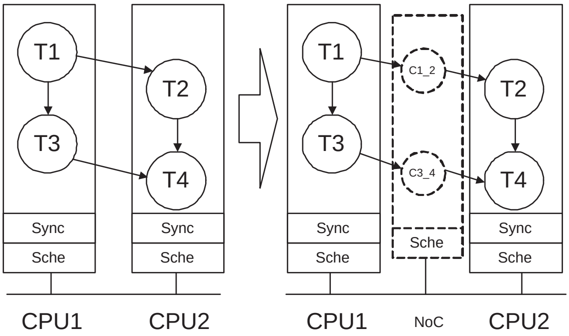


Figure 5.9: ARTS communication task

task dependency can become a communication task. Furthermore, the communicating source and destination is not fixed on the RU array, thus the physical system and the model should have a varying communication latency. In our model, we assume all the task dependencies to be a communication task, and each communication task has a base latency. Each time a task is reallocated, the communication task that is linked to the reallocated task update its communication source or destination's coordinates, depending on how the task is linked to the communication task. If a communication task's source and destination are allocated on the same S-Node, the communication will be finished in one simulation clock cycle, which is negligible. If the source and destination of a communication task are not allocated on the same S-node, the communication latency is the product of the base communication task latency multiplied by the number of hops between the source and the destination s-nodes.

5.5 Demonstrative simulation

To demonstrate the function of the model, we set up the architecture and application as shown in Figure 5.10. The architecture is a 3x3 RU array with one C-node, one M-node and seven S-nodes, each of which supports dual-context. Application 1, 2, and 3, whose task graphs are shown in Figure 5.10C, start their execution at t=T1, T2 and T3, respectively, as shown in Figure 5.10A. The application 1 is assigned a slightly earlier deadline compared to the other 2 applications for demonstrative purpose. We assume all the communication tasks to have a single-hop latency of two clock cycles, and the NoC scheduler can only handle one communication message at a time. The latencies for task initial configuration and task reallocation are assumed to be 5 cycles. The latencies of task staying in **reconfig_preempt** state and **reconfig_run** state are assumed to be 3 cycles. All the numbers presented here, including the size of architecture and various timing figures, are only for demonstration purpose and only serves the purpose of helping readers to understand the function of the model. COSMOS is a flexible model, and there is no constraints on how these number can be decided.

An optimal system's reallocation strategy should minimize the occurrence of task reallocation while keeping the overall communication overhead small. But for our experiment, in order to demonstrate the scenario of task reallocation with a simple setting, we select a reallocation strategy that is far from optimal. We define the M-node to be the only cluster center for all three applications. By doing so, we make the M-node into an allocation "hot spot," thus cause frequent reallocations.

When each application is initialized, its task will be allocated as close to the M-nodes as possible, resulting in the lower priority task to be reallocated on the S-nodes farther away from the M-node. This is achieved by weighing the resource with the distance between the S-node and the M-node during resource evaluation, selecting the most resource-optimal S-node for allocation and selecting the second-most resource-optimal S-node for reallocation.

At t=14, CPU requests to start application 1. The M-node first check the application's distribution matrix for allocation guidance. According to the application 1's distribution matrix, which suggests that task T0_0 and T0_1 should be allocated on the same RU, the M-node initialize both tasks on S-node(0,2). Task T0_2 and T0_3 are both allocated on S-node(1,1) for the same reason. After the tasks finishes the initialization and get ready for execution, only task T0_0 goes into the "run" state, since it's the only task without unsolved dependencies. All the other tasks are blocked by the synchronizer for the time being.

At t=22 the application 2 is initialized. Since the application 2 has the same priority as application 1, it does not cause any task reallocation. At t=30, application 3 starts its initialization. Since this application has a higher allocation priority, previously allocated tasks have to be reallocated to more remote S-nodes. As shown in Figure 5.10B, task T0_0, T0_1 and T0_2 are replaced by task T2_0, T2_1 and T2_2, respectively. From the waveform, we can see that the reallocated tasks enter and exit "realloc" state at the same time. Since task T0_0 is running when being reallocated, after it finishes the reallocation, it goes into "preempted" state and wait for synchronizer and scheduler to start it again, as shown in Figure 5.4. The other two reallocated tasks go back to "ready" state and wait for their dependencies to be solved.

After the reallocation, communication task c0_0_1 and c0_2_3 become non-local,

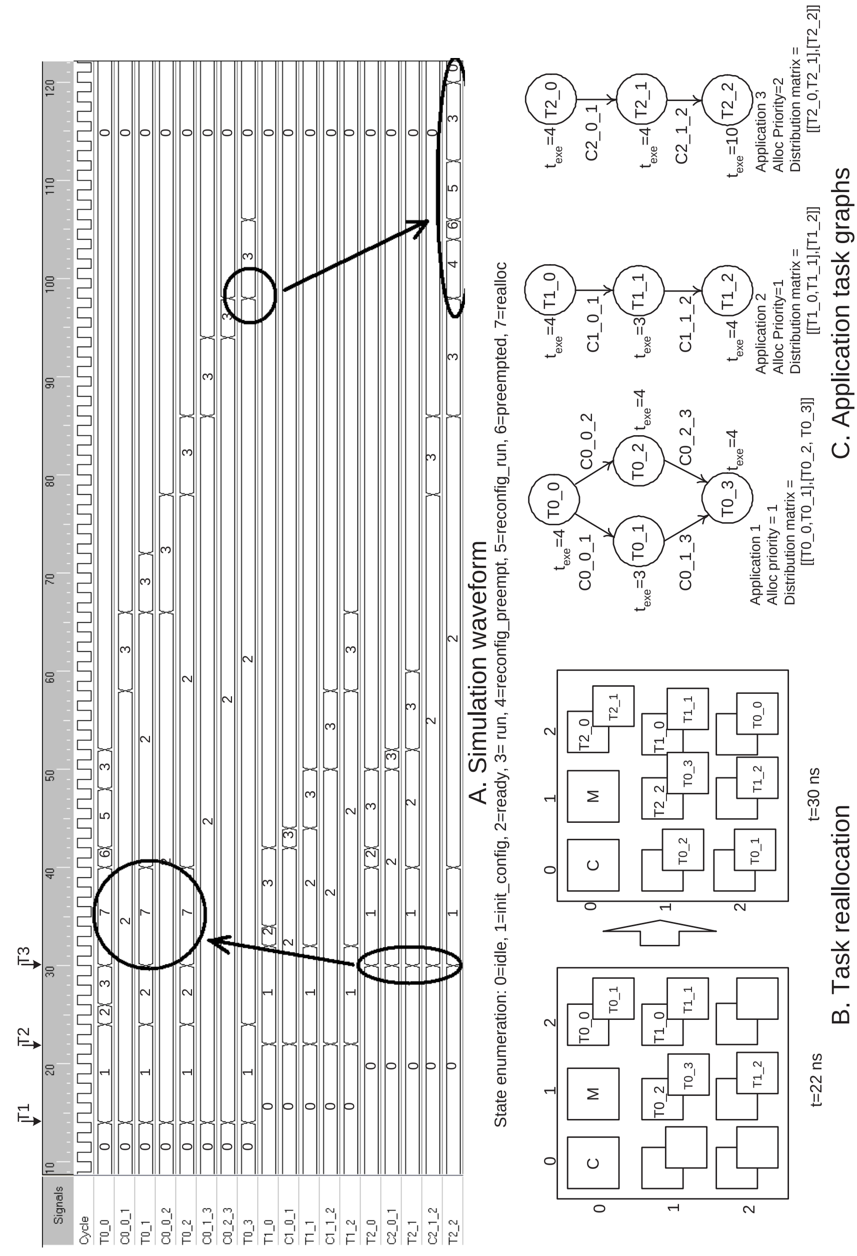


Figure 5.10: Demonstrative simulation

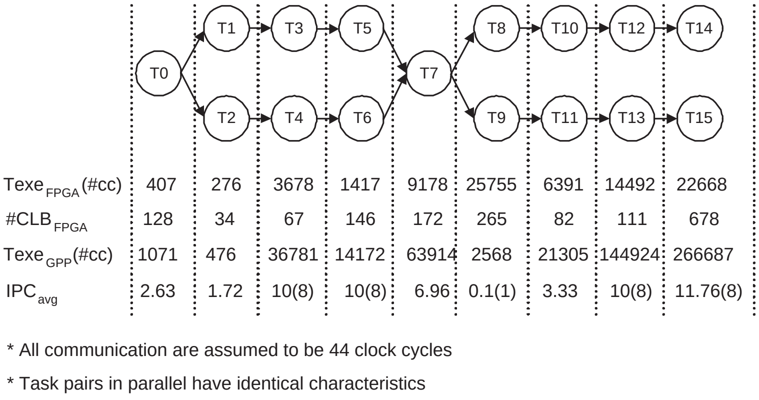
and the communication task c0_0_2's latency is increased by one hop. The communication task c2_0_1, which is made local by the distribution matrix and reallocation, cost only one clock cycle to finish.

At t=98, task T2_2 goes through a few state changes, which is caused by task T0_3. As shown in Figure 5.10B, these two tasks are allocated on the same S-node. At t=86, task T2_2's dependency is solved, and the synchronizer starts its execution. When the simulation time reaches 98, task T0_3's dependency is also solved, and the scheduler decides that T0_3 should start its execution since it has an earlier deadline. Task T2_2 goes through a long preempt phase and return to the "run" state after task T0_3 finishes its execution.

5.6 MP3 Experiments

5.6.1 The reference MP3 task graph

Schmitz et al. [87] studied the implementation of several benchmark programs for ASICs, FPGAs and General Purpose Processors (GPP). Their work generalized each benchmark into a task graph with various parameters annotated to each task, e.g. the area cost of a task's ASIC implementation or the execution time of a task's GPP implementation. In our study, we adopt their MP3 task graph, as shown in figure 5.11, for our experimentation. The whole MP3 cost 2408 CLBs to map onto an FPGA, and the execution time on the FPGA is 61739 clock cycles (cc). The state-of-the-art commercial FPGAs have more than 10,000 CLBs, thus the MP3 can easily be implemented on a single FPGA.



$Texe_{FPGA}$(#cc)	407	276	3678	1417	9178	25755	6391	14492	22668
$\#CLB_{FPGA}$	128	34	67	146	172	265	82	111	678
$Texe_{GPP}$(#cc)	1071	476	36781	14172	63914	2568	21305	144924	266687
IPC_{avg}	2.63	1.72	10(8)	10(8)	6.96	0.1(1)	3.33	10(8)	11.76(8)

* All communication are assumed to be 44 clock cycles

* Task pairs in parallel have identical characteristics

Figure 5.11: Original MP3 task graph and task implementation parameters

The purpose of our experiments is not to demonstrate how the MP3 can be implemented on a reconfigurable system, but to demonstrate how the design space and platform space exploration can be done given a reconfigurable architecture with constant area. Unlike the original MP3 task graph, which assumes that the same task graph can be used for any means of implementation, the tasks in our application model have to optimally use the RUs of various sizes. The original MP3 task graph needs to be transformed to fit the RUs by using the same principle discussed in figure 5.6 with hypothetical assumptions on how the splitting and the merging change the communication cost.

When simulated in COSMOS, the task allocation scheme tries to allocate as many tasks with data dependencies onto the same RU as possible in order to reduce the communication among RUs, and allocate parallel tasks onto different RUs to achieve high performance. When more than one instance of MP3 is executed on the array, reallocation is needed if the second MP3 instance has a higher allocation priority. Currently we assume that the task synchronization, the scheduling and the resource management take no simulation time, so we can focus our study on the interplay between the tasks and the architecture.

5.6.2 Fine-grained architecture study

For a reconfigurable architecture, the size of the RU and the number of hardware contexts have significant impact on system performance. The size of the RU decides the latency of loading a configuration from the memory to the RU, the latency of the task reallocation, and the inter-task communication latency. Under the assumption of having a fixed number of computation resources on a chip, having larger RU leads to having less RUs, which in turn reduces the flexibility of the task allocation. The number of the contexts has significant impact on the system parallelism and the data locality. Reducing the number of contexts leaves more chip area for having more RUs, thus result in having more parallel computation power. But reducing the number of contexts also results in more communication among tasks, since it is harder to allocate the tasks with dependencies into the same RU and localize the communication. In general, there are trade-offs to be made among different architecture settings, and due to the complexity of the applications and the architecture, the impacts of various trade-offs should be assessed by simulation in an early development stage.

5.6.2.1 Varying the RU size

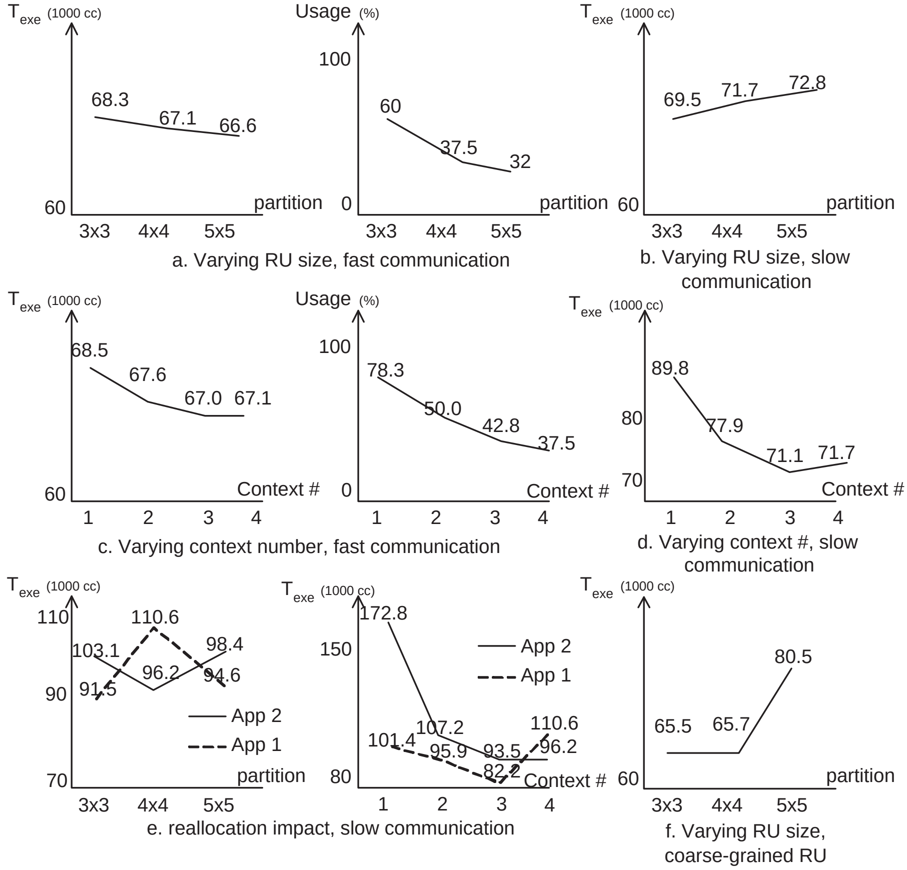
We assume that the total chip area $A_{total} = 10K$ in the number of CLBs. 30% of the area is used on the NoC, thus leave us 7K CLB-equivalent chip area for RUs. We assume that the $\frac{A_{logic}}{A_{context}} = \frac{10}{3}$ for fine-grained RU, and the number of context for each RU is 4. The total chip area spent on logic is about 3.2K CLBs, and the total chip area spent on context storage is about 3.8K CLBs. We assume two Master nodes and two Coordinator nodes are assigned to the RU array.

In our experiment, the whole array is divided into 3x3, 4x4 and 5x5 arrays. The reference task graph in figure 5.11 is transformed for each partition according to the size of the RU. Communication latencies caused by task splitting $T_{comm\_split} = 0.5cc/CLB \times A_{logic}$, and the initial configuration latency of each task $T_{init} = 10cc/CLB \times A_{logic}$, since current FPGA still needs great improvement on reducing reconfiguration latency.

From figure 5.12a we can see that the 5x5 architecture is slightly faster that the others. We expect the 5x5 array to have a higher communication latency than that of the other two, since its task graph has more tasks resulted from transformation. The main reason why this doesn't happen is that the communication latency of the MP3 is very low compared to the initial configuration latency, thus has little effect on the overall execution time. On the other hand, since the 5x5 RU is smaller than the others, the tasks' initial configuration latency is lower than that of the others, thus results in better performance. However, both the initial configuration latency and the communication latency are decided by many design factors, and which latency dominates the timing overhead depends on the NoC and the configuration memory design. We carry out the same simulation with all the communication latency increased by 10 times, and got a completely opposite result, as shown in figure 5.12b.

Another mean of measuring the system optimality is the percentile of the contexts being used by the application. We observed that the usage of the whole array when partitioned into 5X5 array is lower than that of the other two partitions. This leaves room for more flexible allocation of the next running application. From our observation, as long as the communication latency caused by the task splitting is acceptable, having higher number of RUs achieves better overall system usage.

The 5x5 partition's execution time is around 66,600 clock cycle, which is 7.4% more than the 61,739 clock cycle execution time of the traditional FPGA implementation. The application uses 32% of the total 10K CLB chip area, which is around 800 CLBs more than the traditional FPGA implementation's cost.



a. Varying RU size, fast communication

b. Varying RU size, slow communication

c. Varying context number, fast communication

d. Varying context #, slow communication

e. reallocation impact, slow communication

f. Varying RU size, coarse-grained RU

Figure 5.12: Simulation results of MP3 experiments

The performance of the reconfigurable system is comparable to the dedicated FPGA implementation, but the flexibility of the reconfigurable system is a great advantage over the traditional FPGA based design. With dedicated design that aim for reducing the overhead, we can expect the reconfigurable systems to have even closer performance and cost to the traditional FPGA based designs.

5.6.2.2 Varying the number of contexts

We assume that the $A_{logic} = 200CLB$ is a constant for each RU, and we trade contexts for more RUs. With the previously assumed total RU area of 7K CLB, we can have 18 RUs with 3 contexts, 22 RUs with 2 contexts or 27 RUs with single context. Figure 5.12c shows the performance and the context usage of each simulation. We observed that the single context design suffers greatly from the high usage rate, since logics and contexts have one-to-one correspondence. Also, we observed that the more contexts the RUs have, the more communication can be localized, thus the shorter the application execution time can be. The effect of the data localization is amplified on slow communication simulation as shown in figure 5.12d, where the simulation assume the communication bandwidth is reduced by 10 times. However, localizing the communication through task allocation doesn't guarantee the performance gain, since applications may demand parallel computation resources or high number of RU for more speed up.

5.6.2.3 Impact of the reallocation

We experiment on how the partitioning influences the reallocation. As shown in figure 5.12e, we run two instances of MP3 in each simulation to cause task reallocation. To guarantee the occurrence of reallocation, we only define one M-node in the system. The second instance of the MP3 is started after the first instance is run for 20K clock cycles. We expect the first MP3 instance to finish before the second instance finishes, and designs with more RUs and contexts are less influenced by the reallocation. What we observed is somewhat different from expected. For instance, the 4X4 array with 4 contexts finishes running the second MP3 instance before the first one. This happens because the reallocation accidentally cause the tasks that should be executed in parallel be reallocated to the same RU. The penalty of such reallocation is significant enough to cause serious performance degradation. Such observation leads to the consideration of designing more advanced reallocation strategy. A good reallocation strategy not only needs to take the resource distribution and task execution time into account, but the task parallelism as well.

5.6.3 Coarse-grained architecture study

Most of the medium-grained and coarse-grain reconfigurable systems have similar characteristics as fine-grained ones, except that they have lower requirements on the configuration memory size and bandwidth. However, for coarse-grained architectures that use FU as logics, the configuration management becomes a even less demanding problem, since the configuration is small in size. Instead, the efficient use of RU become a more serious issue, as shown in our experiment.

The original MP3 study gave out the execution time of GPP implementation of each task, under the assumption that the instructions are executed in sequential order. To transform the original MP3 into a realistic task graph that fits coarse-grained architecture, we need to find out the instruction-level parallelism of each task. This can be estimated by comparing a task's execution time of the FPGA implementation and that of the GPP implementation, e.g. $IPC_{avg} = \frac{T_{exe_{GPP}}}{T_{exe_{FPGA}}}$. The average IPC of the MP3 tasks varies between 1.7 and 11.8, as shown in figure 5.11, which indicates that the RU can not be efficiently used all the time.

We assume that an FU costs 16 CLBs to implement, including the data routing that fetches the computation results from the neighboring FUs. Based on this assumption, the 3x3, 4x4 and 5x5 partitions can offer the maximum IPCs (IPC_{max}) of 22, 12 and 8 per RU, respectively. As mentioned before, the IPCs of MP3 task are estimated around 1.7 and 11.8, thus high-parallelism tasks' performance is capped at 8 IPC when running on 5x5 partition RUs. In this case, we assume that the execution time of each task is:

$$T_{exe_{coarse}} = \begin{cases} T_{exe_{FPGA}} = \frac{T_{exe_{GPP}}}{IPC_{avg}}, & \text{if } IPC_{avg} \leq IPC_{max}; \\ \frac{T_{exe_{GPP}}}{IPC_{max}}, & \text{if } IPC_{avg} > IPC_{max}; \end{cases}$$

We still assume that only 4 tasks can be allocated on the same RU, and the inter-task communication latency is the same as in fine-grained architectures.

Our simulation is done on various number of RUs, and the result is shown in figure 5.12f. The 3x3 and 4x4 partition systems have very close performance, since both partitions offer enough parallel FUs for all tasks. The 5x5 architecture has a significant performance penalty due to the IPC cap of 8. However, 3x3 and 4x4 partitions result in a large waste of RUs, since many tasks can only use a few FUs, even if up to 22 FUs are available.

The system efficiency boils down again to the matching of the size of the RU and the demand of the tasks. Source code transformation or aggressive compile-time loop-level optimization can improve the task IPC, but these tasks are not trivial. Before upscaling such architecture, designers need to estimate how frequently the added functional unit is being used.

5.7 Advanced allocation strategies

Our previous experiments employed a rather simple task allocation strategy. We tightly cluster all the tasks around the M-nodes to reduce the communication, but the M-nodes become the allocation hot spots and cause high occurrence of reallocation. In this section, we propose a few other (re)allocation strategies and discuss how (re)allocation impacts the system performance.

5.7.1 Task clustering

Instead of using M-nodes as the cluster centers of any application, each application should find its own cluster center and form its own cluster, as shown in figure 5.13. By forming clusters separately, we expect that applications will disrupt each other less frequently, hence result in less communication and reallocation.

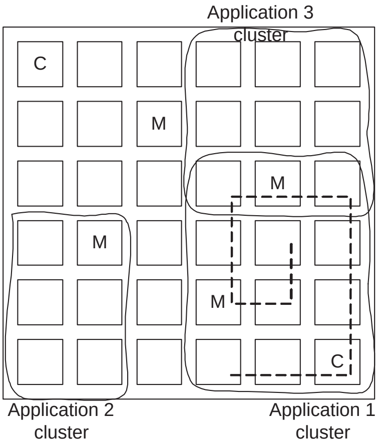


Figure 5.13: Clustering based on applications

The clustering of each application is made by one of the M-nodes when the application is being allocated. The M-node being selected to synchronize and schedule the application has a resource distribution map of the whole coprocessor, thus is able to search for an area on the coprocessor that has condensed free resources. Since the RU array is quite small, the search time is reasonably short.

During the search, the selected M-node assumes that any slave node can be the cluster center, and builds a helix search path on each slave. Figure 5.13 shows one of the possible helix search paths that formed application 1's cluster. The helix search path is extended one hop at a time from the assumed cluster center. Each time the helix is extended, the total number of free resources chained on the helix is calculated. If there are enough free resources on the helix chain to run the application, the search stops. Clearly, if the helix extended from a slave node has the shorter length when the search stops, the cluster has more condensed free resources, thus is a better place to allocate the new application.

5.7.2 Allocation priority

The reallocation is a costly process, thus should only occur if there is little penalty received from it. Our previous reallocation strategy is completely priority-based, e.g. a lower-priority task gets reallocated to guarantee that higher-priority tasks suffer less from communication overhead. Based on some of our observation, such a simple strategy may cause starvation or unnecessarily prolong the task execution time. For instance, a low-priority task can be reallocated repeatedly if several high-priority tasks request to be allocated, resulting in the low-priority task to suffer greatly from reallocation latency and miss the deadline, even if the task is near its completion.

To solve this problem, the allocation has to take not only spatial characteristic of the resource distribution into account, but the temporal one as well. This is a complicated issue to address at run-time, since the temporal characteristic of the resource distribution is not simple to capture. Our strategy to solve this problem is to employ dynamic task prioritization. E.g. we should increase the allocation priority of an allocated task when a large portion of it has been executed, or when its deadline approaches. Currently we are still analyzing how to optimally change a task priority. In our experiment here, we simply increase the task's priority if its remaining execution time is comparable to the reallocation latency.

5.7.3 Critical path guided allocation

After the system is running for a while, free resources will eventually be evenly distributed on the array, causing a 3D-space resource fragmentation. Therefore, we attempt to find an alternative strategy that can reduce the occurrence of the reallocation even further. Critical path is often used to optimize the task partition and scheduling [88] etc. In our system, it can be used to guide the task allocation as well.

The critical path of a task graph can be easily identified when the task graph is statically constructed. During an idealistic allocation scenario, the M-node should spend more effort to guarantee the allocation quality of the tasks on the critical path. Non-critical tasks could be allocated in a more relaxed manner. Reallocation should only be invoked when the task being allocated is critical in order to avoid causing further complication.

However, due to the non-deterministic communication latency, the critical path of the application running on a reconfigurable system may vary dynamically.

Finding one “absolute” critical path is infeasible for such system, so we need to find some more advanced solutions. In practice, we can either dynamically monitor the changes during task (re)allocation and identify the critical path on the fly, or statically identify several critical path candidates and optimally allocate several of them. When compared to the dynamical approach, the static approach is not equally efficient at reducing reallocation’s occurrence, but has the advantage of reducing the run-time management overhead.

We experimented on the static approach during our study. For a given task graph as shown in figure 5.14, based on the execution time analysis, a preliminary critical path can be identified as T1→T2→T3→T4→T5→T6. We assume such path is highly probable to be the critical path in run-time, thus assign the highest allocation priority to it (P=3). Paths that are branched out of the critical path, especially the ones that join the critical path later on, have high probability to become another critical path, thus we back trace some of these paths and assign them a moderately high priority (P=2). The rest of the task graph get the lowest priority, which should not cause any reallocation. We also allow a task graph to have more than one critical path in order to improve the reallocation.

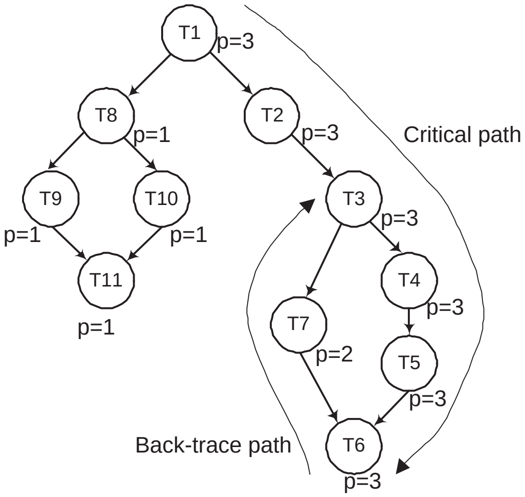


Figure 5.14: Critical path based task prioritization

5.7.4 Simulation and analysis

To evaluate the effectiveness of our various management strategies, we set up the following simulations. We generated 5 application task graphs by using Task Graph For Free (TGFF)[10], as shown in appendix A. Then we randomly instantiate 100 applications from these five application task graphs to construct an input application set. During simulation time, as soon as there is a certain amount of resources available on the coprocessor, one of the applications in the input application set is started. We set up the architecture as an 8x8 RU array with two C-nodes, two M-nodes and 60 four-context S-nodes. We assume that the NoC can handle up to 64 messages concurrently, and the reconfiguration latency is comparable to the average execution time of all tasks. We run the simulation several times, each time with a different input application set, and show the average result in this section.

We believe that (re)allocation will not be very efficient when the coprocessor is overly stressed. E.g. starting a new application when there are just enough resources available can be bad for the overall system performance, since there is little a reallocation strategy can do. However, how many resources should be reserved for (re)allocation and how the reserved resources can impact the system are unclear to us. In our experiments, we reserved up to 120 resources, which is equivalent to up to 50% of total resources, and observed the impacts on the total execution time of these 100 applications.

The simulation results are shown in figure 5.15. The **Basic_R** simulation employs the M-node centered reallocation strategy. In this simulation, an allocation priority ranging from 1 to 5 is randomly assigned to each application. The **Helix_NR** simulation employs the helix-path based allocation strategy to optimize for the task allocation. The matching reallocation strategy for the helix-path based allocation is still under research, thus the reallocation for **Helix_NR** simulation is disabled. The **Dynamic_priority** simulation increases the basic task priority, which is assigned the same way as in the **Baisc_R** simulation, when a task's remaining execution time is less than the reconfiguration time. The **Critical_path** simulation assigns task priority based on the principle introduced in figure 5.14.

The most interesting observation is that a small amount of reserved resources reduce the execution time of all the simulation. Without any reserved resources, the execution time of all the tested strategies are very close, since none of the (re)allocation strategies is effective. With 20-30 resources (~10%) reserved for allocation, the **Basic_R**, **Critical_path** and **Dynamic_priority** gets some benefit. The **Helix_NR** simulation performs no reallocation, thus requires more free resources to get to the optimal point. All simulations show that reserving

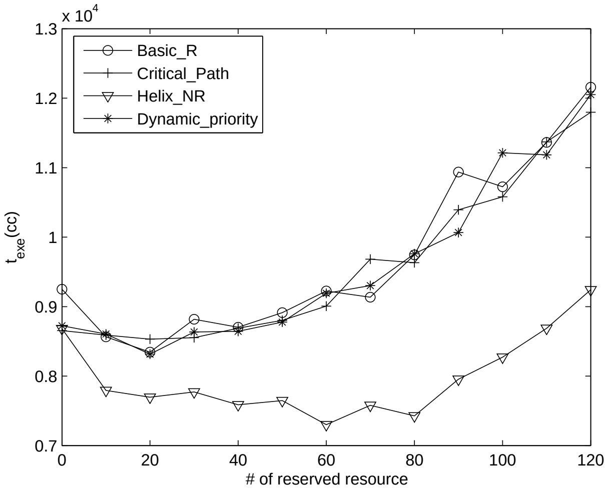


Figure 5.15: Running 100 applications with various management strategies

fewer resources linearly reduces the application execution time, but little performance gain can be achieved from reserved resources deduction when less than 80 resources (∼30%) are reserved for (re)allocation.

The helix-path based allocation strategy outperforms the other three management strategies. The system seems more rigid when fewer resources are reserved for allocation, but the overall performance gain proves that distributed clustering is a right allocation strategy. However, the matching task reallocation strategy is hard to devise, since the task allocation is already very optimal, and reallocation has a high chance to cause more communication rather than reducing any.

The **Basic_R**, **Critical_path** and **Dynamic_priority** simulation rely more on reallocation rather than allocation. The simulations show that their results are very close. From our analysis, we discovered that reallocation frequently cause longer overall communication time. Higher priority applications often get efficiently executed, but the lower priority tasks suffer from exceedingly more communication latencies than necessary. The crucial issues are that we still can't choose the right low priority task to reallocate, and we still can't find an optimal S-node to reallocate a task to. The reallocation issue is a complicated NP-complete problem that is challenging to analyze at run-time, and is one of the major topics for future study.

To investigate how fragmentation impacts our various resource management strategies, we run the same simulations with different numbers of input application and observe the task execution time. The optimal number of reserved resources is granted to each of these simulations. As shown in figure 5.16, due to the flexibility of our architecture, the fragmentation does not change the linear relationship between the execution time and the number of tasks being executed. The **Helix_NR** simulation still outperforms the others, especially when fewer tasks are executed on the coprocessor.

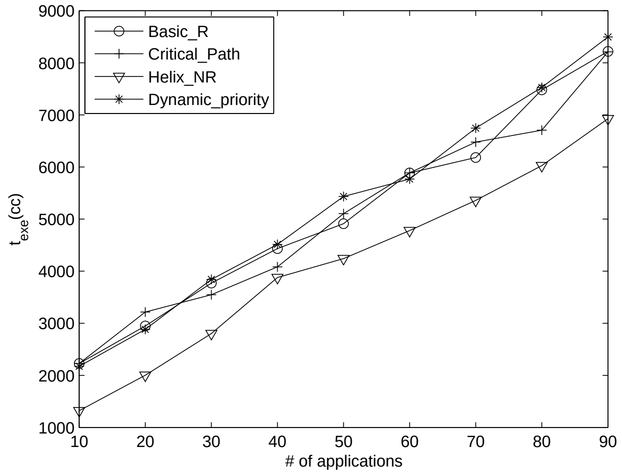


Figure 5.16: Running various number of applications with various management strategies

From our experiments, we demonstrated that our clustering strategy is a simple and effective solution to address the allocation issue. The reallocation issue, however, is much more complicated to understand. All our attempts show that the execution time of higher-priority applications can only be reduced when the overall system performance is compromised. So far, many parts of the reallocation are not well-addressed, and we are still on our journey towards understanding the nature of the reallocation.

5.8 Future work

Our current work concentrates on the real-time behavior of the task being executed on the coprocessor, but the timing characteristic of the management strategy has not been thoroughly addressed. Take reallocation for an example, the decision of which task should be reallocated to which S-node greatly impacts the system performance. If the M-nodes can thoroughly analyze the current resource distribution status, the reallocation strategy can better improve the system performance. However, the decision is made by M-nodes at run-time, thus the latency on making the decision becomes an overhead to the task execution as well. There are trade-offs between reallocation optimality and the reallocation latency, and such trade-off is not well-understood at the moment. In our model, we currently assume that all the resource management algorithms are executed in no time, which can be too optimistic for a large system. In the future, some improvement can be made on this aspect of our model.

With some modification on the allocation strategy and task model, COSMOS can be used to study the systems with heterogeneous RU array. But currently, understanding the behavior of a homogeneous system and optimizing such a system already pose great challenges, and heterogeneous system is far more complicated than the homogeneous system. When the homogeneous system study is more mature, we believe it will be interesting to study the behavior of the heterogeneous systems and compare it with that of the homogeneous ones.

From the applications' allocation/execution scenario, we can identify plenty of issues to be addressed in the future. The strategies of task allocation, scheduling and reallocation etc. are open for further study. From our experience, we see that the (re)allocation issue can be highly complicated. Employing complicated allocation algorithm when an application requests to run not only increases timing overhead, but also improves little when the resource is fragmented. Also, the computation power of the C- or M-nodes is wasted when no application is requesting to run. Proactive run-time management is a promising strategy to make better use of the management nodes and ease the allocation. Currently we are studying how a proactive resource management strategy can be applied to the reconfigurable system, and what kind of complexity such strategy can handle.

5.9 Conclusion

The most important lessen we learnt from reconfigurable system study is that, even if reconfigurable system design issues have much resemblance to that of many traditional systems and technologies, applying previously used solutions to reconfigurable system is rarely appropriate. Being highly dynamic systems with non-deterministic behavior, reconfigurable system should be treated more delicately, or even be approached from a different angle. Before proposing advanced solutions based on previous research or intuitive reasoning, we often need to withdraw our conclusion based on the study of the traditional architectures and try to understand the reconfigurable systems a little better, especially when we are facing such complexity.

Under this circumstance, we developed the COSMOS framework in hopes of helping us to evaluate our design and inspiring us to create more ideas. The COSMOS model is a flexible platform that can be partially customized to fit the needs of the user. It is also a rather friendly tool to update once the underlying message passing mechanism is understood. Through our simulation, we demonstrated how our simulator can be used for studying system-level design, and pinpointed what architecture design issues can impact the application execution performance.

Even if we have only discussed a few key architecture design issues, considerable complexity has already been seen. At the current stage, we are still in the process of understanding how the reconfigurable systems' dynamic behavior is affected by various design issues. Based on our observation, we proposed some run-time management strategies and demonstrated that these strategies can impact the system performance in certain scenario. Currently, plenty of work still remains to be done in both understanding and improving such system, and we anticipate that COSMOS can be a handy tool to assist our future work.

Chapter 6

Conclusion

6.1 Contribution

The research on reconfigurable systems has been growing for more than a decade. The architecture study has taken several different directions, each of which results in its own relevant methodology study and run-time management study. Enormous amount of ideas have been proposed, and a few industrial practices have been seen. However, due to the diversity of the ongoing researches, it is not a trivial task to pinpoint what the urgent needs are for the current research and how to contribute to it.

Under such circumstance, we divide our study into the following three phases. In the first phase, we made a survey of the area and experienced the partial reconfiguration on commercial FPGA. From this phase, we understood some of the crucial characteristics of reconfiguration and identified some of the unaddressed issues that hinder the future research. The outcome of this phase is the foundation of our follow-up works.

In the second phase, we propose to use the simultaneous multi-threading to increase the performance and the scalability of datapath-coupled reconfigurable architectures. The scalability is the crucial issue that restrains the promotion of many datapath-coupled architectures. We expanded the existing ADRES architecture to support the partition-based threading, and proposed a strategy to improve the ADRES tool chain, DRESC, to ease the architecture partitioning and application compilation. By upgrading the MPEG2 decoder into a dual-threaded program and running the decoder on a threaded ADRES instance, we demonstrated that the simultaneous multi-threading leverages the scalability issue of the datapath-coupled system, thus deserves further investigation.

The last phase of our study focuses on the system-level modelling and simulation of the coprocessor-coupled systems. Being the most complicated category of the reconfigurable system, coprocessor-coupled systems have highly unpredictable dynamic behavior. Our work focuses on the homogeneously modularized and partitioned systems, and tries to capture the behavior of such system. With the help of our modelling and simulation framework, we studied how several important early-stage architecture design decisions can impact system performance, and pointed out several pitfalls and tradeoffs. Through some simulation, we've also acquired better understanding of the dynamic resource management issue of such system, and proposed several task reallocation strategies to increase the system performance. We are positive that our simulation framework, COSMOS, is a valuable tool that can help us to understand such system and optimize them in many aspects.

6.2 Outlook

The reconfigurable system research is a promising yet challenging area. These architectures have great potential in achieving both high flexibility and high performance, but also risk suffering from reconfiguration penalties. Understanding the source of these penalties requires us to have ample knowledge of the dynamic behavior of these systems. however, such behavior has not been thoroughly investigated before.

The datapath-coupled systems are relatively more practical to use. These systems' structures are frequently regular and simple, and are suitable to be used for embedded system design. When compared to more traditional embedded systems, the programming and compilation of these reconfigurable systems are more complicated, but thanks to the regularity of these systems, the design automation is still feasible. Due to the scaling limitation of these systems, the performance penalty caused by the run-time management is reasonably small.

The coprocessor-coupled systems tend to be large and complicated. When combined with MPSoC and NoC, these systems offer great deal of parallel computing power to speed up the execution of several applications concurrently, but these systems are also highly complicated in every design aspect. Our COSMOS model currently has been used to study several important architecture design factors and run-time task reallocation issues of these systems, but many other run-time characteristics have not been investigated. The interaction among the host processor, the memory hierarchy and the reconfigurable coprocessor is another interesting topic to study, especially the memory bandwidth issue, and COSMOS is a suitable framework for investigating such issue.

The COSMOS framework models the homogeneous systems. Whether or not the homogeneous systems should be the focus of future research depends on two issues. These issues are: how difficult it is to efficiently manage a heterogeneous system's resource; and how difficult it is to partition an application into a task graph that can efficiently use the homogeneous resource. Both of these issues are known to be very challenging to study at the moment, and we expect nothing less than long term investigation to address these issues.

Even if the reconfigurable systems have been studied by many, we still see quite a few fundamental issues not well-addressed, especially for coprocessor-coupled architectures. As potentially powerful as it is tricky to take advantage of, we believe that many aspects of the reconfigurable system still require much work to be truly understood. Nonetheless, the reconfigurable system is one of the most promising paradigms for future architectures, and its potential of taking the advantages of all the major ongoing researches makes it hard to be overlooked. In time, we wish to see them being better recognized and appreciated.

Appendix A

TGFF files

A.1 Input file

```
_cnt 5
sk_cnt 12 6
sk_degree 2 2
riod_laxity 1
riod_mul 1, 0.5, 2
_write
_write
s_write
g_write

ble_label COMMUN
ble_cnt 1
ble_attrib price 200 40
pe_attrib exec_time 60 20
ans_write
```

A.2 Output file

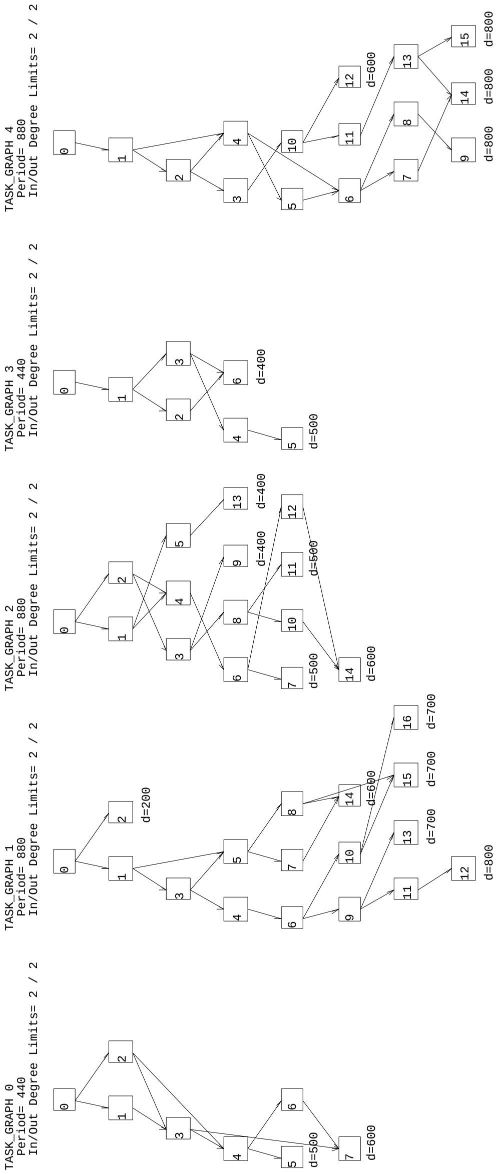


Figure A.1: Task graph used in COSMOS management strategy study

Bibliography

[1] Fpslic on-chip partial reconfiguration of the embedded at40k fpga. *www.atmel.com.*

[2] http://suif.stanford.edu/.

[3] Ise 7.1i development system reference guide. *www.xilinx.com.*

[4] *See http://www.celoxica.com/.*

[5] *See http://www.cray.com/products/xd1/index.html.*

[6] *See http://www.nallatech.com/.*

[7] *See http://www.siliconhive.com/.*

[8] *See http://www.srccomp.com/.*

[9] *See http://www.xilinx.com.*

[10] *See http://ziyang.ece.northwestern.edu/tgff/.*

[11] Two flows for partial reconfiguration: Module based or difference based application note. xapp290 v1.1. *www.xilinx.com.*

[12] Virtex series configuration architecture user guide. xapp151 v1.7 october 20, 2004. *www.xilinx.com.*

[13] H. Akkary and M.A. Driscoll. A dynamic multithreading processor. *31st Annual ACM/IEEE International Symposium on Microarchitecture. MICRO-31. Proceedings*, pages Page(s):226 – 236, 1998.

[14] G.M. Amdahl. Validity of the single processor approach to achieve large-scale computing capabilities. *Proc. AFIPS Spring Joint Computer Conf. 30*, pages 483–485, 1967.

[15] P. Bellows and B. Hutchings. Jhdl - an hdl for reconfigurable system. *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 175, 1998.

[16] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan. A self-reconfigurable platform. *Field-Programmable Logic and Applications FPL'03*, 2003.

[17] Kiran Bondalapati. Parallelizing dsp nested loops on reconfigurable architectures using data context switching. *DAC '01: Proceedings of the 38th conference on Design automation*, pages 273–276, 2001.

[18] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 262–269, 2002.

[19] Mihai Budiu, Mahim Mishra, Ashwin R. Bharambe, and Seth Copen Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 57, 2002.

[20] D. Burger, S.W. Keckler, and K.S. McKinley. Scaling to the end of silicon with edge architectures. *IEEE Computer (37)*, pages 44 – 55, 2004.

[21] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *Computer Volume: 33 Issue: 4*, pages 62–69, 2000.

[22] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for vliws: A preliminary analysis of tradeoffs. *The 25th Annual International Symposium on Microarchitecture*, pages 103–114, 1992.

[23] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[24] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):209–220, 2002.

[25] J.G.F. Coutinho and W. Luk. Source-directed transformations for hardware compilation. *Field-Programmable Technology (FPT). Proceedings. IEEE International Conference on*, pages 278–285, 2003.

[26] R. David, D. Chillet, S. Pillement, and O. Sentieys. A dynamically reconfigurable architecture for low-power multimedia terminals. *VLSI-SOC '01: Proceedings of the IFIP TC10/WG10.5 Eleventh International Conference on Very Large Scale Integration of Systems-on/Chip*, pages 51–62, 2001.

[27] R. David, D. Chillet, S. Pillement, and O. Sentieys. A compilation framework for a dynamically reconfigurable architecture. *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 1058–1067, 2002.

[28] R. David, D. Chillet, S. Pillement, and O. Sentieys. Dart: A dynamically reconfigurable architecture dealing with future mobile telecommunications constraint. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS*, pages 156–163, 2002.

[29] R. Dimond, O. Mencer, and W. Luk. Custard - a customisable threaded fpga soft processor and tools. *International Conference on Field Programmable Logic and Applications*, pages 1 – 6, 2005.

[30] Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, and Heidi Ziegler. Bridging the gap between compilation and synthesis in the defacto system. *In Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC'2001),*, pages 570–578, 2001.

[31] B. Draper, W. Najjar, W. Bohm, J. Hammes, B. Rinker, C. Ross, M. Chawathe, and J. Bins. compiling and optimizing image processing algorithms for fpgas. *Computer Architectures for Machine Perception. Proceedings. Fifth IEEE International Workshop on*, pages 222–231, 2000.

[32] S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, and D.M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *Micro, IEEE Volume 17, Issue 5*, pages 12 – 19, 1997.

[33] Gerald Estrin. Reconfigurable computer origins: The ucla fixed-plus-variable (f+v) structure computer. *IEEE annals of the history of computing*, pages 773–783, 1988.

[34] Jong eun Lee, Kiyoung Choi, and Nikil D. Dutt. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 38(7):183–188, 2003.

[35] K. Furuta, T. Fujii, M. Motomura, K. Wakabayashi, and M. Yamashina. Spatial-temporal mapping of real applications on a dynamically reconfigurable logic engine (drle) lsi. *Custom Integrated Circuits Conference, CICC. Proceedings of the IEEE*, pages 151–154, 2000.

[36] K.M. GajjalaPurna and D. Bhatia. Partitioning in time: A paradigm for reconfigurable computing. *ICCD '98: Proceedings of the International Conference on Computer Design*, page 340, 1998.

[37] Manuel G. Gericota, Gustavo R. Alves, Miguel L. Silva, and Jose M. Ferreira. Run-time management of logic resources on reconfigurable system. *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pages 974 – 979, 2003.

[38] D.B. Gottlieb, J.J. Cook, J.D. Walstrom, S. Ferrera, Chi-Wei Wang, and N.P. Carter. Clustered programmable-reconfigurable processors. *Field-Programmable Technology, (FPT). Proceedings. IEEE International Conference on*, pages 134– 141, 2002.

[39] S. Guccione, D. Levi, and P. Sundararajan. Jbits: Java based interface for reconfigurable computing. *http://www.io.com/ guccione/Papers/MAPPLD/JBitsMAPPLD.pdf.*

[40] Steven A. Guccione and Delon Levi. Jbits: A java-based interface to fpga hardware. *http://www.io.com/ guccione/Papers/JBits/JBits.html.*

[41] Steven A. Guccione and Delon Levi. Run-time parameterizable cores. *Proceedings of the ACM/SIGDA seventh international symposium on Field programmable gate arrays*, page 252, 1999.

[42] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. *Intl. Conf. on VLSI Design, 2003*, pages 461– 466, 2003.

[43] Yajun Ha, Radovan Hipik, Serge Vernalde, Diederik Verkest, Marc Engels, Rudy Lauwereins, and Hugo De Man. Adding hardware support to the hotspot virtual machine for domain specific applications. *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 1135–1138, 2002.

[44] Jeffrey P. Hammes, Robert Rinker, Walid A. Najjar, and Bruce Draper. A high level, algorithmic programming language and compiler for reconfigurable systems. *The 2nd International Workshop on the Engineering of Reconfigurable Hardware/Software Objects (ENREGLE)*, 2000.

[45] Manish Handa and Ranga Vemuri. An efficient algorithm for finding empty space for online fpga placement. *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 960–965, 2004.

[46] J. Harkin, T.M. McGinnity, and L.P. Maguire. Genetic algorithm driven hardware-software partitioning for dynamically recofigurable embedded systems. *Microprocessor and Microsystems*, pages 263–274, 2001.

[47] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The chimaera reconfigurable functional unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 206–217, 2004.

[48] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21. IEEE Computer Society Press, 1997.

[49] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. Dynamic hardware plugins in an fpga with partial run-time reconfiguration. *DAC '02: Proceedings of the 39th conference on Design automation*, pages 343–348, 2002.

[50] Zhining Huang and Sharad Malik. Exploiting operation level parallelism through dynamically reconfigurable datapaths. *DAC '02: Proceedings of the 39th conference on Design automation*, pages 337–342, 2002.

[51] E. Iwata and K. Olukotun. Exploiting coarse-grain parallelism in the mpeg-2 algorithm. *Stanford University Computer Systems Lab Technical Report CSL-TR-98-771*, 1998.

[52] Adam Kaplan, Philip Brisk, and Ryan Kastner. Data communication estimation and reduction for reconfigurable systems. *DAC '03: Proceedings of the 40th conference on Design automation*, pages 616–621, 2003.

[53] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4):605–627, 2002.

[54] Eric Keller. Jroute: A run-time routing api for fpga hardware. *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 874–881, 2000.

[55] Richard B. Kujoth, Chi-Wei Wang, Derek B. Gottlieb, Jeffrey J. Cook, and Nicholas P. Carter. A reconfigurable unit for a clustered programmable-reconfigurable processor. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 200–209. ACM Press, 2004.

[56] Chidamber Kulkarni, Gordon Brebner, and Graham Schelle. Mapping a domain specific language to a platform fpga. *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 924–927, 2004.

[57] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Fast area estimation to support compiler optimizations in fpga-based reconfigurable systems. *FCCM '02: Proceedings of the 10th Annual IEEE*

Symposium on Field-Programmable Custom Computing Machines, page 239, 2002.

[58] Luciano Lavagno. The programmer's view of a dynamically reconfigurable architecture. *MPSoC'04 workshop invited presentation*, 2004.

[59] Sunghyun Lee, Sungjoo Yoo, and Kiyoung Choi. Reconfigurable soc design with hierarchical fsm and synchronous dataflow model. *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 199–204, 2002.

[60] L. Levinson, R. Manner, M. Sessler, and H. Simmler. Preemptive multitasking on fpgas. *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 301, 2000.

[61] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. *DAC '00: Proceedings of the 37th conference on Design automation*, pages 507–512, 2000.

[62] Huiqun Liu and D. F. Wong. Network flow based circuit partitioning for time-multiplexed fpgas. *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 497–504, 1998.

[63] Huiqun Liu and D. F. Wong. A graph theoretic optimal algorithm for schedule compression in time-multiplexed fpga partitioning. *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 400–405, 1999.

[64] A. Lodi, M. Toma, and F Campi. A pipelined configurable gate array for embedded processors. *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays FPGA'03*, pages 21–30, 2003.

[65] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri. A vliw processor with reconfigurable instruction set for embedded applications. *Solid-State Circuits, IEEE Journal of*, pages 1876–1886, 2003.

[66] Roman Lysecky and Frank Vahid. A configurable logic architecture for dynamic hardware/software partitioning. *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, pages 480–485, 2004.

[67] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic fpga routing for just-in-time fpga compilation. *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 954–959, 2004.

[68] J. Madsen, S. Mahadevan, and K. Virk. Network-centric system-level model for multiprocessor system-on-chip simulation. *Interconnect-Centric Design for Advanced SoC and NoC, Springer*, pages 341–365, 2004.

[69] J. Madsen, K. Virk, and M. J. Gonzalez. A systemc-based abstract real-time operating system model for multiprocessor system-on-chips. *Multiprocessor System-on-Chips Morgan Kaufmann*, page 2004, 283-311.

[70] T. Marescaux, J-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on chip as hardware components of an os for reconfigurable systems. *Field-Programmable Logic and Applications FPL'03*, pages 595–605, 2003.

[71] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143, New York, NY, USA, 1999. ACM Press.

[72] P. Master. Keynote: the next big leap in reconfigurable systems. *Field-Programmable Technology, (FPT). Proceedings. IEEE International Conference on*, pages 17–22, 2002.

[73] B. Mei. A coarse-grained reconfigurable architecture template and its compilation techniques. *Ph.D. thesis, IMEC, Belgium*, 2005.

[74] B. Mei, S. Kim, and R. Pasko. A new multi-bank memory organization to reduce bank conflicts in coarse-grained reconfigurable architectures. *IMEC, Technical report*, 2006.

[75] B. Mei, S. Vernalde, D. Verkest, H.D. Man, and R. Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. *International Conference on Field Programmable Technology*, pages 166–173, 2002.

[76] B. Mei, S. Vernalde, D. Verkest, H.D. Man, and R. Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. *Field-Programmable Logic and Applications FPL'03*, 2003.

[77] J. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins. Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances. *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture*, 2002.

[78] J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pages 986–991, 2003.

[79] Gaurav Mittal, David C. Zaretsky, Xiaoyong Tang, and P. Banerjee. Automatic translation of software binaries onto fpgas. *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 389–394, 2004.

[80] Walid A. Najjar, Wim Bohm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69, 2003.

[81] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable soc. *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 174.1, 2003.

[82] V. Nollet, T. Marescaux, P. Avasare, and J-Y. Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 234–239, March 2005.

[83] E. Ozer and T.M. Conte. High-performance and low-cost dual-thread vliw processor using weld architecture paradigm. *IEEE Transactions on Parallel and Distributed Systems, Volume 16, Issue 12*, pages 1132 – 1142, 2005.

[84] Gerard K. Rauwerda, Paul M. Heysters, and Gerard J. M. Smit. Mapping wireless communication algorithms onto a reconfigurable architecture. *J. Supercomput.*, 30(3):263–282, 2004.

[85] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80, 1994.

[86] Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, and Wim Bohm. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):130–139, 2001.

[87] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[88] Mingsheng Shang, Shixin Sun, and Qingxian Wang. An efficient parallel scheduling algorithm of dependent task graphs. *Parallel and Distributed Computing, Applications and Technologies, . PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 595– 598, 2003.

[89] Lesley Shannon and Paul Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 190–199, 2004.

[90] H. Singh, M. Lee, G Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: An integrated reconfigurable system for data-parallel computation-intensive applications. *Computers, IEEE Transactions on*, pages 465–481, 2000.

[91] Greg Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 177–186, 2002.

[92] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Comput.*, 53(11):1393–1407, 2004.

[93] Arvind Sudarsanam, Mayur Srinivasan, and Sethuraman Panchanathan. Resource estimation and task scheduling for multithreaded reconfigurable architectures. In *ICPADS '04: Proceedings of the Parallel and Distributed Systems, Tenth International Conference on (ICPADS'04)*, page 323, Washington, DC, USA, 2004. IEEE Computer Society.

[94] Dinesh C. Suresh, Walid A. Najjar, Frank Vahid, Jason R. Villarreal, and Greg Stitt. Profiling tools for hardware/software partitioning of embedded applications. *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 189–198, 2003.

[95] Nozar Tabrizi, Nader Bagherzadeh, Amir H. Kamalizad, and Haitao Du. Mars: A macro-pipelined reconfigurable system. *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 343–349, 2004.

[96] X. Tang, M. Aalsma, and R. Jou. A compiler directed approach to hiding configuration latency in chameleon processors. *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 29–38, 2000.

[97] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[98] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 22, 1997.

[99] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer. Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. *International Parallel and Distributed Processing Symposium. IPDPS*, page 4 pp, 2006.

[100] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens. Hardware/software partitioning of embedded system in ocapi-xl. *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 30–35, 2001.

[101] D. Verkest. Machine chameleon. *Spectrum, IEEE Volume 40, Issue 12*, pages 41–46, 2003.

[102] Miljan Vuletic;, Laura Pozzi, and Paolo Ienne. Virtual memory window for application-specific reconfigurable coprocessors. *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 948–953, 2004.

[103] Markus Weinhardt and Wayne Luk. Pipeline vectorization for reconfigurable systems. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 52–62, 1999.

[104] J. Williams and N. Bergmann. Embedded linux as a platform for dynamically self-reconfiguring system-on-chip. *Engineering of Reconfigurable Systems and Algorithms, ERSA'04*, 2004.

[105] Guang-Ming Wu, Jai-Ming Lin, and Yao-Wen Chang. Generic ilp-based approaches for time-multiplexed fpga partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 1266–1274, 2001.

[106] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Hierarchical clustered register file organization for vliw processors. *International Parallel and Distributed Processing Symposium, 2003. Proceedings.*, page 10 pp, 2003.

[107] Heidi Ziegler, Byoungro So, Mary Hall, and Pedro C. Diniz. Coarse-grain pipelining on multiple fpga architectures. *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 77, 2002.