Mapping Applications to a Coarse-Grained Reconfigurable Architecture

Yuanqing Guo

Composition of the Graduation Committee:

Prof. Dr. Ir.	Th.	Krol (promotor), faculty of EEMCS
Dr. Ir.	G.J.M.	Smit (assistant-promotor), faculty of EEMCS
Prof. Dr.	C.	Hoede, University of Twente, faculty of EEMCS
Prof. Dr. Ir.	C.H.	Slump, University of Twente, faculty of EEMCS
Prof. Dr. Ir.	D.	Verkest, IMEC, Belgium
Ir.	J.A.	Huisken, Silicon Hive, Eindhoven
Prof. Dr. Ir.	H.	Corporaal, Technical University of Eindhoven
Prof. Dr. Ir.	A.J.	Mouthaan, UT, Faculty of EEMCS (chairman and secretary)

This research is conducted within the Gecko project (612.064.103) supported by the Dutch organization for Scientific Research NWO.

Group of Computer Architecture, Design & Test for Embedded Systems. P.O. Box 217, 7500 AE Enschede, The Netherlands.

Keywords: mapping and scheduling, instruction generation, compiler, coarse-grained reconfigurable architecture.

Email: guoyuanqing@gmail.com

Printed by Wöhrmann Print Service, Zutphen, The Netherlands.
ISBN 90-365-2390-7

MAPPING APPLICATIONS TO A COARSE-GRAINED RECONFIGURABLE ARCHITECTURE

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof.dr. W.H.M. Zijm,
on account of the decision of the graduation committee,
to be publicly defended
on Friday, September 8th, 2006, at 15.00

by

Yuanqing Guo

born on June 27, 1973
in Beiliu, Chang'an, Shaanxi, China

This dissertation is approved by:

Prof. Dr. Ir.	Thijs Krol	(promotor)
Dr. Ir.	Gerard Smit	(assistant promotor)

Acknowledgements

The research presented in this thesis could not have been accomplished without the help and support from many people. I would like to express my sincere gratitude to all of them. Here I would like to mention some of them in particular.

First of all I am very grateful to my assistant promotor, Gerard Smit, for his guidance. Thanks to Gerard, I did not experience the common problem among many Ph.D students: do not know what to do at the very beginning. When I started the project Gerard had many discussions with me to help me to get into the subject. I am also very impressed by his optimism. He always tried to find a direction when I felt "no solution". He suffered a lot from my writing. And on the other hand, I benefited a lot from his comments.

Prof. Thijs Krol is my promotor. With his extensive knowledge on modeling and graph transformation, he helped me a lot in solving the problems that hindered me for a long while by his simple yet clear way of explanation.

I feel so lucky to meet Prof. Cornelis (Kees) Hoede. Kees helped me with the scheduling algorithms. His mathematical background and the way of thinking kept on benefitting me. He taught me the skills of scientific writing. And he helped me to revise my writing. I cannot express my gratitude enough to him for all the help I received from him.

I appreciate Prof. Dr. Hajo Broersma for his supervision on the clustering algorithm. He understood my points even before me!

Paul Heysters is the main designer of the Montium. Unavoidably I asked him many questions and I always received clear answers. My work is most close to that of Michel Rosien's. We had a nice cooperation. And furthermore, he always helped me with C++ and discussed me with my problem. My roommate, Omar Mansour, also had many discussions with me. Many other colleagues in the Chameleon-related projects, Lodewijk Smit, Gerard Rauwerda, Qiwei Zhang, Nikolay Kavaldjiev, Maarten Wiggers, Pas-

cal Wolkotte, Sjoerd Peerlkamp, gave me a lot of help during my work and made the working environment lovely.

Seven years ago, Prof. Jack van Lint gave me an opportunity to came to the Netherlands. That opportunity changed my career life a lot. I am very sorry to hear the bad news that he has passed away. I can only keep my appreciation to him in my mind.

I also want to thank Wang Xinmei, my supervisor in Xidian University for my master program and Frans Willems, my supervisor in Technical University of Eindhoven for my MTD program (master of technological design). They guided me in the information theory domain, which is still my favorite subject.

There is life outside the university. Friends are treasures of the lifetime. The parties with Paul Volf, Mortaza Barge, Behnaz and Kimia are always accompanied by lots of fun and laugh. I especially treasure the gatherings that we had together.

I got to know many friends during my stay in Enschede. I can only mention the names of some of them here: Wu Jian, Wang Yu, He Hong, Zhang Dongsheng, Xu Jiang, Liu Fei, Wang Wuyang, Luo Dun, Pei Linlin, Jin Yan, Liang Weifeng, Liu Di, Wu Lixin, Cheng Jieyin, Luo Yiwei, Zhang Junnian.

Last but not least, I want to thank all my family members. My husband (Jianbo Zhang) always supported unconditionally ; My son (Andy Zhang) made me understand the real meaning of "love"; My parents give me all the love and help they can offer; My brothers (Guo Baiyi and Guo Baiwei), my sisters in law (Jia Baoying and Yao Na) and my lovely nephew (Guo Tiancheng) are always there when I need them.

Enschede, September 2006
Yuanqing Guo

Abstract

Today the most commonly used system architectures in data processing can be divided into three categories: general purpose processors, application specific architectures and reconfigurable architectures. General purpose processors are flexible, but inefficient and for some applications do not offer enough performance. Application specific architectures are efficient and give good performance, but are inflexible. Recently reconfigurable systems have drawn increasing attention due to their combination of flexibility and efficiency. Reconfigurable architectures limit their flexibility to a particular algorithm domain. Two types of reconfigurable architectures exist: fine-grained in which the functionality of the hardware is specified at the bit level and coarse-grained in which the functionality of the hardware is specified at the word level. High-level design entry tools are essential for reconfigurable systems, especially coarse-grained reconfigurable architectures. However, the tools for coarse-grained reconfigurable architectures are far from mature.

This thesis proposes a method for mapping applications onto a coarse-grained reconfigurable architecture. This is a heuristic method which tackles this complex problem in four phases: translation, clustering, scheduling and allocation. In this thesis, the Montium tile processor, a coarse-grained reconfigurable architecture, is used to demonstrate the proposed mapping method. In the translation phase, an input program written in a high-level language is translated into a control data flow graph; and some transformations and simplifications are done on the control data flow graph. In the clustering phase, the data flow graph is partitioned into clusters and mapped onto an unbounded number of fully connected Arithmetic Logic Units (ALUs). The ALU structure is the main concern of this phase and we do not take the inter-ALU communication into consideration. In the scheduling phase the graph obtained from the clustering phase is scheduled taking the maximum number of ALUs into account. The scheduling algorithm tries to minimize the num-

ber of clock cycles used for the given application under the constraints of the number of distinct reconfigurations of ALUs. In the allocation phase, variables are allocated to memories or registers, and data moves are scheduled. The main concern in this phase is the details of the target architecture.

There are many constraints and optimization objectives that need to be considered during the mapping and scheduling procedure. According to the above mentioned division only parts of constraints and optimization goals are considered in each phase, which simplifies the problem dramatically. Furthermore, after the division, part of our work relates to the existing research work, especially the work in the clustering and the scheduling phases. This connection is very important because on one hand, we can build our work on the results of others; on the other hand, the result of our work can also be used by others.

Different levels of heuristics are used in the mapping procedure. In our opinion, using a greedy method is the only practical choice because the original mapping task is too complex to be handled in an optimal way. To decrease the negative consequences caused by the sequential order of the heuristics, when tackling each subproblem, we take the requirements of other subproblems also into consideration.

In the Montium, a two-layer decoding technique is used. This technique limits the configuration space to simplify the structure of instruction fetching, which is good for reducing energy consumption and cost. However, the compiler has to face the challenge of decreasing the number of distinct configurations, i.e., generating a limited number of different instructions, which is the most difficult requirement to the Montium compiler. This is also the main difference between the Montium compiler and a compiler for other architectures. However, these constraints have a good reason: they are inherent to energy efficiency, so we believe that it will be used more and more in future energy-efficient computing architectures. Therefore, we have to find ways to cope with these constraints.

To generate only a few different instructions, our approach tries to find the regularity in the algorithms. Therefore, regularity selection algorithms are used (e.g., template selection in the clustering phase, pattern selection in the scheduling phase). These algorithms find the most frequently used templates or patterns in an application. By using these templates or patterns, the number of different instructions is small.

Samenvatting

De systeem architecturen die vandaag de dag het meest in gebruik zijn, kunnen worden onderverdeeld in drie categorieën: processoren voor algemene doeleinden, architecturen voor specifieke applicaties en herconfigureerbare architecturen. Processoren voor algemene doeleinden zijn flexibel maar niet efficiënt en voor sommige applicaties presteren ze niet genoeg. Architecturen voor specifieke applicaties zijn efficiënt en presteren goed maar zijn niet flexibel. De laatste tijd krijgen herconfigureerbare systemen steeds meer aandacht vanwege de combinatie van efficiëntie en flexibiliteit. Herconfigureerbare architecturen beperken hun flexibiliteit tot een bepaald algorithme domein. Er bestaan twee soorten herconfigureerbare architecturen: fijn-mazig, waar de functionaliteit is gespecificeerd op bit niveau en grof-mazig, waar de functionaliteit is gespecificeerd op woord niveau. Goede software ondersteuning voor het ontwerpen op hoog abstractie niveau is essentieel voor herconfigureerbare systemen, in het bijzonder grof-mazige herconfigureerbare architecturen. Echter, de software ondersteuning voor grof-mazige herconfigureerbare architecturen zijn verre van volwassen.

Dit proefschrift stelt een methode voor om applicaties op een grof-mazige herconfigureerbare architectuur af te beelden. Dit is een op heuristieken gebaseerde methode die dit probleem probeert op te lossen in vier fases: vertaling, groeperen, plannen en allocatie. In dit proefschrift wordt de Montium tile processor, een grof-mazige herconfigureerbare architectuur, gebruikt om de voorgestelde afbeelding te demonstreren. In de vertalingfase wordt een invoer programma, geschreven in een hoog niveau programmeertaal, vertaald in een “Control Data Flow” graaf. Op die graaf worden dan een aantal transformaties en simplificaties uitgevoerd. In de groeperingfase wordt de graaf gepartitioneerd in groepen en afgebeeld op een onbegrensd aantal volledig verbonden “Arithmetic Logic Units (ALUs)”. De ALU structuur is het hoofd doel van deze fase. We houden geen rekening met communi-

catie tussen de ALUs in deze fase. In de planningfase word de graaf, die verkregen is uit de groepering fase, ingepland, rekening houdend met het maximale aantal ALUs. Het planningalgorithme probeert het aantal klok cycles voor een gegeven applicatie te minimaliseren, rekening houdend met de beperkingen die worden opgelegd door het beperkt aantal herconfiguraties die mogelijk zijn per ALU per applicatie. In de allocatie fase worden variabelen toegewezen aan geheugens of registers en data transporten worden ingepland. De hoofdzaak in deze fase zijn de details van de doel architectuur.

Er zijn veel beperkingen en optimalisatie doelen die in overweging moeten worden genomen tijdens het afbeelden en inplannen. Volgens de onderverdeling die hierboven is genoemd worden in elke fase slechts een deel van de beperkingen bekeken. Dit simplificeert het probleem enorm. Tevens, na de onderverdeling kan een deel van ons werk gerelateerd worden aan eerder onderzoek, in het bijzonder de groepering en planning fases. Dit is erg belangrijk omdat we ons onderzoek dus kunnen baseren op het werk van anderen en anderen ons werk kunnen gebruiken in hun onderzoek.

Verschillende niveaus van heuristiek worden gebruikt tijdens de afbeelding. Volgens ons is een "greedy" methode de enige practische keuze omdat de oorspronkelijke afbeelding te complex is om een optimale oplossing te vinden. Om de negatieve consequenties, veroorzaakt door de sequentiële volgorde van de heuristieken tijdens het oplossen van elk sub-probleem, te verminderen, nemen we de eisen en beperkingen van andere sub-problemen ook in overweging.

In de Montium wordt een twee-laags decoderingstechniek gebruikt. Deze techniek beperkt de configuratieruimte om de structuur van het ophalen van instructies te simplificeren. Dit is goed om het energie verbruik en de kosten te reduceren. Echter, de compiler heeft dan het probleem om het aantal verschillende configuraties te verminderen, d.w.z., om een beperkt aantal verschillende configuraties te genereren. Dit is de moeilijkste eis waar de Montium Compiler aan moet voldoen. Dit is ook het grootste verschil tussen de Montium Compiler en een compiler voor andere architecturen. Deze beperkingen hebben echter een goede reden, ze zijn inherent aan energie efficiëntie. Wij geloven dus dat het in de toekomst meer en meer gebruikt zal worden in toekomstige energie efficiënte architecturen. Daarom moeten we een manier vinden om met deze beperkingen te kunnen werken.

Om het voor elkaar te krijgen om een beperkt aantal verschillende instructies te genereren, is onze aanpak het zoeken van de regelmaat in algo-

rithmes door middel van algorithmes die selecteren op basis van regelmaat (b.v. template selectie in de groeperingfase, patroon selectie in de planningfase). Deze algorithmes vinden de meest gebruikte patronen en templates in een applicatie. Door het gebruik van deze templates en patronen is het aantal verschillende instructies klein.

Table of Contents

Chapter 1

Introduction

Reconfigurable computing has been gaining more and more attention over the last decade. To automate the application design flow for coarse-grained reconfigurable systems, the CADTES group at the University of Twente is designing a compiler for such systems. The procedure of mapping computationally intensive tasks to coarse-grained reconfigurable systems is the key part in the compiler, which is the topic of this thesis.

1.1 Reconfigurable architectures

A computer system executes user programs and solves user problems. The most commonly used computer architectures in data processing can be divided into three categories: general purpose processors, application-specific integrated circuits and reconfigurable architectures.

In this thesis, by *performance* of a processor, we mean the amount of clock cycles it takes to run a specific application program under certain specified

conditions; *flexibility* refers to the programmability of the system; *energy-efficiency* refers to the relative energy it takes to perform a certain program on a certain architecture.

The designs of these architectures make different tradeoffs between flexibility, performance and energy-efficiency. General purpose processors are flexible, but inefficient and offer relatively poor performance, whereas application specific architectures are efficient and give good performance, but are inflexible. Reconfigurable architectures make a tradeoff between these two extremes thereby limiting their flexibility to a particular algorithm domain. Two types of reconfigurable architectures exist: coarse-grained and fine-grained. In fine-grained reconfigurable architectures, such as Field Programmable Gate Arrays (FPGAs), the functionality of the hardware is specified at the bit level. Therefore, they are efficient for bit-level algorithms. However, word-level algorithms are more suitable for coarse-grained reconfigurable architectures. Coarse-grained reconfigurable elements also provide an energy advantage over fine-grained reconfigurable elements because the control part is small.

To compete with application specific architectures, which are notorious for their long design cycles, high-level design entry tools are needed for reconfigurable architectures. Without proper tools, developing these architectures is a waste of time and money. In recent years, a number of companies that developed reconfigurable systems have gone bankrupt (e.g., Chameleon systems [80] [86], Quicksilver [42]) because they designed systems without proper development tools.

1.2 Design automation

In a computer system, the hardware provides the basic computing resources. The applications define the way in which these resources are used to solve the computing problems. Tools act as bridges between hardware resources and applications. According to the abstraction level of the source language, tools are divided into low-level design entry tools and high-level design entry tools. Low-level design entry tools such as Hardware Description Languages (HDLs) and assemblers require the programmer to have a thorough understanding of the underlying hardware organization. The programmer has a long learning curve before he or she is experienced enough to exploit all the features of the architecture. The design effort and costs are high and the design cycle

is long. High-level design entry languages (e.g., C or MATLAB) provide an abstraction of the underlying hardware organization. Two main advantages of a high-level design entry tool are:

- Architecture independent application. With the help of such tools, application engineers can create algorithms without being an expert on the underlying hardware architectures. This makes it possible for application engineers to focus on the algorithms instead of on a particular architecture. This also reduces the design cycle of new applications considerably and saves engineering costs.

- Automatic exploitation of the desirable features (such as parallelism) of the hardware architectures. Specifications of algorithms are often provided in a sequential high-level language. Human beings are accustomed to thinking sequentially and have difficulty in extracting the maximum amount of parallelism achievable on a specific architecture from a sequential description. Good high-level design entry tools automatically manage the allocation of low level hardware, and allow the resources to cooperate in the best way.

High-level design entry tools for general purpose processors have been studied for a long time and the techniques are already very mature, giving general purpose processors strong support. Although high-level design entry tools are essential for reconfigurable systems, especially coarse-grained reconfigurable architectures, the tools for these architectures are far from mature.

1.3 Chameleon system-on-chip

The application domain of our research is energy-efficient mobile computing, e.g., multi-standard communicators or mobile multimedia players. A key challenge in mobile computing is that many attributes of the environment vary dynamically. For example, mobile devices operate in a dynamically changing environment and must be able to adapt to each new environment. A mobile computer will have to deal with unpredictable network outages and be able to switch to a different network without changing the application. Therefore, it should have the flexibility to handle a variety of multimedia services and standards (such as different communication protocols or video

decompression schemes) and the ability to adapt to the nomadic environment and available resources. As more and more these applications are added to mobile computing, devices need more processing power. Therefore, performance is another important requirement for mobile computing. Furthermore, a handheld device should be able to find its place in a user's pocket. Therefore, its size should be small enough. Finally, for compelling business reasons, extending the time between battery recharges has long been one of the highest priorities in the design of mobile devices. This implies a need for ultra-low energy consumption. To summarize: computer architectures for mobile computing should be flexible, with high performance and high energy-efficiency.

The Chameleon System-on-Chip (SoC) [43][82] is a heterogeneous reconfigurable SoC designed for mobile applications. Such a SoC consists of several tiles (e.g., 16 tiles). These tiles can be heterogeneous, for example, general-purpose processors (such as ARM cores), bit-level reconfigurable parts (such as FPGAs), word-level reconfigurable parts (such as Montium tiles described in detail later), general Digital Signal Processors (DSPs) or some application-specific integrated circuits (such as a Turbo-decoder). This tile-based architecture combines the advantages of all types of architectures by mapping application tasks (or kernels) onto the most suitable processing entity. We believe that the efficiency (in terms of performance and energy) of the system can be improved significantly by a flexible mapping of processes to processing entities [84].

The Chameleon SoC design concentrates on streaming applications. Examples of these applications are video compression (discrete cosine transforms, motion estimation), graphics and image processing, data encryption, error correction and demodulation, which are the key techniques in mobile computing.

1.4 Montium tile processor

As an example in this section, the Montium tile processor, a coarse-grained reconfigurable architecture designed by the Computer Architecture Design and Test for Embedded Systems (CADTES) group at the University of Twente [43][82], is described. The Montium is used throughout this thesis as an example of a coarse-grained building block. The Montium tile is characterized by its coarse-grained reconfigurability, high performance and

low energy consumption. The Montium achieves flexibility through reconfigurability. High performance is achieved by *parallelism*, because the Montium has several parallel processing elements. Energy-efficiency is achieved by the *locality of reference* which means data are stored close to the processing part that uses them. It costs considerably less energy for a processing part to access the data stored in an on-tile storage location compared to the access of data from off-chip memory. The Montium tiles allow different levels of storage: local register, local memory, global memory and global register. This allows several levels of locality of reference. More details about the Montium structure can be found in Chapter 2.

A new technique that is used extensively in the Montium tile design is the two-layer decoding method (see Chapter 3). This method limits configuration spaces to simplify the instruction fetch part, which is good for energy consumption and cost. The disadvantage incurred by this technique is the limited flexibility. However, the Montium does not aim to provide a general purpose hardware for all types of complex algorithms found in conventional microprocessors. Instead, the Montium is designed for DSP-like algorithms found in mobile applications. Such algorithms are usually regular and have high computational density.

1.5 A compiler for coarse-grained reconfigurable systems

The philosophy of the design of a coarse-grained reconfigurable system is to keep the hardware simple and let the compiler bear more responsibilities. However, it is not an easy task to use all the desirable characteristics of a coarse-grained reconfigurable architecture. If those characteristics are not used properly, the advantages of an architecture might become its disadvantages. Therefore, a strong compiling support designed alongside the work of the hardware design is a necessity.

There are many other compilers for coarse-grained reconfigurable architectures described in literatures, which will be introduced in Chapter 3. The most remarkable point that distinguishes our work from the other compilers for coarse-grained reconfigurable architectures is that we limited the size of configuration spaces for energy-efficiency. In most microprocessors, instructions are stored in an instruction memory. At run time, instructions

are fetched and decoded, and fed to all components. In the Montium, instructions for each component are stored in a local-configuration memory. A sequencer acts as a state machine to choose the instructions for each component. Therefore, the number of different instructions is limited by both the size of sequencer-instruction memory and the size of local-configuration memory. This property can be summarized in an easily-understood sentence: "we prefer the same instructions to be executed repeatedly rather than many different instructions to be executed once." This instruction fetching technique is very energy-efficient and, we believe, it will be used more and more in future computing architectures. Therefore, new mapping techniques that can handle constraints of the configuration space are being investigated.

This thesis is about the compiler for a coarse-grained reconfigurable architecture. The Montium tile processor is used as a sample architecture. This thesis concentrates on mapping an application program written in a high level language (C++ or Matlab) to a coarse-grained reconfigurable processing tile within the constraints of the configuration space. It focuses on decreasing the number of clock cycles under all the constraints of the processor structure.

1.6 Contributions of this thesis

The major contributions of this thesis can be summarized as follows:

- **Mapping and scheduling framework:** The framework for the mapping applications to a coarse-grained reconfigurable architecture is proposed. It has four phases: Translation, clustering, scheduling and allocation.

- **Data value representation:** The presented new data value representation is very convenient for generating AGU instructions.

- **Template generation algorithm:** We present a template generation which can generate all the connected templates and finding all the matches in a given graph. The shape of templates is not limited as in the previous work. The algorithm can be used in application-specific integrated circuit design or generating CLBs in FPGAs.

- **Template selection algorithm:** The template selection algorithm presented choose the generated templates to cover a graph, which min-

imizes the number of distinct templates that are used as well as the number of instances of each template.

- **Color-constrained scheduling:** The scheduling problem for the Montium is modeled as a color-constrained scheduling problem. A three-step approach for color-constrained scheduling problem is proposed.
- **Multi-pattern list scheduling algorithm:** A modified version of the traditional list scheduling algorithm is presented. The modified algorithm schedules the nodes of a graph using a set of given patterns.
- **Pattern selection algorithm:** The algorithm finds the most frequently occurring patterns in a graph. By using these selected patterns, the multi-pattern list scheduling can schedule a graph using fewer clock cycles.
- **Column arrangement algorithm:** This algorithm is developed to decrease the number of one-ALU configurations of each ALU.
- **Resource allocation approach:** The procedure of resource allocation is presented.
- **Modeling of the crossbar allocation, register allocation, memory allocation problems:** The crossbar allocation, register allocation, memory allocation problems are modeled.

1.7 Organization of this thesis

The rest of this thesis is organized as follows:

- Chapter 2 presents coarse-grained reconfigurable architectures and in particular, the structure of the Montium tile processor architecture, which is the sample architecture of the mapping work described in this thesis.
- Chapter 3 gives an overview of other work related to the compilers for coarse-grained reconfigurable architectures. Further, the challenges of the mapping problem for coarse-grained reconfigurable architectures are discussed. The framework of our four-phase approach consists of transformation, clustering, scheduling and allocation.

- Chapter 4 presents the translation and transformation phase. In this phase, an input C program is first translated into a control data flow graph. After that some transformations and simplifications are performed on the control data flow graph.
- Chapter 5 presents the clustering phase. In this phase, the primitive operations of a data flow graph are partitioned into clusters and mapped to an unbounded number of fully connected ALUs.
- Chapter 6 presents the scheduling algorithm, which schedules the clusters within the constraint of the configuration space.
- Chapter 7 presents the allocation algorithm, which focuses on the details of the Montium structure. Two mains jobs executed during the allocation phase are the allocation of variables and the arrangement of communications.
- Chapter 8 summarizes the thesis and presents the final conclusions.

Chapter 2

Coarse-grained reconfigurable architectures

This chapter presents the commonly used computer architectures: general-purpose processor, application specific integrated circuits and reconfigurable architectures. Reconfigurable architectures can be divided into two groups: fine-grained and coarse-grained. The advantages and disadvantages of these architectures are discussed. Coarse-grained reconfigurable architectures are the focus of the chapter. Several sample architectures are described.

2.1 Introduction

A computer system has many hardware resources that may be required to solve a problem: arithmetic and logic units, storage space, input/output devices, etc. The most commonly used computer system architectures in data

processing can be divided into three categories: general purpose processors, application specific architectures and coarse-grained reconfigurable architectures.

2.2 General purpose processor

General-purpose computers have served us well over the past couple of decades. The architecture of a general purpose processor is widely studied, and many optimizations of processor performance have been done. Current general purpose Central Processing Units (CPUs) are many orders of magnitude more powerful than the first ones. They are also the most flexible hardware architectures. Simply by writing the right software, an application programmer can map a large set of applications on general purpose processor hardware. Moreover, tooling support for general-purpose processors has been researched for a long period and many mature tools are now available. Compared with other architectures, general purpose processors are easier to program. The application design procedure is also faster and cheaper.

All general purpose processors rely on the von Neumann instruction fetch-and-execute model. Although general purpose processors can be programmed to perform virtually any computational task, with the von Neumann model, they have to pay for this flexibility with a high energy consumption and significant overhead of fetching, decoding and executing a stream of instructions on complex general purpose data paths. The model has some significant drawbacks. Firstly, the energy overhead due to its programmability most often dominates the energy dissipation of the intended computation. Every single computational step, (e.g., addition of two numbers), requires fetching and decoding an instruction from the instruction memory, accessing the required operands from the data memory, and executing the specified computation on a general-purpose functional unit. Secondly, the clock speed of processors has grown much faster than the speed of the memory. Therefore, sequential fetching of every control instruction from the memory hampers the performance of the function unit. The gap between memory access speed and processor speed is known as the von Neumann bottleneck. Some techniques, such as using cashes or separating instruction and data memories, have been used to relieve the von Neumann bottleneck. However, these efforts are not sufficient for mobile applications where the computational density is very high. Thirdly, to achieve high performance, a general-purpose

processor must run at a high clock frequency; therefore, the supply voltage cannot be aggressively reduced to save energy. Finally, although the execution speed of general purpose processors has been increased many times over the last decades, it has, as a general rule, led to inefficiency compared to an application specific implementation of a particular algorithm. The reason is that many optimizations in general purpose processors are for general cases but not for a specific algorithm.

A digital signal processor is a processor optimized for digital signal processing algorithms. We consider it as a general purpose processor because it still uses the von Neumann model. The need of fetching every single instruction from memory and decoding it brings also in these processors a considerable amount of power consumption overhead.

2.3 Application specific integrated circuit

An application specific integrated circuit, as opposed to a general purpose processor, is a circuit designed for a specific application rather than general use. The goal of application specific modules is to optimize the overall performance by only focusing on their dedicated use. Also due to the application-oriented design goal, the application specific integrated circuit presents the most effective way of reducing energy consumption and has shown to lead to huge power savings. Performing complex multimedia data processing functions in dedicated hardware, optimized for energy-efficient operation, reduces the energy per operation by several orders of magnitude compared with a software implementation on a general purpose processor. Furthermore, for a specific use, an application-specific integrated circuit has lower chip area costs compared to a general purpose processor. However, the disadvantage of dedicated hardware is the lack of flexibility and programmability. Their functionality is restricted to the capabilities of the hardware. For each new application, the hardware has to be redesigned and built. The technological challenges in the design of custom application specific architectures are usually significantly smaller than the design of general purpose circuits. This may compensate for the disadvantages in some applications. However, the smaller flexibility, and consequently the fact that a new chip design is needed for even the smallest change in functionality, is still the fatal shortcoming of application specific integrated circuits.

2.4 Reconfigurable hardware

An application specific architecture solution is too rigid, and a general purpose processor solution is too inefficient. Neither general purpose processors nor application specific architectures are capable of satisfying the power and flexibility requirements of future mobile devices. Instead, we want to make the machine fit the algorithm, as opposed to making the algorithm fit the machine. This is the area of reconfigurable computing systems.

Reconfigurable hardware is ideal for use in System-on-Chips (SoCs) as it executes applications efficiently, and yet maintains a level of flexibility not available with more traditional full custom circuitry. This flexibility allows for both hardware reuse and post fabrication modification. Hardware reuse allows a single reconfigurable architecture to implement many potential applications, rather than requiring a separate custom circuit for each. Furthermore, post-fabrication modification allows for alterations in the target applications, bug-fixes, and reuse of the SoC across multiple similar deployments to amortize design costs. Unlike microprocessors in which functionality is programmable through 'instructions', reconfigurable hardware changes its functionality within its application domain through 'configuration bits', which means the programmability is lower than that of a general purpose processor.

Fine-grained: Reconfigurable processors have been widely associated with Field Programmable Gate Array (FPGA)-based system designs. An FPGA consists of a matrix of programmable logic cells with a grid of interconnecting lines running between them. In addition, there are I/O pins on the perimeter that provide an interface between the FPGA, the interconnecting lines and the chip's external pins. However, FPGAs tend to be somewhat fine-grained in order to achieve a high degree of flexibility. This flexibility has its place for situations where the computational requirements are either not known in advance or vary considerably among the needed applications. However, in many cases this extreme level of flexibility is unnecessary and would result in significant overheads of area, delay and power consumption.

Coarse-grained: Contrasted with FPGAs, the data-path width of coarse-grained reconfigurable architectures is more than one bit. Over the last 15 years, many projects have investigated and successfully built systems where

the reconfiguration is coarse-grained and is performed within a processor or amongst processors [39] [40]. In such systems the reconfigurable unit is a specialized hardware architecture that supports logic reconfiguration. The reconfiguration procedure is much faster than that found in FPGAs. Because the application domain is known, full custom data paths could be designed, which are drastically more area-efficient.

2.5 Select overview of coarse-grained reconfigurable architectures

Many coarse-grained reconfigurable architectures have been developed over the last decades. Here a selected overview is given. For a more comprehensive overview of reconfigurable architectures, we refer to [15] and [39].

2.5.1 The Pleiades architecture

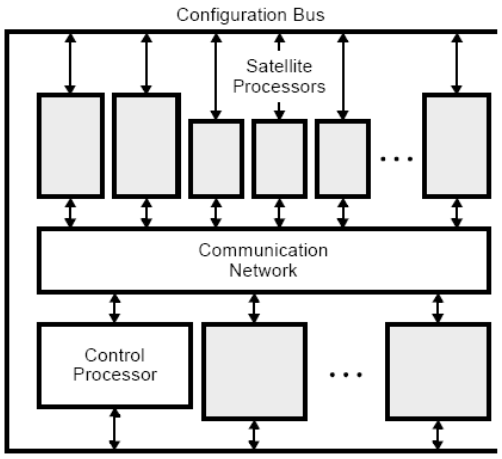

Figure 2.1: The Pleiades Architecture Template

The Pleiades architecture was designed by the University of California at Berkeley for digital signal processing algorithms [1] [75] [76]. The architecture is centered around a reconfigurable communication network (see Figure 2.1). Connected to the network are a control processor that is a general-purpose

microprocessor core, and an array of heterogeneous autonomous processing elements called satellite processors. The satellites can be fixed components or reconfigurable data paths. The control processor configures the available satellite processors and the communication network at run-time to construct the dataflow graph corresponding to a given computational kernel directly in the hardware.

The dominant, energy-intensive computational kernels of a given DSP algorithm are implemented on the satellite processors as a set of independent, concurrent threads of computation. The remainder of the algorithm, which is not compute-intensive, is executed on the control processor.

2.5.2 Silicon Hive's reconfigurable accelerators

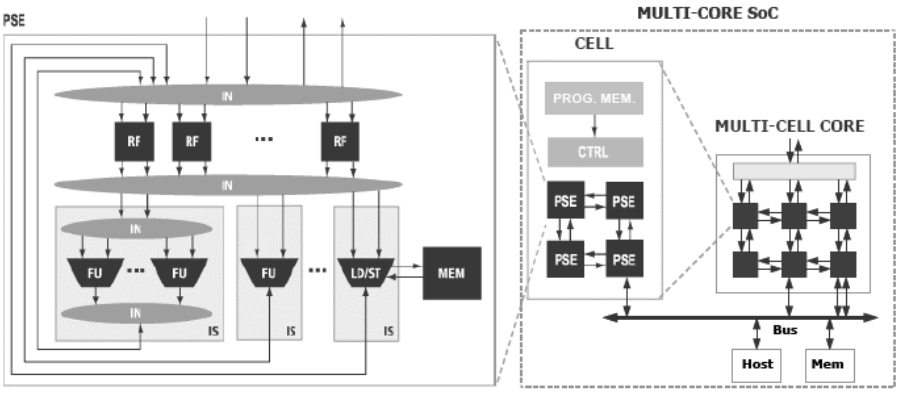

Figure 2.2: Hierarchy of Silicon Hive's Processor Cell Template

The Silicon Hive processor cell template is a hierarchical structure (see Figure 2.2) [13] [37]. A core consists of multiple cells, each of which has its own thread of control. A cell has a VLIW-like controller (CTRL), a configuration memory and multiple Processing and Storage Elements (PSEs). Cells can have streaming interfaces, which allow the cells to be interconnected. For scalability reasons, usually a nearest-neighbor interconnect strategy is chosen, leading to a mesh structure. PSEs consist of register files and issue slots. Issue slots consist of interconnect networks and function units.

A cell is a fully-operational processor capable of computing complete algorithms. A cell typically executes one algorithm at a time.

2.5.3 The MorphoSys reconfigurable processor

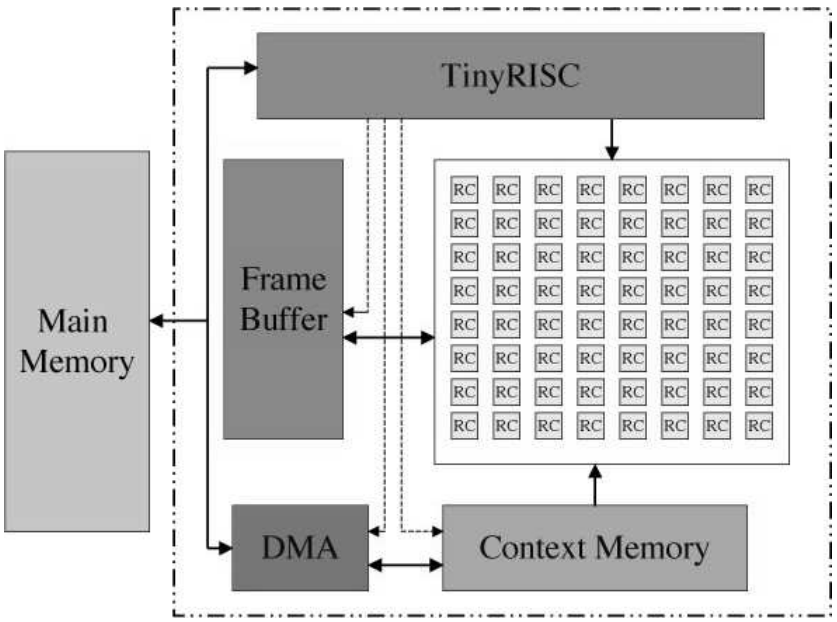

Figure 2.3: The architecture of MorphoSys systems

The MorphoSys reconfigurable processor was developed by researchers in the University of California, targeted at applications with inherent data-parallelism, high regularity and high throughput requirements. The MorphoSys architecture, shown in Figure 2.3, comprises an array of Reconfigurable Cells (RC Array) with configuration memory (Context Memory), a control processor (TinyRISC), data buffer (Frame Buffer) and DMA controller. The main component of MorphoSys is the 8 x 8 RC array. Each RC has an ALU-multiplier, a register file and is configured through a 32-bit context word. The RC Array is dedicated to the exploitation of parallelism available in an applications algorithm. The tiny RISC handles serial operations, initiates data transfers and controls operation of the RC array. The Frame Buffer enables stream-lined data transfers between the RC Array and main memory, by overlap of computation with data loading and storing.

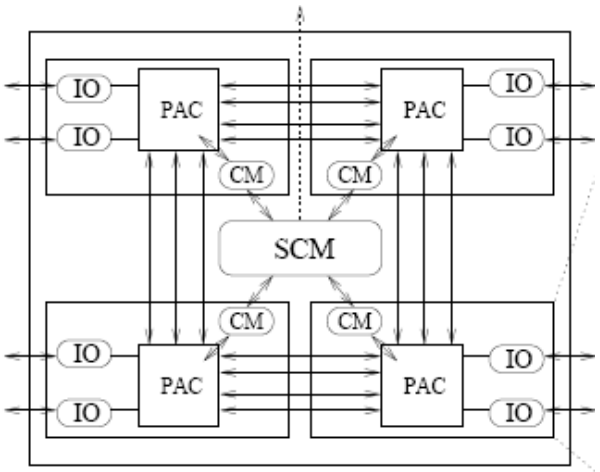

Figure 2.4: XPP device

2.5.4 PACT's extreme processor platform

The eXtreme Processing Platform (XPP) [7] [69] of PACT is a data processing architecture based on an array of coarse-grained, adaptive computing elements called Processing Array Elements (PAEs) and a packet-oriented communication network.

An XPP device contains one or several Processing Array Clusters (PACs) (see Figure 2.4). Each PAC is a rectangular array of PAEs, attached to a control manager responsible for writing configuration data into the PAEs (see Figure 2.5). PAEs can be configured rapidly in parallel while neighboring PAEs are processing data. Entire applications can be configured and run independently on different parts of the array. A PAE is a template for either an ALU named ALU-PAE or for a memory named RAM-PAE. Those in the center of the array are ALU-PAEs, and those at the left and right side are RAM-PAEs with I/O.

A flow graph of an algorithm can be mapped onto a PAC in a natural way. The nodes of the data flow graph are mapped on the PAEs and the edges are mapped on the data network.

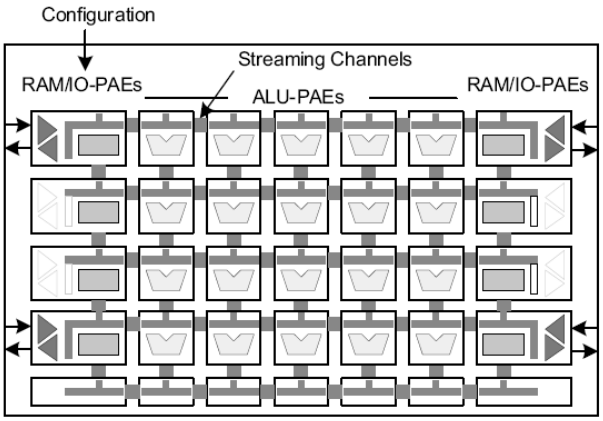

Figure 2.5: Structure of a sample XPP Core

2.5.5 Montium tile processor

The Montium tile processor is being designed at the University of Twente. We will describe the Montium tile processor in more detail than other architectures because it functions as the sample architecture of our compiler. The Montium tile is acting as a tile in the Chameleon system-on-chip, which is especially designed for mobile computing.

Chameleon System-on-Chip

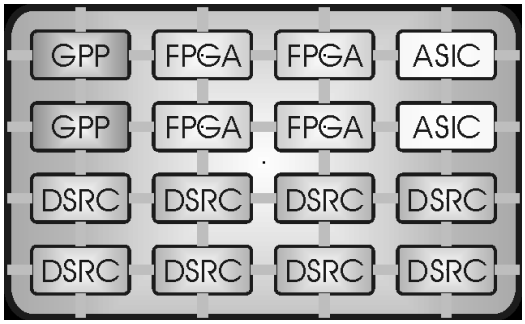

Figure 2.6: Chameleon heterogeneous SoC architecture

In the Chameleon project we are designing a heterogeneous reconfigurable

SoC [82] (see Figure 2.6). This SoC contains a general purpose processor (e.g. ARM core), a bit-level reconfigurable part (e.g., FPGA) and several word-level reconfigurable parts (e.g., Montium tiles).

We believe that in the future 3G/4G terminals, heterogeneous reconfigurable architectures are needed. The main reason is that the efficiency (in terms of performance or energy) of the system can be improved significantly by mapping application tasks (or kernels) onto the most suitable processing entity. The design of the above-mentioned architecture is useless without a proper tool chain supported by a solid design methodology. At various levels of abstraction, modern computing systems are defined in terms of processes and communication (or synchronization) between processes. These processes can be executed on various platforms (e.g., general purpose CPU, Montium, FPGA, etc). In this thesis we will concentrate on the tools for mapping one such process onto a coarse-grained reconfigurable processing tile (Montium). However, the presented algorithms can also be used for programming FPGAs or other coarse-grained architectures.

Montium Architecture

The design of the Montium focuses on:

- Keeping each processing part small to maximize the number of processing parts that can fit on a chip;
- Providing sufficient flexibility;
- Low energy consumption;
- Exploiting the maximum amount of parallelism;
- A strong support tool for Montium-based applications.

In this section we give a brief overview of the Montium architecture [43]. Figure 2.7 depicts a single Montium processor tile. The hardware organization within a tile is very regular and resembles a very long instruction word architecture. The five identical arithmetic and logic units (ALU1$\cdots$ALU5) in a tile can exploit spatial concurrency to enhance performance. This parallelism demands a very high memory bandwidth, which is obtained by having 10 local memories (M01$\cdots$M10) in parallel. The small local memories are also motivated by the locality of reference principle. The Arithmetic and

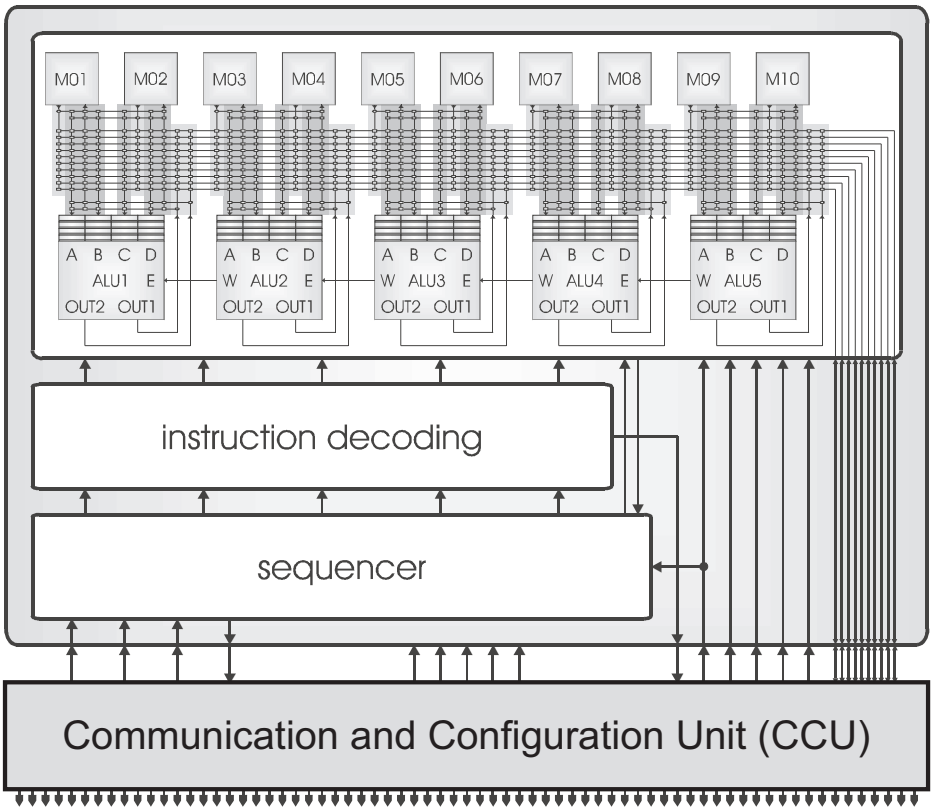

Figure 2.7: Montium processor tile

Logic Unit (ALU) input registers provide an even more local level of storage. Locality of reference is one of the guiding principles applied to obtain energy-efficiency within the Montium. A vertical segment that contains one ALU together with its associated input register files, a part of the interconnect and two local memories is called a Processing Part (PP). The five processing parts together are called the processing part array. A relatively simple sequencer controls the entire processing part array. The Communication and Configuration Unit (CCU) implements the interface with the world outside the tile. The Montium has a datapath width of 16-bits and supports both integer and fixed-point arithmetic. Each local static random access memory

is 16-bit wide and has a depth of 512 positions. Because a processing part contains two memories, this adds up to a storage capacity of 16 Kbit per local memory. A memory has only a single address port that is used for either reading or writing. A reconfigurable Address Generation Unit (AGU) accompanies each memory. The AGU contains an address register that can be modified using base and modification registers.

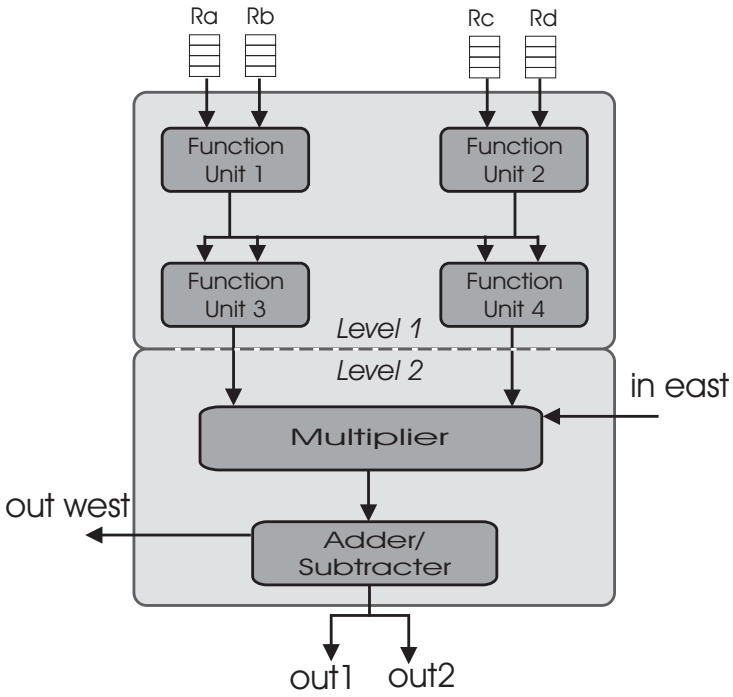

Figure 2.8: Montium ALU

It is also possible to use the memory as a lookup table for complicated functions that cannot be calculated using an ALU, such as sine or division (with one constant). A memory can be used for both integer and fixed-point lookups. The interconnect provides flexible routing within a tile. The configuration of the interconnect can change every clock cycle. There are ten buses that are used for inter-processing part array communication. Note that the span of these buses is only the processing part array within a single tile. The CCU is also connected to the global buses. The CCU uses the global buses to access the local memories and to handle data in streaming algorithms. Communication within a PP uses the more energy-efficient local buses. A single

ALU has four 16-bit inputs. Each input has a private input register file that can store up to four operands. The input register file cannot be bypassed, i.e., an operand is always read from an input register. Input registers can be written by various sources via a flexible interconnect. An ALU has two 16-bit outputs, which are connected to the interconnect. The ALU is entirely combinatorial and consequently there are no pipeline registers within the ALU. The diagram of the Montium ALU in Figure 2.8 identifies two different levels in the ALU. Level 1 contains four function units. A function unit implements the general arithmetic and logic operations that are available in languages like C (except multiplication and division). Level 2 contains the Multiply-ACcumulate (MAC) unit and is optimized for algorithms such as FFT and FIR. Levels can be bypassed (in software) when they are not needed.

Neighboring ALUs can also communicate directly on level 2. The West-output of an ALU connects to the East-input of the ALU neighboring on the left (the West-output of the leftmost ALU is not connected and the East-input of the rightmost ALU is always zero). The 32-bit wide East-West connection makes it possible to accumulate the MAC result of the right neighbor to the multiplier result (note that this is also a MAC operation). This is particularly useful when performing a complex multiplication, or when adding up a large amount of numbers (up to 20 in one clock cycle). The East-West connection does not introduce a delay or pipeline, as it does not contain registers.

2.5.6 Summary

Many coarse-grained reconfigurable architectures have been developed over the last 15 years. Although their details are different, they share several similarities:

- Coarse-grained reconfigurable architectures contain word-level function units, such as multipliers and arithmetic logic units.
- A major benefit of using word-level function units is a considerable reduction of configuration memory and configuration time.
- Reconfigurable communication networks are used in coarse-grained reconfigurable architectures. These networks support rich communication resources for efficient parallelism.

- Distributed storages (memories and registers) are used:
  - to obtain a reasonable locality of reference
  - to achieve a high memory bandwidth
  - to relieve the von Neumann bottleneck
- The designs of coarse-grained reconfigurable architectures are kept simple to achieve high performance and low energy consumption. Automatic design or compiling tools are developed alongside the hardware designs, which map applications to the target architectures, using the particular features of the hardware.

2.6 Conclusion

The existing computing architectures are divided into three groups: general purpose processers, application specific integrated circuits and reconfigurable architectures. General purpose processers are flexible, easy-to-use, but inefficient in performance and energy consumption. Application specific integrated circuits are the most efficient architectures, but inflexible. Reconfigurable architectures make a tradeoff between the efficiency and flexibility.

According to the width of the data-path, reconfigurable architectures are classified into two types: fine-grained and coarse-grained. In fine-grained reconfigurable architectures, the functionality of the hardware is specified at the bit level. In contrast, the data-path width in coarse-grained reconfigurable architectures is always more than one bit. This coarse granularity greatly reduces the delay, area, power and configuration time, compared with fine-grained architectures. However, these advantages come at the expense of flexibility compared with fine-grained architectures.

Compiler techniques are crucial for the future of coarse-grained reconfigurable architectures. Such tools allow application engineers to create algorithms without being an expert of the underlying hardware architectures. This reduces the design cycle and cost of new applications.

Chapter 3

The framework of a compiler for a coarse-grained reconfigurable architecture

This chapter[1] *first gives an overview of the related work for mapping applications to coarse-grained reconfigurable architectures. Then we select the Montium tile processor as a sample architecture for our compiler. After that the challenges of the mapping problem for the Montium tile are discussed. Finally the framework of our mapping procedure is presented, which adopts a four-phase division: transformation, clustering, scheduling and allocation.*

[1] Parts of this chapter have been published in publication [2] [5].

3.1 Introduction

In the process of designing a hardware architecture that is reconfigurable at run-time and is capable of performing many complex operations efficiently, the hardware engineers have come up with many trade-offs that often result in severe constraints being imposed on programming the processor. In this scenario, the compiler plays an important role of hiding the complex details of programming the embedded processor by allowing the users to write embedded applications in high level languages. The compiler is also responsible for ensuring that the embedded applications are translated to short and near-optimal code sequences. Only when the characteristics of a target reconfigurable architecture are used properly its strong points can be embodied during the application design. Otherwise the advantages of a system might become its disadvantages.

In the Gecko[2] project, a compiler is being developed to map applications to a coarse-grained reconfigurable architecture. The compiler is a high-level design entry tool which is capable of implementing programs written in a high-level language, such as C/C++ or Java, directly onto an array of reconfigurable hardware modules on a System-on-Chip (SoC).

3.2 Overview of mapping methods

Many coarse-grained reconfigurable architectures are actually clustered micro-architectures. Compilers for coarse-grained reconfigurable architectures compilers could use some techniques that exist for clustered micro-architectures. Therefore, we first give an overview of some related design methods for clustered micro-architectures.

3.2.1 Mapping techniques for clustered micro-architectures

In a clustered micro-architecture, the register file and functional units are partitioned and grouped into clusters. Each cluster is connected to an interconnection network to allow communication with other clusters.

[2]This research is supported by the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affaires and the technology foundation STW.

The problem of scheduling for clustered architectures consists of two main subproblems: (1) the assignment of data and operations to specific clusters; and (2) the coordination and scheduling of the moves of data between clusters.

In some of the previously published papers [24] [27] [49], cluster assignment and instruction scheduling are done sequentially. In other papers cluster assignment and scheduling are done in the reverse order. For example, Capitanio et al. considered the partitioning problems in [17] [18] for the Limited-Connectivity VLIW (LC-VLIW) architecture, where limited connectivity and register-to-register moves are allowed. According to [17] a partitioning algorithm is applied to a graph in order to divide the code into sub-streams that minimize a given cost function. Next inter-substream data movement operations are inserted and the code is recompacted.

In [68], the Unified Assign and Scheduling UAS algorithm is presented to perform clustering and scheduling on a clustered micro-architecture in a single step.

3.2.2 Compiling for coarse-grained reconfigurable architectures

Compiling applications written in a high-level language to coarse-grained reconfigurable platforms has been an active field of research in the recent past. The work in this domain is mostly highly dependent on the target architecture.

Typically, an application consists of computationally intensive 'kernels' that communicate with each other. Such an application can be described as a dataflow graph in which the nodes are 'kernels' or operations and the edges define the dependence between the nodes. The mapping work for the Pleiades SoC is done by directly mapping the dataflow graph of a kernel onto a set of satellite processors [1]. In this approach, each node or cluster of nodes in the dataflow graph corresponds to a satellite processor. Edges of the dataflow graph correspond to links in the communication network, connecting the satellite processors.

In Garp [41] [91], the reconfigurable hardware is an array of computing elements. The compiler draws heavily from techniques used in compilers for Very Long Instruction Word (VLIW) architectures to identify Instruction Level Parallelism (ILP) in the source program, and then schedules code

partitions for execution on the array of computing elements.

In CHIMAERA [94], the reconfigurable hardware is a collection of programmable logic blocks organized as interconnected rows. The focus of the compiler is to identify frequently executed instruction sequences and map them onto a Reconfigurable Functional Unit Operation (RFUOP) that will execute on the reconfigurable hardware.

PipeRench [12] [35] is a structure of configurable logic and storage elements interconnected by a network. The software development approach is to analyze the application's virtual pipeline, which is mapped onto physical pipe stages to maximize execution throughput. The compiler uses a greedy place and route algorithm to map these pipe stages onto the reconfigurable fabric.

The Reconfigurable Architecture Workstation (RAW) micro-architecture [90][6][58] comprises a set of inter-connected replicated tiles, each tile contains its own program and data memories, ALUs, registers, configurable logic and a programmable switch that can support both static and dynamic routing. The compiler partitions the program into multiple, coarse-grained parallel threads, each of which is then mapped onto a set of tiles.

The RaPiD architecture is a general coarse-grained reconfigurable architecture architecture with function units and buses. These are interconnected and controlled using a combination of static and dynamic control. The compilation techniques for RaPid are presented in [26].

Architecture for Dynamically Reconfigurable Embedded Systems (ADRES) is a template which couples a very-long instruction word (VLIW) processor and a coarse-grained array by providing two functional views on the same physical resources. DRESC is a compiler framework designed by the Interuniversity Microelectronics Center (IMEC) in Belgium for their architecture template ADRES [64].

The Morphosys architecture [81] consists of a general purpose processor core and an array of ALU-based reconfigurable processing elements. In their approach they use a language SA-C, an expression-oriented single assignment language. The paper [54] addresses the problem of compiling a program written in a high-level language, SA-C, to a coarse-grained reconfigurable architecture, Morphosys.

The Dynamically Reconfigurable ALU Array (DRAA) architecture [59][60] has some processing elements placed in a two dimensional array. Each processing element is a word-level reconfigurable function block. The processing elements communicate with each other by the interconnections. The spe-

cial requirement of DRAA to its compiler is that all element rows (columns) should have the same interconnection scheme. Therefore, the compiler of DRAA should exploit the regularity of an application algorithm.

Some research efforts [63], [74] have been focused on generic issues and problems in compilation like optimal code partitioning, and optimal scheduling of computation kernels for maximum throughput. While [74] proposes dynamic programming to generate an optimal kernel schedule, [63] proposes an exploration algorithm to produce the optimal linear schedule of kernels.

3.2.3 Summary

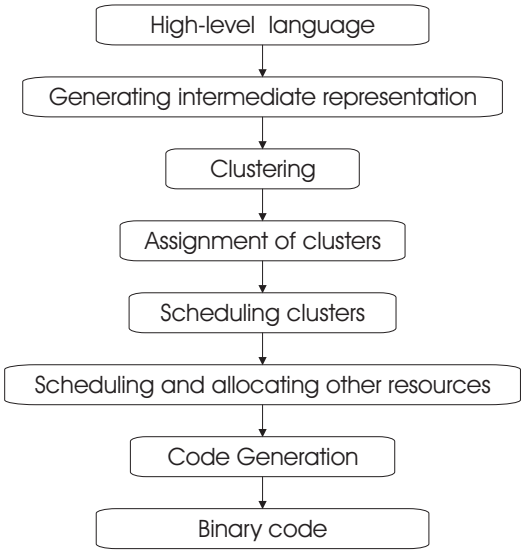

Figure 3.1: General structure of compilers for a coarse-grained reconfigurable architecture

Compilers are usually highly dependent on the structure of the target architecture. As we discussed in Chapter 2, there are some similarities among all these architectures. Therefore, their compilers have some commonness as well. The general structure of a compiler can be presented by Figure 3.1. At the "Generating intermediate representation" phase, the initial application program, which is written in a high level language, will be translated into an intermediate representation. The intermediate representation is usually

convenient for parallelism extraction because most coarse-grained reconfigurable architectures have many parallel function units. When the function unit (or ALU) can execute more than one primary function, the "Clustering" phase is needed to combine the primary operations into clusters. The clusters are allocated to a function unit at the "Assignment of clusters" phase. And then the execution order of these clusters is scheduled at the "Scheduling of clusters" phase. After that, the communications are scheduled, and register and memory allocations are done at the "Scheduling and allocating other resources" phase. Finally the binary code is generated at the "Code generation" phase.

Not all compilers have the complete stream as described in Figure 3.1. For example, DIL is designed as the input language of the PipeRench [12] [35] compiler, so the "Generating intermediate representation" phase is not needed there. If the function units cannot run several primary functions in one clock cycle, the "Clustering" phase can also be skipped. For complex ALUs such as the ALU of the Montium tile processor (see Figure 2.8), the "Clustering" phase plays a very important role in reducing execution clock cycles. The"Assignment of clusters" and "Scheduling of clusters" are often combined [68]. The "Scheduling and allocating other resources" phase highly depends on target architectures, and different architectures have different constraints and requirements.

In our compiler we start from a high language such as C. The reason to choose the high-level language C is that the majority of DSP algorithms is specified in C or Matlab. The system designers often start with an executable C or Matlab reference specification, and the final implementation can be verified against this reference specification.

3.3 A sample architecture

In Chapter 2, we saw that there are many coarse-grained architectures designed during the last 15 years. They are different in detail although they have some common features. To test the algorithms of our compiler, we choose the Montium tile processor as the sample target architecture. There are several reasons for that.

- Firstly, the Montium tile processor has similar features compared to most other coarse-grained reconfigurable architectures. The methodology used in a Montium compiler for taking care of those common

constraints can also be used in compilers for most other coarse-grained reconfigurable architectures.

- Secondly, in the Montium tile processor a hierarchy of small programmable decoders is used to reduce the control overhead significantly. This control system reduces the energy consumption considerably. We believe that this technique will be used more and more in future designs. However, this technique comes as a new requirement for the compiler, i.e., the number of distinct instructions should be as small as possible (more details can be found in Section 3.4.3). This requirement has never been studied before.
- Thirdly, it is believed that the architecture design and compiler design should be done simultaneously. The Montium tile processor has been designed in our group, therefore we have detailed knowledge of the architecture. We are even in the position to suggest changes in the architecture to simplify the compiler design.

3.4 Challenges of designing a Montium compiler

The Montium has been designed for very high speed and low-power applications, and therefore there are some special requirements to the Montium compiler. For example, the Montium allows a limited number of configurations. This limitation has never appeared in one of the previous architectures. New solutions must be found for the architectural concepts applied in the Montium design.

3.4.1 Optimization goals

Before designing a good Montium mapping tool, we need to answer one question first: what does "good" mean? The answer goes back to the motivation of the Montium design – speed and energy-efficiency. That is to reduce the amount of clock cycles for the execution and to reduce the energy consumption of an application. The Montium architecture offers the following opportunities for these objectives.

- **High speed through parallelism.**

  - One ALU can execute several primitive operations within one clock cycle.
  - Five parallel ALUs can run at the same time.
  - The crossbar and distributed storage places allow for parallel fetching and storing of data.

- **Energy-efficiency through locality of reference.**
  - Using the data in a local register is more energy-efficient than using the data in a non-local register.
  - Using the data in a local memory is more energy-efficient than using the data in a non-local memory.
  - Using the data in a local register is more energy-efficient than using the data in a local memory.

A good compiler should find a mechanism to exploit the above mentioned opportunities given by the Montium architecture to generate a fast and low-power mapping. However, sometimes certain conflicts between the reduction of clock cycles and locality of reference may appear, i.e., the mapping scheme with locality of reference could take longer execution time. In that case, we choose a minimal number of clock cycles as a more important criterion mainly because the performance is a more important concern in most cases, and furthermore because more clock cycles definitely cost more energy.

3.4.2 Constraints

Besides the above optimization objectives, the hardware of the Montium defines some constraints for the mapping problem. The constraints can be divided into two groups. One group refers to the constraints given by the number of resources such as number of memories, number of ALUs, etc, which are presented in Chapter 2. Another group is the limitation of configuration spaces. As a reconfigurable architecture, a Montium does not support as many different instructions as a general purpose processor does. The instructions in an application for a Montium should be as regular as possible. In other words, we like to have the same instruction being repeated for many times instead of having many different instructions being used once. The regularity is embodied by the limitations to the size of configuration spaces which will be explained in more detail in Section 3.4.3.

The first type of constraints has been considered in many other compilers for coarse-grained reconfigurable architectures. The second type of constraints has never been studied before. This is mainly because no other architecture has used the same control technique as the Montium. For example, most mapping techniques for clustered micro-architectures focus on minimizing the impact of inter-cluster communications, e.g., [24], because the need to communicate values between clusters can result in a significant performance loss for clustered micro-architectures. However, the regularity of the initial graph might be destroyed in the mapping procedure. As a result, the constraint of configuration space might not be satisfied. The only compiler that deals with the regularity of the target architecture is the compiler for the Dynamically Reconfigurable ALU Array (DRAA) [59][60]. In DRAA architecture, all rows should have the same communication pattern. This architecture has some similarity with the Montium in the sense that there are some special requirements to the pattern (the concept of pattern is discussed below).

3.4.3 Configuration space

The constraints of the sizes of configuration spaces to the compiler in the Montium are given by the control part of the Montium. We first describe the control part for the Montium ALUs in detail. And then we give the idea of configuration spaces for other parts briefly.

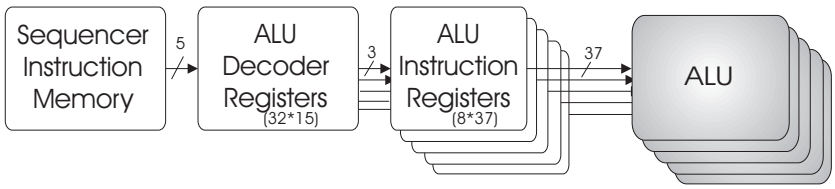

Figure 3.2: Control part for ALUs in the Montium tile

Control part for the Montium ALUs A function or several primitive functions that can be run by one ALU within one clock cycle are called an *one-ALU configuration*. In the Montium, there are 37 control signals for each ALU, so there are 2^{37} possible functions, i.e., 2^{37} one-ALU configurations. In practical algorithms only a few of them are used. The one-ALU

configurations that an ALU needs to execute for an algorithm are stored in configuration registers, named *ALU instruction registers* that are located close to the ALU. Totally, there are 8 such ALU instruction registers for each ALU. The contents of the registers are written at configuration time. At runtime, in every clock cycle, one of these 8 ALU instruction registers is selected to control the function of the ALU. An *ALU decoder register*, which is also a configuration register, determines which ALU instruction register is selected for all five ALUs. A combination of the functions of the five ALUs is called a *pattern*, also called a *five-ALU configuration*. As there are 5 ALUs in the Montium, there are 8^5 combinations of the functions for all the ALUs. In other words, an ALU decoder register could have 8^5 different patterns. However, in practice, not all these 8^5 patterns are used for one application. Therefore, there are only 32 ALU decoder registers in the Montium, which can only store 32 distinct patterns. Finally, sequencer instructions will select an ALU decoder register of a particular pattern in every clock cycle. Note that if there is not enough configuration register space for a specific application, a reconfiguration or partial reconfiguration has to be done. This is very energy-inefficient and time-consuming, and we would like to avoid it. In summary, by using two layers of configuration registers, the control signals for the ALUs in the Montium are reduced from 5×37 to 5 (in the sequencer instruction). On the other hand, the compiler has to face the challenge of decreasing the number of distinct instructions. It is the compiler's responsibility to consider all these constraints and generate proper configurations at compile time.

Control part of the Montium tile processor The controls for other parts in the Montium tile are quite similar to the above described ALU control. Configuration registers can be programmed with microcode instructions that control a part of the data-path. During the execution of a program, in each clock cycle an instruction is selected from a configuration register by the instructions in a decoder (see Figure 3.3). Instruction decoders contain microcode instructions that select instructions from the configurations registers. At each clock cycle an instruction is selected from a decoder by the sequencer. The size of configuration registers determines the number of configurations each configurable entity can have. The size of an instruction decoder determines the number of combinations of the corresponding configurations. The requirements to the compiler can be translated into the following items:

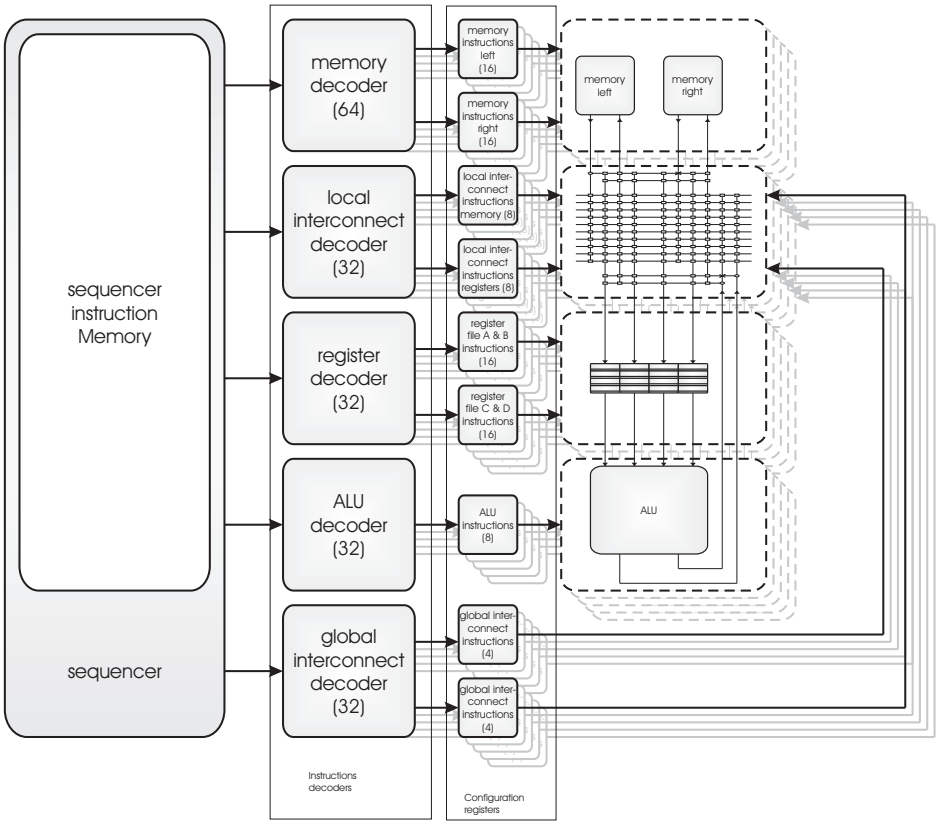

Figure 3.3: Control part of the Montium tile processor

One-ALU configuration(8)
Each ALU can be configured to run 8 different one-ALU configurations within one application. The maximum number of one-ALU configurations that can be executed by the five ALUs is less than 40.

Five-ALU configuration(32)
The number of distinct five-ALU configurations is limited to 32.

The control parts for others in the Montium tile are quite similar to the control part for ALUs shown in Figure 3.2. Details about implementations are beyond the scope of this thesis.

One-AGU configuration (16)
In the Montium, there are are 13 control signals for each AGU. The in-

struction (13 bits) to control each AGU is called a **one-AGU configuration**. Each memory is allowed to have 16 different AGU instructions because the size of register instruction memory is 16.

Ten-AGU configuration(64)
A combination of the ten one-AGU configurations forms a *ten-AGU configuration*. For all the ten memories, there are totally 16^{10} possible different combinations of one-AGU configurations. However at most 64 ten-AGU configurations are allowed in the Montium due to the limitation of the memory decoder, which has 64 entries.

Two-Register configuration (16)
One two-register configuration corresponds to register files Ra and Rb, or Rc and Rd. It decides the reading and writing elements of Ra and Rb or Rc and Rd. For instance, a two-register configuration could be: Ra1 read, Ra2 write, Rb1 write, Rb1 read.

Twenty-Register configuration (32)
There are totally 20 register files, which form 10 two-Register configurations. The combination of these 10 two-Register configurations is a twenty-Register configuration. The number of twenty-Register configurations is limited to 32.

One-global bus configuration (4)
A one-global bus configuration defines one source that writes to the bus in one clock cycle. Each global bus can have at most four different bus configurations.

Ten-global bus configuration (32)
The combination of the ten one-global bus configurations is called a ten-global bus configuration. At most 32 ten-global bus configurations are allowed in the system.

One-local interconnection configuration(8)
An local interconnect instruction of a processing part defines the input buses (global buses or local buses) for the memories and registers of the processing part. Each local interconnect configuration register can store up to 8 instructions for connecting the inputs of the two local memories of a processing part to the local and global buses within the processing part.

Five-local interconnection configuration(32)
For all five processing parts, 32 combinations of one-local interconnection configurations are allowed.

3.4.4 Tasks of a Montium compiler

A Montium compiler has the following tasks:

1. A Montium is designed for executing programs from the digital signal processing domain and ALUs are the units to perform mathematical operations. The central problem of a compiler is to let ALUs execute the operations of a program.

2. A program usually has inputs and will generate some results (also some intermediate results), these values should be put to appropriate storage places.

3. Since an ALU can only use the inputs from its local registers and the outputs of an ALU have to be saved somewhere, communications between storage spaces and ALUs should also be constructed.

3.5 Mapping procedure

This section presents a framework for efficient compilation of applications written in a high-level language (C++) to a reconfigurable computing architecture.

Our compiler consists of all phases in Figure 3.1. The last phase "'Code generation" is not described in this thesis. Fourier transform transforms a signal from the time domain to the frequency domain. For digital signal processing, we are particularly interested in the discrete Fourier transform. The Fast Fourier Transform (FFT) can be used to calculate a discrete Fourier transform efficiently. The FFT is of great importance to a wide variety of applications, e.g., it is the most essential technique used in Orthogonal Frequency Division Multiplex (OFDM) systems. Therefore, we use an FFT algorithm as an example to illustrate our approach.

The objective of this chapter is to give the main idea of our compiler in a simple way. To let the reader get the main point directly without reading complex definitions, the graphs used below are not strictly defined. We

believe that they can be understood easily intuitively. The formal definitions of graphs will be given in the chapters where they are used.

1 **Translating the source code to a control data flow graph**, translation for short. This is the "Generating intermediate representation" phase in Figure 3.1. The input C program is first translated into a Control Data Flow Graph (CDFG) [79]; and then some transformations and simplifications are done on the CDFG. The focus of this phase is the input program (source code) and is largely independent of the target architecture. In general, CDFGs are acyclic. In the first phase we decompose the general CDFG into acyclic blocks and cyclic control information.

   The source C code of a 4-point FFT algorithm is given in Figure 3.4 and Figure 3.5 shows the automatically generated CDFG.

   The translation and transformation phase is the subject of Chapter 4.

2 **Task clustering and ALU data-path mapping**, clustering for short. This is the "Clustering" phase in Figure 3.1.

   The corresponding DFG of the CDFG is partitioned into clusters such that it can be mapped to an unbounded number of fully connected ALUs. The ALU structure is the main concern of this phase and we do not take the inter-cluster communication into consideration. Goals of clustering are:

   - minimization of the number of ALUs required;
   - minimization of the number of distinct ALU configurations; and
   - minimization of the length of the critical path of the graph.

   The clustering phase is implemented by a graph-covering algorithm [33]. The procedure of clustering is the procedure of finding a cover for a CDFG. The clustering scheme for the 4-point FFT algorithm is given in Figure 3.6. After clustering, we get a clustered graph shown in Figure 3.7.

   A cluster on a clustered graph can be presented by an ALU configuration. In Figure 3.6, three types of clusters are found that lead to three distinct configurations:❶, ❷ and ❸, which are shown in Figure 3.8. Configurations ❶ and ❷ have four inputs and generate two outputs.

```
const int N = 4;
float D_r[N]; //real part of input data
float D_i[N]; //imaginary part of input data
float O_r[N]; //real part of output data
float O_i[N]; //imaginary part of output data
float W_r[N/2]; //real part of twist factor
float W_i[N/2]; //imaginary part of twist factor
void main(){
 int xi, xip;
 float u_re, u_im;
 float x_tmp_xi_re, x_tmp_xip_re, x_tmp_xi_im, x_tmp_xip_im;
 for ( int le = N/2; le > 0; le = le/2){
  for ( int j = 0; j < le; j++){
   int step = N/le;
   for ( int i = 0; i < step/2; i++){
    xi = i + j * step; xip = xi + step/2;
    u_re = W_r[le * i]; u_im = W_i[le * i];
    x_tmp_xi_re = D_r[xi]; x_tmp_xip_re = D_r[xip];
    x_tmp_xi_im = D_i[xi]; x_tmp_xip_im = D_i[xip];
    D_r[xi] = x_tmp_xi_re + (u_re * x_tmp_xip_re - u_im * x_tmp_xip_im);
    D_r[xip] = x_tmp_xi_re - (u_re * x_tmp_xip_re - u_im * x_tmp_xip_im);
    D_i[xi] = x_tmp_xi_im + (u_re * x_tmp_xip_im + u_im * x_tmp_xip_re);
    D_i[xip] = x_tmp_xi_im - (u_re * x_tmp_xip_im + u_im * x_tmp_xip_re);
   }
  }
 }
 for ( int j = 0; j < N; j++){
  O_r[j] = D_r[j];   O_i[j] = D_i[j];
 }
}
```

Figure 3.4: The source code of the 4-point FFT program

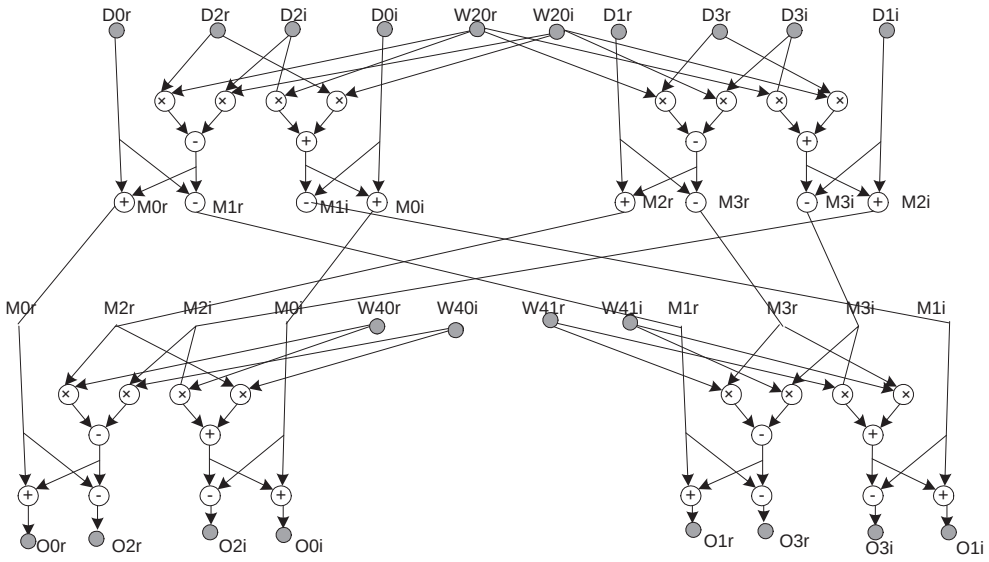

Figure 3.5: The generated graph of a 4-point FFT after complete loop unrolling and full simplification.

One input is from the east input connection (see Chapter 2). Configuration ❸ computes the multiplication of two inputs and sends the result to the left adjacent ALU by the west output.

The clustering phase is the subject of Chapter 5.

3 Scheduling. This consists of the "Assignment of clusters" and the "Scheduling clusters" phases in Figure 3.1. In the scheduling phase, the clustered graph obtained from the clustering phase is scheduled taking the maximum number of ALUs (which is 5 in our case) into account. The scheduling phase fixes the relative execution order of the clusters and fixes the physical ALU allocation for a cluster. The result of the scheduling phase is a time-space table which specifies at which clock cycle and by which ALU a cluster is executed.

To facilitate the scheduling of clusters, all clusters get a *level* number (two types of levels will be defined in Chapter 6). The level numbers are assigned to clusters with the following restrictions:

- For a cluster A that is dependent on a cluster B with level number

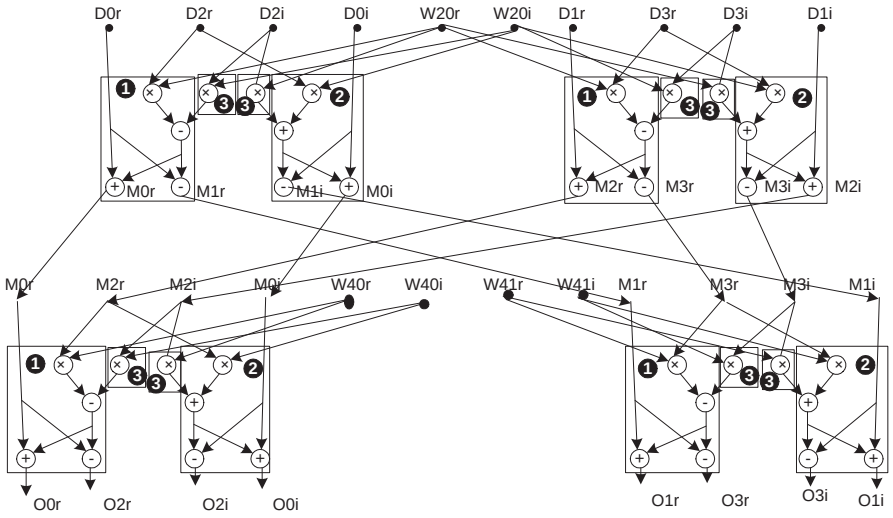

Figure 3.6: The clustering schedule for a 4-point FFT-algorithm

i, cluster A must get a level number $> i$ if the two clusters cannot be connected by the west-east connection (see Figure 2.8).

- Clusters that can be executed in parallel can have equal level numbers.
- Clusters that depend only on in-ports have level number one.

The objective of the clustering phase is to minimize the number of different configurations for separate ALUs, i.e. to minimize the number of different one-ALU configurations. Since our Montium tile is a kind of very long instruction word (VLIW) processor, the number of distinct five-ALU configurations should be minimized as well. At the same time, the maximum amount of parallelism is preferable within the restrictions of the target architecture. In our architecture, at most five clusters can be on the same level. If there are more than 5 clusters at some level, one or more clusters should be moved down one level. Sometimes one or more extra clock cycles have to be inserted. Take Figure 3.7 as an example, where, in level one, the clusters with configuration ❶ and ❷ are dependent on clusters with configuration ❸. Therefore, type 3 clusters should be executed before the corresponding type ❶ or type ❷

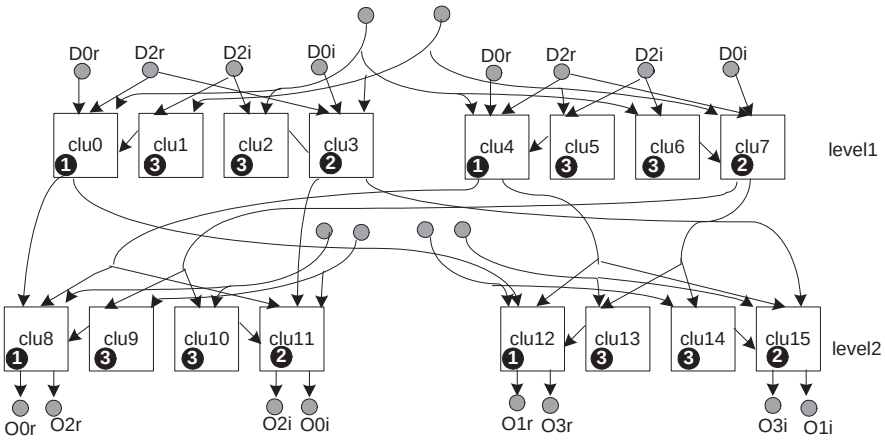

Figure 3.7: The clustering result for the CDFG of Figure 3.6.

cluster, or they are executed by two adjacent ALUs in the same clock cycle, in which case type ❸ clusters must stay east to the connected type ❶ or type ❷ cluster. Because there are too many clusters in level 1 and level 2 of Figure 3.7, we have to split them. Figure 3.9(a) shows a possible scheduling scheme where not all five ALUs are used. This scheme consists of only one five-ALU configuration: $C1$={❶❸❷❸○}. As a result, using the scheme of 3.9(a), the configuration of ALUs stays the same during the execution. The scheduling scheme of Figure 3.9(b) consists of four levels as well, but it is not preferable because it needs two distinct five-ALU configurations: $C2$={❶❸❷❸❸} and $C3$={❶❷❸○○}. Switching configurations adds to the energy overhead of control.

At the scheduling phase, one cluster clu1 is given two attributes: Clock(clu1) and ALU(clu1). The former one represents the clock cycle in which clu1 will be executed and the latter one is the ALU that will run clu1.

The scheduling phase is the subject of Chapter 6.

4 **Resource allocation**: allocation for short. This is the "Scheduling and allocating other resources" phase in Figure 3.1. The resource allocation phase is to allocate variables and arrays to storage places and to schedule data moves. The locality of reference should be exploited whenever

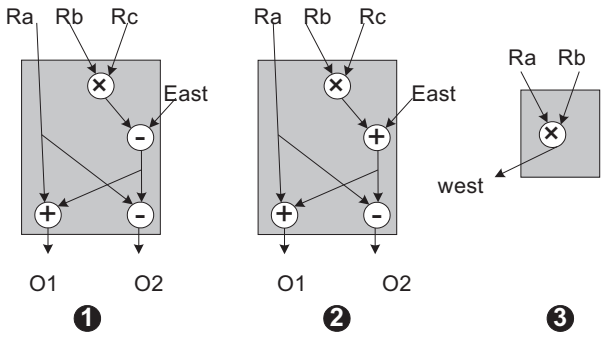

Figure 3.8: ALU Configurations for the clusters used in 4-point FFT

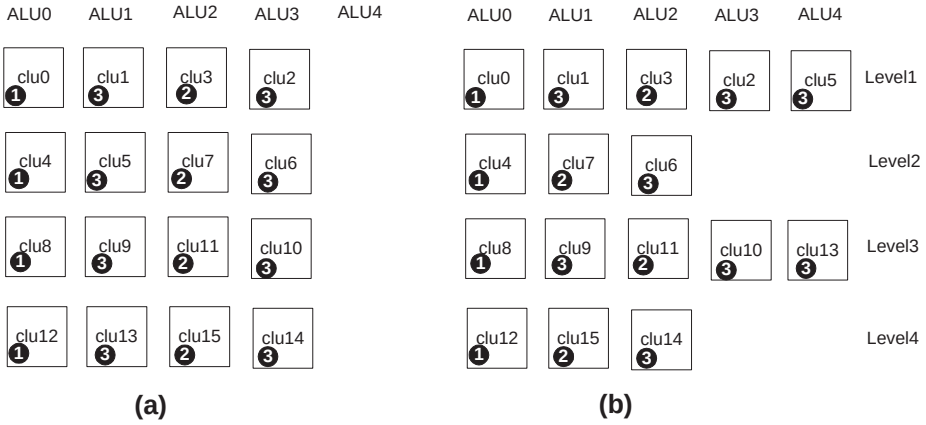

Figure 3.9: Scheduling the ALUs of Figure 3.7

possible, which is important for performance and energy reasons. The main challenge in this phase is the limitation of the configuration space of register banks and memories, the number of buses of the crossbar and the number of reading and writing ports of memories and register banks. CDFGs are used in this section. The output of the resource allocation phase will be assembler code for the Montium.

The allocation phase is the subject of Chapter 7.

3.6 Conclusion

Due to the similarity in the target hardware architectures, there is some commonness in the compilers for coarse-grained reconfigurable architectures. Some novel technique used in the Montium tile processor, i.e., a reconfigurable control part that limits the control space, puts forward some new requirements to the compiler. This new technique is very energy-efficient, and we believe that it will be used more often in future system designs. Therefore the compilation techniques that can handle the new mapping requirements are worthwhile to be investigated. Therefore, we take the Montium tile processor as the sample architecture for our compiler. Code efficiency and power consumption are the major concerns of the Montium compiler. The configuration space is the most challenging constraint. The mapping task is tackled by a heuristic approach, which consists of four phases. At each phase only part of the big task is considered. We use an example in this chapter to give readers a basic idea of our approach.

Chapter 4

Translation and transformation

This chapter presents the translation and transformation phase. In this phase an input C program is first translated into a control data flow graph. A control data flow graph is used as the intermediate representation of an application program. Some transformations and simplifications are performed on the control data flow graph to extract information needed for the mapping phase.

The Montium compiler front-end is based on a graph transformational design method described in [53] [65] [66]. Control Data Flow Graphs (CDFGs) are the design representations used in the transformational design flow, where a design is described in terms of hyper-graphs. The semantics of the design representation are fully based on a set theoretic calculus and applies the concept of the mathematical representation of tables. The work presented in this thesis and in [83] is a continuation of the work presented in [53]. The transformational design approach described in [53] is not the main focus of this thesis. This part is adopted here for completeness and to show

a possible path from the C program to the low level executable Montium assembler code.

CDFGs are the intermediate representation of applications. In this thesis a CDFG is modeled as a hydra-graph defined below. The hydra-graph model is built on the hyper-graph model in [53]: hydra-edges in the hydra-graph model are nodes in the hyper-graph model, hydra-nodes are edges in the hyper-graph model and super nodes are hyper-edges in the hyper-graph model. We changed the names used in the hyper-graph model in this thesis because most people working on compilers are used to using the word "node" (or "vertex") representing a function and using the word "edge" indicating dependence between nodes.

A Control Data Flow Graph, just as its name suggests, includes both control and data flow information. The clustering phase and scheduling phase described in Chapter 5 and Chapter 6 only deal with the data-path information. For the clarity of describing those two parts, instead of using complete CDFGs, Data Flow Graphs (DFGs) are used there, which are actually subgraphs of CDFGs, in which the control part is removed. We will describe DFGs in Section 4.3.

4.1 Definition of Control Data Flow Graphs

In this section, we give the definition of an original CDFG, the CDFG obtained after the translation step. In Section 4.2 some transformation will be done on the original CDFGs.

A CDFG is a hydra-graph. A hydra-graph $G = (N_G, A_G)$ consists of a finite non-empty set of *hydra-nodes* N_G and a set A_G of so-called *hydra-edges*.

Hydra-node

A hydra-node $n \in N_G$ can represent an input/output state space place holder, a label, a constant, an operation, or a hierarchical basic block that is described later, etc. A hydra-node n has a labeled input port set $IP(n)$ and a labeled output port set $OP(n)$. For example, in Figure 4.1, "n1" "n2" and "n3" are three hydra-nodes. The use of labeled input and output ports is the main difference between hydra-nodes and nodes defined in traditional graph theory books. In the rest of the thesis, nodes always refer to hydra-nodes. For node "n1" of Figure 4.1, IP(n1) = {In1, In2, In3} and OP(n1) = {Out1,

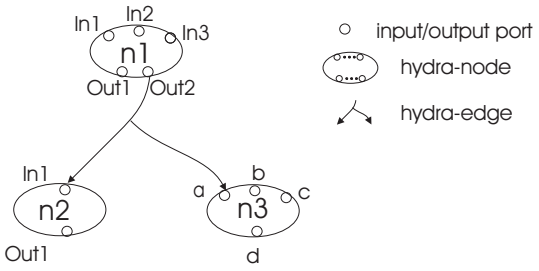

Figure 4.1: Hydra-node and hydra-edge

Out2}. For node "n3", IP(n3) = {a, b, c} and OP(n3) = {d}.

To distinguish input or output ports of different nodes, the input or output port p of node n can be written as $n.p$. In Figure 4.1, there are two ports named "In1". To distinguish them, one can be written as "n1.In1", the other as "n2.In1".

For a port p (input or outport) of node n, $p \in IP(n)$ or $p \in OP(n)$, we use $Node(p)$ to denote the node to which p belongs. Therefore, $Node(n.p) = n$. In Figure 4.1, $Node$(a) = n3. For a set of input ports $IP = \{p_1, p_2, \cdots, p_n\}$, we use $NODE(IP)$ to denote the node set

$$NODE(IP) = \bigcup_{p \in IP} \{Node(p)\}.$$

Similarly, for a set of output ports $OP = \{p_1, p_2, \cdots, p_n\}$,

$$NODE(OP) = \bigcup_{p \in OP} \{Node(p)\}.$$

For example, for input port set IP = {n1.In1, n2.In1, n3.a}, $NODE(IP)$ = {n1, n2, n3}.

When there is no confusion, for simplicity, we do not explicitly indicate input ports and output ports for each node in figures in this thesis.

Hydra-edge

Hydra-edges are named after Hydra, a water-snake from Greek mythology with many heads, just like hydra-edges.

A hydra-edge $e = (t_e, H_e)$ has one *tail port*

$$t_e \in \bigcup_{n \in N_G} OP(n)$$

and a non-empty set of *head ports*

$$H_e \subset \bigcup_{n \in N_G} IP(n).$$

A hydra-edge is directed from its tail to its heads. Because an operand might be the input for more than one operation, a hydra-edge is allowed to have multiple heads although it always has only one tail. Edge "e" in Figure 4.1 can be written as edge (n1.Out2,{n2.In1, n3.a}), where t_e = n1.Out2 and H_e = {n2.In1, n3.a}.

An edge e is called an *input edge* of a node n when one input port of the node is a head port of the edge, i.e., there exists a port $p \in H_e$ and $p \in IP(n)$. An edge n is called an *output edge* of a node n when one output port of the node is the tail port of the edge, i.e., $t_e \in OP(n)$. In Figure 4.1, edge "e" is an output edge of "n1" and an input edge of "n2" or "n3".

Hydra-edges indicate the dependence between nodes. The tail node is called the *predecessor* of the head nodes and the head nodes are called the *successors* of the tail node. In the above mentioned example, nodes "n2" and "n3" are successors of node "n1"; correspondingly "n1" is the predecessor of "n2" and "n3". If there exists a sequence $u_0, \ldots, u_k$ of nodes from N_G such that $u_0 = u$, $u_k = v$, and u_{i+1} is a successor of u_i for all $i \in \{0, \ldots, k-1\}$, v is called a *follower* of u.

There are two types of hydra-edges: *thin edges* and *thick edges*. Thin edges drawn as thin lines on a CDFG represent data and thick edges drawn as thick lines are used to represent state space, which we will explain in more detail below.

The difference between hydra-edges and the edges defined in traditional graph theory books are that traditionally an edge has only one head. In the rest of the thesis, for simplicity, "edge"s are used to refer to hydra-edges.

State space

In a CDFG, the mathematical abstraction of a memory model, called the state space, is presented in [53]. The state space is a set of tuples: $\{(ad_1,da_1)$, $(ad_2,da_2), \cdots\}$. A tuple consists of an *ad* field, which represents the address, and a *da* field, which represents the data at that address. Therefore, the state space is a function mapping address on data values. Interaction with the state space is, in principal, done with two primitive operations called *store* (st) and *fetch* (fe). State space operations implicitly carry the information

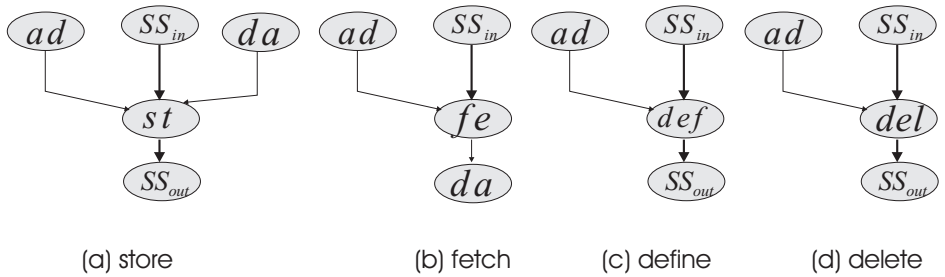

Figure 4.2: Primitive operations on the state space

for allocation and scheduling. The *st* is used to add a tuple to the state space or to overwrite a tuple if a tuple with the same address already exists (see Figure 4.2(a)). It has three inputs: the incoming state space (SS_{in}), an address (*ad*) and the data (*da*) to be written on address (*ad*). It has a single output. If the state space SS_{in} already contains a tuple with the address *ad*, i.e., there exists a *d* such that $(ad, d) \in SS_{in}$, then $SS_{out} = (SS_{in} - \{(ad, d)\})$ $\cup \{(ad,da)\}$. Otherwise, the modified state space $SS_{out} = SS_{in} \cup \{(ad,da)\}$. The fe operation is used to read data from an address (see Figure 4.2(b)). It has two inputs: the incoming state space (SS_{in}) and the address from which to read (*ad*). The output is the data *da* at address *ad*. In this simple model we assume that all tuples (*ad*,*da*) have the same size in bits. This model can be extended with variables with different bit-width. An fe operation does not modify the state space. Two extensions to the before mentioned operations have been defined to be able to support scope levels/local variables more easily, a delete operation (*del*) and a define operation (def) (see Figure 4.2(c) and (d)). The def operation does not actually do anything but it signals when the lifetime of a specific tuple begins. The *del* operation signals when a specific tuple is no longer valid. The def and *del* operations are very helpful when performing a dependency analysis or other simplifications. A graph always starts from SS_{in}, which is the input state space and ends with SS_{out}, which represents the output state space. Besides these operations, basic blocks and jumps have influence on state spaces, which will be discussed below. The state space part without basic blocks and jumps always has the structure as shown in Figure 4.3(a). Operations *st*, def and *del* may only appear on the main stream from input state space SS_{in} to the output state space SS_{out}, while fe does not. Several fe's may connect to the same hydra-edge. However, for a non-jump and non-loop structure, only one *st*, def or

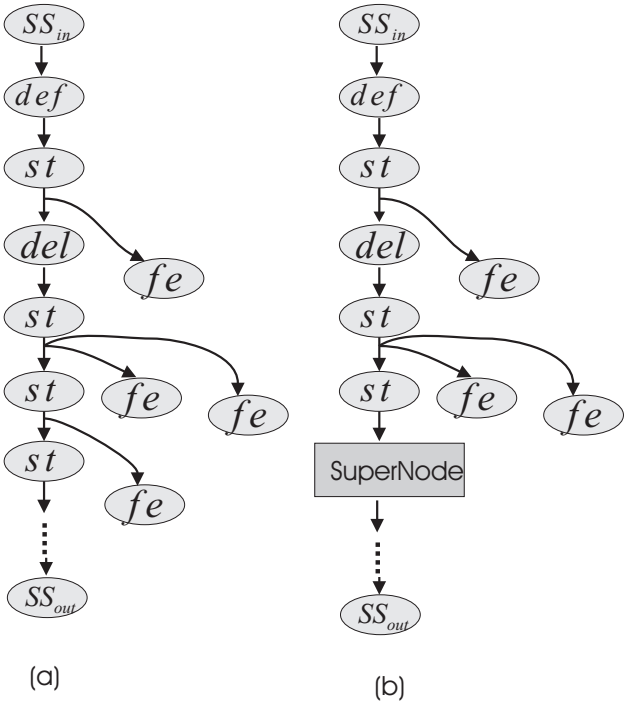

Figure 4.3: Operations on state spaces

del is allowed to use the same hydra-edge as their inputs. The advantage of the state space structure presented in Figure 4.3(a) is that with this clear structure, the state space of variables can be separated into smaller state spaces, which is presented in Section 4.2.

Hierarchical structure

The above described state space stream is extended by allowing a *super node* on the main stream (see Figure 4.3(b)). A super node represents a basic block, which is a subgraph that has only one input SS_{in} and only one output SS_{out}. This subgraph is called a basic block. Within this subgraph, other super nodes could exist. A hierarchical structure is formed by using super nodes. A super node could be considered as a black box, in which something could have been done on the state space. The hierarchical structure makes the graph more readable. Furthermore, the mapping task is simplified dramatically by tackling basic block by basic block. The difference of a basic

```
float C1 = C2 = C3 = C4 = C5 = 1;
float T0 = T1 = T2 = T3 = T4 = 0;
int SIZE = 8;
float input[8]; //this is the input array.
float output[8]; //this is the output array.
Fir5(){
  for(int i=0; i<SIZE; i++){
     T4 = input[i]×C4 + T3;
     T3 = input[i]×C3 + T2;
     T2 = input[i]×C2 + T1;
     T1 = input[i]×C1 + T0;
     T0 = input[i]×C0 ;
     output[i] = T4;
  }
}
```

Figure 4.4: The source code of the FIR5 program

block and a subgraph is that a subgraph can be any part of a graph, while a basic block has a clear state space structure, i.e., only one input SS_{in} and only one output SS_{out}. Let us take a 5-tap Finite Impulse Response (FIR5) filter as an example. The source code of a FIR5 filter is given in Figure 4.4. Figure 4.5 is the root graph of a 5-tap FIR filter, which has a clear state space structure. Node "SS_in" is the input state space and node "SS_out" the output state space. The node "main199_stmnt_iter_for:1#41" is a super node which represents the basic block shown in Figure 4.6.

Operator

All unary and binary C operators (like +, −, ×, /, <<, >> can directly be mapped to a node in the CDFG.

Multiplexer

In a CDFG, an "if/then/else" statement is modeled by a multiplexer (MUX). Figure 4.7 shows an example: (a) shows the C code and (b) shows the corresponding CDFG. "Part1" and "Part2" represent two basic blocks. When the condition holds, the output state space of "Part1" is selected; otherwise, the output state space of "Part2" is selected. The "if/then/else" part does not have the state space structure shown in Figure 4.3, however, in each separate part, "Part1" or "Part2", the state space structure exists.

Iterations are modeled by recursions. A while statement, for example, calls itself recursively to go to the next iteration instead of actually 'jumping' back to the starting point of the loop body (see Figure 4.8 for the template

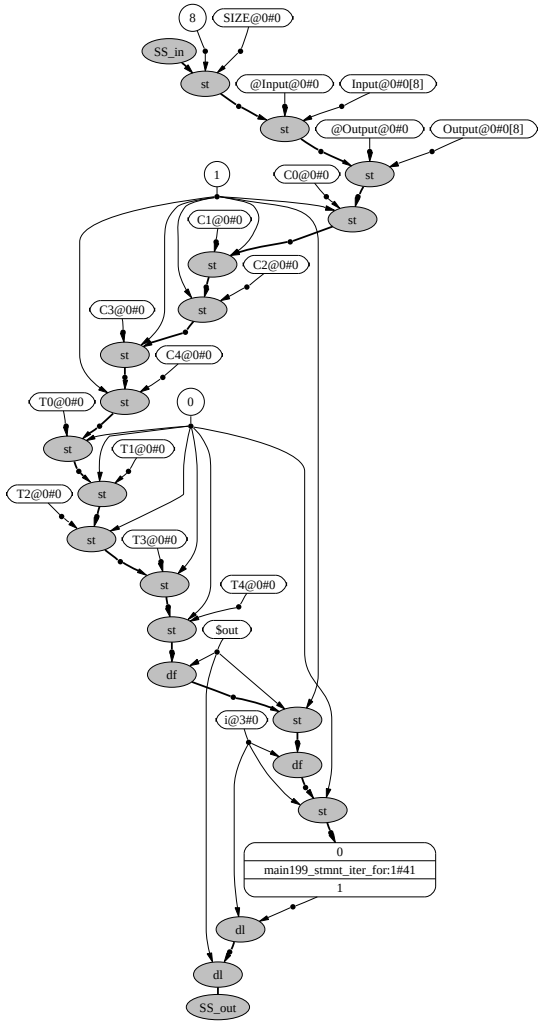

Figure 4.5: The root graph of a 5-tap FIR filter, where the loop body is a super node. Here "$out" represents the return value of a function which is not used here.

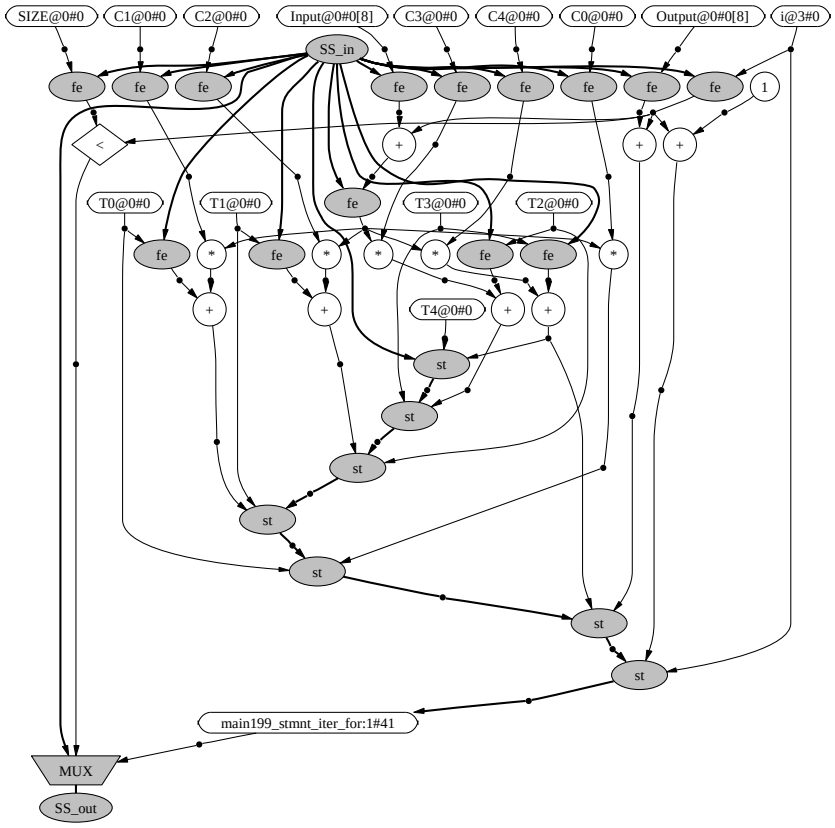

Figure 4.6: The loop body of a 5-tap FIR filter

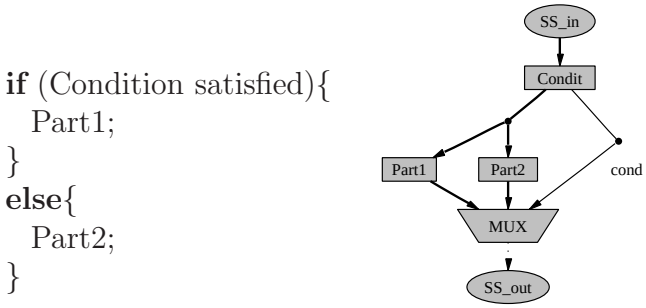

Figure 4.7: An example of an if/then/else statement

of a while state). In Figure 4.8 "$0" denotes the subgraph for evaluating the while condition, "$1" denotes the while body subgraph and "$R" denotes the recursive call of the while statement. Depending on the result of "$0" the MUX takes another iteration or exits the while loop. Other jump statements such as "goto", "break", "continue" and "return" can also be modeled by recursion [46].

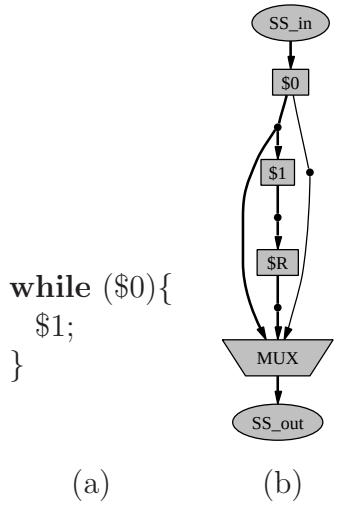

Figure 4.8: While statement

4.2 Information extraction and transformation

By graph transformation we mean that if a certain structure exists in a given graph then it may be transformed into a new one, which can be either semantics preserving or non-semantics preserving [83]. Figure 4.9 is a small example of a transformation. The edges $e1$ and $e2$ in Figure 4.9(a) have the same value. Therefore they can be combined as shown in Figure 4.9(b).

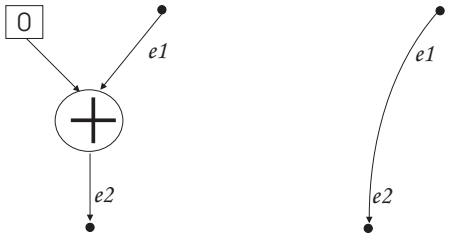

Figure 4.9: A transformation

Transformations are applied because some information needed for the mapping procedure is hidden in the CDFG that is directly generated from a high level programming language. After transformations, this kind of information is more clear. Furthermore, the result of each transformation should be sufficiently close to the previous expression. Therefore, the effort of verifying the transformation is not excessive. More work on a tool chain to support a transformational design methodology can be found in [83].

To extract this type of mapping information from the original CDFG, several important transformations are described below.

4.2.1 Finding arrays

In the initial generated CDFG, an array cannot be easily distinguished from a scalar variable because the array accesses are expanded into arithmetic address calculations. Therefore extracting data dependence information is difficult in the original generated CDFG. This is also inconvenient for generating the Address Generation Unit (AGU) instructions (to be described later). An array is a structure that holds multiple values of the same type. To identify an array, the following structure should be recognized: There is always an *st* operation $st(label, @label)$ that stores an address $@label$ to

the *label*, and the name of the address is always the name of a label with a prefix @, we use this characteristic to find arrays. For example, in Figure 4.5 "Input@0#0" and "Output@0#0" can be identified to be arrays by this method.

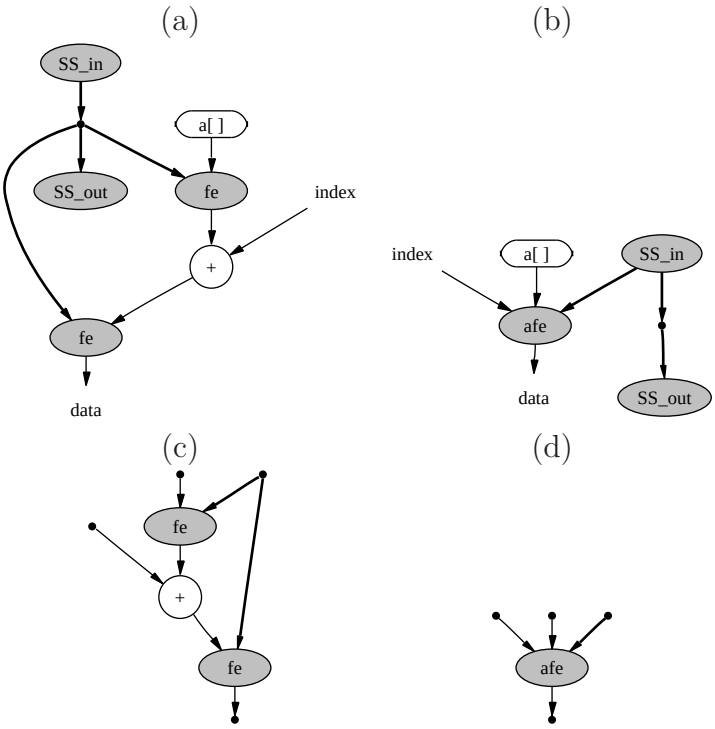

Figure 4.10: Using array fetch

Using array fetch and array store: In the original CDFG, to obtain an element of an array we need two fe operations (see Figure 4.10(a)). The first fe reads the initial address of the array, i.e., @*label*; the second fe operation reads the content of the desired element using address (@*label* + index). Storing a value to an array element also takes two operations. First an fe operation is used to get the initial address of the array; and then an *st* operation is used to write the content to the desired element at address (@*label* + index). After identifying arrays, we define a new operation for fetching an element of an array, *array fetch* (afe). An afe has three inputs: the incoming state space (SS_{in}), the label of the array (*label*) and the index of the element *index* (see Figure 4.10 (b)). The node afe is equivalent to (can replace) the original 3 operations (see Figure 4.10 (c) and (d)). This is

an example of behavior preserving transformation. Similarly, an *array store* (*ast*) operation is introduced to store a value to an element of an array.

4.2.2 New value representation

Some variables can be computed or are known at compile time. Knowing these value is very useful for instruction generation at compile time. In most cases, those variables are integers, such as the number of iterations of a loop body, or the index of an element of an array. For example, for the FIR5 filter, the iteration times of the loop body is controlled by the variable "SIZE" (see Figure 4.6), which is set to be 8 in Figure 4.5. When the iteration time is known at compile time, the loop counter of the Montium sequencer can be used for controlling the iterations. Therefore, the loop counter of the Montium tile can be set to 8.

Motivation – array addressing

In a CDFG, one single value is not always enough to represent the value of a variable. For example, we cannot find a single integer value to represent the index of the array element "Output[i]" in the loop body of the FIR5 filter program (see Figure 4.4) although we do know that the index of "Output[i]" equals 0 at the first iteration, 1 at the second iteration, $\cdots$, 7 at the 8th iteration. This kind of knowledge is needed to generate AGU instructions. Therefore, we introduce a new type of value representation.

New value representation

We use a new type of data structure to represent the value of above mentioned variables $v =< init, step, upper, iter, loop >$. Here, *init* is the first value of a variable or intermediate result; *step* is the difference between the next value and the current one; *upper* is the upper bound of the value (usually it refers to the largest index of an array); *iter* specifies the amount of values; *loop* refers to a loop body. Usually relative values are used in loop bodies, therefore *loop* is used here to indicate whether the value will change and for which loop. We will use the index of the array "Output[ ]" in the FIR5 filter as an example. The index "i" has the values 0, 1, 2, $\cdots$, 7. At each iteration of the loop body, the index takes the next number. $v =< 0, 1, 8, 8, loop1 >$.

Memory access pattern

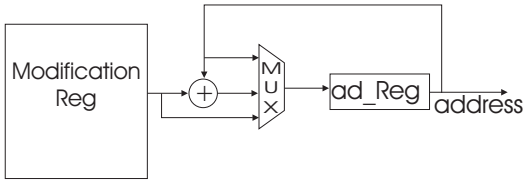

Figure 4.11: Address generation unit

The new variable representation is designed based on commonly used memory access patterns. In the Montium, the address of a memory is generated by an AGU unit. An AGU has one address register, and several modification registers (see Figure 4.11). The address register holds the address that was issued in the previous clock cycle. Addresses are typically modified by selecting an modification address (ad_reg ← modification(i)) or by adding this modification to the current address (ad_reg← modification(i) + ad_reg).

Correspondingly, there are two commonly used ways to access an element of a memory. One method is to use the absolute address of the element; the other one is to use a relative address. For example, to access the 5th element of a memory, the first method is to use one modification register to keep 5 (modification(i) ← 5), then set the address register to the content of the modification register (ad_reg← modification(i)). If the address at the previous clock cycle is 3 (ad_reg is 3), the second method is to use one modification register to keep 2 (modification(i) ← 2), and to increment the previous address by the content of the modification register (ad_reg← modification(i) + ad_reg).

Using *init* and *step* in the above described value representation, the AGU information can easily be extracted: The *init* is used to set the address register before the loop body and at the end of each loop body, the address is incremented by *step*. The Montium memory can be used as a cyclic buffer. Item *upper* is used to set the size of a cyclic buffer. For example:

```
        int a[8], b[8];
loop1:  for (int i = 0; i < 8; i++){b[i mod 8] = a[2i mod 8];}
```

Assume array "a[ ]" and array "b[ ]" are allocated to different memories. The index of a[ ] has the value <0, 2, 8, 8, loop1> which represents the list {0,

2, 4, 6, 0, 2, 4, 6}. The index of b[ ] has the value <0, 1, 8, 8, loop1> which represents the list {0, 1, 2, 3, 4, 5, 6, 7}. In this way we can build a cyclic buffer of size 8 for array "a[ ]". The address register is set to a[0] and b[0] before the loop body, and during each iteration, the address of array "a[ ]" is incremented by 2 and the address of array "b[ ]" is incremented by 1.

4.2.3 Separating state space

The state space in the original CDFG generated from a high level language represents the whole data space of all variables. The input and output of a basic block are always a state space. From such a CDFG, the data flow of a variable cannot always be clearly seen at higher place in the hierarchy. As a result it is not trival to analyze the lifetime of a variable. For example, in Figure 4.5, we do not have a clue which variables have been fetched and which variables have been changed by the super node "main199_stmnt_iter_for:1#41". This kind of information is presented in the corresponding basic block of the super node. For a large application which has many basic blocks, it is preferable to do the mapping work block by block. To avoid the inconvenience of checking information from other basic blocks when one specific basic block is mapped, it is very useful that the interface of a super node shows all the information needed for mapping other basic blocks. The information consists of the variables that are fetched and the variables that are stored in a basic block.

To get this information easily, we separate the state space into several small state spaces. Each small state space represents the mathematical abstraction of a memory model for one specific variable or array. Figure 4.13 and Figure 4.14 are the transformation results of Figure 4.5 and Figure 4.6. The input state space of a variable is labeled by the name of the variable with a prefix "SS_in_", and the output state space by a prefix "SS_out_". Input state spaces and output spaces are drawn as grey ellipses. For example, "SS_in_T2@0#0" and "SS_out_T2@0#0"are the input and output state space for variable "T2@0#0".

Two new type of nodes for state space operations are introduced here: *virtual fetch* (vfe) and *virtual store* (vst) (see Figure 4.12). They are not real operations such as a fetch or a store; they are used to indicate what kind of state space operations have been done inside a basic block. The vfe operation is always followed by a super node, which indicates that the content in the address *ad* is fetched within the subgraph corresponding to the

super node. The *vst* indicates that the content in the address *ad* is changed by an *st* operation within the subgraph corresponding to the super node.

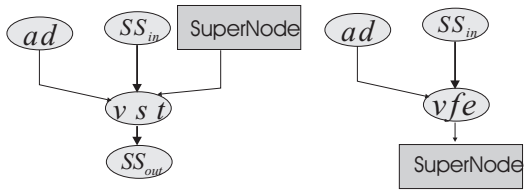

Figure 4.12: Virtual fetch and virtual store

With the use of vfe and vst, the data flow of a variable becomes very clear at a higher level in the hierarchy. For example, only from Figure 4.13 we can conclude that variable "T2@0#0" is first stored as a constant "0"; the stored value is used in the loop body "main199_stmnt_iter_for:1#41" because of the presence of the vfe; the value is also changed within the loop body because of the presence of the vst. Therefore, "0" must be stored to "T2@0#0" before the execution of the loop. For another example of the variable "T4@0#0", which has no vfe operation, we can conclude that the *st* operation can be removed because the value after *st* is not used before another *st* that is inside the loop body.

4.2.4 Finding loops:

There are four loop counters in the Montium. If a loop is known to be a manifest loop (the iteration time of a loop is fixed and known at compile time), a loop counter could be used. Otherwise, we have to use conditional jumps to implement loops, which is slower and more energy inefficient than using a loop counter. In a CDFG, loops are modeled as subgraphs with a multiplexer and a recursion. They could be found by identifying the structure shown in Figure 4.8. The number of iterations of a loop should be computed at compile time whenever possible. In Figure 4.14, the transformation result of Figure 4.6, the "MUX" structure is removed because we already know that this is a loop and the iteration time is 8.

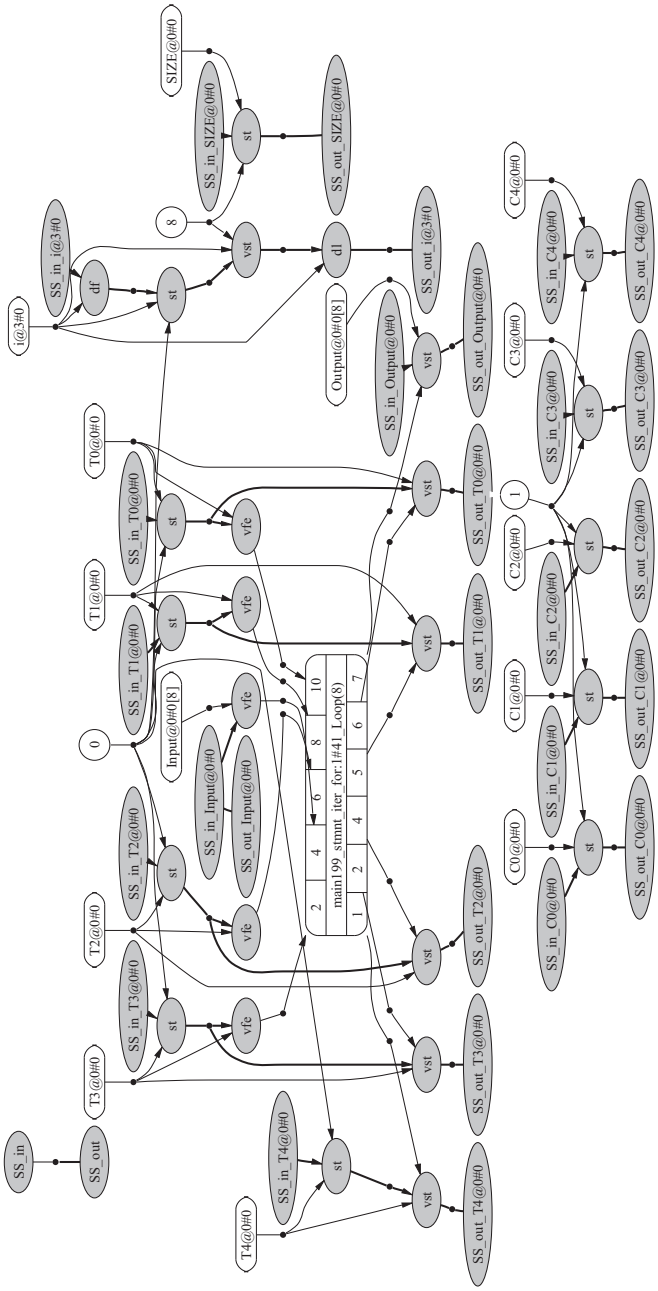

Figure 4.13: After transformation, the new root graph of the 5-tap FIR filter

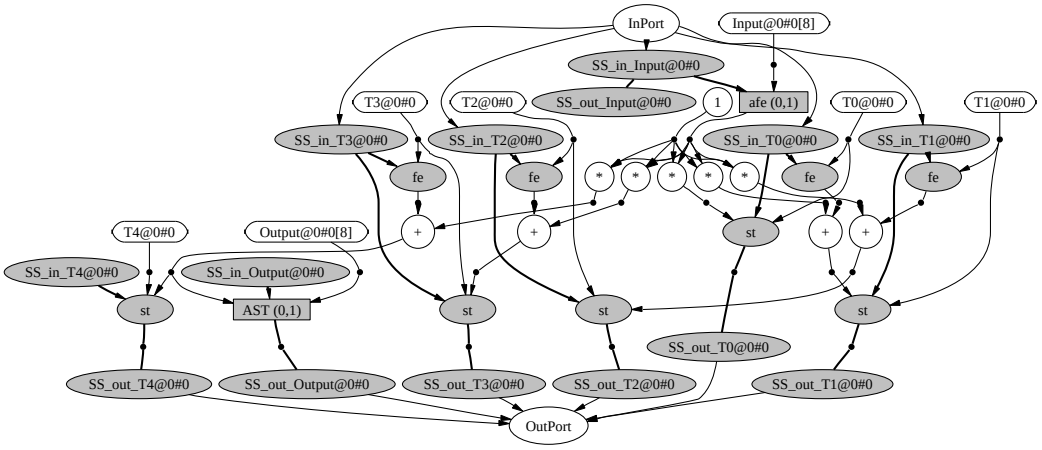

Figure 4.14: After transformation, the new loop body of the 5-tap FIR filter

4.3 Data flow graph

The clustering and scheduling phases only handle the nodes representing operations on the data-path of an application. The nodes related to state spaces in a CDFG are not used there. Therefore, instead of using a complete CDFG, to describe clustering and scheduling algorithms we use a subgraph of the corresponding CDFG, the *Data Flow Graph* (DFG). The DFG is defined here. The procedure to obtain the corresponding DFG from a CDFG is presented as well.

Please note again that edges refer to hydra-edges and nodes refer to hydra-nodes.

In a DFG G, two new types of entities, an *input terminal set* IN_G and an *output terminal set* OT_G, are defined. IT_G represents the inputs of G and OT_G the outputs of G. A DFG communicates with external systems through its terminals.

A DFG $G = (N_G, IT_G, OT_G, A_G)$ consists of a node set N_G, an input terminal set IT_G, an output terminal set OT_G and an edge set A_G. A node $n \in N_G$ can only represent a mathematical computation, such as an addition, a multiplication or a subtraction, but not a state space operation. This is because a DFG represents the data-path part of an application program. An edge $e = (t_e, H_e)$ has one tail port that is either an input terminal or an

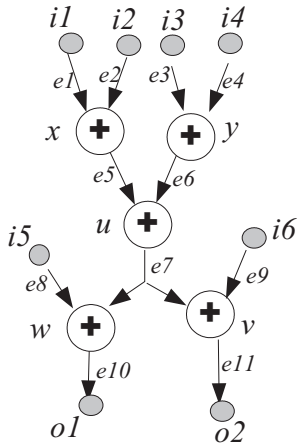

Figure 4.15: DFG example

output port of a node, i.e.,

$$t_e \in (\bigcup_{n \in N_G} OP(n)) \bigcup IT_G$$

and a non-empty set of head ports that could be output terminals and/or input ports of nodes, i.e.,

$$H_e \subset (\bigcup_{n \in N_G} IP(n)) \bigcup OT_G.$$

For the example in Figure 4.15,

$$\begin{aligned} N_G &= \{x, y, u, w, v\}, \\ IT_G &= \{i1, i2, i3, i4, i5, i6\}, \\ OT_G &= \{o1, o2\}, \\ A_G &= \{e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11\}. \end{aligned}$$

If $t_e \in IT_G$, the edge (t_e, H_e) represents an external input of the DFG. In Figure 4.15, $e1, e2, e3, e4, e8$ and $e9$ represent external inputs because tail ports are $i1, i2, i3, i4, i5$ and $i6$ respectively. If

$$t_e \in (\bigcup_{n \in N_G} OP(n)),$$

the edge (t_e, H_e) represents an output of an operation, such as edge $e5, e6, e7, e10$ and $e11$. If H_e contains a terminal of OT_G, the value represented by this edge is an output of the DFG G, such as edges $e10$ and $e11$ in Figure 4.15.

The corresponding DFG of a given CDFG can be obtained by:

1. For each edge $e = (t_e, H_e)$, if $Node(t_e)$ is a fetch or array fetch node, introduce a new input terminal t', replace the edge $e = (t_e, H_e)$ by $e' = \{t', H_e\}$.

2. For each edge $e = (t_e, H_e)$, construct a new port set in the following way: For each head port h_e of e, e.g., $h_e \in H_e$, if $Node(h_e)$ is a store or array store node, introduce a new output terminal t', and add t' to H'_e, i.e., $H'_e = H'_e \cup \{t'\}$; if $Node(h_e)$ is not a store or array store node, add h_e to H'_e, i.e., $H'_e = H'_e \cup \{h_e\}$. Replace the edge $e = (t_e, H_e)$ by $e' = \{t_e, H'_e\}$.

3. After step 1 and 2, the state space operations are separated from the data-path. Delete the state space related nodes and edges and a DFG will be obtained.

Figure 4.16 shows how to obtain the corresponding DFG from a CDFG. In Figure 4.16(b) small grey circles p_{a} and p_{b} are two input terminals. p_{c} is an output terminal. In the example, edge $(n1.OutPort, \{n.InPort1\})$ is replaced by $(p_{\mathrm{a}}, \{n.InPort1\})$, edge $(n2.OutPort, \{n.InPort2\})$ is replaced by $(p_{\mathrm{b}}, \{n.InPort2\})$, and edge $(n.OutPort, \{n3.InPort3\})$ is replaced by $(n.OutPort, \{p_{\mathrm{c}}\})$. From Figure 4.16 we can see that the DFG is simpler than the corresponding CDFG and meanwhile it contains all the necessary information for the clustering phase and scheduling phase.

4.4 Related work

Many systems use the Stanford University Intermediate Format (SUIF) compiler developed by the Stanford Compiler Group [6][41][58][91]. SUIF [85] is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. SUIFs cfg library can divide a program into its basic blocks. Basic blocks are merged into super basic blocks which contain sufficient parallelism to be parallelized profitably.

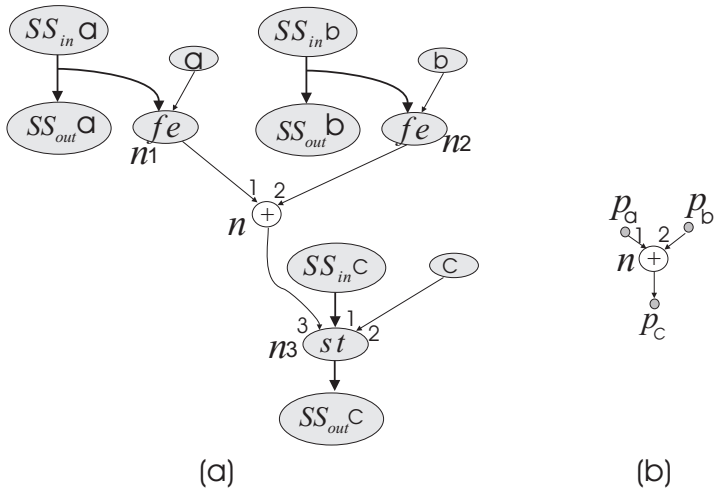

Figure 4.16: (b) is the corresponding DFG of the CDFG in (a)

Another widely used front end is the GNU Compiler Collection (GCC) [31]. The GCC includes front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages.

LLVM is also a collection of source code that implements the language and compilation strategy [56] [61]. The primary components of the LLVM infrastructure are a GCC-based C and C++ front-end. The LLVM code representation describes a program using an abstract RISC-like instruction set but with key higher level information for effective analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation.

As the input language of the PipeRench [12][34][35] compiler, DIL is designed. A programmer can use DIL to describe algorithms for the PipeRench reconfigurable computing systems.

We select the CDFG model because of its simplicity. For instance, control information is carried by state space operations; loops and branches are modeled by MUX operations. This simple model allows more flexibility to the mapping procedure. Furthermore, our research group has mature tools to translate a C program to a CDFG, which is very convenient.

4.5 Conclusion and discussion

For the convenience of mapping an application program to a coarse-grained reconfigurable architecture, an input application program written in a high level language is first translated into an intermediate representation, a control data flow graph. Control data flow graphs are modeled by hydra-graphs. The definition of hydra-graphs is given in this chapter. A number of transformations are done on the CDFG to extract information from the CDFG which is needed for the mapping procedure.

The CDFG representation is very convenient for parallelism extraction, dead code removal and other simplifications [53]. Because both loops and branches are modeled by multiplexors and recursions, the CDFG model is very simple and clean. The representations of arrays are flexible such that pointer operations are supported. However, one disadvantage of the CDFG representation is that much control information is obscured in the CDFG. For example, loops and arrays have to be rediscovered. Another disadvantage is, because every variable is fetched from and stored to a state space, we cannot easily identify its producer and consumers, which is very important for the mapping and scheduling of parallel processors. To overcome these disadvantages, some transformations are introduced in CDFGs, such as separating state space. These transformations are helpful for the mapping procedure. On the other hand, these transformations make the model of CDFGs more complicated than the model initially defined in [53]. How to balance them is another topic that will be considered in future work.

Chapter 5

A clustering algorithm

This chapter[1] *presents a clustering algorithm. The algorithm partitions the nodes of a data flow graph into clusters. The clustering was initially developed for the Montium architecture. However, the clustering algorithm can also be used in other architectures such as ASIC designs or FPGA designs. The clustering phase consists of two steps: template generation and template selection. The objective of the template generation step is to extract functionally equivalent structures, i.e. templates, from a data flow graph. By inspecting the graph, the algorithm generates all the possible templates and the corresponding matches. Unlike other similar algorithms in literature, our method does not limit the shape of the templates. The objective of the presented template selection algorithm is to find an efficient cover for an application graph with a minimal number of distinct templates and minimal number of matches.*

---

[1] Parts of this chapter have been published in publications [3] [4].

In the clustering phase, the nodes of a DFG are partitioned into clusters. Each cluster can be executed by one Montium ALU within one clock cycle.

5.1 Definitions

First we give some definitions that will be used in this chapter.

An ALU on a Montium can execute several primary operations, e.g., $x = a \times b + c + d$, $y = a \times b + c - d$, in one clock cycle. A node on a CDFG usually represents a single primary operation, such as an addition, a multiplication or a subtraction. Therefore, the compiler should divide the data-path part of a CDFG into groups such that each group can be executed by one ALU in one clock cycle. One such group is called a *cluster*.

The clustering phase only handles the nodes that represent operations on the data-path. Therefore, in this chapter we use a subgraph of the corresponding CDFG, the DFG (see the definition of DFG in Section 4.3).

Two distinct nodes u and v from N_G are called *neighbors* if there exists an edge (t, H) with $\{u, v\} \subset NODE(H) \cup \{Node(t)\}$. These nodes are called *connected within a DFG* G if there exists a sequence $u_0, \ldots, u_k$ of nodes from N_G such that $u_0 = u$, $u_k = v$, and u_i and u_{i+1} are neighbors for all $i \in \{0, \ldots, k-1\}$. If u and v are connected within G, then the smallest k for which such a sequence exists is called the *distance of* u *and* v *within the DFG* G, denoted by $\mathrm{Dis}(u, v|G)$; the distance is 0 if $u = v$.

We call u and v *connected within a subset* $S \subset N_G$, if there exists a sequence $u_0, \ldots, u_k$ of nodes from S such that $u_0 = u$, $u_k = v$, and u_i and u_{i+1} are neighbors for all $i \in \{0, \ldots, k-1\}$. Correspondingly, the *distance within a subset* S is defined, denoted by $\mathrm{Dis}(u, v|S)$.

A subset S of the nodes of a DFG is called *connected* if all pairs of distinct elements from S are connected within S. A DFG is called *connected* if all pairs of distinct elements from N_G are connected within G, i.e., if N_G is a connected set.

When describing the clustering algorithm, subgraphs are represented only by node sets. Here we present a method to generate a unique DFG from a node set.

Let $S \subset N_G$ be a non-empty connected set of nodes of the DFG G. Then S generates a connected DFG $T(N_T, IT_T, OT_T, A_G)$ in the following natural way:

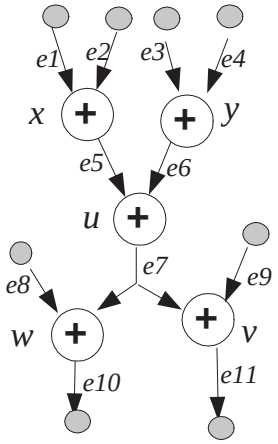

Figure 5.1: A small DFG

- The node set of T is formed by S, i.e., $N_T = S$;
- For each edge $a = (t_e, H_e)$ of G, separate H_e into two small port/terminal sets H' and H'', i.e., $H_e = H' \cup H''$, where H' keeps the ports which belong to a node of S, and H'' keeps the rest, i.e., $H' = \{p | Node(p) \in S\}$.
  - If $Node(t_e)$ is an element of S and H'' is not empty, i.e., $Node(t_e) \in S$ and $H'' \neq \phi$, we introduce a new output terminal t' and add it to OT_T, i.e., $OT_T = OT_T \cup \{t'\}$, and add the edge $(t_v, H' \cup \{t'\})$

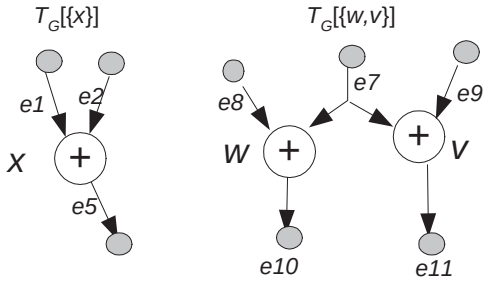

Figure 5.2: Two templates of the DFG of Figure 5.1

to A_T.

- If $Node(t_e) \in S$ and $H'' = \phi$, add (t_v, H_v) to A_T.
- If $Node(t_e) \notin S$ and $H' \neq \phi$, we introduce a new output terminal t' and add it to IT_T, i.e., $IT_T = IT_T \cup \{t'\}$. Add the edge (t', H') to A_T.
- If $Node(t_e) \notin S$ and $H' = \phi$, do nothing.

By doing so we obtain a unique DFG which is referred to as the *template* generated by S in G. We denote it by $T_G[S]$ and say that S is a *match* of the template $T_G[S]$. In the sequel only connected templates are considered without always stating this explicitly.

For example, in Figure 5.2 we see two templates of the DFG from Figure 5.1: the left one is generated by the set $\{x\}$, the right one by $\{v, w\}$. Compared with the original DFG from Figure 5.1, in the left one, the newly added terminal is a head for edge $e5$, while in the right one the newly added terminal is a tail for edge $e7$.

Two DFGs G and F are said to be *isomorphic* if there is a bijection $\phi : N_G \cup IT_G \cup OT_G \rightarrow N_F \cup IT_F \cup OT_F$ such that: $\phi(N_G) = N_F$, $\phi(IT_G) = IT_F$, $\phi(OT_G) = OT_F$, and $(t_e, H_e) \in A_G$ if and only if $(\phi(t_e), \phi(H_e)) \in A_F$. We use $G \cong F$ to denote that G and F are isomorphic.

A DFG T is a *template of the DFG* G if, for some $S \subset N_G$, $T_G[S] \cong T$. Of course, the same template could have different matches in G. As an example, Figure 5.3 shows all the connected templates and matches of up to two nodes generated from Figure 5.1.

Note that, in general, a template is not a subgraph of a DFG, because some ports of nodes may have been replaced by terminals.

The important property of templates of a DFG is that they are themselves DFGs that model part of the algorithm modeled by the whole DFG: the template $T_G[S]$ models the part of the algorithm characterized by the operations represented by the nodes of S, together with the inputs and outputs of that part. Templates are chosen such that they represent one-ALU configurations as we discussed in Chapter 3. A template should match an ALU configuration. Because of this property, templates are the natural objects to consider if one wants to break up a large algorithm represented by a DFG into smaller parts that have to be executed on ALUs.

In order to be able to schedule and execute the algorithm represented by the DFG in as few clock cycles on the ALU-architecture as possible,

it is preferable to break up the DFG in as few parts as possible. On the other hand, due to the limitation on the number of one-ALU configurations, the number of different templates should be as small as possible. In other words, we would like to use a small number of rather large templates, just fitting on an ALU, and a rather small number of matches to partition the nodes of the DFG. This is an NP-hard problem [30] and we cannot expect to solve this complex optimization problem easily. We would be quite happy with a solution that gives approximate solutions of a reasonable quality, and that is flexible enough to allow for several solutions to choose from. For these reasons, we propose to start the search for a good solution by first generating all different matches (up to a certain number of nodes because of the restrictions set by the ALU-architecture) of non-isomorphic templates for the DFG. To avoid unnecessarily computations we will use a clever labeling of the nodes during the generation process; we will describe this in more detail in the next section.

The set of all possible matches of admissible templates will be used to serve as a starting point for an approach to solve the following problem.

Given a non-isomorphic template set $\{T_1, \ldots, T_\ell\}$ of DFG G, if there exists a partition of N_G into mutually k disjoint sets $S_1, \ldots, S_k$ such that $T_G[S_i] \cong T_i, T_i \in \{T_1, \ldots, T_\ell\}$ for all $i \in \{1, \ldots, k\}$, the match set $\{S_1, \ldots, S_k\}$ is called a (k, ℓ)-*cover* of G, and the template set $\{T_1, \ldots, T_\ell\}$ is called a *(k, ℓ)-tiling* of G.

DFG clustering problem Given a DFG G, find an optimal (k, ℓ)-cover $(S_1, S_2, \ldots, S_k)$ and corresponding (k, ℓ)-tiling $\{T_1, \ldots, T_\ell\}$ of G.

This problem can be treated as two subproblems:

Problem A: Template generation problem Given a DFG, generate the complete set of non-isomorphic templates (that satisfy certain properties, e.g., which can be executed on the ALU-architecture in one clock cycle), and find all their corresponding matches.

Problem B: Template selection problem Given a DFG G and a set of (matches of) templates, find an 'optimal' (k, ℓ)-cover of G.

However, the meaning of 'optimal' is not easy to define because of several reasons that have partly been discussed before.

Firstly, the values of k and ℓ will depend on each other: an optimal value for one could imply a bad value for the other.

Secondly, how should one define optimal in the formulation of the DFG clustering problem? The overall aim is to schedule and execute the algorithm represented by the DFG on the ALU-architecture in as few clock cycles as possible. This aim cannot be translated in a simple statement about the values of k and ℓ in an optimal solution.

In the sequel we first focus on generating all matches of non-isomorphic templates, i.e., on part A of the problem. For part B, instead of optimizing a particular pair of k and ℓ, we will describe an approach which solves part B in a flexible way. By simply adjusting an objective function, the method can change the optimization goal.

5.2 Template Generation and Selection Algorithms

For convenience let us call a template (match) an *i-template* (*i-match*) if the number of its nodes is i. The objective of the template generation algorithm is to find all non-isomorphic i-templates with $1 \leq i \leq$*maxsize* for some predefined value *maxsize* depending on the application, and their corresponding matches from G. Given the input DFG of Figure 5.1 and *maxsize*=2, the algorithm should find the templates and matches as shown in Figure 5.3.

A clear approach for the generating procedure is:

- Generate a set of connected i-matches by adding a neighbor node to the $(i-1)$-matches.
- For all i-matches, consider their generated i-templates. Choose the set of non-isomorphic i-templates and list all matches of each of them.
- Starting with the 1-templates, repeat the above steps until all templates and matches up to *maxsize* nodes have been generated.

By a clever use of a predetermined labeling of the nodes, a choice of a so-called *leading node* for each match, and using the distance of the newly added node to this leading node, we are able to save on execution time and memory during the generation procedure. Of course we cannot avoid that it is time

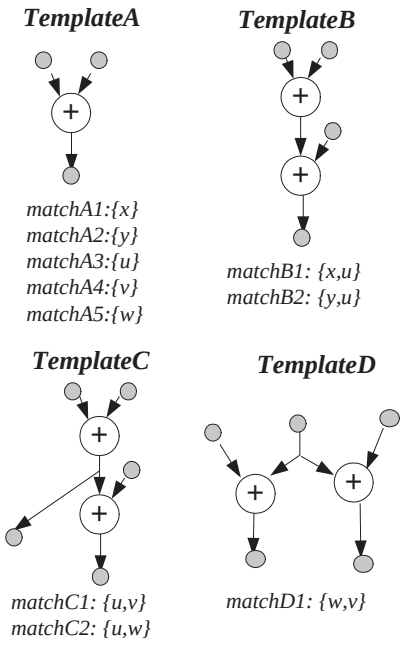

Figure 5.3: Templates and matches for the DFG of Figure 5.1

consuming since G can have exponentially many (matches of) templates. We will give more details on the suggested procedure below.

The union of a connected match and one of its neighbor nodes is a larger connected set, which is a match for some larger template. The newly obtained match is called a *child match* of the old match, and correspondingly, the old one is called a *father match* of the new one. Each match has one *leading node*, which, for the case of a 1-match, is the unique DFG node. An i-match ($i > 1$) inherits the leading node from its father match. In Figure 5.3 the set $\{x,u\}$, which is a match for *templateB*, is obtained when the match *matchA1:*$\{x\}$ absorbs its neighbor u. Thus, the match *matchB1:*$\{x,u\}$ is a child match and *matchA1:*$\{x\}$ is a father match of *matchB1.* The leading node for *matchA1:*$\{x\}$ is x, which is also the leading node for *matchB1:*$\{x,u\}$. On the other hand, *matchB1:*$\{x,u\}$ can be treated as a child match for *matchA3:*$\{u\}$ (see Figure 5.3 and Figure 5.1). In this case, u is the leading node for *matchB1:*$\{x,u\}$. In short, a match might have different father matches and the leading node for a match cannot be determined in a unique

way without knowing all its father matches. In Section 5.2.1 we will introduce a technique to solve this.

Within a match S, each graph node $n \in S$ is given a *circle number*, denoted by Cir($n|S$), which is the distance between the leading node and n within S, i.e., Cir($n|S$)=Dis(S.LeadingNode, $n|S$). Except for the leading node, which has circle number 0, the circle numbers of other nodes might be different in the predecessor and child matches. Suppose the match $S_1 = \{A, B, C, D\}$ is a father match of the match $S_2 = \{A, B, C, D, E\}$ and their corresponding templates are those given in Figure 5.4. If A is the leading

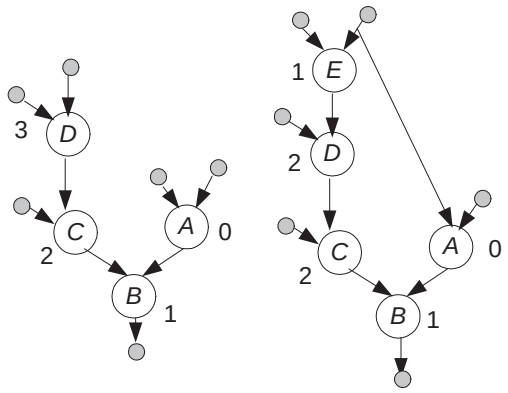

Figure 5.4: A node with different circle numbers in predecessor and child matches.

node of S_1 and S_2, for instance, then Cir($D|S_1$)=3 while Cir($D|S_2$)=2. In Section 5.2.1 an algorithm is presented to skip those child matches where the circle numbers of nodes are different in the father match and the child match. This is done to avoid the generation of multiple identical matches.

5.2.1 The Template Generation Algorithm

The template generation algorithm we propose is shown by the pseudo-code in Figure 5.5. The result of the algorithm (all the different matches of non-isomorphic templates) are stored in *TemplateList* array. The structure of *TemplateList[i]* is:

```
    //Input: graph G.
    //Outputs: TemplateList[1],TemplateList[2],

    ···,TemplateList[maxsize].
    //TemplateList[i] stores i-templates and their matches.
1   Main(){
2     Give each node a unique serial number;
3     FindAllTemplatesWithOneNode(G).
4     for(i=2; i ≤ maxsize; i++){
5       FindAllTemplatesWithMoreNodes(i, G);
6     }
7   }
```

Figure 5.5: Pseudo-code for the template generation algorithm

```
8   FindAllTemplatesWithOneNode(G){
9     TemplateList[1]=φ;
10    foreach node currentNode in graph G{
11      tentativeTemplate = new template (currentNode);
12      isomorphicTemplate
        =TemplateList[1].FindIsomorphicTemplate(tentativeTemplate);
13      if(isomorphicTemplate==φ){
14
15        newTemplate = tentativeTemplate;
16        newMatch = GenerateOneNodeMatch(currentNode);
17        add newMatch to newTemplate;
18        add newTemplate to TemplateList[1];
19      }
20      else{
21        newMatch = GenerateOneNodeMatch(currentNode);
22        add newMatch to isomorphicTemplate;
23      }
24    }
25  }

26  match GenerateOneNodeMatch (newNode){
27    newMatch ={newNode};
28    newMatch.leadingNode = newNode;
29    Cir(newNode|newMatch)=0;
30    return(newMatch);
31  }
```

Figure 5.6: Finding one node template

```
32  FindAllTemplatesWithMoreNodes(nodes number: i, graph: G){
33     TemplateList[i]=φ;
34     foreach (i-1)-match currentMatch
35     and foreach neighbor currentNeighbor∈Neighbor(currentMatch) {
36        if(CanMatchTakeInNode(currentMatch, currentNeighbor)==yes){
37           tentativeTemplate = new template (currentMatch "+" currentNeighbor);
38           if(tentativeTemplate can be mapped to one ALU){
39              isomorphicTemplate
                 =TemplateList[i].FindIsomorphicTemplate(tentativeTemplate);
40              if(isomorphicTemplate== φ){
41                 newTemplate = tentativeTemplate ;
42                 newMatch
                   =GenerateMoreNodesMatch(currentMatch,currentNeighbor);
43                 add newMatch to newTemplate;
44                 add newTemplate to TemplateList[i];
45              }
46              else{
47                 newMatch
                   =GenerateMoreNodesMatch(currentMatch,currentNeighbor);
48                 add newMatch to isomorphicTemplate;
49              }
50           }
51        }
52     }
53  }
54  match GenerateMoreNodesMatch (oldMatch, newNode){
55     newMatch=oldMatch∪{newNode};
56     newMatch.LeadingNode=oldMatch.LeadingNode;
57     foreach n∈oldMatch, let Cir(n|newMatch)=Cir(n|oldMatch);
58     Cir(newNode|newMatch)=1+min(Cir(n|oldMatch),
         where n∈oldMatch and n is a neighbor of newNode.
59     return(newMatch);
60  }
```

Figure 5.7: Finding more nodes template

template1, match11, match12, match13, $\cdots$
template2, match21, match22, match23, $\cdots$
template3, match31, match32, match33, $\cdots$
$\cdots$

First, each node is given a unique serial number (see Figure 5.8). To distinguish between serial numbers and circle numbers within some specific match, the former ones are denoted by the numbers enclosed in small boxes.

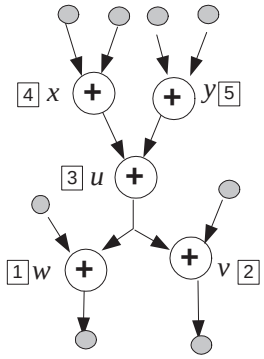

Figure 5.8: Giving each node a unique serial number

1-templates are generated by single nodes of a DFG (line 10 in Figure 5.6). The unions of all possible $(i-1)$-matches and all possible neighbors of those matches cover the whole i-matches space (lines 34, 35 in Figure 5.7). Each i-match generates a corresponding i-template (*tentativeTemplate*) (lines 11 in Figure 5.6, line 37 in Figure 5.7). Templates that cannot be mapped onto one ALU within one clock cycle are filtered out (line 38 in Figure 5.7). All i-templates (*tentativeTemplate*s) are then checked against the templates in *TemplateList*[i] (lines 12 in Figure 5.6, 39 in Figure 5.7). If an isomorphic template (*isomorphicTemplate*) does not exist in the list, a new template (*newTemplate*) is inserted into *TemplateList*[i] (lines 13-19, 40-45); otherwise a match (*newMatch*) for that isomorphic template (*isomorphicTemplate*) is added (lines 20-23, 46-49). For a 1-match, the leading node is the only node (line 28) and its circle number is 0 (line 29). For an i-match ($i > 1$) *newMatch* (line 56), the leading node is inherited from the father match *oldMatch*.

In this algorithm, multiple identical matches are avoided by the function "CanMatchTakeInNode(*oldMatch*, *newNode*)" (line 36), which, making use

of the unique serial numbers and the circle numbers, filters out other father matches except the one (*oldMatch*) that satisfies the following conditions:

1 *oldMatch*.LeadingNode.Serial<*newNode*.Serial;
2 Dis(*oldMatch*.LeadingNode, *newNode*|*oldMatch* ∪ {*newNode*}) is not smaller than Cir(*n*|*oldMatch*) for any $n \in$ *oldMatch*;
3 For every *n* which satisfies *n*∈*oldMatch* and Cir(*n*|*oldMatch*) = Dis(*oldMatch*.LeadingNode,*newNode*|*oldMatch*∪ {*newNode*}), *n*.Serial< *newNode*.Serial.

Condition 1 makes sure that the leading node is the one with the smallest serial number in a match; Condition 2 makes sure the newly added node *newNode* always has the largest possible circle number among all choices. The nodes for a father match, as a result, keep the same circle number in the child match (line 57); Condition 3 makes sure that *newNode* is the one with the largest serial number among the nodes with the same circle numbers. For a match *newMatch*, these conditions decide a unique pair (*oldMatch*, *newNode*) that satisfies (*newMatch*=*oldMatch*∪ {*newNode*}), where *oldMatch* corresponds with a connected $(i-1)$-template, and *newNode* is a neighbor of *oldMatch*.

In Table 5.1, the procedure of finding all the i-matches from $(i-1)$-matches of Figure 5.8 is given. The symbols in bold are the names of the leading nodes and the newly added nodes are underlined. In each row, the match in the left column is the father match of the match in the right column. The matches that do not satisfy the conditions of the function "CanMatch-TakeInNode" are discarded. The numbers next to the discarded matches indicate which of the above three conditions is violated.

Theorem 5.1 *Given a DFG G and an integer maxsize, the template generation algorithm of Figure 5.5 generates all the possible templates and their corresponding matches with at most maxsize nodes. None of the generated templates for G are isomorphic and no matches appear more than once.*

Proof We use induction on the number of nodes of the matches. Clearly, the algorithm checks all 1-matches of G; the non-isomorphic 1-template are listed once, and for each 1-template, all of its 1-matches.

The induction hypothesis is that the algorithm finds all $(k-1)$-matches, grouped according to non-isomorphic $(k-1)$-templates. We want to prove

1-matches	2-matches	3-matches	4-matches
$\{\boldsymbol{x}\}$	$\{\boldsymbol{x},\underline{u}\}$ 1		
$\{\boldsymbol{y}\}$	$\{\boldsymbol{y},\underline{u}\}$ 1		
$\{\boldsymbol{u}\}$	$\{\boldsymbol{u},\underline{w}\}$ 1		
	$\{\boldsymbol{u},\underline{v}\}$ 1		
	$\{\boldsymbol{u},\underline{x}\}$	$\{\boldsymbol{u},x,\underline{y}\}$	$\{\boldsymbol{u},x,y,\underline{w}\}$ 1
			$\{\boldsymbol{u},x,y,\underline{v}\}$ 1
		$\{\boldsymbol{u},x,\underline{w}\}$ 1	
		$\{\boldsymbol{u},x,\underline{v}\}$ 1	
	$\{\boldsymbol{u},\underline{y}\}$	$\{\boldsymbol{u},y,\underline{x}\}$ 3	
		$\{\boldsymbol{u},y,\underline{w}\}$ 1	
		$\{\boldsymbol{u},y,\underline{v}\}$ 1	
$\{\boldsymbol{v}\}$	$\{\boldsymbol{v},\underline{u}\}$	$\{\boldsymbol{v},u,\underline{x}\}$	$\{\boldsymbol{v},u,x,\underline{y}\}$
			$\{\boldsymbol{v},u,x,\underline{w}\}$ 1
		$\{\boldsymbol{v},u,\underline{y}\}$	$\{\boldsymbol{v},u,y,\underline{x}\}$ 3
			$\{\boldsymbol{v},u,y,\underline{w}\}$ 1
		$\{\boldsymbol{v},u,\underline{w}\}$ 1	
	$\{\boldsymbol{v},\underline{w}\}$ 1		
$\{\boldsymbol{w}\}$	$\{\boldsymbol{w},\underline{u}\}$	$\{\boldsymbol{w},u,\underline{x}\}$	$\{\boldsymbol{w},u,x,\underline{y}\}$
			$\{\boldsymbol{w},u,x,\underline{v}\}$ 2
		$\{\boldsymbol{w},u,\underline{y}\}$	$\{\boldsymbol{w},u,y,\underline{x}\}$ 3
			$\{\boldsymbol{w},u,y,\underline{v}\}$ 2
		$\{\boldsymbol{w},u,\underline{v}\}$ 3	
	$\{\boldsymbol{w},\underline{v}\}$	$\{\boldsymbol{w},v,\underline{u}\}$	$\{\boldsymbol{w},v,u,\underline{x}\}$
			$\{\boldsymbol{w},v,u,\underline{y}\}$

Table 5.1: Multiple copies of a match are filtered out by the function "CanMatchTakeInNode(*oldMatch*, *newNode*)".

that the algorithm also finds all k-matches, grouped according to non-isomorphic k-templates. For this purpose, let $S \subset N_G$ be a match of the k-template $T_G[S]$. Recall that S is supposed to be a connected set. Order its nodes $\{n_1, n_2, \cdots, n_k\}$ as follows: The first node n_1 is the node with the smallest serial number. The set of all the other nodes is ordered according to their distance to n_1, in order of increasing distance, where the nodes with the same distance to n_1 are sorted by their serial numbers, again in increasing order. It is obvious that this ordering of k nodes is unique once the serial numbers are fixed.

It is also clear that the set $S' = S \backslash \{n_k\}$ is connected and hence is a match of a $(k-1)$-template with leading node n_1. By the induction hypothesis, S' and a $(k-1)$-template isomorphic to $T_G[S']$ are found by our algorithm. Now, since S is a child match of S', the pair (*oldMatch*=$\{n_1, n_2, \cdots, n_{k-1}\}$, *newNode*=$n_k$) will be passed on to the function "CanMatchTakeInNode(*oldMatch*, *newNode*)" and one easily checks that it satisfies the three conditions. Therefore $\{n_1, n_2, \cdots, n_k\}$ is a match, and either $T_G[S]$ will be listed by the algorithm as a new template or S will be listed as a match of an previously found isomorphic template. This shows that all connected sets with at most *maxsize* nodes will be listed as a match of some template.

Next we prove that no isomorphic templates are listed more than once as a template by the algorithm. This is clear because for each potential template $T_G[S]$, the algorithm checks the already listed templates for isomorphism and either list $T_G[S]$ as a new template or lists S as a new match for an already listed template.

To complete the proof, we will show that S is not listed twice. Suppose that both S' and S, $S' = S$, are both listed as a match by the algorithm and suppose the nodes of S' have be ordered $\{n'_1, n'_2, \cdots, n'_k\}$, which is different from $\{n_1, n_2, \cdots, n_k\}$. Note that the order presents the sequence in which nodes enter a match. If the node in S' with the smallest serial number is not at the first position n'_1, i.e., n_1 is not the leading node, it will never be taken into a match due to condition 1 of the function 'CanMatchTakeInNode". So, we may assume that $n'_1 = n_1$. If a node with label n_b appears before a node with label n_e in the order $\{n'_1, n'_2, \cdots, n'_k\}$ of S' while n_b appears after n_e in the order $\{n_1, n_2, \cdots, n_k\}$ of S, this could have had two reasons: (a) the distance between n_b and the leading node in S is larger than the distance between n_e and the leading node; or (b) these distances are equal in S, and the serial number of n_b is larger than that of n_e. The former situation conflicts with condition 2 and the latter one conflicts with condition 3 of the

function 'CanMatchTakeInNode". This completes the proof of Theorem 5.1. ∎

For a random DFG, the number of templates and matches can be very large, i.e., exponential in the number of nodes of the DFG [14]. Of course, if *maxsize* is fixed, the number of matches is always bounded by n^k, where $n = |N_G|$ is the total number of nodes and k =*maxsize*, so polynomial in n. In practice, the admissible match space can indeed be restricted significantly by adding constraints to the templates. In our Montium tile, for instance, the number of inputs, outputs, and operations are limited. The templates that do not satisfy such limitations will be discarded. When DSP-like applications are considered, the size of the set of admissible matches can be further decreased.

Graph isomorphism algorithm A graph isomorphism algorithm, Gemini, is presented in [25]. In the Gecko project we use a modified algorithm presented in [50] which uses small initial checks to accelerate the procedure.

5.2.2 The Template Selection Algorithm

So far, we have presented an algorithm to generate a set of templates Ω and their corresponding matches $M(\Omega)$ from G. Now we present a heuristic to find an approximate solution to the template selection problem. The scheduling of the matches of the templates on the ALU-architecture in as few clock cycles as possible is not part of the solution concept given here. This last phase will be dealt with in the next chapter.

Given G, $\Omega = \{T_1, T_2, \cdots, T_p\}$ and the matches $M(\Omega)$, the objective is to find a subset C of the set $M(\Omega)$ that forms a 'good' cover of G. Here by 'good' cover we mean a (k, ℓ)-cover with preferably both minimum k and ℓ.

Since the generated set $M(\Omega)$ can be quite large, the template and match selection problem is computationally intensive. We adopt a heuristic based on one for the maximum independent set problem, and apply it to a conflict graph related to our problem, similarly as it was done in [50][77].

A *conflict graph* is an undirected graph $\tilde{G} = (V, E)$. Each match $S \in M(\Omega)$ for a template of the DFG G is represented by a vertex v_S in the conflict graph $\tilde{G}$. If two matches S_1 and S_2 have one or more nodes in common, there will be an edge between the two corresponding vertices v_{S_1} and v_{S_2} in the conflict graph $\tilde{G}$. The *weight* $w(v_S)$ of a conflict graph vertex v_S is the number of DFG nodes $|S|$ within the corresponding match S. The

vertex set of the conflict graph is partitioned into subsets, each of which corresponds to a certain template (see Figure 5.9). Therefore, on the conflict graph, vertices of the same subset have the same weight. The *maximum independent set (MIS) for a subset* $T \subset V(\tilde{G})$ is defined as the largest subset of vertices within T that are mutually nonadjacent. There might exist more than one MIS for T. Corresponding to each MIS for T on $\tilde{G}$, there exists a set of node-disjoint matches in G for the template corresponding to T; we call this set of matches a *maximum non-overlapping match set* (MNOMS). To determine a cover of G with a small number of distinct templates, the

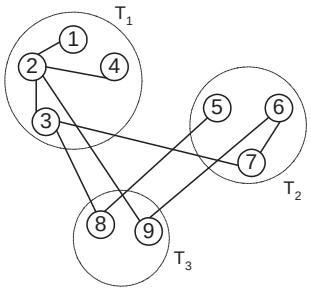

Figure 5.9: A conflict graph. The weight of each node is 4.

templates should cover a rather large number of DFG nodes, on average. The following theorem expresses the relationship between this goal and the concept of MNOMSs.

Theorem 5.2 *Given a DFG G and a template T of G, the template matches of a MNOMS (corresponding to a MIS in $\tilde{G}$) yield an optimal cover of G, where optimal is to be understood as covering a maximum number of nodes of G.*

The proof to this theorem is straightforward and can be found in [50] or [51].

An MNOMS corresponds to a MIS on the conflict graph. Finding a MIS in a general graph, however, is an NP-hard problem [30]. Fortunately, there are several heuristics for this problem that give reasonably good solutions in practical situations. One of these heuristics is a simple minimum-degree based algorithm used in [38], where it has been shown to give good results. Therefore, we adopted this algorithm as a first approach to finding 'good' coverings for the DFGs within our research project.

For each template T, an *objective function* is defined by:

$$g(T) = g(w, s),$$

where w is the weight of each vertex and s is the size of an approximate solution for a MIS within the subset corresponding to T on the conflict graph. The outcome of our heuristic will highly depend on the choice of this objective function, as we will see later.

The pseudo-code of the selection algorithm is shown in Figure 5.10. This is an iterative procedure, similar to the methods in [14][21][77]. At each round, after computing an approximate solution for the MISs within each subset, out of all templates in Ω, the heuristic approach selects a template T with a maximum value of the objective function, depending on the weights and approximate solutions for the MISs. After that, on the conflict graph, the neighbor vertices of the selected approximate MIS and of the approximate MIS itself are deleted. This procedure is repeated until the set of matches C corresponding to the union of the chosen approximate MISs, covers the whole DFG G.

1 Cover $C = \phi$;
2 Build the conflict graph;
3 Find a MIS for each group on the conflict graph;
4 Compute the value of the objective function for each template;
5 The T with the largest value of the objective function is the selected template. Its MIS is the selected MIS. The MNOMS corresponding to the MIS is put into C.
6 On the conflict graph, delete the neighbor vertices of the selected MIS, and then delete the selected MIS;
7 Can C cover DFG totally? If no, go back to 3; if yes, end the program.

Figure 5.10: The pseudo-code for the proposed template selection algorithm

Theorem 5.3 *Using the template generation algorithm, when there is no vertex left on the conflict graph $\tilde{G}$ after removing the vertices and neighbors of the approximate MISs corresponding to the MNOMSs of C, the matches of the generated C cover the whole DFG G and vice versa.*

Proof Every node in the DFG G forms a 1-node match, which is represented by a vertex in the conflict graph $\tilde{G}$. Suppose a node n of G is not covered. Then this node n is still part of a match (or several matches) that are represented by vertices in the graph that results from $\tilde{G}$ after removing the vertices and neighbors of the approximate MISs. (At least the vertex v_n representing the 1-match n will not have been removed, because it does only conflict with matches that contain n.) Hence there is at least one vertex left in the remaining graph.

On the other hand, if there is a vertex v of the conflict graph left, then v corresponds to a match S, such that none of the nodes of S is a node of a match of a MNOMS corresponding to an already chosen approximate MIS (otherwise v would have been removed as (a neighbor vertex of) a vertex contained in one of the chosen approximate MISs). Therefore S is uncovered. ∎

We used the following objective function:

$$g(T) = w^{1.2} \cdot s = ws \cdot w^{0.2}. \tag{5.1}$$

In this function, for a template T, ws equals the total number of DFG nodes covered by a MNOMS, which expresses a preference for the template whose MNOMS covers the largest number of nodes. Furthermore, due to the extra factor $w^{0.2}$, the larger templates, i.e., the templates with more template nodes, are more likely to be chosen than the smaller templates. As described before, the template selection algorithm should be developed to find a cover with, in some sense, a minimum number of (matches of) templates. It is obvious that this is more likely the more nodes each match (template) contains, but it could conflict with finding suitable covers; the latter seems to be easier when the matches (templates) contain fewer nodes.

An example is shown in Figure 5.11, in which terminals are omitted. For this DFG, the following holds:

$$w_A s_A = 3 \cdot 5 = 15,$$

$$w_B s_B = 1 \cdot 15 = 15.$$

We prefer *Cover A* to *Cover B* since it consists of fewer matches of templates, which is caused by the factor $w^{0.2}$.

Example: Suppose every vertex of the conflict graph in Figure 5.9 represents a 4-match on the DFG. The sets {1,3,4}, {5,7} and {8,9} are MISs

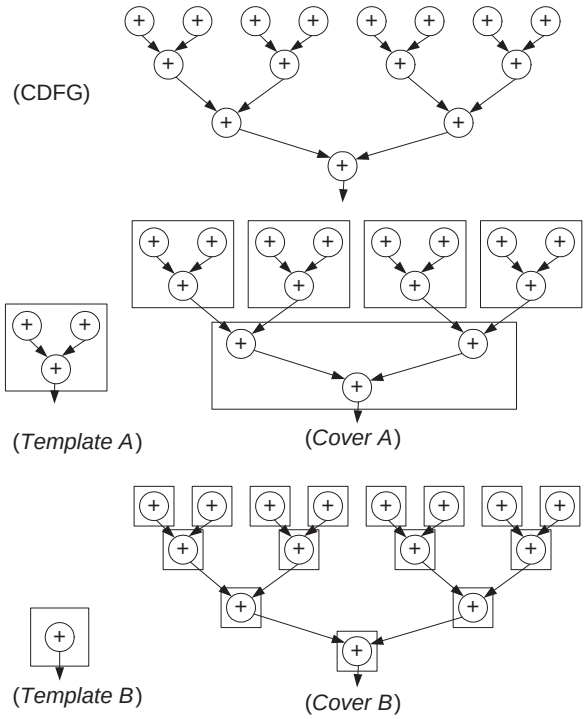

Figure 5.11: Giving the template with more DFG nodes more preference

for T_1, T_2, T_3 respectively. Furthermore, the objective function is given by Equation (5.1). We get:

$$g(T_1) = 4^{1.2} \cdot 3,$$
$$g(T_2) = 4^{1.2} \cdot 2,$$
$$g(T_3) = 4^{1.2} \cdot 2.$$

T_1 has the largest value of the objective function, so the selected MIS is {1,3,4}. The matches corresponding to vertex 1, 3 and 4 are chosen to be put into the cover C first. After that, we remove the neighbor vertices of the selected MIS and the vertices of the selected MIS itself. The newly obtained conflict graph is shown in Figure 5.12, which is the input for the next round. This time {5,6} will be selected and the algorithm terminates.

After the template selection, the selected matches will be scheduled and mapped onto Montium structure and each match can be executed by one ALU in one clock cycle [79].

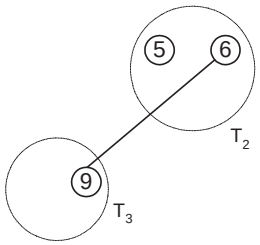

Figure 5.12: The conflict graph after the first round

5.3 Experiments

We tested the template generation and selection algorithms using some test DFGs that are typical DSP functions, namely:

- *FFT4*: 4-point FFT;
- *FFT8*: 8-point FFT;
- *FFT16*: 16-point FFT;
- *FIR*: FIR filter with 128 taps;
- *DCT*: one-dimensional, 8-point discrete cosine transform;
- *TEA*: tiny encryption algorithm.

For each test DFG, a set of matches of templates was generated and a cover was produced using the generated matches. Equation (5.1) in Section 5.2.2 was taken as the objective function. All the generated templates can be executed by one Montium ALU (Figure 2.8), unlike the experimental result in [32], where the generated templates fitted on one tile (Figure 2.7).

As an example Figure 5.13 presents the produced cover for FFT4. The letters inside the dark circles indicate the templates. For this DFG, among all the templates, three have the highest objective function value. The DFG is completely covered by them. This result is the same as our manual solution. The same templates are chosen for an n-point FFT ($n = 2^d$).

The other DFGs are too large to be presented here. The results are summarized in Table 5.2.
From the table, it can be seen that:

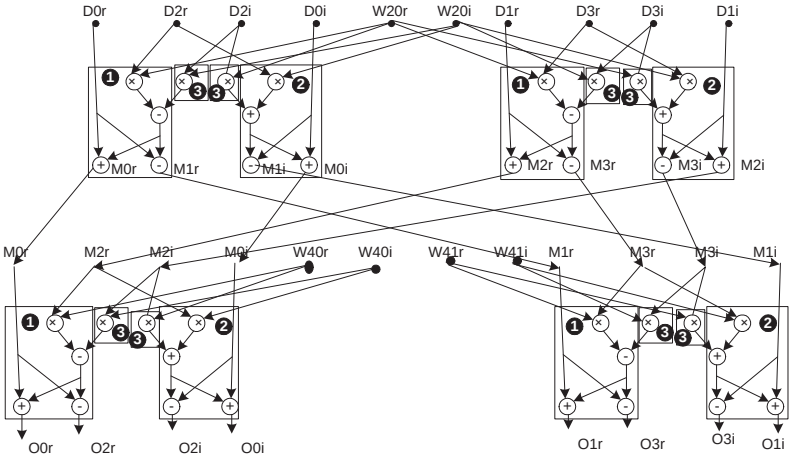

Figure 5.13: The DFG for a 4-bit FFT-algorithm

- Compared to the total number of nodes in the DFG, the number of templates needed to cover a DFG of a DSP function is small. This is due to the highly regular structure of the DFGs of DSP functions.
- The numbers of templates for FFT4, FFT8 and FFT16 are almost the same although they have different number of fundamental operators. This is because of the similarity in the structures of their DFGs.
- The number of templates will not increase unlimited with the total number of nodes. This is due to limitation of the *maxsize* of templates and the ALU structure.
- The more regular a DFG is, the fewer templates are generated by the algorithm.

5.4 Introducing cluster nodes in DFG

Now we use a *cluster node*, shown in Figure 5.14, to represent a selected match. The corresponding template of the match is a one-ALU configuration. Each cluster node has the following attributes:

- Name, which represents the node;

Application	Number of nodes	Number of generated matches	Number of generated templates	Number of selected matches	Number of selected templates
FFT4	40	≈ 500	≈ 200	16	3
FFT8	120	≈ 1500	≈ 200	48	3
FFT16	320	≈ 4400	≈200	128	3
FIR	255	≈ 3700	≈ 35	139	4
DCT	43	≈ 1000	≈ 500	21	7
TEA	80	≈ 700	≈ 80	42	5

Table 5.2: Using template coverings for DSP-like applications

- 5 inputs: Ra, Rb, Rc, Rd and East. They are denoted as Clu1.Ra, Clu1.Rb, Clu1.Rc, Clu1.Rc and Clu1.East;
- 3 outputs: Out1, Out2 and West. They are denoted as Clu1.Out1, Clu1.Out2 and Clu1.West;
- One-ALU configuration,which configuration will represent the structure of the corresponding template of the selected match which the cluster represents. The function can be presented by a graph or it can also be presented by several formulas. The One-ALU configuration is written as Conf(Clu1) or Clu1.Conf.

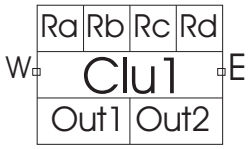

Figure 5.14: A Cluster Node

For simplicity, the west output and the east input are not drawn in figures in the rest of the thesis when they are not considered. After replacing selected matches by cluster nodes, the DFG is called *clustered DFG*. Note that there are only cluster nodes in a clustered DFG. The clustered graph is the input for the scheduling algorithm presented in Chapter 6.

5.5 Overview of related work

There are two kinds of research that relate to the work presented in this chapter: finding regular patterns and graph clustering.

Finding or using templates This is done in the areas of high-level synthesis and FPGA logic synthesis. Here the work is focused on finding or using some regular templates in a graph. The found templates can be used for generating CLBs in FPGAs or an application-specific integrated circuit design.

In [16][22], a template library is assumed to be available and the template matching is the focus of their work. However, this assumption is not always valid, and hence an automatic compiler must determine the possible templates by itself before coming up with suitable matchings. [4][55][77] give some methods to generate templates. These approaches choose one node as an initial template and subsequently add more operators to the template. There is no restriction on the shape of the templates. The drawback is that the generated templates are highly dependent on the choice of the initial template. The heuristic algorithm in [50] generates and maps templates simultaneously, but cannot avoid ill-fated decisions. The clustering algorithm in [60] started from one node template and it limited the shape to the template with one output.

The algorithms in [14][21] provide all templates of a DFG. The complete set of tree templates and single-PO (single principle output) templates are generated in [21] and all the single-sink templates (possibly multiple outputs) are found by the configuration profiling tool in [14]. The central problem for template generation algorithms is how to generate and enumerate all the (connected) subgraphs of a DFG. The methods employed in [21] and [14] can only enumerate the subgraphs of specific shapes (tree shape, single output or single sink) and as a result, templates with multiple outputs or multiple sinks cannot be generated. In reconfigurable architectures, the ALUs can have more than one outputs. For example, in the Montium, each ALU has three outputs, so the existing algorithms cannot be used.

As far as we know, no algorithm has been designed to generate the complete set of templates without limitations to the shapes. In the Montium, complex shapes of templates are allowed, which asks for new template generating and matching techniques.

Graph clustering The work of graph clustering is done for generating instructions for some clustered register file micro-architectures, where the register file and functional units are partitioned and grouped into clusters [17] [27] [49] [68]. All these works consider not only the clustering phase but also the scheduling phase. The focus is to decrease the number of clock cycles of the scheduling. The number of clock cycles is also one of our minimization goals. However, the regularity is not considered there, which is more important for the Montium tile processor due to the limitation of the configuration space.

5.6 Conclusion and future work

The major contributions of this chapter are the algorithms developed to generate all possible non-isomorphic templates and the corresponding matches. The generated templates are not limited in shapes (as opposed to the approaches in [14][21]), i.e., we can deal with templates with multiple outputs or multiple sinks. By applying a clever labeling technique, using unique serial numbers and so-called circle numbers, we were able to save on execution time and memory, and avoid listing multiple copies of isomorphic templates or matches. This approach is applicable to data flow graphs as well as the general netlists in circuit design.

Using the templates and corresponding matches, an efficient cover for the control data flow graphs can be found, which tries to minimize the number of distinct templates that are used as well as the number of instances of each template. The experiments showed promising results.

The clustering algorithm partitions a DFG into subgraphs. Nodes of each such subgraph can be replaced by a cluster node which represents the behavior of the subgraph.

The following work for the clustering phase could be done in the future:

- More DFGs can be tested by the presented method, and then the results can be compared with the manual solutions. By checking the difference between the automatical results and manual results, we can further manipulate and optimize the objective function, and therefore improve the template selection algorithm.

- We believe that the performance of clustering could be further improved when templates are selected two by two. That is, we define an objective

function for each combination of two templates. At each round the two templates corresponding to the largest objective functions are selected.

- Another research direction is to decrease the complexity. Our suggestion is not to generate all templates first. Instead, we should start from the best one-node template (according to some priority function). This approach is more greedy than the one presented in this chapter, which unavoidably will decrease the performance of clustering. For instance, many templates might get no opportunity to be investigated. On the other hand the complexity might be much lower than the presented algorithm because not all matches and templates are checked.
- An east-west connection of the Montium allows an ALU to use the west result of its east neighbor ALU without buffering the result to a register. To use this function, two connected selected matches should be investigated to find out whether they can fit into two adjacent ALUs.

Chapter 6

Scheduling of clusters

The color-constrained scheduling problem is presented in this chapter[1]*. The problem is tackled by three algorithms: the multi-pattern scheduling algorithm, the column arrangement algorithm and the pattern selection algorithm. Simulations show that the presented algorithms lead to good results.*

The scheduling problem is concerned with associating nodes of a DFG to resources and clock cycles such that certain constraints are met. For the scheduling problem in the Montium tile, the resources refer to reconfigurable ALUs.

In Chapter 5, we presented a clustering algorithm. After the clustering phase, all nodes of the corresponding DFG graph of a CDFG are clusters. In this chapter, the clusters are scheduled on reconfigurable ALUs. During the scheduling phase, we have to take the constraints and limitation of the processor resources into consideration. For example, in the Montium, the numbers of both one-ALU configurations and five-ALU configurations are limited. The

[1]Parts of this chapter have been published in publications [12] [13].

scheduling problem with this kind of limitations is called a *color-constrained scheduling problem* which has never been studied before. As discussed in Chapter 3, the energy consumption overhead of the decoding part can be reduced dramatically by techniques which limit the configuration space. We believe that techniques similar to the Montium decoding technique will be used more and more in future architecture designs. Therefore, studying the color-constrained scheduling problem is important not only for the Montium, but also for future architectures. To generalize the research problem, instead of using the terms one-ALU configuration and five-ALU configuration, we use the terms color and pattern to describe our approach in the rest of this chapter. The term color corresponds to a one-ALU configuration and pattern corresponds to a five-ALU configuration.

6.1 Definition

If each node of a graph is given a color, the DFG is called a *colored graph* The graph is called *L-colored* if the total number of different colors is L. The clustered DFG is actually a colored graph if a color is used to represent an one-ALU configuration. As we mentioned in Chapter 5, a clustered DFG only consists of cluster nodes. Therefore, every node has a color. L is the number of selected templates.

In a system with a fixed number (denoted by C, which is 5 in the Montium architecture) of reconfigurable ALUs, each individual ALU can have a limited number of different one-ALU configurations (the number of different configurations is 8 in the current Montium architecture). We use a *color* to represent the type of a function that a reconfigurable ALU can execute (for the Montium the color of a cluster is the corresponding one-ALU configuration of the cluster). Scheduling is to associate each node of the colored graph to an ALU within a clock cycle, given some constraints. We can look at this kind of scheduling problem in another way, i.e., putting each node of the DFG into a table of C columns and as few as possible rows such that the constraints are satisfied. A scheduling solution with R clock cycles can thus be written as an R by C *schedule table* $\mathbf{S}$:

$$\mathbf{S} = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,C} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,C} \\ \cdots & \cdots & \cdots & \cdots \\ s_{R,1} & s_{R,2} & \cdots & s_{R,C} \end{pmatrix}. \tag{6.1}$$

Each $s_{i,j}$ is a node n of the input colored graph or "-" which means "empty". In this chapter, we use $|N|$ to denote the number of nodes of a DFG $G(N, E)$. When $|N| < R \times C$, $(R \times C - |N|)$ positions of the schedule table will be "-".

Let $l(n)$ denote the color of a node n. The *color table* of a schedule $\mathbf{S}$ is defined as:

$$\mathbf{L} = \mathbf{L}(\mathbf{S}) = \begin{pmatrix} l(s_{1,1}) & l(s_{1,2}) & \cdots & l(s_{1,C}) \\ l(s_{2,1}) & l(s_{2,2}) & \cdots & l(s_{2,C}) \\ \cdots & \cdots & \cdots & \cdots \\ l(s_{R,1}) & l(s_{R,2}) & \cdots & l(s_{R,C}) \end{pmatrix}. \tag{6.2}$$

We define a color dummy (meaning do not care), denoted by "*", for the empty node "-", which means this position can be any color.

As mentioned, C functions that can be run by the C parallel reconfigurable ALUs form a pattern. (In the Montium, a pattern corresponds to a five-ALU configuration.) A pattern is therefore a vector of C elements. Each row on the color table $\mathbf{L}$ defines a pattern. We use $\vec{p} = \{p_1, p_2, \cdots, p_C\}$ to represent a pattern.

For two patterns $\vec{p1} = \{p1_1, p1_2, \cdots, p1_C\}$ and $\vec{p2} = \{p2_1, p2_2, \cdots, p2_C\}$, we say that *pattern* $\vec{p1}$ *is a subpattern of* $\vec{p2}$ if at each position i, $1 \leq i \leq C$, $p1_i =$ "$*$" or $p1_i = p2_i$. For instance, $p1 =$ {a, a, b, *, c} is a subpattern of $p2 =$ {a, a, b, b, c}. Notice that a pattern is always a subpattern of itself.

If we delete those patterns which are subpatterns of another pattern in a color table $\mathbf{L}(\mathbf{S})$, we can obtain a *pattern table* $\mathbf{P}$ for a schedule $\mathbf{S}$. We use P to denote the number of different patterns used in a schedule $\mathbf{S}$. P is actually equal to the number of rows in $\mathbf{P}$.

In a pattern table $\mathbf{P}$, all different colors in column i form a *column color set* $u_i(\mathbf{P})$, $1 \leq i \leq C$, which represents all configurations the ALU i has. $|u_i(\mathbf{P})|$ denotes the number of elements in $u_i(\mathbf{P})$.

On a DFG, two nodes n_1 and n_2 are called *parallelizable* if neither n_1 is a follower of n_2 nor n_2 is a follower of n_1 (the definition of follower is given in Chapter 4). If $\mathcal{A}$ is a set of pairwise parallelizable nodes we say that $\mathcal{A}$

is an *antichain*. (this concept of antichain is borrowed from the theory of posets, i.e. partially ordered sets. We refer to [95] for more information.) If the size of an antichain $\mathcal{A}$ is smaller than or equal to the number of columns of the expected scheduled table $\mathbf{S}$, we call $\mathcal{A}$ is *executable*. Notice that the elements of each row of the schedule table $\mathbf{S}$ form an executable antichain. In this chapter, we only discuss executable antichains because only executable antichains are useful in practice.

All elements of a pattern $\vec{p}_1 = \{p_{11}, p_{12}, \cdots, p_{1C}\}$ form a bag[2] of colors. It is called a *non-ordered pattern* and is written as $\bar{p}$. If the colors of the elements of an executable antichain $\mathcal{A}$ form a non-ordered antichain $\bar{p}$, $\mathcal{A}$ is called an *instance* or *antichain* of $\mathcal{S}$. $\mathcal{S}$ is called the corresponding pattern of $\mathcal{A}$. A sub-bag (see [9] for the definition of sub-bag) of a non-ordered pattern forms another non-ordered pattern, which is called a subpattern of the original non-ordered pattern.

For clarity, we list the mentioned symbols here again:

- **S:** schedule table;
- **L or L(S):** color table for schedule **S**;
- **P or P(S):** pattern table for schedule **S**;
- C**:** number of columns of schedule **S**;
- R**:** number of rows of schedule **S**, i.e., number of clock cycles of the schedule **S**;
- G**:** input DFG G;
- $|N|$**:** number of nodes in the input DFG G;
- n**:** a node in the DFG G;
- N_G**:** the set of all the nodes in the input DFG G;
- n_i**,** $1 \leq i \leq |N|$**:** ith node in the input DFG G;
- L**:** number of different colors in the graph G;

[2] A bag or multiset is a collection of objects the members of which need not be distinct [5].

- $\mathcal{L}$**:** the set of all the colors in the input DFG;
- l**:** a color;
- l_i**,** $1 \leq i \leq L$**:** ith color;
- $l(n)$**:** the color of node n;
- P**:** number of patterns of a schedule $\mathbf{S}$, i.e., number of rows in the pattern table $\mathbf{P}(\mathbf{S})$;
- P_{def}**:** maximum number of allowed patterns;
- $\vec{p} = \{p_1, p_2, \cdots, p_C\}$: a pattern;
- $|\vec{p}|$: the number of colors of the pattern $\vec{p}$;
- $\vec{p_i} = \{p_{i1}, p_{i2}, \cdots, p_{iC}\}, 1 \leq i \leq P$: ith pattern;
- $\bar{p} = \{p_1, p_2, \cdots, p_C\}$: a non-ordered pattern;
- $|\bar{p}|$: the number of colors of the non-ordered pattern $\bar{p}$;
- $\bar{p_i} = \{p_{i1}p_{i2}, \cdots, p_{iC}\}, 1 \leq i \leq P$: ith non-ordered pattern;
- $u_i(\mathbf{P})$**,** $1 \leq i \leq P$**:** the column color set for ALU i according to pattern table $\mathbf{P}$, i.e., the set of configurations for ALU i;
- $|u_i(\mathbf{P})|$**,** $1 \leq i \leq P$**:** the size of the column color set for ALU i according to pattern table $\mathbf{P}$;
- U_{def}**:** maximum number of allowed elements in each column color set;

Now we define some terms that will be used in this chapter.

The As Soon As Possible level ($ASAP(n)$) attribute indicates the earliest time that the node n may be scheduled. It is computed as:

$$ASAP(n) = \begin{cases} 0 & \text{if } Pred(n) = \phi; \\ \max\limits_{\forall n_i \in Pred(n)} (ASAP(n_i) + 1) & otherwise. \end{cases} \tag{6.3}$$

Here, $Pred(n)$ represents the node set consisting of all predecessors of node n.

The As Late As Possible level ($ALAP(n)$) attribute determines the latest clock cycle the node n may be scheduled. It is computed using the following:

$$ALAP(n) = \begin{cases} ASAP_{max} & \text{if } Succ(n) = \phi; \\ \min\limits_{\forall n_i \in Succ(n)} (ALAP(n_i) - 1) & otherwise. \end{cases} \quad (6.4)$$

Here $ASAP_{max} = \max\limits_{\forall n_i \in N_G} (ASAP(n_i))$, and $Succ(n)$ represents the node set consisting of followers of node n.

The *Height* of a node n is the maximum distance between n and a follower without successors. It is calculated as follows:

$$Height(n) = \begin{cases} 1 & \text{if } Succ(n) = \phi; \\ \max\limits_{\forall n_i \in Succ(n)} (Height(n_i) + 1) & otherwise. \end{cases} \quad (6.5)$$

Figure 6.2 shows the ASAP, ALAP and Height attributes of the nodes of the DFG shown in Figure 6.1.

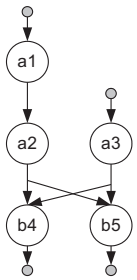

Figure 6.1: A small example

6.2 Problem description

The *color-constrained scheduling problem* is defined as putting each node of the DFG to a table $\mathbf{S}$ of C columns with as few as possible rows such that the following constraints are satisfied:

1. Pattern number condition: The number of different patterns P in the pattern $\mathbf{P}(\mathbf{S})$ is smaller than a defined number P_{def}, i.e., $P \leq P_{def}$ ($P_{def} = 32$ in the Montium).

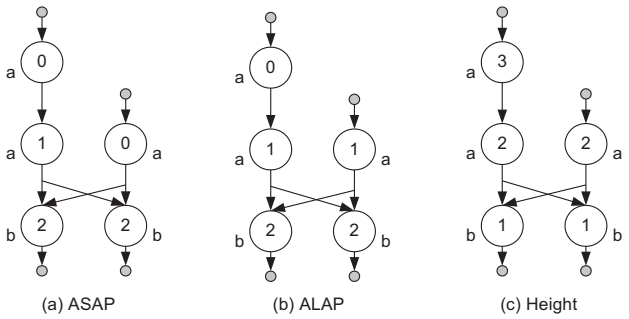

Figure 6.2: ASAP, ALAP and Height attributes of the nodes in the DFG shown in Figure 6.1

2. Column condition: The column color set for every ALU should contain at most U_{def} elements ($U_{def} = 8$ in the Montium), i.e., $|u_i(\mathbf{P})| \leq U_{def}$, $1 \leq i \leq C$.

3. The graph constraint: The successors of a node should be scheduled after the node has been scheduled.

Because the number of configurations for each ALU cannot be larger than the number of total patterns, all the discussions in this chapter are based on the assumption that:

$$U_{def} \leq P_{def}. \tag{6.6}$$

We also assume that

$$P_{def} \leq (U_{def})^C. \tag{6.7}$$

These assumptions hold in Montium, where $U_{def} = 8$, $P_{def} = 32$ and $C = 5$. If the assumption given in Equation (6.7) is not true, the pattern number condition will be satisfied automatically when the column condition is satisfied. With this assumption, the scheduler has to consider both conditions.

Theorem 6.1 *Given a graph of $|N|$ nodes, the number of necessary rows R in the schedule table $\mathbf{S}$ is smaller than or equal to the number of nodes $|N|$ and larger than $\lceil |N|/C \rceil$, i.e., $\lceil |N|/C \rceil \leq R \leq |N|$. $\lceil x \rceil$ refers to the smallest integer larger than x.*

Proof $|N|$ nodes can occupy at most $|N|$ clock cycles. Therefore, $R \leq |N|$.

In each clock cycle, at most C nodes can be executed. Therefore, at least $\lceil |N|/C \rceil$ clock cycles are needed to execute all $|N|$ nodes. ∎

Theorem 6.1 gives an indication of the range of the number of clock cycles of a schedule.

Theorem 6.2 *Given an L-colored $|N|$-node graph, valid schedules with less than or equal to $|N|$ rows exist for the color-constrained scheduling problem, if and only if $L \leq U_{def} \times C$, i.e., $\lceil L/C \rceil \leq U_{def}$.*

Proof First we prove that the statement that $\lceil L/C \rceil \leq U_{def}$ is a necessary condition of the statement that valid schedules exist. At least $\lceil L/C \rceil$ patterns are needed to contain all L colors. If $L > U_{def} \times C$,

$$\lceil L/C \rceil > \lceil \frac{U_{def} \times C}{C} \rceil = U_{def}.$$

Because U_{def} is a natural number, $\lceil L/C \rceil > U_{def}$ is equivalent to $\lceil L/C \rceil \geq U_{def} + 1$. Therefore, it is not possible to build U_{def} patterns which contain all colors in the graph. Thus a valid solution does not exist if $L > U_{def} \times C$.

Now we prove that the statement that $\lceil L/C \rceil \leq U_{def}$ is a sufficient condition, of the statement that valid schedules exist, by finding one schedule under the condition.

- Firstly build a color pattern table in this way:

$$\begin{array}{rcl} \vec{p}_1 & = & \{ \; l_1, \qquad\qquad l_2, \quad \cdots, \quad l_C \; \} \\ \vec{p}_2 & = & \{ \; l_{C+1}, \qquad l_{C+2}, \quad \cdots, \quad l_{2C} \; \} \\ & \vdots & \\ \vec{p}_{\lceil L/C \rceil} & = & \{ \; l_{(\lceil L/C \rceil - 1) \times C + 1}, \quad \cdots, \quad l_L, *, \cdots, \quad * \; \} \end{array} \tag{6.8}$$

  This color pattern table consists of $\lceil L/C \rceil$ rows. From the color pattern table, we can obtain the column color set for each ALU. For example,

$$u_1(\mathbf{P}) = \{l_1, l_{C+1}, l_{2C+1}, \cdots, l_{(\lceil L/C \rceil - 1) \times C + 1}\}.$$

  The C column color sets are non-overlapping, i.e., no color appears in two different column color sets.

- Secondly order the nodes in the graph according to their ASAP level. Order the nodes with the same ASAP level randomly.

- Finally schedule the ordered nodes one by one. Schedule each node n to a new clock cycle following the clock cycle of the previous node, i.e., put each node to a new row of the schedule table. Among C columns, choose the one whose column color set contains the color of the node, i.e., $l(n)$.

After these three steps, we obtain a schedule table of $|N|$ rows and C columns. Only one element of each row is not "-". Because the nodes are ordered according to their ASAP level, and the ASAP level of all predecessors of a node is always smaller than or equal to the ASAP of the node, predecessors of a node are always scheduled in earlier clock cycles. Therefore, the generated table is a schedule. ∎

In the rest of this chapter we assume that $\lceil L/C \rceil \leq U_{def}$.

The more nodes are allowed to be put into one row, the shorter the schedule table will be. Larger U_{def} and P_{def} allow more possible combinations of colors. When U_{def} and P_{def} are fixed, the selection of patterns becomes crucial. Some patterns are better than others. Figure 6.1 shows a small example. The first letter of each node name represents the color of the node. If patterns {a, a} and {b, b} are used, nodes "a1" and "a3", "a2" and "a3", or "b4" and "b5" can be scheduled at the same clock cycle. However, if patterns {a, b} and {b, b} are used "a1", "a2" and "a3" must be scheduled at three different clock cycles. Therefore, we should use "good" patterns to schedule a graph.

The maximum number of allowed patterns is P_{def}. The pattern matrix $\mathbf{P}$ has therefore at most $P_{def} \times C$ elements. From Equation (6.6) and Theorem 6.2, it can be easily seen that

$$L \leq P_{def} \times C. \tag{6.9}$$

This means that some colors may appear more than once in the pattern matrix. In other words, at most $(P_{def} \times C - L)$ elements in the P_{def} patterns are duplicated versions of others.

Based on the above analysis, we define the color-constrained scheduling problem which includes the following two aspects: Choosing at most P_{def} patterns which satisfy the column condition; and scheduling the graph using the given patterns. The procedure of our approach is shown in Figure 6.3 This approach consists of three algorithms:

1 Pattern selection: Select P non-ordered patterns ($P \leq P_{def}$) based on the graph. The chosen patterns should occur often in the input DFG, and should contain all L colors of the DFG.

2 Column arrangement: Given P patterns, allocate the elements of each pattern to a column to minimize (1) the maximum number of colors

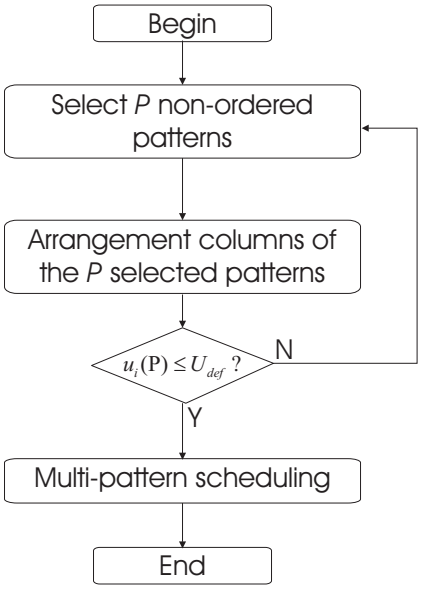

Figure 6.3: Scheduling procedure

per column; or (2) the sum of the numbers of colors of all columns. When these two minimization criteria cannot be reached at the same time, we try to reach a point where both of them are reasonably small.

3 Multi-pattern scheduling: Schedule the DFG using the patterns obtained from step 1 and step 2.

The pattern selection algorithm is designed based on the multi-pattern scheduling algorithm. Therefore, instead of describing the three algorithms in their execution order, we first present the multi-pattern scheduling algorithm in Section 6.3. Then the pattern selection algorithm is presented in Section 6.4. Finally, the column arrangement algorithm is described in Section 6.5.

6.3 A multi-pattern scheduling algorithm

Given a colored graph G and a pattern table $\mathbf{P}$ of at most P_{def} patterns, the multi-pattern algorithm is to schedule the nodes in the graph such that each row in the color table $\mathbf{L}(\mathbf{S})$ is the same as one row of $\mathbf{P}$. The list algorithm is the most commonly used scheduling algorithm for resource-constrained

scheduling problems [45]. The multi-pattern scheduling problem is actually a resource-constrained scheduling problem with extra constraints. We propose a multi-pattern scheduling algorithm for this problem, which is actually a modified version of the traditional list scheduling algorithm.

6.3.1 Algorithm description

A conventional list based algorithm maintains a *candidate list* CL of *candidate nodes*, i.e., nodes whose predecessors have already been scheduled. The candidate list is sorted according to a priority function of these nodes. In each iteration, nodes with higher priority are scheduled first and lower priority nodes are deferred to a later clock cycle. Scheduling a node within a clock cycle makes its successor nodes candidates, which will then be added to the candidate list.

For multi-pattern scheduling, for one clock cycle, not only nodes but also a pattern should be selected. The selected nodes should not use more colors than the colors presented in the selected pattern. For a specific candidate list CL and a pattern $\vec{p}_i$, a *selected set* $S(\vec{p}_i, CL)$ is defined as the set of nodes from CL that will be scheduled provided the colors given by $\vec{p}_i$.

The multi-pattern scheduling algorithm is given in Figure 6.4. In total two types of priority functions are defined here, the *node priority* and the *pattern priority*. The former is for each node in the graph and the latter is for scheduling elements from the candidate list by one specific pattern.

1. Compute the priority function for each node in the graph.
2. Get the initial candidate list.
3. Sort the nodes in the candidate list according to their priority functions.
4. Schedule the nodes in the candidate list from high priority to low priority using each given pattern.
5. Compute the priority function for each pattern and keep the one with highest priority function.
6. Update the candidate list.
7. If the candidate list is not empty, go back to 3; else end the program.

Figure 6.4: Multi-Pattern List Scheduling Algorithm

Node priority

In the algorithm, the following priority function for graph nodes is used:

$$\begin{aligned} f(n) &= s \times height(n) + t \times \#\text{successors(n)} \\ &\quad +\#\text{followers(n)}. \end{aligned} \tag{6.10}$$

The definitions of successor and follower are given in Chapter 4. Here #successors is the number of the successors that follow the node n directly, and #followers is the number of followers of n. Parameters s and t are used to distinguish the importance of the factors. s and t should satisfy the following conditions:

$$\begin{aligned} s &\geq \max\{t \times \#\text{successors(n)} \\ &\qquad +\#\text{followers(n)}\} \\ t &\geq \max\{\#\text{followers(n)}\}. \end{aligned} \tag{6.11}$$

These conditions guarantee that: (1) the node with largest height will always have the highest priority; (2) for the nodes with the same height, the one with more successors will have higher priority; (3) for the nodes with both the same height and the same number of successors, the one with highest number of followers will have highest priority.

The height of a node reflects its scheduling flexibility. For a given candidate list, the node with smaller height is more flexible in the sense that it might be scheduled at a later clock cycle. Nodes with largest height are given the preference to be scheduled earlier. The scheduling of the nodes with more successors will assume that more nodes go to the candidate list, they are therefore given higher priority. Furthermore, the node with more successors is given higher priority since the delaying of the scheduling of this node will cause the delaying of the scheduling of more successors.

Pattern priority

Intuitively for each clock cycle we want to choose the pattern that can cover most nodes in the candidate list. This leads to a definition of the priority function for a pattern $\vec{p}$ corresponding to a candidate list CL.

$$F_1(\vec{p}, CL) = \text{number of nodes in selected set } S(\vec{p}, CL). \tag{6.12}$$

On the other hand, the nodes with higher priorities should be scheduled before those with lower priorities. That means that we prefer the pattern

that covers more high priority nodes. Thus we define the priority of a pattern as the sum of priorities of all nodes in the selected set.

$$F_2(\vec{p}, CL) = \sum_{n \in S(\vec{p}, CL)} f(n). \tag{6.13}$$

6.3.2 Example

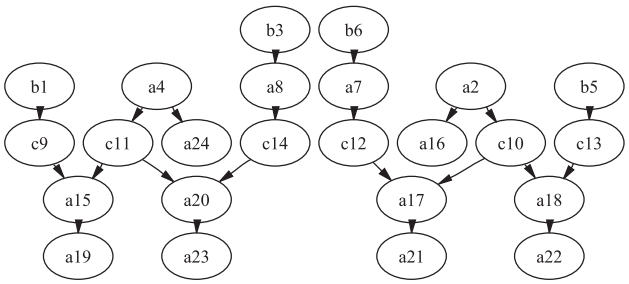

Figure 6.5: Multi-pattern scheduling example: 3FFT algorithm

clock cycle	candidate list	pattern1 =“aabcc”	pattern2 =“aaacc”	selected pattern
1	a2,a4,b1,b3,b5,b6	a2,a4,b6	a2,a4	*1*
2	b1,b3,b5,c11,a24, a16,c10,a7	a7,a24,b3,c10, c11	a24,a16,a7,c11, c10	*1*
3	a8,a16,b1,b5,c12	a8,a16,b5,c12	a8,a16,c12	*1*
4	b1,c14,a17,c13	a17,b1,c13,c14	a17,c13,c14	*1*
5	a18,a20,a21,c9	a18,a20,c9	a18,a20,a21,c9	*2*
6	a15,a22,a23	a15,a22	a15,a22,a23	*2*
7	a19	a19	a19	*1*

Table 6.1: Scheduling Procedure

We explain the algorithm with the help of the 3-point Fast Fourier Transform (3FFT) algorithm. The DFG of 3FFT consists of additions, subtractions and multiplications, as shown in Figure 6.5. The nodes denoted by “a” are additions; while those with “b” represent subtractions and the nodes with “c” multiplications. Two patterns are assumed to be given here: pattern1 = “aabcc” and pattern2 = “aaacc”. In Section 6.4, we will see how these patterns are selected. The scheduling procedure is shown in Table 6.1. Initially,

clock cycle	scheduled nodes
1	a2,a4,b6
2	a7,a24,b3,c10,c11
3	a8,a16,b5,c12
4	a17,b1,c13,c14
5	a18,a20,a21,c9
6	a15,a22,a23
7	a19

Table 6.2: Final scheduling

there are six candidates: {a2, a4, b1, b3, b5, b6}. If we use pattern1 {a2, a4, b6} will be scheduled, and if we use pattern2 {a2, a4} will be scheduled. Because the priority function of pattern1 is larger than that of pattern2, pattern1 is selected. For the second clock cycle, pattern1 covers nodes {a7, a24, b3, c10, c11} while pattern2 covers {a7, a16, a24, c10, c11}. The difference between the use of the two patterns lies in the difference between b3 and a16. If we use the pattern priority function $F_1(\vec{p}, CL)$ defined in Equation (6.12), the two patterns are equally good. The algorithm will pick one at random. If we use $F_2(\vec{p}, CL)$ defined in Equation (6.13) as pattern priority function, pattern1 will be chosen because the height of b3 is larger than that of a16. The final scheduling result, using $F_2(\vec{p}, CL)$ as pattern priority, is shown in Table 6.2.

6.3.3 Complexity comparison with fixed-pattern list scheduling

For each clock cycle, the multi-pattern scheduling algorithm schedules the nodes in the candidate list using every given pattern. The rest is the same as in the traditional resource constrained list scheduling algorithm which uses a fixed pattern. The computational complexity of the multi-pattern scheduling algorithm is therefore $O(R \times$ *Complexity of fixed-pattern list algorithm*$)$, where R is the number of given patterns.

	3FFT		5FFT		15FFT	
number of nodes	24		62		544	
pattern priority	F_1	F_2	F_1	F_2	F_1	F_2
"aabcc"	8		17		153	
{aabcc},{aaacc}	7	7	16	16	139	139
{aabcc},{aaacc},{aaaac}	7	7	16	15	143	134
{aabcc},{aaacc},{aaaac},{aabbc}	6	7	14	14	127	117
{acccc},{abbbc},{aaaaa},{aabbc}					119	110
{aaabc},{ccccc},{aaaaa},{bbbbb}					109	109

Table 6.3: Experimental results: Number of clock cycles for the final scheduling

6.3.4 Experiment

We ran the multi-pattern scheduling algorithm on the 3-, 5- and 15-point Fast Fourier Transform (3FFT, 5FFT and 15FFT) algorithms. For each algorithm two different pattern priorities were tested. The experimental results are given in Table 6.3 where the number indicates the number of clock cycles needed. From the simulation results we have the following observations:

- As more patterns are allowed the number of needed clock cycles gets smaller. This is the benefit we get from reconfiguration.
- In most cases the pattern priority function $F_2(\vec{p}, CL)$ leads to better scheduling than the priority function $F_1(\vec{p}, CL)$. However, because of the greedy strategy of the list algorithm, there is no single priority function which can guarantee to find the best solution.
- The selection of patterns has a very strong influence on the scheduling results!

6.4 Pattern selection

We saw in the previous section that the selection of patterns is very important. In this section we present the method to choose at most P_{def} patterns.

The requirements to the selected patterns are:

1. The selected patterns cover all the colors that appear in the DFG;

2. The selected patterns appear frequently in the DFG (have many instances in the DFG).

3. The selected patterns satisfy the column condition (see page 97).

The procedure of the selection algorithm is shown in Figure 6.6. The selection algorithm first selects P_{def} color bags (non-ordered patterns). Each color bag has at most C colors. This is the topic of this section. A column arrangement is then used to order the colors of each bag to decrease the size of each column color set, i.e., $|u_i(\mathbf{P})|$, which part is presented in section 6.5.

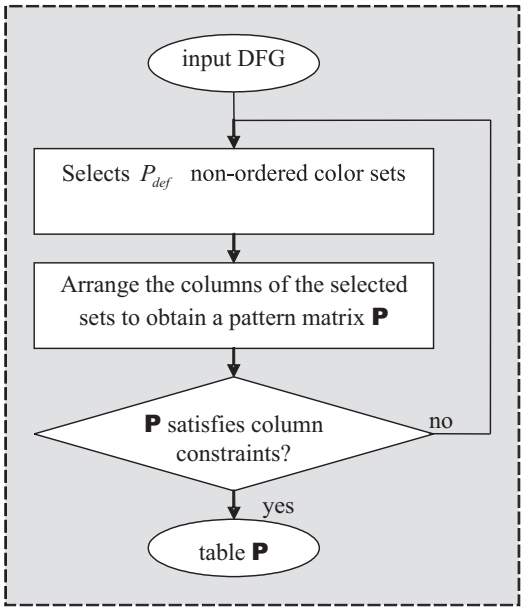

Figure 6.6: The structure of the pattern selection algorithm

Similar to what we have done in the clustering phase (Chapter 5), we first find all the possible non-ordered patterns and their instances in the DFG (pattern generation) in Section 6.4.1, and then make the selection from them (pattern selection) in Section 6.4.2.

6.4.1 Pattern generation

The pattern generation method finds all antichains of size C first and then the antichains are classified according to their non-ordered patterns. The

non-ordered patterns and antichains are written to a pattern-antichain table as follows:

non-ordered pattern1,	instance1,	instance2,	instance3,	$\cdots$
non-ordered pattern2,	instance1,	instance2,	instance3,	$\cdots$
non-ordered pattern3,	instance1,	instance2,	instance3,	$\cdots$
$\vdots$				

The procedure for finding antichains starts from one node antichains, which consist of one single graph node. Larger size antichains are obtained by extending smaller antichains (Figure 6.7). To avoid multiple counting of an antichain, a serial number (unique identity number) is given to each node. The boolean function "CanAntichainTakeInNode($\mathcal{A}$, n)" is used to make sure that an antichain is counted only once. The function is true only when the serial number of the new node n is larger than the serial number of any other node in $\mathcal{A}$. The function is much easier than the function "CanMatchTakeInNode(*oldMatch*, *newNode*)" given on Page 76.

Theorem 6.3 *By using the function "CanAntichainTakeInNode($\mathcal{A}$, n)", all non-ordered patterns and antichains for a DFG are generated. Furthermore, no antichains appear more than once in the pattern-antichain table.*

We do not give the proof here because it is similar to the proof of Theorem 5.1 on Page 76.

```
1  foreach antichain A with i nodes {
2     foreach node n ∉ A {
3        if CanAntichainTakeInNode(A, n) {
4           if(n is parallelizable with every node in A)
5              B = A ∪ {n} is a new antichain with i + 1 nodes;
6        }
7     }
8  }
```

Figure 6.7: Finding antichains of $i + 1$ nodes from an antichain of i nodes

For the graph in Figure 6.1, the classified antichains are listed in Table 6.4:

The number of antichains increases very fast with the size of the pattern (See Table 6.5).

patterns	antichains
$\bar{p}_1 = \{a\}$:	{a1},{a2},{a3}
$\bar{p}_2 = \{b\}$:	{b4},{b5}
$\bar{p}_3 = \{aa\}$:	{a1,a3},{a2,a3}
$\bar{p}_4 = \{bb\}$:	{b4,b5}

Table 6.4: Patterns and antichains in the DFG of Figure 6.1

Number of nodes	1	2	3	4	5
3FFT	22	187	782	1713	1936
5FFT	62	1646	24878	240622	1586004

Table 6.5: The number of antichains

The elements of an antichain may have been chosen from different levels of the DFG. The concept *span* for an antichain $\mathcal{A}$ captures the difference in level, which is defined as follows:

$$Span(\mathcal{A}) = U(\max_{n\in\mathcal{A}}\{ASAP(n)\} - \min_{n\in\mathcal{A}}\{ALAP(n)\}),$$

where, $U(x)$ is a function defined as follows:

$$U(x) = \begin{cases} 0 & x < 0; \\ x & x \geq 0. \end{cases}$$

Looking at an antichain $\mathcal{A} = \{a24, b3\}$ in Figure 6.5, the levels of the nodes are: $ASAP$(a24) = 1, $ALAP$(a24) = 4, $ASAP$(b3) = 0 and $ALAP$(b3) = 0. Therefore,

$$\max_{n\in\mathcal{A}}\{ASAP(n)\} = \max\{1, 0\} = 1;$$

$$\min_{n\in\mathcal{A}}\{ALAP(n)\} = \min\{0, 4\} = 0.$$

The span is

$$Span(\mathcal{A}) = U(1 - 0) = 1.$$

Theorem 6.4 *If the nodes of an antichain $\mathcal{A}$ are scheduled in one clock cycle, the total number of clock cycles will be at least $ASAP_{max}+Span(\mathcal{A})+1$.*

Proof Assume that node $n1$ has the minimal ALAP level and node $n2$ has the maximal ASAP level (see Figure 6.8). Before $n2$, there are at

least $ASAP(n2)$ clock cycles and after $n1$, there are at least $ASAP_{max} - ALAP(n1)$ clock cycles. If $n1$ and $n2$ are run at the same clock cycle, when $ASAP(n2)$ is larger than $ALAP(n1)$ as is the case in Figure 6.8, totally at least $ASAP(n2) + ASAP_{max} - ALAP(n1) + 1$ clock cycles are required for the whole schedule, where the extra 1 is for the clock cycle in which the $n1$ and $n2$ are executed. However, the total number of clock cycles cannot be smaller than $ASAP_{max} + 1$, which is the length of the longest path on the graph. Thus when $ASAP(n2) \leq ALAP(n1)$, the length of the schedule is larger than or equal to $ASAP_{max} + 1$. ∎

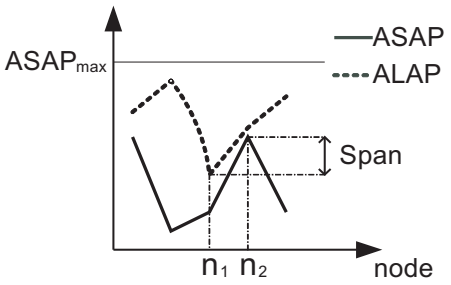

Figure 6.8: Span

Theorem 6.4 shows that to run the nodes of an antichain $\mathcal{A}$ with a too large span in parallel will decrease the performance of the scheduling. A non-ordered pattern with many antichains, all of which are with very large span, is therefore not a favorable pattern. We will see soon that antichains of a non-ordered pattern will contribute to the preference to take the pattern. Due to the above analysis, it is not useful to take antichains with a large span into consideration. For instance, in the graph of Figure 6.5 node "a19" and node "b3" are unlikely to be scheduled to the same clock cycle although they are parallelizable. The number of antichains decreases by setting a limitation to the span of antichains. As a result, the computational complexity is decreased as well. Table 6.6 shows the number of antichains for the 3FFT satisfying the span limitation.

6.4.2 Pattern selection

In the pattern selection phase, the "good" generated non-ordered patterns in the pattern antichain table are selected. The "good" patterns are those

Number of nodes	1	2	3	4	5
$Span(\mathcal{A}) = 4$	24	224	1034	2500	3104
$Span(\mathcal{A}) = 3$	24	222	1010	2404	2954
$Span(\mathcal{A}) = 2$	24	208	870	1926	2282
$Span(\mathcal{A}) = 1$	24	178	632	1232	1364
$Span(\mathcal{A}) = 0$	24	124	304	425	356

Table 6.6: The number of antichains that satisfy the span limitation

which satisfy the requirements given on Page 105.

The pseudo-code for selecting patterns is given in Figure 6.9. Non-ordered

```
1  for(i = 0; i < P_def; i++) {
2     Compute the priority function for each pattern;
3     Choose the pattern with the largest priority
      function;
4     Delete the subpatterns of the selected pattern.
5  }
```

Figure 6.9: The pseudo-code for the pattern selection procedure

patterns are selected one by one based on priority functions. The key technique is the computation of the priority function for each non-ordered pattern (line 2 in Figure 6.9), which is decisive for the potential use of the selected non-ordered pattern. After one non-ordered pattern is selected, all its subpatterns are deleted (line 4) because we can use the selected non-ordered pattern at the place where a subpattern is needed. For example, if a non-ordered pattern {a, b, c, d, e} is selected, it can be used at the place where {a, b, c} is needed.

In the multi-pattern list scheduling algorithm given in Section 6.3, a node which forms a non-ordered pattern with other parallelizable nodes can be scheduled. If the allowed patterns for the multi-pattern list scheduling algorithm have more instances including a node n, it is more easy to schedule node n. Therefore, the number of instances of the selected non-ordered patterns that cover a node should be as large as possible. However, the number should be balanced among all nodes because some unscheduled nodes might decrease the performance of the scheduling.

For each non-ordered pattern $\bar{p}$, a *node frequency*, $h(\bar{p}, n)$ is defined to rep-

resent the number of antichains that include a node n. The node frequencies of all nodes form an array:

$$\vec{h}(\bar{p}) = (h(\bar{p}, n_1), h(\bar{p}, n_2), \cdots, h(\bar{p}, n_{|N|})).$$

$h(\bar{p}, n)$ tells how many different ways there are to schedule n by the non-ordered pattern $\bar{p}$, or we can say that $h(\bar{p}, n)$ indicates the flexibility to schedule the node n by the non-ordered pattern $\bar{p}$. The vector $\vec{h}(\bar{p})$ indicates not only the number but also the distribution of the antichains over all nodes.

Suppose t non-ordered patterns have been selected and they are represented by $\mathbf{P}_s = \{\bar{p}_1, \bar{p}_2, \cdots, \bar{p}_t\}$. The priority function of the remaining non-ordered patterns for selecting the $(t+1)$th non-ordered pattern $\bar{p}_{t+1}$ is defined as:

$$f(\bar{p}_j) = \left(\sum_{n \in N_G} \frac{h(\bar{p}_j, n)}{\left(\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, n) \right) + \varepsilon} \right) + \alpha \times |\bar{p}_j|^2, \qquad \text{for } \bar{p}_j \notin \mathbf{P}_s. \quad (6.14)$$

We want to choose the non-ordered pattern that occurs more often in the DFG. Therefore the priority function is larger when a node frequency $h(\bar{p}_j, n)$ is larger. To balance the node frequencies for all nodes, $\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, n)$ is used, which is the number of antichains containing node n among all the selected non-ordered patterns. When other non-ordered patterns already have many antichains to cover node n, the effect of the node frequency in the next non-ordered pattern becomes less. ε is a constant value to avoid that 0 is used as the divisor. The size of a non-ordered pattern $|\bar{p}_j|$ means the number of colors in non-ordered pattern $\bar{p}_j$. α is a parameter. By $\alpha \times |\bar{p}_j|^2$, larger non-ordered patterns are given higher priority than smaller ones. We will see the reason in the following example. In our system $\varepsilon = 0.5$ and $\alpha = 20$.

Let us use the example in Figure 6.1 to demonstrate the above mentioned algorithm. The node frequencies are given in Table 6.7.

At the very beginning, there is no selected non-ordered pattern, i.e., $\mathbf{P}_s =$

	a1	a2	a3	b4	b5
$\bar{p}_1 = \{\text{a}\}$	1	1	1	0	0
$\bar{p}_2 = \{\text{b}\}$	0	0	0	1	1
$\bar{p}_3 = \{\text{aa}\}$	1	1	2	0	0
$\bar{p}_4 = \{\text{bb}\}$	0	0	0	1	1

Table 6.7: Node frequencies $h(\bar{p}, n)$

ϕ. $\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, n)$ is therefore always zero. The priorities are:

$$\begin{aligned}
f(\bar{p}_1) &= \frac{1}{\varepsilon} + \frac{1}{\varepsilon} + \frac{1}{\varepsilon} + 0 + 0 + 20 \times 1^2 = 26; \\
f(\bar{p}_2) &= 0 + 0 + 0 + \frac{1}{\varepsilon} + \frac{1}{\varepsilon} + 20 \times 1^2 = 24; \\
f(\bar{p}_3) &= \frac{1}{\varepsilon} + \frac{1}{\varepsilon} + \frac{2}{\varepsilon} + 0 + 0 + 20 \times 2^2 = 88; \\
f(\bar{p}_4) &= 0 + 0 + 0 + \frac{1}{\varepsilon} + \frac{1}{\varepsilon} + 20 \times 2^2 = 84;
\end{aligned}$$

Obviously $\bar{p}_3$ is the first selected non-ordered pattern. Correspondingly $\bar{p}_1$ is deleted because it is a subpattern of $\bar{p}_3$. For choosing the second non-ordered pattern, we have

$$\begin{aligned}
\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, \text{a1}) = 1; \\
\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, \text{a2}) = 1; \\
\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, \text{a3}) = 2; \\
\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, \text{b4}) = 0; \\
\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, \text{b5}) = 0.
\end{aligned}$$

The priorities become:

$$f(\bar{p}_2) = 0+0+0+\frac{1}{\varepsilon}+\frac{1}{\varepsilon}+20\times 1^2 = 24;$$
$$f(\bar{p}_4) = 0+0+0+\frac{1}{\varepsilon}+\frac{1}{\varepsilon}+20\times 2^2 = 84;$$

The priority functions for $\bar{p}_2$ and $\bar{p}_4$ keep the old value. The reason is that non-ordered pattern $\bar{p}_3$ has antichains that cover nodes "a1", "a2" and "a3", while $\bar{p}_2$ and $\bar{p}_4$ only relate to "b4" and "b5". If there were another non-ordered pattern which covered node "a1", "a2" or "a3", the value of its priority function would go down because of the increase of $h(\bar{p}_i, \text{a1})$, $h(\bar{p}_i, \text{a2})$ and $h(\bar{p}_i, \text{a3})$. Of course $\bar{p}_4$ is chosen as the second non-ordered pattern. If $\alpha \times |\bar{p}_j|^2$ is not part of the priority function, both $f(\bar{p}_2)$ and $f(\bar{p}_4)$ will be 4, i.e., there is no preference to make a choice among these two. A random one will be taken. However, we can easily see that $\bar{p}_4$ is better than $\bar{p}_2$ in that $\bar{p}_4$ allows "b4" and "b5" to run in parallel.

Now a problem arises: How about $P_{def} = 1$ in the above example? That means only one non-ordered pattern is allowed. Of course we have to use the non-ordered pattern $\bar{p} = \{\text{ab}\}$ to be able to include all colors. Unfortunately there is no antichain with color set {a, b}, therefore non-ordered pattern {ab} is not even a candidate! To solve this problem, the column number condition is used in the priority function, which will be explained below. The priority function is modified as follows:

$$f(\bar{p}_j) = \begin{cases} \displaystyle\sum_{n\in N_G} \frac{h(\bar{p}_j, n)}{\displaystyle\sum_{\bar{p}_i \in \mathbf{P}_s} h(\bar{p}_i, n) + \varepsilon} + \alpha \times |\bar{p}_j|^2 & \text{if } \bar{p} \text{ satisfies the color number condition;} \\ 0 & \text{otherwise.} \end{cases}$$

Let the complete color set $\mathcal{L}$ represent all the colors that appear in the DFG,

$$\mathcal{L} = \{l(n)| \text{ for all } n \text{ in DFG}\},$$

and let the selected color set $\mathcal{L}_s$ represent all the colors that appear in one of already selected non-ordered patterns, i.e.,

$$\mathcal{L}_s = \{l | l \in \bar{p}_j \text{ for } \bar{p}_j \in \mathbf{P}_s\}.$$

The new color set $\mathcal{L}_n(\bar{p})$ of the candidate non-ordered pattern $\bar{p}$ consists of the colors that exist in $\bar{p}$ but not in the selected color set $\mathcal{L}_s$, i.e.,

$$\mathcal{L}_n(\bar{p}) = \{l | l \in \bar{p} \text{ and } l \notin \mathcal{L}_s\}.$$

We say that a candidate non-ordered pattern satisfies the color pattern condition if the inequality (6.15) holds.

$$|\mathcal{L}_n(\bar{p})| \geq |\mathcal{L}| - |\mathcal{L}_s| - C \times (P_{def} - |\mathbf{P}_s| - 1). \tag{6.15}$$

$|\mathcal{L}| - |\mathcal{L}_s|$ is the number of colors that have not been covered by the $|\mathcal{L}_s|$ non-ordered patterns. Except for the non-ordered pattern that is going to be selected, there are another $(P_{def} - |\mathbf{P}_s| - 1)$ to be selected later, which can cover at most $C \times (P_{def} - |\mathbf{P}_s| - 1)$ uncovered different colors. Therefore the right part of the inequality is the minimum number of new colors that should be covered by the candidate non-ordered pattern.

If we use a candidate pattern $\bar{p}$ which does not satisfy the inequality (6.15), some colors will not appear in the final chosen P_{def} non-ordered patterns. For example, after selecting $(P_{def} - 1)$ non-ordered patterns, there are still $(C + 2)$ colors that have never appeared in the selected $(P_{def} - 1)$ non-ordered patterns. We can put at most C colors in the last non-ordered pattern. Therefore the last two colors cannot appear in the non-ordered patterns. To avoid this, when the inequality (6.15) is not satisfied for non-ordered pattern $\bar{p}$, we do not select $\bar{p}$ by setting its priority function $f(\bar{p})$ to zero. If the priority function for all candidate non-ordered patterns are zero, we have to make a non-ordered pattern using C colors that have not appeared in the selected color set $\mathcal{L}_s$. The selection algorithm is modified and shown in Figure 6.10.

```
1  for(i = 0; i < P_def; i++) {
2      Compute the priority function for each non-ordered
       pattern.
3      Choose the non-ordered pattern with the largest
       nonzero priority function. If there is no non-ordered
       pattern with nonzero priority function, take C uncov-
       ered colors to make a non-ordered pattern.
4      Delete the subpatterns of the selected non-ordered
       pattern.
5  }
```

Figure 6.10: The pseudo-code for the modified pattern selection procedure

Now let us do the example given in Figure 6.1 again, assuming that only one non-ordered pattern is allowed. In the inequality (6.15), $\mathcal{L} = \{\mathrm{a}, \mathrm{b}\}$,

$\mathcal{L}_s = \Phi$, $P_{def} = 1$ and $\mathbf{P}_s = \phi$. The right side of the inequality is therefore 2. All the non-ordered patterns generated from the graph have only one color. The new color sets for the four non-ordered patterns are: $\mathcal{L}_n(\bar{p}_1) = \{a\}$, $\mathcal{L}_n(\bar{p}_2) = \{b\}$, $\mathcal{L}_n(\bar{p}_3) = \{a\}$, $\mathcal{L}_n(\bar{p}_4) = \{b\}$. Thus, $|\mathcal{L}_n(\bar{p}_1)| = |\mathcal{L}_n(\bar{p}_2)| = |\mathcal{L}_n(\bar{p}_3)| = |\mathcal{L}_n(\bar{p}_4)| = 1$. The inequality does not hold for any of them. Due to the presented modification a new non-ordered pattern {ab} is made.

6.4.3 Experiment

We applied the presented pattern selection method on the 3- and 5- point FFT algorithms. The result is shown in Table 6.8. In the experiment, we always used the priority function given by Equation (6.13) for the multi-pattern list scheduling.

	3FFT		5FFT	
P_{def}	Patterns	#Clock Cycles	Patterns	#Clock Cycles
1	{aabcc}	8	{aaabc}	19
2	{aabcc}{aaaac}	7	{aaabc}{aaacc}	16
3	{aabcc}{aaaac}{aaabc}	7	{aaabc}{aaacc}{aaaab}	16
4	{aabcc}{aaaac}{aaabc} {aaacc}	7	{aaabc}{aaacc}{aaaab} {aabcc}	15
5	{aabcc}{aaaac}{aaabc} {aaacc}{aabbc}	6	{aaabc}{aaacc}{aaaab} {aabcc}{aaaac}	15

Table 6.8: Experimental result of the pattern selection algorithm. Here $C = 5$

6.4.4 Discussions

We can make the following observations regarding our simulation results, which are worthwhile to be investigated in the future:

- The performance of the selection algorithm can be improved by taking the information of the value of spans of antichains into consideration. The antichains with smaller spans are more likely to be used. Therefore, we might obtain better performance by modifying the definition of node frequency $h(\bar{p}, n)$ such that the instances with n with smaller spans

contribute more to the value of $h(\bar{p}, n)$ than the instances with larger spans.

- According to our current algorithm, when there is no candidate non-ordered pattern which satisfies the inequality (6.15), the new non-ordered pattern is made randomly. However, if the new non-ordered pattern is made based on the information of smaller non-ordered patterns, better performance might be achieved. For instance, suppose that "a", "b", "c", "d" and "e" are five colors that have not be covered by one of the selected patterns, if we find that {abcd} and {bcde} are two non-ordered patterns with large priority functions, then we can make a non-ordered pattern {abcde}. Of course, how to use the information of smaller non-ordered patterns is another hard topic.

6.5 A column arrangement algorithm

In Section 6.4, we presented an algorithm to select non-ordered patterns from a colored graph. In this section, the elements of each selected non-ordered pattern are allocated to columns.

Given a non-ordered pattern $\vec{p}_i = \{p_{i1}, p_{i2}, \cdots, p_{iC}\}$, one order of its elements is written as $Ord(\vec{p}_i) = \{p_{io(1)}, p_{io(2)}, \cdots, p_{io(C)}\}$. The column arrangement algorithm is to order elements of selected non-ordered patterns to get a table $\mathbf{P}_{ou}$ such that the column condition

$$\max_{1 \le i \le C} |u_i(\mathbf{P}_{ou})| \le U_{def}$$

is satisfied.

In a real application, the whole application might be scheduled part by part. If we use fewer configurations in one part, we might have more freedom in other parts. Therefore, in the column arrangement phase, we not only aim at satisfying the constraints, i.e., to minimize the function defined in Equation (6.16),

$$f_{max} = \max_{1 \le i \le C} |u_i(\mathbf{P}_{ou})|, \tag{6.16}$$

but also aim at minimizing the sum of the size of the column color sets, i.e., to minimize the function defined in Equation (6.17),

$$f_{sum} = \sum_{1 \le i \le C} |u_i(\mathbf{P}_{ou})|. \tag{6.17}$$

6.5.1 Lower bound

The *number of concurrences* of a color l in a pattern $\vec{p}_i$ (or a non-ordered pattern $\bar{p}_i$) is denoted by $Con(l, \vec{p}_i)$ (or $Con(l, \bar{p}_i)$), $l \in \mathcal{L}$, $1 \leq i \leq R$. The *maximum concurrence number of a color l in a pattern matrix* $\mathbf{P}$ is defined as

$$Con_{max}(l, \mathbf{P}) = \max_{1 \leq i \leq R} Con(l, \vec{p}_i), \quad l \in \mathcal{L}. \tag{6.18}$$

For instance, the maximum concurrence number of "f" in Table 6.10 (a) is 2 since pattern 4 has two "f"s, the maximum concurrence number of "b" is one.

Theorem 6.5 *For a pattern matrix* $\mathbf{P}$*, the sum of the number of column colors in each column of* $\mathbf{P}$ *cannot be smaller than the sum of the maximum concurrence number for all colors, i.e.,*

$$f_{sum} \geq \sum_{l \in \mathcal{L}} Con_{max}(l, \mathbf{P}). \tag{6.19}$$

The maximum size of the column color sets is thus satisfying

$$f_{max} \geq \lceil \frac{1}{C} \sum_{l \in \mathcal{L}} Con_{max}(l, \mathbf{P}) \rceil. \tag{6.20}$$

Proof If Equation (6.19) is not satisfied, there must be a color l which appears a times in the C column color sets and its maximum concurrence number satisfies $Con_{max}(l, \mathbf{P}) > a$. According to the definition of the maximum concurrence number given by Equation (6.18), there must be a pattern $\vec{p}_i$ with $Con_{max}(l, \mathbf{P})$ times color l. In other words, there must be a row in which l appears $Con_{max}(l, \mathbf{P})$ times in the color table $\mathbf{L}$ defined by Equation (6.2). This conflicts with the conclusion $Con_{max}(l, \mathbf{P}) > a$ given above.

Equation (6.20) can directly be obtained from Equation (6.19). ∎

6.5.2 Algorithm description

The column arrangement algorithm is given in Figure 6.11. The algorithm moves patterns from the input non-ordered pattern set $\mathbf{P}_{in}$ to the output pattern table $\mathbf{P}_{ou}$ one by one according to a cost function. The key point is the cost function, which determines which pattern and in which ordering is going to be put to the output table at the next step.

//	The input non-ordered pattern set is $\mathbf{P}_{in}$, and the output table is $\mathbf{P}_{ou}$;
1.	Take one pattern from $\mathbf{P}_{in}$ as the starting pattern and put it to $\mathbf{P}_{ou}$ without taking care of the ordering of the pattern; (We will explain how to select the pattern later.)
2.	For each ordering of each remaining pattern in $\mathbf{P}_{in}$, compute the cost function.
3.	Choose the pattern and the ordering with the lowest cost and put it to $\mathbf{P}_{ou}$.
4.	Repeat step 2 and step 3 until $\mathbf{P}_{in}$ is empty.

Figure 6.11: A Columns Arrangement Algorithm

Cost function To build the cost function used in step 2 and step 3 of Figure 6.11, the following aspects have to be considered:

- It is preferable to put a color to the column which already has the same color because we do not have to add a new color.

(a)$\mathbf{P}_{in}$ before patterns 2 and 3 are ordered

pattern	1	2	3	4	5
2	a	b	c	f	g
3	d	f	*	*	*

(b)$\mathbf{P}_{ou}$ before patterns 2 and 3 are ordered

pattern	1	2	3	4	5
1	a	b	c	d	e

(c)$\mathbf{P}_{ou}$ if pattern 2 is ordered as {abcgf}

pattern	1	2	3	4	5
1	a	b	c	d	e
2	a	b	c	g	f
3	*	*	*	d	f
$\|u_i\|$	1	1	1	2	2

(d)$\mathbf{P}_{ou}$ if pattern 2 is ordered as {abcfg}

pattern	1	2	3	4	5
1	a	b	c	d	e
2	a	b	c	f	g
3	*	*	f	d	*
$\|u_i\|$	1	1	2	2	2

Table 6.9: Example

For example, suppose Table 6.9 (a) is the current non-ordered pattern set and Table 6.9 (b) is the current output pattern table. For ordering pattern 2, it is preferable to allocate "a" to column 1, "b" to column 2 and "c" to column 3 because these three colors already exist in $u_1(\mathbf{P}_{ou})$, $u_2(\mathbf{P}_{ou})$ and $u_3(\mathbf{P}_{ou})$ respectively (see Table 6.9(b)).

- To allocate a color l_1 to column i, for each color l_2 in column i, i.e., $l_2 \in u_i(\mathbf{P}_{ou})$, we should first check whether l_1 and l_2 exist together in other non-ordered patterns. If yes, we should better not allocate l_1

in column i. Otherwise, one of them has to appear at least twice in the output configurable sets. An exception is that $Con(l_2, \mathbf{P}_{in}) > 1$ or $Con(l_2, \mathbf{P}_{ou}) > 1$ because in this case a color l has to appear in at least $Con(l, \mathbf{P}_{in})$ configurable sets.

In the example shown in Table 6.9, order {abcgf} for pattern 2 is better than order {abcfg} because after order {abcgf} is selected for pattern 2, pattern 3 can be ordered as {***df} (see Table 6.9(c)). Therefore we have $|u_1(\mathbf{P}_{ou})| = 1$, $|u_2(\mathbf{P}_{ou})| = 1$, $|u_3(\mathbf{P}_{ou})| = 1$, $|u_4(\mathbf{P}_{ou})| = 2$, $|u_5(\mathbf{P}_{ou})| = 2$ and $f_{sum} = 7$. However, if we select order {abcfg} for pattern 2, for pattern 3 we have to allocate "d" or "f" to a column other than column 4 (see Table 6.9(d)). As a result, $f_{sum} = 8$.

- The patterns with more dummies have more freedom to arrange, so they are given lower priority.

- For a color l, the non-ordered pattern $\bar{p}_i$ is given higher priority to arrange when $Con_{max}(l, \mathbf{P}_{in}) = Con(l, \bar{p}_i)$. Then other patterns which have fewer l's can be arranged according to the l's already arranged.

(a) $\mathbf{P}_{in}$ before patterns 3 and 4 are ordered

pattern	1	2	3	4	5
3	a	b	c	e	f
4	a	b	c	f	f

(b) $\mathbf{P}_{ou}$ before patterns 3 and 4 are ordered

pattern	1	2	3	4	5
1	a	b	c	d	e
2	a	b	c	d	f

(c) $\mathbf{P}_{ou}$ after first ordering pattern 3 then ordering pattern 4

pattern	1	2	3	4	5
1	a	b	c	d	e
2	a	b	c	d	f
3	a	b	c	e	f
4	a	b	c	f	f
$\|u_i\|$	1	1	1	3	2

(d) $\mathbf{P}_{ou}$ after first ordering pattern 4 then ordering pattern 3

pattern	1	2	3	4	5
1	a	b	c	d	e
2	a	b	c	d	f
4	a	b	c	f	f
3	a	b	c	f	e
$\|u_i\|$	1	1	1	2	2

Table 6.10: Example

Look at the example shown in Table 6.10. Suppose Table 6.10 (a) is the current non-ordered pattern set and Table 6.10 (b) is the output pattern table. The maximum concurrence of color "f" is 2, which happens in pattern 4. If pattern 3 is arranged first, we might select the order {abcef} for pattern 3. Finally there are three colors in column 3 (see

6.10(c)). However, if we first arrange pattern 4, order {abcff} will be used. As a result, column 3 has only 2 different colors (see 6.10(d)).

Now we define the conflict function. The intuitive notion of this cost function is based on the above aspects, which will be explained later.

The *conflict factor* between two colors l and y:

$$Conflict(l_i, l_j) = \begin{cases} 2000 & \text{if } l_i \text{ and } l_j \text{ appear in one pattern and} \\ & Con(l_i, \mathbf{P}_{in}) = 1, Con(l_j, \mathbf{P}_{in}) = 1; \\ 200 & \text{if } l_i \text{ and } l_j \text{ appear in one pattern and} \\ & Con(l_i, \mathbf{P}_{in}) \geq 2 \text{ or } Con(l_j, \mathbf{P}_{in}) \geq 2; \\ 0 & \text{otherwise.} \end{cases} \quad (6.21)$$

Remark: The actual values of the numbers are only relevant in relative to each other. The important point is that the cost function makes it possible to choose the best order for each pattern.

The *benefit* of assigning one color l to a column i is defined as

$$Benefit(l, i) = \begin{cases} 2000 & \text{if } l \in u_i(\mathbf{P}_{ou}) \\ 0 & \text{if } l \notin u_i(\mathbf{P}_{ou}) \end{cases}$$

The *size factor* of assigning an element to a column i is:

$$SizeFactor(l, i) = \begin{cases} |u_i(\mathbf{P}_{ou}) + 1|^2 & \text{if } l \notin u_i(\mathbf{P}_{ou}); \\ 0 & \text{if } l \in u_i(\mathbf{P}_{ou}). \end{cases} \quad (6.22)$$

The *cost function of assigning one color l to a column i of a table* $\mathbf{P}_{ou}$ is defined as:

$$Cost(l, i) = -Benefit(l, i) + \sum_{y \in u_i(\mathbf{P}_{ou})} Conflict(l, y) + SizeFactor(l, i).$$

For a color l, the pattern $\vec{p}_i$ is given higher priority to be selected when $Con_{max}(l, \mathbf{P}_{ou}) = Con(l, \vec{p}_i)$. Then other patterns which have l's can arrange themselves according to the existing l's. The concurrency factor of pattern $\vec{p}_i$ is defined as

$$Concurrency(\vec{p}_i) = \begin{cases} (Con_{max}(l, \mathbf{P}_{ou}))^2 \times 500 & \text{if there is a color } l \text{ with} \\ & Con(l, \vec{p}_i) = Con_{max}(l, \mathbf{P}_{ou}) \\ & \text{and } Con_{max}(l, \mathbf{P}_{ou}) > 1 \\ 0 & otherwise \end{cases}$$

The cost of assigning one pattern $\vec{p_i}$ to the output pattern table $\mathbf{P}_{ou}$ in the ordering

$$Ord(\vec{p_i}) = \{p_{i,o(1)}, p_{i,o(2)}, \cdots, p_{i,o(C)}\}$$

is:

$$\begin{aligned} &Cost(Ord(\vec{p_i}) = \{p_{i,o(1)}, p_{i,o(2)}, \cdots, p_{i,o(C)}\}) \\ &= \sum_{1 \le j \le C} Cost(p_{i,o(j)}, j) + Concurrency(\vec{p_i}) \\ &\quad + 200 \times \text{ Number of dummies in } \vec{p_i}. \end{aligned} \tag{6.23}$$

When a pattern has many colors that do not exist in the output pattern table, it is more likely to make a bad decision when this pattern is arranged earlier since the output pattern table has no information on the new colors. The philosophy in choosing the constants in the functions is to let the newly arranged pattern bring in as little as possible information at each iteration. The colors enter the output table as slowly as possible.

The ordering of the starting pattern will not decrease the performance of the final result. However, the choice of the starting pattern will significantly influence the speed in which all colors enter the output table. In the experiments, we also found that the experimental result is very sensitive to the selection of the starting pattern, especially the value of f_{max}. For the same set of patterns given by Table 6.11, if we use the pattern 1 as the first pattern, the result will be as shown in Table 6.13. $f_{sum} = 14$ and $f_{max} = 3$. If we use pattern 7 as the starting pattern, the result will be $f_{sum} = 14$ and $f_{max} = 5$. Observing that the number of patterns is not very large in reality (at most 32 for the Montium), instead of designing an algorithm to find the best starting pattern, we just try all of them, and then take the best one.

Theoretically, the two optimality critera given by Equations (6.16) and (6.17) may conflict in some cases. The design of the above cost function gives attention to balancing the size of column color sets among columns as well as decreasing the total number of all column color sets. For instance, according to the size factor in Equation (6.22), the cost of putting a new color to a column with more colors is larger than that of putting it to a column with fewer colors.

Now we use the example in Tables 6.11, 6.12 and 6.13 to demonstrate the column arrangement algorithm. Here we assume $C = 5$ which is the value in the Montium architecture. The non-ordered pattern set is shown in

pattern	1	2	3	4	5
1	a	a	b	c	d
2	h	i	g	g	f
3	a	f	d	h	*
4	d	i	g	*	*
5	d	b	c	a	e
6	f	g	k	i	l
7	a	k	l	*	*
8	c	f	i	j	d

Table 6.11: Non-ordered pattern set

Table 6.11, where the letters represent colors. We use pattern 1 as the starting pattern. The five column color sets are now: $u_1(\mathbf{P}_{ou}) = \{\text{a}\}, u_2(\mathbf{P}_{ou}) = \{\text{a}\}, u_3(\mathbf{P}_{ou}) = \{\text{b}\}, u_4(\mathbf{P}_{ou}) = \{\text{c}\}, u_5(\mathbf{P}_{ou}) = \{\text{d}\}$. When pattern 5 is ordered as {a,e,b,c,d}, column 1, 3, 4 and 5 will have benefit 2000. The cost of putting "e" to column 2 is:

$$\begin{aligned} Cost(\text{e}, 2) &= Conflict(\text{a}, \text{e}) + SizeFactor(\text{e}, 2) \\ &= 200 + 2^2 = 204. \end{aligned} \tag{6.24}$$

The total cost will be $-8000 + 204 = -7796$. This ordering of the pattern is chosen because all other orderings and other patterns will have higher cost. The column color sets are changed to $u_1(\mathbf{P}_{ou}) = \{\text{a}\}, u_2(\mathbf{P}_{ou}) = \{\text{a,e}\}, u_3(\mathbf{P}_{ou}) = \{\text{b}\}, u_4(\mathbf{P}_{ou}) = \{\text{c}\}, u_5(\mathbf{P}_{ou}) = \{\text{d}\}$. For pattern 8, obviously "c" and "d" will be put in column 4 and column 5 separately. From Table 6.12 we see that $Conflict(\text{a,f}) = 200$, $Conflict(\text{e,f}) = 0$, $Conflict(\text{b,f}) = 0$. Thus it is better to put "f" in column 3. "i" and "j" can be put in any order.

6.5.3 Computational complexity

Since one pattern has $C!$ possible orders, if there are k patterns to arrange, step 2 and step 3 in Figure 6.11 have to choose the best one from $k \times C!$ candidates. If there are R non-ordered patterns, the number of candidates for arranging the second one is $((R-1)\times C!)$, the number for the third one will be $((R-2)\times C!), \cdots$. Totally, the cost function will be computed $R(R-1)/2\times C!$ times. Considering that the program given in Figure 6.11 will be run R times since each pattern can act as a first pattern, the computational complexity of the column arrangement algorithm will be $O(R^2(R-1)/2 \times C!)$. For the

pattern	a	b	c	d	e	f	g	h	i	j	k	l
a	0	200	200	200	200	200	0	200	0	0	200	200
b	200	0	2000	2000	2000	0	0	0	0	0	0	0
c	200	2000	0	2000	2000	2000	0	0	2000	2000	0	0
d	200	2000	2000	0	2000	2000	200	2000	2000	2000	0	0
e	200	2000	2000	2000	0	0	0	0	0	0	0	0
f	200	0	2000	2000	0	0	200	2000	2000	2000	2000	2000
g	0	0	0	200	0	200	0	200	200	0	200	200
h	200	0	0	2000	0	2000	200	0	2000	0	0	0
i	0	0	2000	2000	0	2000	200	2000	0	2000	2000	2000
j	0	0	2000	2000	0	2000	0	0	2000	0	0	0
k	200	0	0	0	0	2000	200	0	2000	0	0	2000
l	200	0	0	0	0	2000	200	0	2000	0	2000	0

Table 6.12: Conflict factor table

step	pattern	1	2	3	4	5
1	1	a	a	b	c	d
2	5	a	e	b	c	d
3	8	j	i	f	c	d
4	3	a	*	f	h	d
5	2	g	i	f	h	g
6	6	g	i	f	k	l
7	4	g	i	*	*	d
8	7	a	*	*	k	l
	$\|u_i\|$	3	3	2	3	3

Table 6.13: Output pattern table: $f_{sum} = 14, f_{max} = 3$

Montium application, $R = 32$ and $C = 5$, the algorithm can be finished within a couple of seconds on a general Pentium 4 computer.

6.5.4 Experiment

Table 6.14 shows the experimental results for the column arrangement algorithm. The input matrices are randomly generated. We can see that the algorithm works very well for minimizing the f_{sum} which is very close to its lower bound. The last column of the table indicates whether the pattern which reaches f_{max} also reaches f_{sum}. In our experiments, the answer is almost always "yes". The lower bound 1 of f_{max} in the table is defined by Equation (6.20). The lower bound 2 is computed by $\lceil f_{sum}/\mathrm{C} \rceil$. The lower bound 2 indicates how good the algorithm balances the size of all column color sets. In the experiment, f_{max} mostly reaches the lower bound 2.

Number of patterns	Number of colors	Lower bound of f_{sum}	f_{sum}	Lower bound1 of f_{max}	Lower bound2 of f_{max}	f_{max}	Simultaneously?
10	10	15	16	3	4	4	Yes
10	10	14	16	3	4	4	Yes
10	9	19	19	4	4	4	Yes
10	10	14	15	3	3	3	Yes
10	9	15	15	3	3	3	Yes
10	8	14	14	3	3	4	No
10	8	14	14	3	3	4	Yes
10	6	13	13	3	3	3	Yes
10	6	12	12	3	3	3	Yes
10	12	18	18	4	4	4	Yes
20	20	29	30	6	6	7	Yes
20	20	29	33	6	7	8	Yes
20	25	31	36	7	8	8	Yes
20	23	27	29	5	6	7	Yes
32	10	22	24	5	5	5	Yes

Table 6.14: Experimental result

6.6 Using the scheduling algorithm on CDFGs

To simplify the scheduling problem, CDFGs are scheduled block by block. A basic block is scheduled to a *virtual clock block*, donoted by VirtualClock(block) (see Figure 6.12). For example, "Block1" is scheduled to "VirtualClock1" in Figure 6.12. The virtual clock block includes some normal clock cycles, which are obtained by scheduling the DFG part of a basic block, as

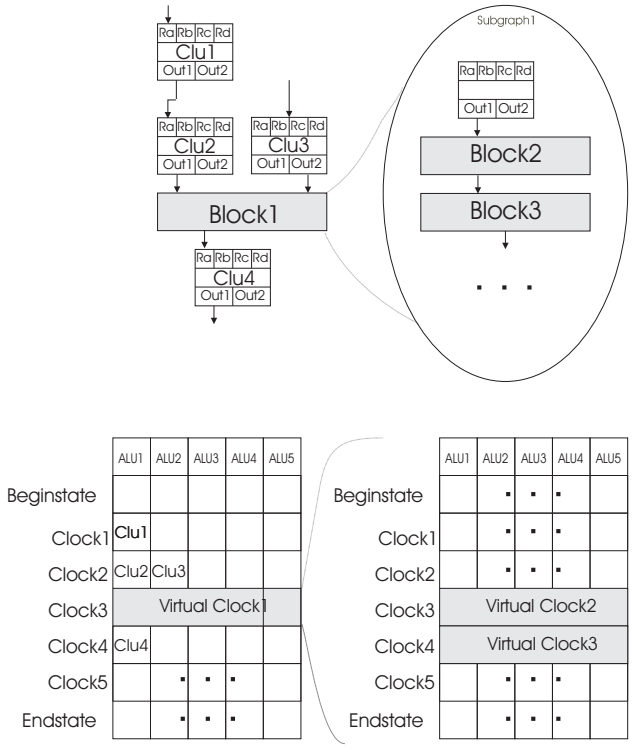

Figure 6.12: The structure of scheduling result

described in this chapter. Furthermore, we define two states for each virtual clock block: a *BeginState* and an *EndState*. A virtual clock block starts with a BeginState and ends with an EndState. No clusters are scheduled in the BeginState and the EndState. BeginState records the state of the Montium tile, including the state of all memories and all registers, before the actions within the block are executed. EndState records the state of the Montium tile after the actions within the block are executed. The BeginState and the EndState act as an interface between the schedule inside a basic block and the schedule outside a block. When scheduling the higher level of a basic block, we only need the information of its BeginState and EndState.

6.7 Related work

Scheduling is a well defined and studied problem in the research area of high-level synthesis. Scheduling is an optimization problem, and is specified in several ways depending on the constraints: (1) *Unconstrained scheduling* finds a feasible schedule that obeys the precedence constraints on the graph. (2) *Time-constrained scheduling* minimizes the number of required resources when the number of clock cycles is fixed. (3) *Resource-constrained scheduling* minimizes the number of clock cycles when the number of resources is given.

Most scheduling problems are NP-complete problems [8]. To solve the scheduling problems, exact algorithms, which find optimal solutions, or heuristic algorithms, which find feasible (possibly suboptimal) solutions have been used. Four commonly used scheduling algorithms are:

- integer linear programming
- As-Soon-As-Possible (ASAP) or As-Late-As-Possible (ASAP) scheduling
- list scheduling
- force-directed scheduling

The Integer Linear Programming is an exact algorithm which guarantees to find the globally optimal solution [36] [67]. An important disadvantage of ILP is the high complexity. These problems are known to be NP-complete in general, and so the existence of a polynomial time algorithm for solving these problem is highly unlikely. As a result, computation times associated to ILP usually become too large for cases that are slightly more complex than trivial [47] [62].

The ASAP algorithm schedules each node on the earliest possible clock cycle. The ALAP scheduling is similar, but schedules each node to the latest possible clock cycle. ASAP and ALAP algorithms are used for solving the unconstrained scheduling problem [89].

The resource-constrained scheduling problem is commonly solved by list scheduling. List scheduling is a generalization of the ASAP algorithm with the inclusion of resource constraints invented 30 years ago [45]. A list algorithm maintains a list of candidate nodes whose predecessors have already been scheduled [2] [70] [87]. In a dynamic list scheduling algorithm, the list changes every clock cycle. In a static list scheduling [48], a single large list is

constructed statically only once before starting scheduling. The complexity is decreased by fixing the candidate list.

The force-directed scheduling was developed as part of the Carleton University's HAL system [72] [73]. It is a common choice for solving the time-constrained scheduling problem.

The scheduling problem for our target architecture is basically a resource-constrained scheduling problem with extra constraints. Therefore, a modified list based scheduling algorithm, a multi-pattern scheduling algorithm, is used.

6.8 Conclusions

The problem of scheduling clusters on a coarse-grained reconfigurable architecture is defined as a color-constrained scheduling problem. In this chapter, the color-constrained scheduling problem is discussed. Our solution consists of three algorithms: pattern selection, column arrangement and multi-pattern scheduling. The pattern selection algorithm selects a set of non-ordered patterns from the input graph; the column arrangement algorithm minimizes the number of configurations for each ALU; and the multi-pattern scheduling algorithm schedules the graph using the selected pattern. The algorithms are heuristic algorithms based on priority functions. The algorithms are verified by simulations. The performance can be further improved by refining the priority functions, which will be our future work.

Chapter 7

Resource allocation

In the allocation phase[1] *variables are allocated to memories or registers, and data moves are scheduled. This phase is the final step before code generation and uses the details of the target architecture.*

7.1 Introduction

After the scheduling phase, each cluster is assigned to an ALU and the relative executing order of clusters has been determined. The number of clock cycles used to run the input application cannot be smaller than the length of the schedule table. The allocation phase takes care of the inputs and outputs of the clusters. Two major tasks performed in the allocation phase are: allocation of variables and arrangement of data moves. Moving data is done for preparing inputs for and saving outputs of clusters. To do so some extra clock cycles might be added to the schedule table due to the limitations of

[1]Parts of this chapter have been published in publication [9].

resources (such as global bus restrictions, reading or writing port limitations of a memory). Our objective is to decrease the number of extra clock cycles for these data moves. Therefore, data moves should be scheduled properly. Furthermore, allocation of variables also affects the scheduling of data moves, which part is also considered at the allocation phase.

The hardware constraints given by the Montium that should be considered in the allocation phase are:

1. The size of register banks;
2. The size and number of memories;
3. The number of reading and writing ports of each register bank;
4. The number of reading and writing ports of each memory;
5. The number of buses of the crossbar;
6. The number of register configurations;
7. The number of AGU configurations;
8. The number of crossbar configurations.

The rest of this chapter is organized as follows: In Section 7.2 some definitions used in this chapter are given; In Section 7.3, the method for allocation of variables is presented; The procedure of scheduling data moves is described in Section 7.4; In Section 7.5.1 the crossbar allocation problem is modeled; In Section 7.5.2 the register arrangement algorithm is modeled; and Section 7.5.3 the memory arrangement problem is modeled; In Section 7.6, an overview of the related work for resource allocation is given; Section 7.7 concludes this chapter.

7.2 Definitions

First some symbols used in this chapter are listed:

- M1, M2, $\cdots$, M10: Ten memories of a Montium tile.
- M: One of the ten memories of a Montium tile.

- M_r: Read port of memory M. M can be replaced by one specific memory, e.g., $M1_r$ is the read port of memory M1.
- M_w: Write port of memory M. M can be replaced by one specific memory.
- PP1, PP2, PP3, PP4, PP5: Five processing parts of a Montium tile.
- PP: One of the five processing parts of a Montium tile.
- PP.Ra, PP.Rb, PP.Rc, PP.Rd: Four register banks of a processing part PP. PP can be replaced by one specific processing part.
- Ra1, Ra2, Ra3, Ra4: Four registers of register bank Ra. Ra can be replaced by other register banks, e.g., PP1.Rb2 denotes register 2 of register bank Rb of processing part 1.
- PP.Out1, PP.Out2: Two outputs of the ALU of processing part PP. PP can be replaced by one specific processing part.
- ALU1, ALU2, ALU3, ALU4, ALU5: Five ALUs of a Montium tile.
- ALU: One of the five ALUs of a Montium tile.
- GB1, GB2, GB3, GB4, GB5, GB6, GB7, GB8, GB9, GB10: Ten global buses.
- LM1, LM2, LM3, LM4, LM5, LM6, LM7, LM8, LM9, LM10: Ten memory buses connected with the ten memories.
- LA11, LA12, LA21, LA22, LA31, LA32, LA41, LA42, LA51, LA52: ten ALU buses connected with ten ALU outputs.

Space-time live ranges Traditionally allocation algorithms work with distinct *live ranges* of variables in a program [11][19]. A single variable can have several distinct values that 'live' in different parts of the program - each of these values will become a separate live range. Clusters that refer to a variable must now refer to a particular 'live range' of that variable. The allocator discovers all the separate live ranges and allocates them to physical registers or memory location on the target machine.

Several clusters allocated to different physical ALUs might use one value at the same time and in our Montium architecture, an ALU can only use the data from its local registers. This situation asks for several copies of a same value stored in several distinct physical registers. We define *Space-Time Live Ranges* (STLRs) for each copy of a value. AN STLR represents a copy of this value from its producer to one of its consumers. On a clustered graph, an STLR always has only one producer and one consumer. The producer of an STLR is either a fetch (*fe*) or an output of a cluster. The consumer of an STLR is either a store (*st*), or an input of a cluster. Figure 7.2 shows all the possible combinations of producers and consumers. Several STLRs might share a common producer, while the consumers of two STLRs cannot be the same one. AN STLR is different from a live range. A live range traditionally extends from the producer to its last consumer while there exists an STLR for every consumer. On a CDFG, a live range corresponds to a hydra-edge which represents a data value, while an STLR corresponds to a producer and consumer pair. One hydra-edge might refer to several STLRs. For example, in Figure 7.1, STLR1 in Figure 7.1(b) and STLR2 in Figure 7.1(c) correspond to the same edge e shown in Figure 7.1(a). Each STLR has three attributes: a value, a producer and a consumer. Notice that not all connections on a CDFG represent an STLR. Only edges that represent data represent STLRs.

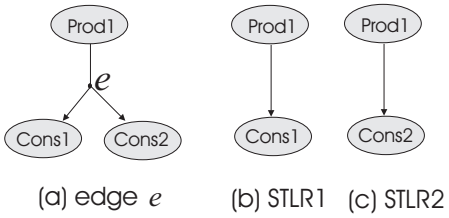

Figure 7.1: Two STLRs corresponding to one edge

Moves Moving a value from one place (called *source*) to another place (called *destination*) is called a *move*, m, which is written as (source $\rightarrow$ destination). The resource entity is either a reading port of a memory or an ALU output. The destination entity is either a register or a writing port of a memory.

Summarizing: a data move has three aspects:

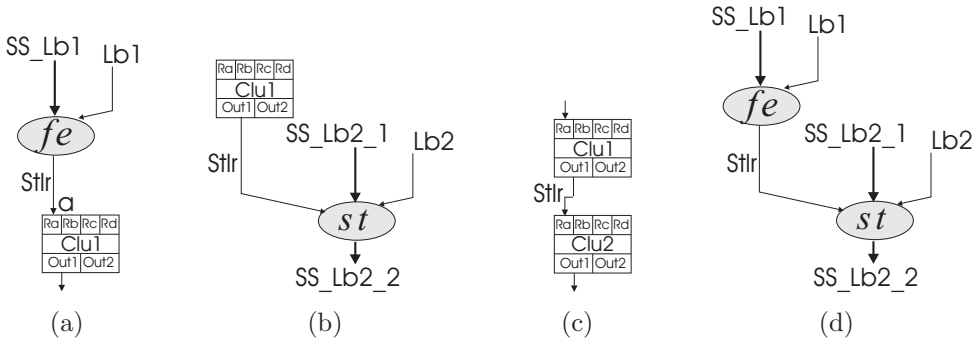

Figure 7.2: STLRs

- Source: PP.out or a read port of a memory;
- Destination: register or a write port of a memory;
- Route: a global or local bus.

As we have seen in Chapter 2, in total there are ten memories, ten ALU outputs and 20 registers in one Montium tile. All possible moves can be represented by a 20 by 30 table, as shown in Table 7.1.

<table>
<tr><th>Source \ Dest</th><th>PP1Ra</th><th>PP1Rb</th><th>PP1Rc</th><th>$\cdots$</th><th>PP5Rd</th><th>$M1_w$</th><th>$\cdots$</th><th>$M10_w$</th></tr>
<tr><td>PP1.Out1</td><td colspan="5" rowspan="4">consignment</td><td colspan="3" rowspan="4">upward move</td></tr>
<tr><td>PP1.Out2</td></tr>
<tr><td>$\vdots$</td></tr>
<tr><td>PP5.Out2</td></tr>
<tr><td>$M1_r$</td><td colspan="5" rowspan="3">downward move</td><td colspan="3" rowspan="3">shift</td></tr>
<tr><td>$\vdots$</td></tr>
<tr><td>$M10_r$</td></tr>
</table>

Table 7.1: Moves

The moves can be divided into four categories. (1) Loading of a copy of a value from a memory to a register is called a *downward move*; (2) Saving an output to a memory is called a *upward move*; (3) Saving an output of an ALU to a register is called a *consignment*; (4) Copying a value from one memory place to another memory place is called a *shift*.

From STLRs to moves Now we are going to define a data move for every STLR in a scheduled graph. We first assume that the value of a variable is kept in memory before a *fe* and saved to memory again after a *st*. With this assumption, an STLR is mapped to a move. According to producers and consumers of STLRs, we divide them into categories and treat them in different ways, which is shown in Table 7.2.

Type	Producer	Consumer	Move	Example
1	fetch(*fe*)	store(*st*)	shift	b=a; b is set to the value of a.
2	fetch(*fe*)	an input of a cluster	downward move	Clu1.Ra = a; Here a is used as the Ra input of Clu1.
3	an output port of a cluster	store(*st*)	upward move	b = Clu1.Out1; The value of Clu1.Out1 is saved to b.
4	an output port of a cluster	an input port of a cluster	consignment	Clu2.Ra = Clu1.Out1; The output of Clu1 is used an the Ra input of Clu2.

Table 7.2: From STLR to data moves: defining the source and destination of a move

Break a data move Every data move can be broken up into several consecutive moves. A reason could be that two values need to be written to the same memory in the same clock cycle. For example, a shift of from $M1_r$ to $M3_w$ ($M1_r \rightarrow M3_w$) can be changed into two moves: first ($M1_r \rightarrow M2_w$), then ($M2_r \rightarrow M3_w$). Breaking moves gives more flexibility to the compiler. However, it also adds more pressure to the crossbar and memory/register ports. To decrease the complexity of the compiler, we only break upward moves and consignments. The source entities of these two types of moves are ALU outputs. The results of ALUs have to be saved immediately. Otherwise the values will be lost. Therefore, when the required moves cannot be executed, an upward move is broken into another upward move followed by a shift; a consignment is broken into an upward move followed by a downward move. For example, (PP1.Out1 $\rightarrow$ PP1.Ra1) can be broken into (PP1.Out1 $\rightarrow$ $M2_w$) and ($M2_r \rightarrow$ PP1.Ra1).

7.3 Allocating variables

A variable or array could be allocated to one of the ten memories or to one of the twenty registers. In our mapping procedure, we first assume that external input values (except for streaming inputs) must be stored in memories at the configuration stage and when the program is finished, external output values must be saved in memories (except for streaming outputs). Actually in the current version of a Montium tile values are allowed to be put to registers directly at configuration phase and the computation result could be read from registers [44]. Therefore it is also possible to allocate a variable to a register. We assume that all variables are allocated in memories at the beginning of the allocation stage because: firstly the number of registers is very small so it is not wise to let a variable stay in a register too long; secondly the assumption is true in the older version of the Montium, therefore the work we have done for the old Montium can be used here; thirdly, at the optimization phase, we will check the possibility of allocating a variable directly to a register.

The following aspects must be taken into consideration when choosing a memory to allocate a variable or an array:

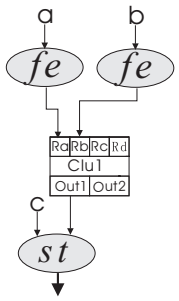

Figure 7.3: A small example

1. The memory occupancy should be balanced. Only one value can be read from or written to a memory within one clock cycle. Having variables distributed over all the ten memories will improve the parallel access to different variables. Without balancing the memory the execution speed might deteriorate due to the reading and writing limitation of the single-ported memory. Take the simple example in Figure 7.3, where cluster "Clu1" takes variable "a" and "b" as its inputs and

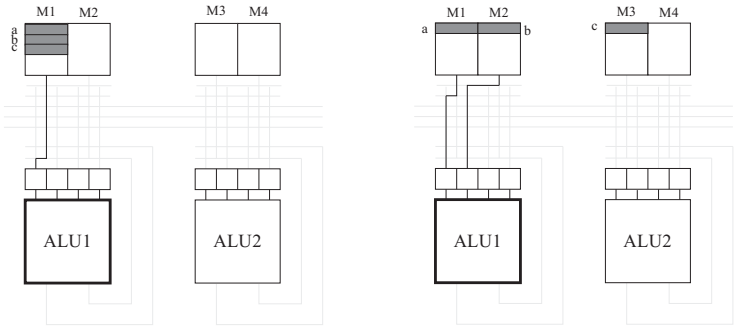

Figure 7.4: Having memories occupied evenly

computes the value for variable "c". Assume that Clu1 is scheduled to ALU1 at the scheduling phase. Figure 7.4 shows two variable allocation schemes. According to solution (a) all three variables are allocated to M1. Two clock cycles are needed to move "a" and "b" to the desired memory: ($M1_r \rightarrow$ PP1.Ra) and ($M1_r \rightarrow$ PP1.Rb). This is because only one value can be read from memory M1 at each clock cycle. According to scheme (b), all variables are allocated to different memories. Therefore "a" and "b" could be moved to Ra and Rb respectively in one clock cycle, i.e., ($M1_r \rightarrow$ PP1.Ra) and ($M2_r \rightarrow$ PP1.Rb) are done in parallel.

2. Exploit the locality of reference. If possible we allocate a variable to a memory local to the ALU that is going to use the variable/array. This is more energy-efficient than allocating the variable/array to a non-local one. For instance, in the example shown in Figure 7.3 and Figure 7.4, if "b" is only used by ALU1, it is preferable to allocate it to M1 or M2 instead of M3. Variable "b" has to be moved to PP1.Rb. The move ($M3_r \rightarrow$ PP1.Rb), using a global bus, costs more energy than the move ($M1_r \rightarrow$ PP1.Rb) using a local memory bus. However, one variable might be used by several ALUs, so it is not always possible to allocate the variable to a memory that is local to all ALUs which use it. For example, if both ALU1 and ALU2 use "a" as one of their inputs, when "a" is allocated to M1, it is not local to ALU2, and vice versa. However, the PP that uses the variable most frequently could be determined. When it is located to a memory of that determined PP,

the frequency of using global buses of the crossbar could be decreased. For example, if ALU1 uses "a" two times and ALU2 uses "a" one time, it is preferable to allocate "a" to M1 or M2.

3. Determine a possible memory access pattern. Some memory access patterns cannot be implemented in the Montium tile because of the AGU limitations. Those type of patterns must be avoided by the compiler. For example, see the C code in Figure 7.5(a), where arrays a$[0\cdots9]$ and b$[0\cdots9]$ are used in a nested loop. If we allocated the two arrays in the same memory, as it is shown in Figure 7.5(b), the order of the accessed address would be: 0, 10, 0, 11, 0, 12, $\cdots$, 0, 19, 1, 10, 1, 11, 1, 12, $\cdots$. Unfortunately it is impossible for the AGU to generate this sequence (see the structure of AGU in Figure 4.11 on Page 56). Therefore, arrays a$[0\cdots9]$ and b$[0\cdots9]$ should be allocated to two different memories.

```
for(i = 0; i < 10; i++){
  for(j = 0; j < 10; j++){
    ...
    a[i] = b[j];
    ...
  }
}
```

(a) C code

address	element
0	a[0]
⋮	⋮
9	a[9]
10	b[0]
⋮	⋮
19	b[9]

(b) Two arrays in the same memory

Figure 7.5: Unaccessible memory pattern

4. Distribute memory access patterns evenly over all memories. The number of AGU configurations is limited. Accessing the same memory for many different variables/arrays we will very likely run out of AGU configurations for that memory.

5. Reduce move patterns. It is preferable for a memory to communicate with the same move entity for many times instead of to communicate with many different entities. For example, suppose that there is a variable "a" which is going to be used by ALU1 as the Ra input and there exist already moves ($M1_r \rightarrow$ PP1.Ra) ($M2_r \rightarrow$ PP1.Ra) and ($M3_r$

→ PP1.Ra) for other reasons. Now if we put the variable "a" to the memories M1, M2 or M3, it is possible that the same crossbar configuration is used. If "a" is allocated to M4, a new configuration for the move ($M4_r$ → PP1.Ra) must be added. To record move patterns, a source-destination table is introduced in Section 7.3.1.

The above mentioned aspects often conflict with each other. For example, a simple move pattern might lead to complex AGU patterns and vice versa. To obtain a feasible allocation result, we have to find a balance between all aspects.

7.3.1 Source-destination table

Source \ Dest	PP1Ra	PP1Rb	PP1Rc	⋯	PP5Rd	$M1_w$	⋯	$M10_w$
PP1.Out1	0	0	0	0	0	0	0	0
PP1.Out2	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
PP5.Out2	0	0	0	0	0	0	0	0
$M1_r$	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$M10_r$	0	0	0	0	0	0	0	0

Table 7.3: Source-destination table

A source-destination table is introduced to record all the moves for an application. It is a 20 by 30 table, as shown in Table 7.3. The integer numbers in the Table represent the frequency of occurrence of moves with the same source entity and destination entity. It is preferable to have one entity communicating repeatedly with fewer other entities than with many different other entities because of the limitation of the crossbar configurations (including both local interconnection configurations and global bus configurations). Having fewer move partners implies fewer move patterns. As a result, the number of needed configurations is smaller. Therefore it is worthwhile for the algorithm for allocating variables to consider this preference.

```
1.  AllocateVariablesAndArrays(){
2.     Find the next array/variable v to allocate.
3.     Compute the corresponding priority function for allocating v
       to all ten memories.
4.     Allocate v to the memory with the largest priority function.
5.  }
```

Figure 7.6: Algorithm for allocating variables and arrays

7.3.2 Algorithm for allocating variables and arrays

Figure 7.6 shows the algorithm for allocating variables and arrays according to the analysis above. This algorithm uses a priority function based heuristic method. Similar methods have been used several times in this thesis.

Line 2 and 3 are the key techniques in the algorithm, which are presented in detail below.

7.3.3 Ordering variables and arrays for allocating

There might be many variables and arrays in one application. The allocation order has a strong influence on the final result.

The order of variables/arrays being allocated is determined by the following factors:

- Usually an array needs a consecutive block of memory. Furthermore, a bad allocation for an array will cause more usage of global buses than a bad allocation for a variable because arrays are usually accessed within loop bodies which are repeatedly executed many times. Therefore the allocation of an array to a local memory is typically more important than the allocation of a variable. For this reason, we allocate arrays before variables to give arrays a higher probability of good allocations. Furthermore, larger arrays are allocated before smaller arrays.

- For each array or variable v we can compute the priorities $p(v, \mathrm{M})$ of allocating it to any of all possible ten memories , $\mathrm{M} \in \{\mathrm{M1}, \cdots, \mathrm{M10}\}$, which is described in Section 7.3.4. The *allocation preference of* v is

computed as:

$$\text{Preference}(v) = \max_{\text{M}\in\{\text{M1},\cdots,\text{M10}\}}\{p(v,\text{M})\} - \min_{\text{M}\in\{\text{M1},\cdots,\text{M10}\}}\{p(v,\text{M})\}. \quad (7.1)$$

If there is not much difference between allocating v to any of the memories, the values of the priority function $p(v,\text{M})$ for all ten memories will be almost equal. Thus, the allocation preference is small. If the allocation preference is large, we can conclude that there is much benefit if we allocate v to its most favorite place instead of to its least favorite place. Therefore, the array with a larger allocation preference is allocated before the one with a smaller allocation preference.

7.3.4 Priority function

The priority of allocating an array or variable v to a memory M is

$$\begin{aligned} p(v,\text{M}) \;=\; & \begin{cases} -\infty & \text{if memory access not possible;} \\ \grave{f}(\text{Number of variables in M}) & \text{otherwise.} \end{cases} \\ & + \grave{f}(\text{Times of access to M}) \\ & + \grave{f}(\text{Number of elements in M}) \\ & + \grave{f}(\text{Number of new source-destination pairs}) \\ & + \acute{f}(\text{Number of existing source-destination pairs}) \\ & + \grave{f}(\text{Number of global moves}). \end{aligned} \quad (7.2)$$

In this equation, $\grave{f}(x)$ represents a decreasing function of x and $\acute{f}(x)$ represents an increasing function of x. All these increasing or decreasing functions do not necessarily have to be the same. Here we give some simple definitions for these functions: all increasing functions are $\acute{f}(x) = x$ and all decreasing functions are $\grave{f}(x) = -x$. These functions will be improved in our future work.

We use the example shown in Figure 7.7 to explain the priority function. Here variable "a" is taken by clusters "Clu1" and "Clu2" as inputs and the result of "Clu1" is saved to "a" again. Assume "Clu1" is scheduled to ALU1

and "Clu2" is scheduled to ALU2. Table 7.4(a) shows all moves related to variable "a" in Figure 7.7. Suppose that all non-zero elements of the source-destination table are shown in Table 7.4(b) and the state of ten memories is listed in Table 7.4(c). The result is presented in Table 7.5.

The priority is set to $-\infty$ when the memory access pattern cannot be generated by the AGU as described in item 3 on Page 137.

$\grave{f}$(Number of variables in M): The number of different variables and arrays in one memory reflects the amount of memory access patterns. The more variables a memory has, the more likely the memory runs out of the AGU configuration space. Therefore, in Equation (7.2) by $\grave{f}$(Number of variables in M) the number of already existing variables/arrays in memory M negatively contributes to the priority of allocating a new variable to memory M.

$\grave{f}$(Times of access to M): A *memory access* to a variable is a fetch (fe), an array fetch (afe), a store (st) or an array store (ast) operation on the state space of that variable. For example, in Figure 7.7, there are two memory accesses to variable "a". If there is an fe within a loop body, the fe will be repeated at each iteration. In that case we count it as one memory access because the AGU configuration is not changed during the execution of the loop. More memory accesses to a memory lead to a higher probability of more address access patterns. Therefore the AGU is more likely to run out of its configuration space. By the factor $\grave{f}$(Times of access to M) a new variable is not likely to be allocated to memory M if M has already been accessed for many times.

The times of accessing memory M have a similar influence on the configuration space of the AGU for M as the number of different variables and arrays in M. Thus the priority function is also a decreasing function of the

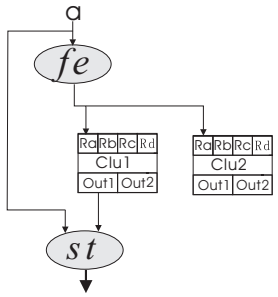

Figure 7.7: A variable allocating example

(a)

$m1$	$M_r \rightarrow$ PP1.Ra
$m2$	$M_r \rightarrow$ PP2.Ra
$m3$	PP1.Out1 $\rightarrow M_w$

(b)

	$\cdots$	$\cdots$	PP1.Ra	$\cdots$	$M6_w$	$\cdots$	$\cdots$	
PP1.Out1	0	0	0	0	2	0	0	0
PP2.Out1	0	0	0	0	2	0	0	0
$M1_r$	0	0	2	0	0	0	0	0
$M5_r$	0	0	5	0	0	0	0	0

(c)

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
#variables	1	0	0	0	1	1	0	0	0	0
#access	1	0	0	0	1	2	0	0	0	0
#elements	1	0	0	0	128	512	0	0	0	0

Table 7.4: (a) New address moves; (b) Source-destination table; (c) Memory states

times of accessing memory M.

$\grave{f}$(Number of elements in M): To have memories evenly occupied, $\grave{f}$(Number of elements in M) is used. Number of elements in M refers to the number of elements before v is allocated to M. Due to this factor, almost full memories have lower priority than almost empty memories to accept a variable or an array.

$\grave{f}$(Number of new source-destination pairs): Usually, there are some moves related to a variable. When allocating a variable, *new moves* refer to the moves for that variable. To decrease the number of crossbar configurations, we prefer that as few as possible new source-destination pairs are added to the source-destination table when allocating a variable. For the example in Figure 7.7, from the source-destination table (Table 7.4(b)), it can be seen that there exist two moves ($M1_r \rightarrow$ PP1.Ra). If "a" is allocated to M1, $m1$ = ($M1_r \rightarrow$ PP1.Ra), an existing configuration might be used for $m1$. However, if "a" is allocated to M2, a new configuration must be used for $m1$. Therefore, memory M1 has some priority for allocating "a" over M2 when we only consider $m1$. In Equation (7.2) the factor $\grave{f}$(Number of new source-

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
#variables	1	0	0	0	1	1	0	0	0	0
#access	1	0	0	0	1	2	0	0	0	0
#elements	1	0	0	0	128	512	0	0	0	0
#new pairs	2	3	3	3	2	2	3	3	3	3
#existing pairs	2	0	0	0	5	2	0	0	0	0
#global moves	1	1	2	2	3	3	3	3	3	3
Priority	-4	-4	-5	-5	-130	-518	-6	-6	-6	-6

Table 7.5: Priority function for allocating a variable

destination pairs) is introduced for this reason. "Number of new source-destination pairs" refers to the number of items that changed from zero to non-zero when the variable is allocated to M in the source-destination table.

On the other hand, even if a new move has the same resource-destination pair as an existing move, it is not always possible for the new move to use the same crossbar bus as the old one. If many existing moves have the same source-destination pair as a new move, it is more likely that the new move will share the configuration of an existing move. In the example, in the source-destination table (Table 7.4(b)) there are two moves ($M1_r \rightarrow$ PP1.Ra) and five moves ($M5_r \rightarrow$ PP1.Ra). When we only consider the configuration for move $m1$, it is better to allocate variable "a" to M5 than to M1 because $m1$ has more chance to use the configuration for one existing move. Therefore, the factor $\acute{f}$(Number of existing source-destination pairs) is introduced here. "Number of existing source-destination pairs" is computed as the sum of the number of existing source-destination pairs for each new move. Consider the priority p("a", M1) for Figure 7.7. In Table 7.4(b), the number of moves ($M1_r \rightarrow$ PP1.Ra) is 2, the number of moves ($M1_r \rightarrow$ PP2.Ra) is 0 and the number of moves (PP1.Out1 $\rightarrow M1_w$) is 0. Therefore, "Number of existing source-destination pairs" = 2+0+0 = 2.

$\grave{f}$(Number of global moves): By the factor $\grave{f}$(Number of global moves), the allocation that leads to fewer global moves is given higher priority. For example, in Table 7.4, if "a" is allocated to M1, moves $m1$(M $\rightarrow$ PP1.Ra) and $m3$(PP1.Out1 $\rightarrow$ M) could use a local bus, and only $m2$ has to use a global bus. If "a" is allocated to M3, $m2$ might use a local bus. However, $m1$ and $m3$ must pass a global bus. Therefore, M1 is a better choice than M3 because it needs fewer global moves.

According to Table 7.5, M1 and M2 are equally good for variable "a". In this case we pick one of them randomly.

7.4 Scheduling data moves

After ALUs are scheduled and variables are allocated, the source and destination of all moves are known. Now we can determine the clock cycle where the move should be executed.

In this part, the order of moves is determined. The attention is on the correctness of the schedule. The limitation of the size of configurations are not considered, which are assumed to be unlimited at this moment. Only the number of reading and writing ports of memories and registers are considered. We avoid to write to or read from the same memory/register at the same clock cycle.

7.4.1 Determining the order of moves

To determine the order of moves, the producer and consumer of the value must be checked.

The *producing clock cycle* of an STLR, ProdCC(STLR), is the clock cycle where the value of the STLR is generated. Correspondingly, the *first consuming clock cycle* ConsCC(STLR) is the first clock cycle where the value of the STLR is used. Figure 7.8 gives all the possible situations of producing clock cycles, and Figure 7.9 shows all the possible situations of consuming clock cycles.

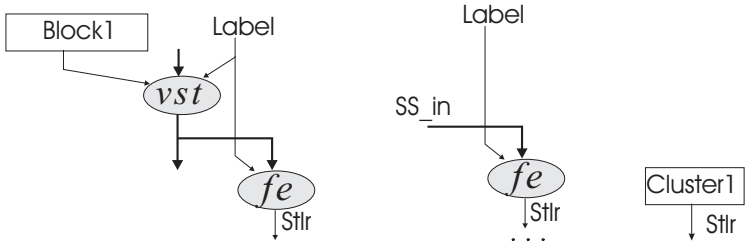

Figure 7.8: Producing clock cycle: (a) ProdCC(STLR) = VirtualClock(Block1); (b) ProdCC(STLR) = External Step; (c) ProdCC(STLR) = Clock(Cluster1)

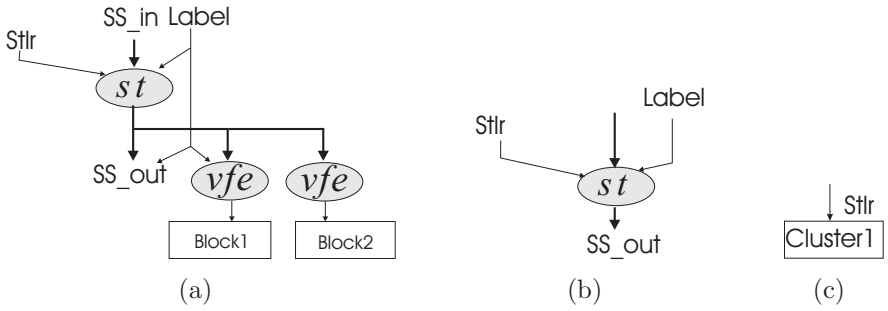

Figure 7.9: First consuming clock cycle: (a) ConsCC(STLR) = The earlier clock between VirtualClock(Block1) and VirtualClock(Block2); (b)= ConsCC(STLR) = External Step; (c) ConsCC(STLR) = Clock(Cluster1)

In the Montium architecture, a constant cannot be loaded from an instruction as in many other processors. Except for several simple numbers that can be generated by ALUs during run time, other constants have to be stored in memories and moved to a register when needed. In our compiler, we treat a constant in the same way as a variable.

Schedule moves The results of a cluster must be saved at the clock cycle immediately after the clock cycle where the cluster is executed. At this specific clock cycle, if a consignment or an upward move for the STLR cannot be executed, the value will be lost. Preparing inputs for ALUs is more flexible in the sense that there are more possible clock cycles. Therefore, type 3 and type 4 moves defined in Table 7.2 are scheduled before type 1 and type 2 moves. This ordering avoids that type 1 and type 2 moves compete with type 3 and type 4 moves for writing ports of registers and memories. For example, suppose that two moves $m1$ = (PP1.Out1→PP1.Ra) and $m2$ = (PP1.M1→PP1.Ra) both want to write to PP1.Ra at a specific clock cycle. If $m2$ is scheduled first, then $m1$ cannot be scheduled because the writing port of PP1.Ra is occupied. The value of PP1.Out1 will be lost. On the other hand, if $m1$ is scheduled to the clock cycle, $m2$ might go to another one, e.g., one clock cycle earlier or later. Although type 1 and type 2 moves are prevented from competing for writing ports with type 3 and type 4 moves, it is still possible that writing ports of destination memories/registers are occupied for type 3 and type 4 moves because two type 3 or type 4 moves want to write to the same destination memory at the same clock cycle. Now we break one of them into two moves (see Page 134), i.e. to save the output

to a temporary memory whose writing port is not busy at that clock cycle, and then copy it to the correct destination when the original writing port is free. The happens, for example, for two moves $m3$ = (PP1.Out1→PP1.Ra) and $m4$ = (PP1.Out2→PP1.Ra), if both of them should be scheduled at the same clock cycle. $m4$ can be broken into $m5$ = (PP1.Out2→M1) and $m6$ = (M1→PP1.Ra). A Montium tile has ten global buses and ten memories. Therefore it is always possible to save all ten ALU outputs within one clock cycle.

7.4.2 Optimization

Moves in Table 7.2 are defined with the assumption that each variable or array element is in a memory before a fetch (*fe*) and will be saved to a memory after a store (*st*). In many cases, especially within a loop, some moves defined by Table 7.2 could be combined. This combination will relieve the pressure on the crossbar and memory ports. We concentrate on loop bodies to discuss combinations of moves.

Combine downward moves: In a loop body, some variables might only be fetched. Then the value of the STLR after a downward move always stays the same during the entire loop. In Figure 4.4 on Page 49, parameters C0, C1, C2, C3 and C4 are such a kind of variables. In this case, instead of defining a downward move for every iteration, a downward move before the loop could substitute the moves for all iterations. The disadvantage of this is that the register which keeps the value cannot be used by others during the execution of the whole loop. Thus, these kind of combinations are only done when there are free registers.

Combine upward moves with downward moves or shifts: If during the previous iteration of a loop an STLR corresponding to a variable is stored and during the next iteration another STRL corresponding to the same variable is fetched, then the consumers of the second STLR can operate directly on the first STLR. If the first STLR is an output of a cluster, the derived downward move or shift according to Table 7.2 will be changed to a consignment or upward move. Take "T0@0#0" in Figure 4.14 as an example. "T0@0#0", is saved to a memory by an upward move in the first iteration and moved to a register by a downward move at the second iter-

ation. Therefore we can just use a consignment to save "T0@0#0" to the register directly.

Combine upward moves: In a loop body, if there is an upward move for a variable for every iteration, then only the upward move of the last iteration is needed. In the following example,

```
for   (int i = 0; i < 10; i++){
      s = i*3
}
```

we can use the upward move for variable "s" of the last iteration to substitute the upward moves of all iterations.

7.4.3 Spilling STLRs

Each register bank in all inputs of all ALUs of the Montium has four entries. The amount of registers is supposed to be enough when we define data moves. If, at some moment, the number of entries of a specific register bank is larger than 4, one or more STLRs should be spilled, i.e., saved in a memory and fetched to the register before being used. The traditional graph coloring method of choosing the proper STLRs can be found in [10][19]. We do not address it here again. For the Montium the graph coloring method has to be used on each register bank (totally 20). Once an STLR representing a move consignment is chosen to spill, the consignment is broken up into an upward move and a downward move as shown on Page 134. And the downward move operation is done just before the STLR is used. If an STLR of a downward move is chosen to be spilled, then we only need to limit the moment of the downward move and let it be done just before it is being used.

7.5 Modeling of crossbar allocation, register allocation and memory allocation

After moves are scheduled, we will do crossbar allocation, register allocation and memory allocation to locally decrease the configuration numbers. These three parts have not yet been implemented. They will be tackled in our future work. Here we only give the modeling of these problems.

7.5.1 Crossbar allocation

The purpose of crossbar allocation is to decrease the number of *crossbar configurations*. Crossbar configurations refer to one-global bus configurations, five-global bus configurations, one-local interconnection configurations and ten-local interconnection configurations. After scheduling the moves described in Section 7.4, the relative order of executing moves is determined. The crossbar allocation is to assign a bus of the crossbar to each move.

The moves of each clock cycle can be written into *an interconnect table*. A row represents a source entity and a column a destination entity. In the

Source \ Dest	PP1Ra	PP1Rb	PP1Rc	⋯	PP5Rd	$M1_w$	⋯	$M10_w$
PP1.Out1	-	-	√	-	-	-	-	-
PP1.Out2	-	-	-	-	-	-	-	√
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
PP5.Out2	-	√	-	-	-	-	-	-
$M1_r$	√	-	-	-	-	-	-	-
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$M10_r$	-	-	-	-	-	√	-	-

Table 7.6: An interconnect table

table "-" means no moves from the source entity to the destination entity, and a "√" is used when there is a move. For each clock cycle, we could build such a table. Notice that there is at most one "√" within one column because only one data can be written to a specific destination entity within one clock cycle. However, more "√"s are allowed in one row. This does not mean that more values can be read from the same memory or register simultaneously. It means that one value is read from a source entity and written to several different destination entities (see the example in Figure 7.10). The interconnect tables of all clock cycles form an *interconnect cube*.

The objective of the crossbar allocation is to assign a bus to each element of an interconnect cube where a "√" occurs such that some constraints are satisfied. The bus is one in the set {GB1, GB2, GB3, GB4, GB5, GB6, GB7, GB8, GB9, GB10, LM1, LM2, LM3, LM4, LM5, LM6, LM7, LM8, LM9, LM10, LA11, LA12, LA21, LA22, LA31, LA32, LA41, LA42, LA51, LA52}. The constraints for crossbar allocation are:

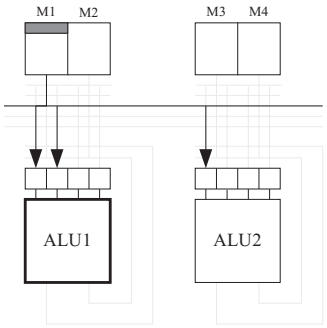

Figure 7.10: From one source entity to multiple destination entities

1. Each local bus from {LM1, LM2, LM3, LM4, LM5, LM6, LM7, LM8, LM9, LM10, LA11, LA12, LA21, LA22, LA31, LA32, LA41, LA42, LA51, LA52} only appears at the row corresponding to the memory or ALU output. For example, LM1 can only appear in row $M1_r$, and LA12 can only appear in row PP1.Out2. Furthermore, a local bus is only allowed to be used when the source entity and the destination entity are local. For example, LM1 can only be used at row $M1_r$ and at columns PP1.Ra, PP1.Rb, PP1.Rc, PP1.Rd, $M1_w$ and $M2_w$.

2. In a row, a bus can repeatedly appear in several columns. This means that a value is written to several different destinations through the same bus. This holds for local buses. For example, in row $M1_r$ LM1 is allowed to appear at columns PP1.Ra, PP1.Rb, PP1.Rc, PP1.Rd, $M1_w$ and $M2_w$ simultaneously.

3. In each clock cycle, one bus may not appear in different rows. This is because a bus cannot transfer different values within one clock cycle.

4. The constraints for global bus configurations are seen clearly using a *global bus configuration table*. This table defines what source is driving each bus in each clock cycle. The global bus configuration table is built, based on the interconnect cube, in this way: Each row in the global bus configuration table corresponds to a clock cycle in the interconnect cube, each column corresponds to a global bus. The element at a specific position is the name of the source entity that uses the global bus within the corresponding clock cycle. Table 7.8 is the global bus configuration table generated from the interconnect cube given in

Table 7.7. GB1 appears at row PP1.Out1 in clock cycle 1 in the interconnect cube, so PP1.Out1 is at row 1 column GB1 in the global bus configuration table. As we said, a global bus may not appear in different rows in one clock cycle. Therefore, each position in the global bus configuration table contains at most one source entity.

Clock cycle 1

	PP1Ra	PP1Rb	PP1Rc	$\cdots$	PP5Rd	$M1_w$	$\cdots$	$M10_w$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
PP1.Out1	GB1	GB2	-	-	-	-	-	-
PP2.Out1	-	-	-	-	-	-	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$M1_r$	-	-	-	-	-	-	-	GB10

Clock cycle 2

	PP1Ra	PP1Rb	PP1Rc	$\cdots$	PP5Rd	$M1_w$	$\cdots$	$M10_w$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
PP1.Out1	GB1		-	-	-	-	-	-
PP2.Out1	-	-	-	-	-	-	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$M1_r$	-	-	-	-	-	-	-	GB10

Clock cycle 3

	PP1Ra	PP1Rb	PP1Rc	$\cdots$	PP5Rd	$M1_w$	$\cdots$	$M10_w$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
PP1.Out1	GB1		-	-	-	-	-	-
PP2.Out1	-	GB2	-	-	-	-	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$M1_r$	-	-	-	-	-	-	-	GB10

Table 7.7: An example interconnect cube with three clock cycles

(a) The elements in each column of the global bus configuration table are the *one-global bus configuration*s for the corresponding global bus. Therefore, each column is allowed to have at most n (In the current Montium $n = 4$.) different elements because there are n configurations for each global bus.

(b) Each row is a *ten-global bus configuration*. The number of distinct

clock	GB1	GB2	$\cdots$	GB10
1	PP1.Out1	PP1.Out1	$\cdots$	$M1_r$
2	PP1.Out1	-	$\cdots$	$M1_r$
3	PP1.Out1	PP2.Out1	$\cdots$	$M1_r$

Table 7.8: Global bus configuration table

rows is limited to m ($m = 32$ in the current Montium). Notice that in Table 7.8, clock cycle 2 can use the ten-global bus configuration of clock cycle one. Therefore, there are two different ten-global bus configurations in Table 7.8.

5. *A local interconnect configuration table* is used to check local interconnect configurations. The local interconnect configuration table is based on the interconnect cube in this way: Each column represents a destination entity, and each row represents a clock cycle. For each destination entity in one clock cycle, write down the local bus that appears in one column in the corresponding to the clock cycle of the interconnect cube. The table has 30 columns corresponding to the 30 destination entities. Table 7.9 is the local interconnect configuration table generated from the interconnect cube given in Table 7.7. Notice that in each clock cycle, at most one bus appears in one column. This ensures that each position of the local interconnect table has at most one element.

clock	PP1Ra	PP1Rb	PP1Rc	$\cdots$	PP5Rd	$M1_w$	$\cdots$	$M10_w$
1	GB1	GB2	-	-	-	-	-	GB10
2	GB1		-	-	-	-	-	GB10
3	GB1	GB2	-	-	-	-	-	GB10

Table 7.9: Local interconnect configuration table

   (a) The elements in one column are one-local interconnect configurations for the corresponding destination entity. The number of different elements in one column is limited to 8 in the current Montium implementation.

   (b) Each row in the local interconnect configuration table forms a thirty-local interconnect configuration. The number of distinct

clock	PP1					PP2	PP3	pp4	pp5
	Ra_r	Ra_w	Rb_r	Rb_w	···	···	···	···	···
1	1	1	2	1	···	···	···	···	···
2	1	2	3	2	···	···	···	···	···
3	1	2	*	2	···	···	···	···	···
4	1	2	*	-	···	···	···	···	···

Table 7.10: Register configuration table

thirty-local interconnect configurations is limited to 32 in the current Montium implementation.

7.5.2 Register arrangement

Register configurations can be written as a table (see an example in Table 7.10). Each row represents a clock cycle and each column represents the read address (with index $_r$) or write address (with index $_w$) of a register file. As we know, in the current Montium implementation, each register file has four entries. The read address could be "1", "2", "3", "4" or "*". "1" to "4" represent the four registers and "*" represents "Do not care", which means the read value is not useful. The written address could be "1", "2", "3", "4" or "-", in which "-" means "not written".

Two-register configuration The four register files of a processing part are divided into two groups: group (Ra and Rb) and group (Rc and Rd). The numbers in the table represent the index of the register in the register bank. Here the group PP1.Ra and PP1.Rb is taken as an example (shown in Table 7.10). A two-register configuration of (Ra and Rb) includes the information about Ra read address, Ra write address, Rb read address and Rb write address. In Table 7.10, clock cycle 3 could use the two-register configuration of clock cycle 2 because Rb_r address in clock cycle 3 is "*" (do not care). However, clock cycle 4 cannot use the two-register configuration of clock cycle 2 or 3. The reason is that the "-" ("not written") cannot be replaced by any other address. In total, there are 3 different two-register configurations for PP1.Ra and PP1.Rb shown in Table 7.10. In the current Montium, the number of two-register configurations is limited to 16.

Twenty-register configuration Each processing part has two two-register configurations. Five processing parts have ten two-register configurations in total. The combination of them forms a *twenty-register configuration*. In the current Montium, the number of twenty-register configurations 32.

The objective of the register configuration algorithm is to select the best register for each intermediate value or variable such that the size constraints for the two-register configurations and the twenty-register configurations are satisfied.

7.5.3 Memory arrangement

Each memory unit contains a reconfigurable AGU to generate address sequences for the associate RAM component. Addresses are typically generated by selecting an modification address or by adding this modification to the current address. Each AGU has 16 modification registers to store modifications. The selection of each AGU is an one-AGU configuration. And the combinations of all ten AGU forms a ten-AGU configuration. The number of one-AGU configurations is currently limited to 16 and the number of ten-AGU configurations is limited by 64. The objective of the memory arrangement problem is to determine the storage positions for variables within a memory such that the number of one-AGU configurations and the number of ten-AGU configurations do no exceed their limitations.

The implementation of the crossbar allocation, register arrangement and memory arrangement algorithm is left to future work. We plan to implement these algorithms in a way similar as the one we used for the clustering and scheduling algorithms in Chapter 5 and Chapter 6, i.e., we first do a pattern selection to extract regular patterns, then perform the crossbar allocation, register arrangement or memory arrangement, using the selected patterns.

7.6 Related work

The storage allocation problem has been studied for a long time. The first paper on this subject appears to be Lavrov's paper of 1961 on minimizing memory usage [57]. Ershov solved storage allocation problems by building an interference graph and using a packing algorithms on it [28]. Some papers focus on register allocation. Chaitin *et al.* first used graph coloring as a paradigm for register allocation and assignment in a compiler [19]. Briggs

et al. described two improvements to Chaitin-style graph coloring register allocators to decrease the number of procedures that require spill code, to reduce the amount of spill code when spilling is unavoidable, and to lower the cost of spilling some values. [10][11]. Chow and Hennessy describe a priority-based coloring scheme [20].

In [71], Pauca *et al.* presented a scheme for optimizing data locality and reducing storage space.

In the Montium, the resource allocation problem includes not only register allocation, memory allocation, but also communication arrangement. The limitation in the tradition register allocation is the number of registers or the size of memories. In the Montium, the limitation on the number of configurations for register access, AGU, and crossbar is a more difficult constraint, which has not been addressed in the existing literature.

7.7 Conclusion

This chapter describes the resource allocation phase for the Montium tile processor. The resource allocation phase uses the details of the Montium tile. The main jobs are allocating variables and scheduling communications. There are many constraints and minimization objectives. It is almost impossible to handle them at the same time. Therefore, we consider them one by one. For example, for each variable, a memory is selected at the variable allocation step. The position where the variable stays in the selected memory is determined at the memory allocation step. The position where an intermediate variable stays in a register is determined at the register allocation step. The clock cycle in which a data move takes place is determined at the scheduling data moves step. At the crossbar allocation step, each move is assigned to a bus. According to our approach, one clear optimization goal is considered at each step. To minimize the negative effects caused by this division, at each sub-step we take requirements for later sub-steps into consideration. For example, at the variable allocation step, we use a source-destination table to record all the moves that exist for the whole application. Variables tend to be allocated in a memory such that as few as possible new communications are needed. This will lead to a better crossbar allocation.

Currently we use a very simple priority function for the variable allocation. In future work, we will work on designing a more suitable priority function to improve the allocation performance. Crossbar allocation, register

allocation and memory allocation parts are only modeled in this chapter. We are convinced that they can be implemented with techniques similar to the techniques described earlier in this thesis in our future work.

Chapter 8

Conclusions

In this chapter, first a summary of the thesis is given in Section 8.1. Then in Section 8.2 we look back at the lessons learned during the work and suggest some future work.

8.1 Summary

This thesis presented a mapping and scheduling method for a coarse-grained configurable processor tile. As many other compilers, a heuristic method with a four-phase division is adopted: transformation, clustering, scheduling and allocation. The performance of the mapping result will be negatively influenced by the sequential concatenation of those four phases. As a result, more clock cycles might be used in the generated code for an application than that in the optimal solution. However, in our opinion, using a greedy method is the only practical choice because the original mapping task is too complex to be handled in an optimal way. To decrease the negative consequences caused by the sequential order of the heuristics, when each phase tackles one subproblem, it also takes the requirements of others subproblems into consideration.

The limitation of the numbers of configurations is the most important aspect which the Montium compiler should consider. This is also the main difference between the Montium compiler and a compiler for other architectures. To generate limited number of instructions for the Montium, regularity selection algorithms are used (e.g., template selection in the clustering phase, pattern selection in the scheduling phase). These algorithms find the most

frequently used templates or patterns in the application. By using these templates or patterns, the number of different instructions is small.

The main challenges faced by the Montium compiler are mentioned in Section 3.4 on Page 29. Now we take a look at how the proposed mapping procedure achieves these challenges.

High speed: We achieve the high speed by exploiting parallelism of the source code: (1) In the transformation phase, sequential codes are transformed to CDFGs which disclose the parallelism among all functions (see Chapter 4); (2) In the clustering phase, the largest possible templates are selected (see Chapter 5). Therefore one ALU can execute more primitive functions within one clock cycle, which reduces the amount of clock cycles; (3) The multi-pattern scheduling algorithm schedules a DFG into as few clock cycles as possible (see Section 6.3); (4) The pattern selection algorithm selects the patterns with more instances, which improves the performance of the multi-pattern scheduling algorithm (see Section 6.4); (5) To have memories occupied evenly we enable several values to be fetched and stored in parallel (see Section 7.3).

Low energy consumption: Low energy consumption is mainly achieved in the following part: (1) The mapping scheme with fewer clock cycles consumes less energy. Therefore, all efforts for achieving high parallelism are also efforts to decrease the energy consumption. (2) In the compiler the locality of reference principle is used extensively. In other words, variables and arrays have the preference to be allocated to the memories or registers local to the ALUs that use the variable or the array (see Section 7.3); (3) Intermediate results have the preference to be allocated to the registers that need it directly. If that is not possible, a local memory is first considered, then a global memory or a global register.

Dealing with Constraints of the target architecture The constraints of the numbers of configurations imposed by the target architecture increase the complexity of the compiler considerably. However, these constraints have a good reason: they are inherent to energy efficiency, which means that we have to find ways to cope with these constraints.

The following efforts decrease the number of one-ALU configurations. In the clustering phase, a minimal number of distinct templates are used to cover

a graph. The distinct templates are one-ALU configurations (see Chapter 5). Therefore the total number of one-ALU configurations for all five ALUs is minimized at the clustering phase. The column arrangement algorithm in the scheduling phase is to decrease the number of one-ALU configurations for each ALU (see Section 6.5).

The following efforts decrease the number of five-ALU configurations: In the clustering phase (see Chapter 5), the effort to decrease the number of distinct templates will make the scheduling problem easier, which indirectly decreases the number of five-ALU configurations. The pattern selection algorithm selects a given number of patterns for scheduling algorithms. Each pattern is a five-ALU configuration (see Section 6.4). The multi-pattern selection algorithm schedules a graph, only using the pattern selected by the pattern selection algorithm (see Section 6.3).

The major efforts for decreasing the number of one- and five- AGU configurations is the memory arrangement algorithm, which arranges the variables and arrays in a memory in such a way that the number of memory access patterns is decreased. This reduces the amount of AGU instructions (see Section 7.5.3).

The register arrangement step is to decrease the number of configurations of a register (see Section 7.5.2).

The crossbar configurations consist of one-local interconnect configurations, five-local interconnect configurations, one-global bus configurations and ten-global bus configurations. When allocating a variable or an array, a source-destination table is used to record the communications (see Section 7.3). A variable or array has the preference to be allocated in such a way that the number of different source-destination pairs of communications is minimized. The crossbar allocation part is to decrease the number of crossbar configurations (see Section 7.5.1).

8.2 Lessons learned and future work

The mapping problem for coarse-grained configurable architectures is NP-hard in general. We tried to build an overall mathematical model that could be optimized. We wanted to use some simplify heuristics based on an optimal mathematical model. However, we soon realized that building an overall mathematical model is a too complex task. Therefore we came up with the four-phase approach.

Currently CDFGs act as the frontend of the mapping procedure. Unfortunately, much important information is hidden in CDFGs. For example, loops cannot be identified easily; the number of iterations is not clearly specified. Some extensions of the CDFG could be considered in the future.

There are still many optimizations that can be done for the mapping and scheduling problem. In each chapter, we have already mentioned some future work for some sub-problem of that chapter. To summarize these, future work should focus on the following aspects:

1. Decrease the computational complexity: Due to the limitation of configuration spaces, selection algorithms are used in many places to find the regular pattern or template. These selection algorithms investigate all the possible candidates to find the best one. The number of candidates is usually very large. Although we have employed some simplification methods, the complexity is still too high. More work will be done on decreasing the computational complexity further.

2. Currently, the mapping work is done block by block. Directed acyclic graphs are the focus of clustering and scheduling algorithms. Although some partsb such as the column arrangement algorithm and pattern selection algorithm, can also be used for loops, the whole structure pays more attention to the regular extraction problem of directed acyclic graphs. Many problems regarding loops will be considered in future work. For example: How to reorganize the structure of a loop? How to handle the prefix and postfix of a loop such that as few as possible new configurations are needed? Is it possible to modify the traditional module scheduling algorithms [78], and use them for the Montium tile?

3. In this thesis many building blocks have been developed. However, a set of building blocks does not make a compiler for a coarse-grained reconfigurable processor. Much work still needs to be done to integrate the building blocks to come to a satisfactory compiler for the Montium.

Bibliography

[1] A. Abnous, H. Zhang, M. Wan, G. Varghese, V. Prabhu and J. Rabaey, "The Pleiades Architecture", *The Application of Programmable DSPs in Mobile Communications*, A. Gatherer and A. Auslander, Eds., Wiley, 2002, pp. 327-360.

[2] T.L. Adam, K.M. Chandy and J.R. Dickson, "A comparison of list schedules for parallel processing systems", *Communication ACM*, vol. 17(12), 1974, pp. 685-690.

[3] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Publisher: Addison-Wesley, 1986, ISBN: 0-201-10088-6.

[4] S.R. Arikati and R. Varadarajan, "A Signature Based Approach to Regularity Extraction", *Proceedings of the International Conference on Computer-Aided Design*, 1997, pp. 542-545.

[5] http://www-users.cs.york.ac.uk/ susan/cyc/b/bag.htm.

[6] R. Barua, W. Lee, S. Amarasinghe and A. Agarwal, "Maps: A Compiler-Managed Memory System for Raw Machines", *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, 1999, pp. 4-15.

[7] V. Baumgarten, G. Ehlers, F. May, A. Nuckel, M. Vorbach and M. Weinhardt, "PACT XPPA Self-Reconfigurable Data Processing Architecture", *The Journal of Supercomputing*, vol. 26(2), 2003, pp. 167-184.

[8] D. Bernstein, M. Rodeh and I. Gertner, "On the Complexity of Scheduling Problems for Parallel/Pipelined Machines", *IEEE Transactions on Computers*, vol. 38(9), 1989, pp. 1308-1318.

[9] R. Biswas, "An application of Yager's bag theory in multicriteria based decision making problems", *International Journal of Intelligent Systems*, vol. 14(12), 1999, pp. 1231-1238.

[10] P. Briggs, *Register Allocation via Graph Coloring*, Ph.D thesis, Rice University, 1992.

[11] P. Briggs, K.D. Cooper and L. Torczon, "Improvements to graph coloring register allocation", *ACM Transactions on Programming Languages and Systems*, vol. 16(3), 1994, pp. 428-455.

[12] M. Budiu and S.C. Goldstein, "Fast compilation for pipelined reconfigurable fabrics", *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, Montery, USA, 1999, pp. 195-205.

[13] G.F. Burns, E. van Dalen, M. Jacobs, M. Lindwer and B. Vandewiele, *Enabling Software-Programmable Multi-Core Systems-on-Chip for Consumer Applications*, http://www.silicon-hive.com/uploads/GSPx2005_Paper1.pdf.

[14] S. Cadambi and S.C. Goldstein, "CPR: A Configuration Profiling Tool", *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 1999, pp. 104-113.

[15] http://www.site.uottawa.ca/ rabielmo/personal/rc.html#calisto.

[16] T.J. Callahan, P. Chong, A. DeHon and J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs", *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, Monterey, USA, 1998, pp. 123-132.

[17] A. Capitanio, N. Dutt and A. Nicolau, "Partitioned register files for VLIWs: A preliminary analysis of tradeoffs", *Proceedings of the 25th International Symposium on Microarchitecture*, 1992, pp. 292-300.

[18] A. Capitanio, N. Dutt and A. Nicolau, "Design considerations for Limited Connectivity VLIW architectures", *Technical Report TR59-92*, Department of Information and Computer Science, University of California at Irvine, USA, 1992.

[19] G.J. Chaitin, "Register allocation and spilling via graph coloring", *Proceedings of the ACM SIGPLAN '82 Symposium on compiler Construction*, Boston, USA, 1982, pp. 98-101.

[20] F.C. Chow and J.L. Hennessy, "The priority-based coloring approach to register allocation", *ACM Transactions on Programming Languages and Systems*, vol. 12(4), 1990, pp. 501-536.

[21] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal and R. Gupta, "A General Approach for Regularity Extraction in Datapath Circuits", *Proceedings of the International Conference on Computer-Aided Design*, San Jose, USA, 1998, pp. 332-339.

[22] M.R. Corazao, M.A. Khalaf, L.M.Guerra, M. Potkonjak and J.M. Rabaey, "Performance Optimization Using Template mapping for Datapath-Intensive High-Level Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15(8), 1996, pp. 877-888.

[23] S. Davidson, D. Landskov, B.D. Shriver and P.W. Mallett, "Some experiments in local microcode compaction for horizontal machines", *IEEE Transactions on Computers*, 1981, pp. 460-477.

[24] G. Desoli, "Instruction assignment for clustered VLIW DSP compilers: A new approach", *Technical Report HPL-98-13*, Hewlett-Packard Co., 1998.

[25] C. Ebeling and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison", *Proceedings of the International Conference on Computer-Aided Design*, 1983, pp. 172-173.

[26] C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky and S.G. Berg, "Mapping Applications to the RaPiD Configurable Architecture", *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 1997, pp. 106.

[27] J. Ellis, "Bulldog: A Compiler for VLIW Architectures", MIT Press, 1986.

[28] A.P. Ershov, "Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs", *Doklady Akademii Nauk S.S.S.R. 142, 4(1962)*, English translation *Soviet Mathematics 3*, 1962, pp. 163-165.

[29] E.M.C. Filho, *The TinyRISC Instruction Set Architecture, Version 2*, University of California, Irvine, 1998, http://www.eng.uci.edu/morphosys/docs/isa.pdf.

[30] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[31] http://gcc.gnu.org/.

[32] Y. Guo, G.J.M. Smit and P.M. Heysters, "Template Generation and Selection Algorithms for High Level Synthesis", *Proceedings of the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, Canada, 2003, pp. 2-5.

[33] Y. Guo, G.J.M. Smit, P.M. Heysters and H. Broersma, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System", *2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, California, USA, 2003, pp. 199-208.

[34] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor and R. Laufer, "Piperench: A coprocessor for streaming multimedia acceleration", *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999, pp. 28-39.

[35] S.C. Goldstein, H. Schmit, M. Budiu, M. Moe and R.R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", *IEEE Computer*, vol. 33, 2000, pp. 70-77.

[36] L.J.Hafer and A.C. Parker, "A Formal Method for the specification, Analysis, and Design of Register-Transfer Level Digital Logic," *IEEE Transactions Computer-Aided Design*, vol. CAD-2, no.1, 1983, pp. 4-18.

[37] T.R. Halfhill, *Silicon Hive Breaks Out: Philips Startup Unveils Configurable Parallel-processing Architecture*, http://www.siliconhive.com/uploads/m48_siliconhive_rprnt.pdf, 2003.

[38] M.M. Halldórsson and J. Radhakrishnan, "Greed is good: Approximating independent sets in sparse and bounded-degree graphs", *Proceedings of the 26th annual ACM symposium on Theory of computing*, Montreal, Quebec, Canada, 1994, pp. 439-448.

[39] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", *Proceedings of the Conference Design Automation and Test in Europe*, Munich, Germany, 2001, pp. 642-649.

[40] R. Hartenstein, "The digital divide of computing", *Proceedings of the ACM International Conference on Computing Frontiers*, 2004, pp. 357-362.

[41] J.R. Hauser and J. Wawrzynek, "Garp: A mips processor with a reconfigurable coprocessor", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, 1997, pp. 24-33.

[42] G. Heidari and K. Lane, "Introducing a paradigm shift in the design and implementation of wireless devices", *Proceedings of Wireless Personal Multimedia Communications*, vol. 1 Aalborg, Denmark, 2001, pp. 225-230.

[43] P.M. Heysters, Ph.D Thesis: *Coarse-Grained Reconfigurable Processors – Flexibility Meets Efficiency*, ISBN 90-365-2076-2.

[44] P.M. Heysters, *Montium Tile Processor Design Specification*, internal technical report, 2005.

[45] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9(6), 1961, pp. 841-848.

[46] P. Holzenspies, *Silc: Sprite Input Language with C(++)*, master thesis, 2006, University of Twente.

[47] C.T. Hwang, J.H. Lee and Y.C. Hsu, "A formal approach to the scheduling problem in high-level synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10(4), 1991, pp. 464-475.

[48] R. Jain, A. Mujumdar, A. Sharma and H. Wang, "Empirical evaluation of some high-level synthesis scheduling heuristics," *Proceedings of the 28th conference on ACM/IEEE design automation*, Munich, Germany, 1991, pp. 686-689.

[49] S. Jang, S. Carr, P. Sweany and D. Kuras, "A code generation framework for VLIW architectures with partitioned register banks", *Proceedings of 3rd International Conference on Massively Parallel Computing Systems*, 1998, pp. 61-69.

[50] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh and M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", *ACM Transactions on Design Automation of Electronic Systems*, vol. 7(4), 2002, pp. 605-627.

[51] *Instruction Generation for Hybrid Reconfigurable Systems*, http://citeseer.nj.nec.com/446997.html.

[52] K. Kennedy and R. Allen, "Automatic Translation of FORTRAN programs to vector forms", *ACM Transactions on Programming Languages and Systems*, vol. 9(4), 1987, pp. 491-542.

[53] Th. Krol and B. Visser, "High-level Synthesis based on Transformational Design", Internal Report, University of Twente, Enschede, The Netherlands.

[54] G. Venkataramani, F. Kurdahi and W. Bohm, "A compiler Framework for Mapping Applications to a coarse-grained Reconfigurable Computer Architecture", *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, Atlanta, Georgia, USA, 2001, pp. 116-125.

[55] T. Kutzschebauch, "Efficient Logic Optimization Using Regularity Extraction", *Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, 2000, pp. 487-493.

[56] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, Palo Alto, California, 2004, pp. 75-86.

[57] S.S. Lavrov, "Store economy in closed operator schemes", *Journal of Computational Mathematics and Mathematical Physics 1*, 4, 1961, pp. 687-701.

[58] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar and S.P. Amarasinghe, "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine", *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, 1998, pp. 46-57.

[59] J. Lee, K. Choi and N.D. Dutt, "Compilation Approach for Coarse-Grained Reconfigurable Architectures", *IEEE Design & Test*, vol. 20(1), 2003, pp. 26-33.

[60] J. Lee, K. Choi and N.D. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures", *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool For Embedded Systems*, San Diego, California, USA, 2003, pp. 183-188.

[61] http://llvm.org/.

[62] V.K. Madisetti and B.A. Curtis, "A Quantitative methodology for rapid prototyping and high-level synthesis of signal processing algorithms", *IEEE Transactions on Signal Processing*, vol. 42(11), 1994, pp. 3188-3208.

[63] R. Maestre, F.J. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida and M. Fernandez, "Kernel Scheduling in Reconfigurable Computing",*Proceedings of the Conference Design Automation and Test in Europe*, Munich, Germany, 1999, pp. 90-96.

[64] B. Mei, S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, "DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures", *International Conference on Field Programmable Technology*, Hong Kong, 2002, pp. 166-173.

[65] P.F.A.Middelhoek, G.E. Mekenkamp, E. Molenkamp and Th. Krol, "A Transformational Approach to VHDL and CDFG Based High-Level Synthesis: a Case Study", *Proceedings of CICC 95*, Santa Clara, CA, USA, 1995, pp. 37-40.

[66] P.F.A.Middelhoek, *Transformational Design: an architecture independent interactive design methodology for the synthesis of correct and efficient digital systems*, Ph.D thesis, University of Twente, 1997.

[67] G.L. Nemhauser and L.A. Wolser, *Integer and Combinatorial Optimization*, John Wiley & Sons, New York, 1988.

[68] E. Ozer, S. Banerjia and T.M. Conte, "Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures", *Proceedings of the International Symposium on Microarchitecture*, 1998, pp. 308-315.

[69] http://www.pactcorp.com/.

[70] B.M. Pangrle and D.D. Gajski, "Design Tools for Intelligent Compilation", *IEEE Transactions Computer-Aided Design*, vol. CAD-6, No. 6, 1987, pp. 1098-1112.

[71] V.P. Pauca, X. Sun, S. Chatterjee and A.R. Lebeck, "Architecture-efficient Strassen's Matrix Multiplication: A Case Study of Divide-and-Conquer Algorithms", *Proceedings of the International Linear Algebra Society Symposium on Fast Algorithms for Control, Signals and Image Processing*, Winnipeg, Manitoba, Canada, 1997.

[72] P.G. Paulin and J.P. Knight, 1989. "Force-directed scheduling for the behavioral synthesis of ASICs", *IEEE Transactions CAD* vol. 8(6), 1989, pp. 661-679.

[73] P.G. Paulin and J.P. Knight, "Algorithms for High-Level Synthesis", *IEEE Design and Test of Computers*, vol. 6(4), 1989, pp. 18-31.

[74] K.Bondalapati, G. Papavassilopoulos and V.K. Prasanna, "Mapping Applications onto Reconfigurable Architectures using Dynamic Programming", *Military and Aerospace Applications of Programmable Devices and Technologies*, Laurel, Maryland, 1999.

[75] J. Rabaey, "Reconfigurable Computing: the Solution to Low Power Programmable DSP", *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Munich, 1997.

[76] J. Rabaey and M. Wan, "An energy-conscious exploration methodology for reconfigurable DSPs", *Proceedings of the Conference Design Automation and Test in Europe*, Munich, Germany, 1998, pp 341-342.

[77] D.S. Rao and F.J. Kurdahi, "On Clustering For Maximal Regularity Extraction", *IEEE Transactions on Computer-Aided Design*, vol. 12(8), 1993, pp. 1198-1208.

[78] B.R. Rau, "Iterative Module Scheduling", Technical Report HPL-94-115, Hewlett–Packard Laboratories, 1995.

[79] M.A.J. Rosien, Y. Guo, G.J.M. Smit and Th. Krol, "Mapping Applications to an FPFA Tile", *Proceedings of the Conference Design Automation and Test in Europe*, Munich, Germany, 2003, pp. 11124-11125.

[80] B. Salefski and L. Caglar, "Re-configurable computing in wireless", *Proceedings of the 38th Conference on Design Automation*, Las Vegas, United States, 2001, pp. 178-183.

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

[82] G.J.M. Smit, P.J.M. Havinga, L.T. Smit, P.M. Heysters, M.A.J. Rosien, "Dynamic Reconfiguration in Mobile Systems", *Proceedings of FPL2002*, Montpellier, France, 2002, pp. 171-181.

[83] G.J.M. Smit and Mi.A.J. Rosien, Y. Guo and P.M. Heysters, "Overview of the tool-flow for the Montium Processing Tile", *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, 2004, pp. 45-51.

[84] L.T. Smit, J.L. Hurink and G.J.M. Smit, "Run-time mapping of applications to a heterogeneous SoC", *Proceedings of the 2005 International Symposium on System-on-Chip*, 2005, pp. 78-81.

[85] http://suif.stanford.edu/suif/suif.html.

[86] X. Tang, M. Aalsma and R. Jou, "A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors", *Proceedings of FPL 2000*, Villach, Austria, 2000, pp. 29-38.

[87] H. Topcuoglu, S. Hariri and M.Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing", *IEEE Transaction on Parallel and Distributed Systems*, 13(3), 2002, pp. 260-274.

[88] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh and W. Bohm, "A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture", *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, Atlanta, USA, 2001, pp. 116-125.

[89] R.A. Walker and S. Chaudhuri, "Introduction to the Scheduling Problem", *IEEE Design and Test of Computers*, vol. (12)2, 1995, pp. 60-69.

[90] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, "Baring it all to software: RAW machines", *IEEE Computer*, vol. 30(9), 1997, pp. 86-93.

[91] J. Wawrzynek and T.J. Callahan, "Instruction-level Parallelism for Reconfigurable Computing", *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, Berlin, Germany, 1998, pp. 248-257.

[92] M.E. Wolf and M. Lam, "A loop transformation theory and an algorithm to maximize parallelism", *IEEE Transactions on Parallel and Distributed Systems*, vol. 2(4), 1991, pp. 452-471.

[93] J. Xue, "On Nonsingular Loop Transformations Using SUIF's Dependence Abstraction", *Proceedings of the 2nd international conference on parallel and distributed computing, applications and technologies*, Taiwan, 2001, pp. 331-336.

[94] A. Ye, A. Moshovos, S. Hauck and P. Banerjee, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit", *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, 2000, pp. 225-235.

[95] E.W. Weisstein, "Antichain", from MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Antichain.html.

Publications

1. Y. Guo and G.J.M. Smit, "Mapping and Scheduling of Directed Acyclic Graphs on An FPFA Tile", *Proceedings of the PROGRESS 2002 workshop*, Utrecht, the Netherlands, 2002, pp. 57-65.

2. M.A.J. Rosien, Y. Guo, G.J.M. Smit and Th. Krol, "Mapping Applications to an FPFA Tile", *Proceedings of the Conference Design Automation and Test in Europe*, Munich, Germany, 2003, pp. 11124-11125.

3. Y. Guo, G.J.M. Smit, P.M. Heysters and H. Broersma, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System", *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, California, USA, 2003, pp. 199-208.

4. Y. Guo, G.J.M. Smit and P.M. Heysters, "Template Generation and Selection Algorithms for High Level Synthesis", *Proceedings of the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, Canada, 2003, pp. 2-5.

5. Y. Guo, G.J.M. Smit, H. Broersma, M.A.J. Rosien and P.M. Heysters, "Mapping Applications to a Coarse Grain Reconfigurable System", *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu, Japan, 2003, pp. 221-235.

6. Y. Guo and G.J.M. Smit, "Template Generation - A Graph Profiling Algorithm", *Proceedings of the PROGRESS 2003 Embedded Systems Symposium*, NBC De Blokhoeve, Nieuwegein, the Netherlands, 2003, pp. 96-101.

7. Yuanqing Guo and Gerard J.M. Smit, "Resource Allocation, Exploiting Locality of Reference of a Reconfigurable Architecture", *Proceedings of*

the PROGRESS 2004 Embedded Systems Symposium, Nieuwegein, the Netherlands, 2004, pp. 49-58.

8. G.J.M. Smit, M.A.J. Rosien, Y. Guo and P.M. Heysters, "Overview of the tool-flow for the Montium Processing Tile", *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, 2004, pp. 45-51.

9. Y. Guo and G.J.M. Smit, "Resource Allocation for A Mobile Application Oriented Architecture", *Proceedings of the Global Mobile Congress 2005*, Chongqing, China, 2005, pp 514-519.

10. Y. Guo, G.J.M. Smit, M.A.J. Rosien, P.M. Heysters, Th. Krol and H. Broersma, book chapter "Mapping Applications to a Coarse Grain Reconfigurable System", *New Algorithms, Architectures and Applications for Reconfigurable Computing*, Springer, 2005, pp 93-104.

11. Y. Guo and G.J.M. Smit, "A Column Arrangement Algorithm for a Coarse-grained Reconfigurable Architecture", *Poster for SIREN 2005*, Eindhoven, the Netherlands, 2005.

12. Y. Guo, C. Hoede and G.J.M. Smit, "A Multi-Pattern Scheduling Algorithm", *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, 2005, pp. 276-279.

13. Y. Guo, C. Hoede and G.J.M. Smit, "A Pattern Selection Algorithm for Multi-Pattern Scheduling", *Proceedings of the 20th International Parallel & Distributed Processing Symposium Reconfigurable Architectures Workshop (RAW 2006)*, Greece.

14. Y. Guo, C. Hoede and G.J.M. Smit, "A Column Arrangement Algorithm for a Coarse-grained Reconfigurable Architecture", *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, 2006.