UNIVERSITY OF CALIFORNIA

SANTA BARBARA

Synthesizing Sequential Programs onto Reconfigurable Computing Systems

A Dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Electrical and Computer Engineering

by

Wenrui Gong

Committee in charge:
Dr. Ryan Kastner, Chair
Dr. Forrest Brewer
Dr. Chandra Krintz
Dr. Margaret Marek-Sadowska

December 2007

The dissertation of Wenrui Gong is approved:

Dr. Forrest Brewer

Dr. Chandra Krintz

Dr. Margaret Marek-Sadowska

Dr. Ryan Kastner, Chair

University of California, Santa Barbara

December 2007

Synthesizing Sequential Programs onto

Reconfigurable Computing Systems

by

Wenrui Gong

To my dearest parents, Zhang Shu and Gong Yiheng,

who instilled me the thirst for knowledge

and supported my pursuit of knowledge.

Abstract

This dissertation focuses on synthesizing sequential programs on FPGA-based fine-grained reconfigurable computing systems. Reconfigurable computing combines the flexibility of software with the high performance of hardware, bridges the gap between general-purpose processors and application-specific systems, and enables higher productivity and shorter time to market. Design flows for reconfigurable computing systems conduct parallelizing compilation and reconfigurable hardware synthesis in an integrated framework.

This work proposes to extend the program dependence graph with the single assignment form as the intermediate representations of the design framework. This form supports most of the known program transforms, enables speculative execution, and exposes the instruction-level parallelism. Experimental results showed an overall 15% speed-up compared with traditional control/data-flow graph given the same optimizations.

To create coarse-grained parallelism and better utilize available storage and computation resources, this work presents a novel approach to partition the data/iteration spaces subject to the architectural constraints. This approach utilizes code analysis techniques and exploits a variety of candidate partitions. Experimental results also show that this approach benefits placement and routing, and improves the finally achieved clock frequencies up to 19.5%.

To effectively generate hardware from the data dependence subgraph, we study the resource allocation and scheduling problem. The proposed

operation scheduling algorithms utilize the max-min ant system (MMAS) optimization. Experimental results show up to 23% speed-up over the list scheduler on the resource constraint scheduling, and 15% smaller area over the force-directed scheduler. We further introduce a general model of the resource allocation and scheduling problem, and present an MMAS-based concurrent resource allocation and scheduling algorithm. Experimental results from over more than 1250 realistic designs show up to 20% smaller area and perform better on larger designs.

To summarize, this work presents an automatic design methodology for reconfigurable computing systems, extends current knowledge in parallelizing compilation and hardware synthesis, and delivers a unified solution to the resource allocation and scheduling problem.

Acknowledgments

This dissertation would not have been possible without the advice and support of my advisor Professor Ryan Kastner. He has, since 2003, guided me with enthusiasm, and supported me to grow as a researcher. I am very grateful for his thorough reviewing of this dissertation and our collaboration that has resulted in a number of publications, of which several are included in this dissertation.

I am also grateful to my colleagues in the ExPRESS group at the Electrical and Computer Engineering Department at the University of California, Santa Barbara: Gang Wang, Anup Hosangadi, Yan Meng, Brian DeRenzi, and Daniel Grund. In particular, I would like to thank Gang Wang for our collaborations on numerous research efforts over the years and many fruitful discussions, and for the fun we had in writing the papers and presenting them at conferences.

I would like to thank Professor Forrest Brewer, Professor Chandra Krintz, and Professor Margaret Marek-Sadowska for being on my Ph.D. committee and for all their help along the way. In particular, I would like to thank Professor Margaret Marek-Sadowska for her constructive comments on early versions of this dissertation.

I would also like to thank Professor Louise Moser, Professor Michael Melliar-Smith, Professor Behrooz Parhami, and Dr. Charles Kenney. In particular, I would like to thank Dr. Charles Kenney for his great help.

Of all my friends and colleagues at the University of California, Santa Barbara, I would like to thank in particular Hailin Jiang, Feng Lu, Xin

Hao, Tao Feng, Wei Cui, Chung-Kuan Tsai, Yang Cao, and so many others for all the help I received over the years.

I would like to acknowledge my colleagues in the Catapult Synthesis group at Mentor Graphics: Andrew Guyler, Peter Gutberlet, Andres Takach, Simon Waters, Welson Sun, Sandeep Garg, Shawn McClaud, Bryan Bowyer, Stuart Clubb, Michael Fingeroff, and many others. Thank you for your cooperativeness and many stimulating discussions.

Finally, thanks to my parents who are my best sources of inspiration and motivation. I am sorry that we have not been able to get together lately, I will probably have more time now.

This work has been partially supported by the National Science Foundation (NSF) under grant 0411321 and the Electrical and Computer Engineering Department at the University of California, Santa Barbara.

Curriculum Vitæ

Education

July 1999, Bachelor of Engineering in Computer Software, Sichuan University (Chengdu, Sichuan, China).

December 2002, Master of Science in Electrical and Computer Engineering, University of California (Santa Barbara, California, USA).

Professional Experiences

January 1999 – July 2001, Research Associate, Sichuan University (Chengdu, Sichuan, China).

September 2001 – June 2005, Teaching Assistant, University of California (Santa Barbara, California, USA).

January 2003 – September 2006, Graduate Student Researcher, University of California (Santa Barbara, California, USA).

Summer 2004, Intern, Mentor Graphics (Wilsonville, Oregon, USA).

Summer 2005, Intern, Mentor Graphics (Wilsonville, Oregon, USA).

October 2006 – present, Software Development Engineer, Mentor Graphics (Wilsonville, Oregon, USA).

Publications

[1] Gang Wang, Wenrui Gong, and Ryan Kastner. A New Approach for Task Level Computational Resource Bi-partitioning. In *International Conference on Parallel and Distributed Computing and Systems* (PDCS), 2003.

[2] Gang Wang, Wenrui Gong, and Ryan Kastner. System Level Partitioning for Programmable Platforms Using the Ant Colony Optimization. In *International Workshop on Logic and Synthesis* (IWLS), 2004.

[3] Wenrui Gong, Gang Wang, and Ryan Kastner. A High Performance Intermediate Representation for Reconfigurable Systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2004.

[4] Gang Wang, Wenrui Gong, and Ryan Kastner. Instruction Scheduling Using MAX–MIN Ant Colony Optimization. In *Great Lakes Symposium on Very Large Scale Integration* (GLSVLSI), 2005.

[5] Wenrui Gong, Gang Wang, and Ryan Kastner. Data Partitioning for Reconfigurable Architectures with Distributed Block RAM. In *International Workshop on Logic and Synthesis* (IWLS), 2005.

[6] Ryan Kastner, Wenrui Gong, Xin Hao, Forrest Brewer, Adam Kaplan, Philip Brisk, and Majid Sarrafzadeh. Physically Aware Data Communication Optimization for Hardware Synthesis. In *International Workshop on Logic and Synthesis* (IWLS), 2005.

[7] Wenrui Gong, Yan Meng, Gang Wang, Ryan Kastner, and Timothy Sherwood. Data Partitioning for Reconfigurable Architectures with Distributed Block RAM. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2005.

[8] Peter Gutberlet, Wenrui Gong, and Andres Takach. C/C++ Loop Transformations for Hardware Synthesis. In *GSPx 2005 Pervasive Signal Processing Conference*, 2005.

[9] Wenrui Gong, Gang Wang, and Ryan Kastner. Storage Assignment during High-level Synthesis for Configurable Architectures. In *IEEE/ACM International Conference on Computer Aided Design* (ICCAD), 2005.

[10] Gang Wang, Wenrui Gong and Ryan Kastner. Application Partitioning on Programmable Platforms Using the Ant Colony Optimization. *Journal of Embedded Computing*, Vol. 2, No. 1. 2006.

[11] Yan Meng, Wenrui Gong, Ryan Kastner, and Timothy Sherwood. Algorithm/Architecture Coexploration for Designing Energy Efficient Wireless Channel Estimator. *Journal of Low Power Electronics*, January, 2006.

[12] Ryan Kastner, Wenrui Gong, Xin Hao, Forrest Brewer, Adam Kaplan, Philip Brisk, and Majid Sarrafzadeh. Layout Driven Data Communication Optimization for High-level Synthesis. In *Design Automation and Test in Europe* (DATE), 2006.

[13] Gang Wang, Wenrui Gong, Brian DeRenzi and Ryan Kastner. Design Space Exploration using Time and Resource Duality with the Ant Colony Optimization. In *the 43rd Design Automation Conference* (DAC), 2006.

[14] Gang Wang, Wenrui Gong, and Ryan Kastner. On the Use of Bloom Filters for Defect Maps in Nanocomputing. In *International Conference on Computer-Aided Design* (ICCAD), 2006.

[15] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. Ant Scheduling Algorithms for Resource and Timing Constrained Instruction

Scheduling. *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems* (TCAD). 2007.

[16] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. Exploring Time/Resource Tradeoffs by Solving Dual Scheduling Problems with the Ant Colony Optimization. *ACM Transactions on Design Automation of Electronic Systems* (TODAES). 2007.

[17] Gang Wang, Wenrui Gong, and Ryan Kastner. Operation Scheduling: Algorithms and Design Space Exploration. To appear in *High Level Synthesis Handbook: The State of Arts*. Springer. New York, NY. 2008.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past five decades, semiconductor technology experienced an unprecedented rapid improvement. The capabilities and performance of integrated circuits grew exponentially [87]. Many physical side effects emerged with the continuing device scaling. Some examples include resistance-capacitance coupling, signal integrity, in-die variation, transistor leakage, and soft error rate.

Today's typical application-specific integrated circuit (ASIC) designs contain millions of gates. They are so complicated that it may take weeks to complete full back-end iteration. In addition, the cost of fabricating an ASIC mask set is more than one million dollars. Fewer companies can bring their designs to fruition, and provide flexible solutions adapting to different design requirements.

On the other hand, most computation tasks can fit in a general-purpose processor. General-purpose processors feature sequential

instruction fetching and decoding, fixed control, and limited memory accesses. The performance is limited by these fundamental attributes, and cannot satisfy increasing computation demands. In addition, when designers implement their general-purpose processor designs to silicon, they encounter the same challenges as those in ASIC designs.

When design application-specific systems become more difficult, and general-purposed processors cannot provide sufficient performance, alternative computing architectures are demanded to fulfill computation tasks.

Reconfigurable computing is a novel computing paradigm. A reconfigurable computer consists of an array of processing elements [39], and allows post-manufacturing customization.. These processing elements can be processor cores, configurable logic blocks, or dedicated hardware. Reprogrammable interconnects connect these processing elements and steer data transmission. Optional general-purpose processors execute infrequent operations and handle exceptions. In addition, dedicated hardware is integrated to support complex computation tasks.

Compared with general-purpose processors, reconfigurable computer systems provide high performance since processing elements work in parallel, and dedicated hardware is optimized for complex tasks. Compared with the ASICs, reconfigurable systems provide superior flexibility and great reliability.

Reconfigurable computing combines the flexibility of software with the high performance of hardware, bridges the gap between general-purpose processors and application-specific systems, and enables higher produc-

tivity and shorter time to market.

Over the past twenty years, many reconfigurable processors have been proposed. Some prototypes among them have been implemented. Examples include PADDI [25], PADDI-2 [134], KressArray [73], RaPiD [35], Garp [57], RAW [124], PipeRench [47], Chameleon [117], CRAY XD1, SGI RASC, and so forth. However, few of them have been widely adapted.

The problem is not with the design of reconfigurable architectures, but with the design tools to implement applications to reconfigurable systems. Good design tools effectively synthesize applications from system specifications into configuration files, and perform architecture-specific optimizations to efficiently utilize available resources and exploit the best performance. The lack of automatic design tools limits design activities and disheartens the adaptation of reconfigurable architectures. Moreover, successful design tools evaluate target reconfigurable processors, identify architectural inefficiencies, and direct future architecture designs.

1.1 Research Objectives

The objective of this dissertation is to *provide designers an automatic design methodology to implement reconfigurable computing systems and achieve design goals*, such as high performance, small area, and short time to market.

1.2 This Dissertation

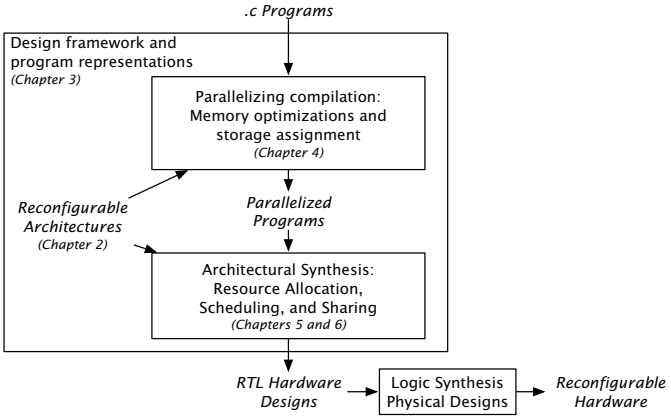


Figure 1.1: Organization of this dissertation

This dissertation begins with a high-level characterization to reconfigurable computing architectures. An introduction to typical design flows follows this characterization. This part helps us understand the characteristics of reconfigurable systems, and identify key challenges in synthesizing applications onto reconfigurable computing systems.

Design tools usually parse functional specifications in high-level language programs. Many sequential languages are available. Popular choices are C and Matlab. This dissertation uses the C programs as the input language because the C programming language is the one with most support and most existing designs.

A design flow for reconfigurable computing systems conducts parallelizing compilation and reconfigurable hardware synthesis in an integrated framework.

The front-end of this framework creates coarse-grained parallelism

and exploits fine-grained parallelism in order to utilize limited hardware resources in an effective and efficient manner. The ability to accomplish these tasks is heavily relied upon the program representations used in the framework. This dissertation investigates various program representations in the literature, and compares their performance in exploring parallelism and synthesizing hardware.

Within the proposed synthesis framework, there are many issues on creating coarse-grained parallelism. In particular, solutions to the memory optimization and storage assignment problem are very valuable in order to utilize precious storage components in modern hybrid architectures. This dissertation characterizes communication-free partitions and other partitioning schemes, explores novel approaches, and provides empirical relations between memory optimizations and system performance.

Synthesizing programs into reconfigurable hardware has many similarities with traditional hardware synthesis. The design tool conducts architectural synthesis, technical mapping, placement, and routing to generate configuration files for reconfigurable hardware. Resource allocation and scheduling, one of the most important problems in hardware synthesis, determines the start time of operations and minimizes the silicon area or latencies subject to timing or resource constraints. The quality of scheduling results greatly affects the quality of completed designs.

With the increasing complexity, it is harder to obtain the optimal solutions using exact solutions and greedy algorithms. Novel algorithms are required to tackle these problems. Evolutionary algorithms are great

candidates to solve these design problems. This dissertation provides algorithm formulations and implementations of evolutionary algorithms to solve these design problems.

Many assumptions and limitations of scheduling algorithms confine their usages in actual hardware designs. This dissertation presents a general model of the concurrent resource allocation and scheduling problem, extends the evolutionary algorithm, and exploits the design space of actual hardware designs.

The main contributions of this dissertation are as follows:

1. The PDG+SSA form is proposed as the intermediate program representation in the front-end of the design framework. This form extends the program dependence graph with the static single-assignment. Based on this form, the design tool is able to explore both coarse-grained and fine-grained parallelism, support loop transformations and optimizations, take advantage of speculative execution, and enable direct mapping to fine-grained reconfigurable hardware. Experimental results show that areas of generated hardware by direct mapping from this form are smaller than from the control/data-flow graph or the predicated static single-assignment form.

2. A novel approach is proposed to partition iteration spaces of nested loops and determine storage assignment of partitioned data arrays. This approach exploits the design space using approaches based on both code analysis and architectural synthesis. Experimental results shows synthesized reconfigurable hardware effectively utilizes precious available

storage components. Proposed optimization techniques, such as hardware pre-fetching, and scalar replacement, further improve system performance.

3. Max-min ant system (MMAS) scheduling algorithms are designed to solve the timing constraint scheduling (TCS) and the resource constraint scheduling (RCS) subject to allocated hardware resources. The MMAS is a multiple-agent meta-heuristics, which is able to solve many combinational optimization problems. Compared to the force-directed scheduler, the MMAS scheduler stably generates area-efficient schedules in TCS. Compared to the list scheduler, the MMAS scheduler generates faster schedulers in RCS. Compared to known optima of some test cases, the MMAS achieves the optima, or generates quantitatively closer results.

4. The MMAS concurrent resource allocation and scheduling (CRAAS) algorithm extends the MMAS scheduling algorithm to explore the design space of actual hardware designs. With few assumptions and limitations, this MMAS CRAAS algorithm supports operation chaining, multiple binding, multiple resource types, pipelined resources, design pipelining, speculative execution, and mutually exclusive sharing. Compared with algorithms in the literature, experimental results show 5 to 20 percent smaller area depending on different design types and optimization goals.

This dissertation does not address other issues in design flows of synthesizing reconfigurable computing systems. Examples include system partitioning, software generation, technology mapping, placement, and routing. The MMAS optimization can resolve some of these problems,

such as software/hardware partitioning [125], multiple-way system partitioning [126], and design space exploration [127]. The others, such as software generation, technology mapping, placement, and routing, are highly architecture-specific, and require knowledge of specific reconfigurable architectures. Because this dissertation focuses on design problems common to most reconfigurable architectures, these problems are out of the range of this dissertation.

1.3 Outline of the Dissertation

The introductory part of this dissertation continues with Chapter 2, which characterizes reconfigurable architectures in the literature and presents designs flows of synthesizing reconfigurable systems. Chapter 3 presents our synthesis framework based on the PDG+SSA form.

The main algorithmic contributions of this research are presented in the next three chapters. Chapter 4 presents the approach partitioning iteration and data spaces and determining storage assignment of data arrays. Chapter 5 presents the MMAS scheduling algorithms using allocated hardware resources. The MMAS CRAAS algorithm, which resolves the actual hardware design problems, is presented in Chapter 6. Finally, Chapter 7 concludes this research and proposes avenues of further research.

Chapter 2

Reconfigurable Computing

In this chapter, we begin with a high-level review of current electronic system designs and describe the gap between the flexibility of ASICs and the performance of general-purpose processors. Based on the granularity of processing elements, reconfigurable architectures are categorized into coarse-grained architectures and fine-grained architectures. We introduce both categories and various typical reconfigurable systems. We present design flows of synthesizing system specifications into reconfigurable computing systems. Much of the discussion takes fine-grained architecture as a basis. Finally, we identify challenges in synthesizing applications into reconfigurable computing systems.

2.1 Computing Systems Design

Researchers and developers are constantly enlarging the application space and improving the performance of computing machinery. Some ex-

amples are weather prediction and nuclear explosion simulation in scientific computations, DNA sequence matching in bioinformatics, high-speed media applications over mobile networks, high-speed switching and routing, and so forth.

There are two ways to fulfill the increased demand of computation tasks. One is to utilize a general-purpose computing platform. Though most computations can fit in a general-purpose processor (GPP), the performance is not satisfactory. The other way is to construct an application-specific integrated circuit (ASIC) to perform such computation tasks. Because all functions are committed during fabrication, the flexibility is tightly constrained.

With rapid advances in semiconductor technology, device geometries are exponentially shrinking. Both ASIC design and microprocessor evolution encounter great challenges.

2.1.1 ASIC design challenges

Over the past five decades, semiconductor technology experienced an unprecedented rapid improvement. The capabilities and performance of integrated circuits grew exponentially [87]. The international technology roadmap for semiconductors (ITRS) [109] estimates that, in ten years (the year 2018), the minimum feature sizes of ASICs are going to be 18nm and the on-chip clock frequency will reach 32 GHz.

However, continued device scaling cannot grow as straightforward as it has been in the past, and there are more challenges that researchers

cannot get around in the design process. First, *system complexity* is constantly increasing when more and more silicon resources are available for designers. It is impossible to handle such complicated designs without clear abstraction and specification, automatic design space exploration, efficient behavioral synthesis, and early performance estimation.

Static and fixed ASIC designs limit the flexibility of the computing devices. It is necessary to synthesize all possible cases, but rarely executed function units occupy considerable resources. In addition, the technology process of ASICs determined that there is no way to exploit adaptability and reconfigurability.

2.1.2 Microprocessor evolution

With the rapid development of semiconductor technology, faster and more plentiful transistors are integrated on one chip. The first microprocessor, the Intel 4004, which arrived in 1971, operates at a frequency of 108 KHz and contains 2,300 transistors. To date, the Intel Xeon processors 7000 sequence operates at a frequency of up to 3.5GHz, and contains up to four processor cores and hundreds of millions of transistors. Intel expects the growth to continue at least through the end of this decade. In order to utilize these transistors, computer architects applied a number of techniques to improve the performance of microprocessors [61, 95]. To increase the processing throughput, *pipelining* overlaps micro-operations of executing different instructions.

Existing research work applies a number of techniques to exploit

instruction-level parallelism (ILP). With multiple functional units, *super-scalar processors* may issue more than one instructions per cycle when such execution does not violate program dependencies. *Dynamic scheduling* with *register renaming* solves data hazards and stalls from anti-dependencies and output dependencies, which enables out-of-order completion. To increase overlapping between blocks of instructions, *speculative execution* allows executing some instructions in branches prior to determining the selected paths given that such execution can be harmlessly reverted. If the prediction is incorrect, the system discards those speculatively executed results.

Computer scientists also exploited program behavior, and developed new techniques. Fundamental observations directed the design of micro-architecture, such as the *principle of locality*. To match the speed of logic and memory, cache memory is integrated on chip, and multilevel caches are applied in the present micro-architectures. For instance, the Intel Xeon processor integrates three-level cache architecture. L1 cache uses trace cache to store a trace of executed decoded instructions, which benefits instruction fetch performance. The Intel Xeon processor also integrates 512 KB L2 cache and up to 2 MB L3 cache.

In the past thirty years, the industry continued to release more powerful microprocessors to improve their performance in general-purpose computation. However, the evolution of microprocessors is limited by current semiconductor technology [102]. Skyrocketing *cost* of CMOS fabrication, continually increasing *power* consumption, and intractable *complexity* of

billions of transistors hinder microprocessors following Moore's law.

The most importantly fundamental attributes in the current micro-architecture, such as sequential instruction fetching and decoding, fixed control, and limited memory accesses, limited the performance of micro-processors.

2.2 Reconfigurable Architectures

Reconfigurable computing is a novel computing paradigm, which combines the flexibility of software with the high performance of hardware, bridges the gap between general-purpose processors and application-specific systems, and enables higher productivity and shorter time to market [14].

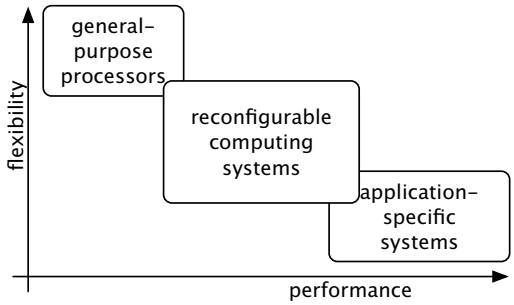


Figure 2.1: Reconfigurable computing bridges the gap between microprocessors and ASICs

A reconfigurable computing system is made of an array of *reconfigurable* processing elements (PEs) and an optional general-purpose processor [39]. The standard processor would control the behavior of these processor elements, i.e. load correct configurations to processor elements at

the correct time to conduct the desired tasks. Processor elements perform all sorts of computation.

The flexibility of reconfigurable computing systems comes from not only their configurable processor elements but also their programmable interconnects. Programmable interconnects connect these processor elements together and steer inputs, outputs, and intermediate results to the correct places. A reconfigurable architecture with adequate interconnects would greatly improve flexibility and increase resource utilizations, and therefore provide superior performance.

The granularity of the reconfigurable architecture is defined as the width of the smallest PE. Based on the granularity, reconfigurable architectures can be categorized as coarse-grained reconfigurable architectures, and fine grained reconfigurable architectures. These two kinds of reconfigurable architectures are further discussed in Sections 2.2.2 and 2.2.1

A reconfigurable system may be configured at deployment time, between execution phases, or during execution [14]. During the configuration process, the system loads configuration files, addresses each PE and programmable interconnects, and writes the new configuration. Partial reconfiguration allows part of the system to be configured, which may reduce the configuration time and keep other portions of the system working. The rate of configuration is proportional to the amount of devices to be programmed. Therefore, it may take less time to configure a coarse-grained reconfigurable system than a fine-grained reconfigurable system.

2.2.1 Fine-grained reconfigurable architectures

Fine-grained reconfigurable computing systems have evolved from field-programmable gate arrays (FPGAs) [14, 103]. FPGAs and other kinds of programmable logic devices (PLDs) provide the initial programmable computation abilities. These devices are mainly used during prototyping or other low-volume scenarios in design and manufacture of ASICs [16]. However, with the fast advances of semiconductor technology, the capabilities and performance of FPGAs have improved dramatically. With SRAM-based FPGAs' infinite reconfiguration ability, it is possible to build dynamic reconfigurable computing systems.

Field-programmable gate arrays

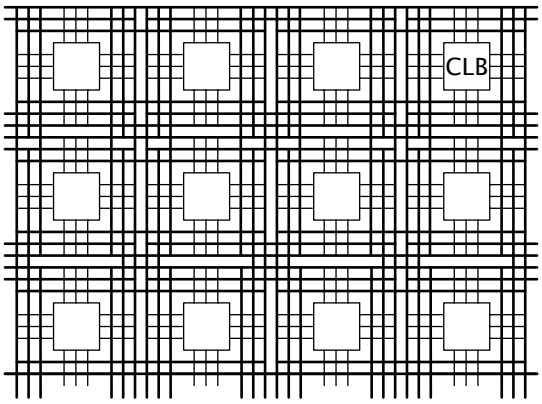


Figure 2.2: An FPGA

An FPGA, as shown in Figure 2.2, consists of an array of lookup tables (LUTs), flip-flops, and programmable interconnect cells. A LUT is a 1-bit-wide memory array. The memory address lines are the LUT input, and the 1-bit memory output is the LUT output. A $2^n \times 1$-bit LUT is programmed

to act as an arbitrary n-input logic function. A fixed number of LUTs and flip-flops are grouped and interconnected with a fixed pattern to form a logic blocks. Programmable interconnects connect these logic blocks to provide the required functionality. When synthesizing arbitrary netlists into FPGAs, the FPGA tools map netlists into LUTs and flip-flops, and bind them with logic blocks. In the physical design phase, proper tools are used to place and route interconnects, and generate configuration files.

However, a performance gap still exists between FPGAs and ASICs. Designs implemented purely using the LUT-based logic elements in an FPGA are approximately 35 times larger and between 3.4 to 4.6 times slower on average than an ASIC implementation [76].

In order to narrow this gap, vendors integrate more components on advanced FPGAs, such as general-purpose processor cores, digital signal processors, dedicated ASICs, and block RAM modules. For example, the Xilinx Virtex-II Pro Platform FPGA [132] provides from 3K to 125K logic cells, coupling with up to four PowerPC processor cores. It also embeds up to 24 multi-gigabit transceivers and hundreds of 18×18 multipliers. Embedded processors may have the capability to do run-time tailoring of applications, and other embedded components can provide better performance, such as dedicated transceivers and multipliers for digital signal processing.

FPGA-based architectures

A number of FPGA-based reconfigurable architectures have been implemented in recent years. Some examples include the Cray XD1, the SRC IMPLICIT+EXPLICIT architecture, and the SGI reconfigurable application specific computing (RASC) platform. In these systems, FPGAs are typically used as accelerator platforms, and couple with other main processors. Main processors control the configuration and execution of FPGAs and handle some rare situations.

The CRAY XD1 system is based on the direct connected processor architectures. Each chassis integrates 12 AMD Opteron processors and 6 Xilinx Virtex-4 FPGAs. RapidArray, a fast-embedded switching fabric, interconnects these FPGAs and processors. Besides the amazing performance of up to 106GFLOPS per chassis, the FPGAs are used as acceleration co-processors to provide massive parallel execution of critical algorithm components, and promise orders of magnitude improvement for target applications.

The SGI RASC platform provides a cost-effective low-power alternative to computer clusters based on general-purpose processors. A SGI RC100 RASC system integrates two Xilinx Virtex-4 FPGAs and interconnects them using a fast system switch fabric. The configuration can be further extended up to 256 FPGAs. SGI claims an average of 18 times speed-up compared with traditional pure software solutions.

2.2.2 Coarse-grained reconfigurable architectures

In contrast to LUTs in FPGA-based architectures, coarse-grained architectures typically consist of reconfigurable PEs with datapaths wider than 4 bits. Based on the arrangement and interconnects among those PEs, coarse-grained reconfigurable architectures are categorized as mesh-based architectures, linear architectures, and crossbar-based architectures. The remainder of this section introduces these architectures and some typical designs.

Mesh-based architectures

In mesh-based architectures, PEs are arranged in a rectangular 2-D array. Adjacent PEs are connected by horizontal and vertical programmable interconnects. In order to avoid using precious PEs to relay data, segmented buses are used to support communications over longer distances.

The KressArray family [73, 56] covers a wide but generic variety of interconnect resources and functional units in coarse-grained reconfigurable architectures. As shown in Figure 2.3, a KressArray chip consists of a mesh of PEs, a.k.a. rDPUs (reconfigurable datapath units), which are connected to their four nearest neighbors by bidirectional links. Beside the nearest neighbor connects, a background communication architecture with a global bus provides additional communication resources.

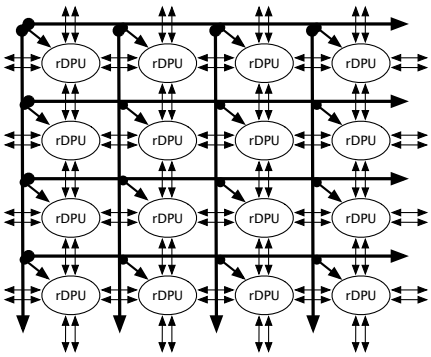


Figure 2.3: A KressArray processor

RAW (Reconfigurable architecture workstation) [118, 124] uses a scalable instruction set architecture (ISA) to provide a parallel software interface to the computing resources of a chip. A RAW chip, as shown in Figure 2.4, contains multiple identical PEs. Each PE consists of an in-order single-issue MIPS processor, a pipelined floating-point unit, data cache, and instruction cache. To connect other tiles and off-chip resources, each tile also contains one static router and two dynamic routers.

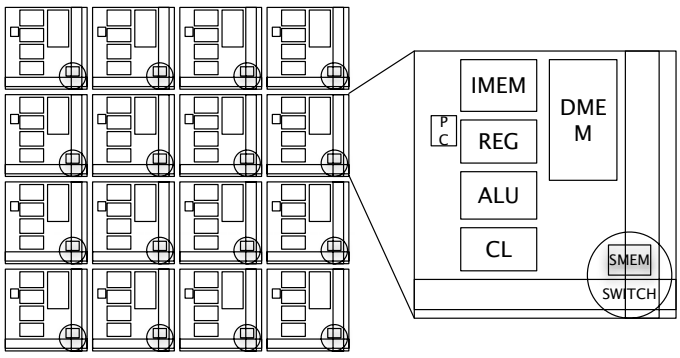


Figure 2.4: A RAW processor

The Garp architecture [57, 21] combines a single-issue MIPS processor core with reconfigurable hardware as a coprocessor. The Garp co-

processor is a two-dimensional reconfigurable array of configurable logic blocks (CLBs). This array is connected with MIPS processor, data cache, and cache using a crossbar.

Chameleon is a coarse-grained reconfigurable system [117]. Chameleon targets to telecommunications and data communications, coupling with a 32-bit RISC processor core. The reconfigurable fabric connects with 84 32-bit ALUs, 24 multipliers, and a distributed memory hierarchy. They can be configured to implemented algorithms with specific word-widths. These systems have rather good performances for specific applications and support faster configuration, but their flexibilities are restricted by the abilities of processor cores and the limited interconnect.

Linear architectures

Linear architectures aim typically at the speed-up of highly regular and computation-intensive tasks by deep pipelining these tasks on its linear array of PEs. In linear architectures, PEs are arranged in a 1-D array. Adjacent PEs are connected by programmable interconnects and segmented buses support communications over longer distances.

RaPiD (Reconfigurable pipelined datapath)[35] is a linear array of PEs, which is configured to form a linear computational pipeline, as shown in Figure 2.5. Each PE comprises an integer multiplier, three integer ALUs,

some general-purpose data registers, and three small RAMs. A typical RaPiD contains up to 32 such PEs. These PEs are interconnected using a set of segmented buses that run the length of the linear array. The buses are segmented into different lengths and placed in different tracks.

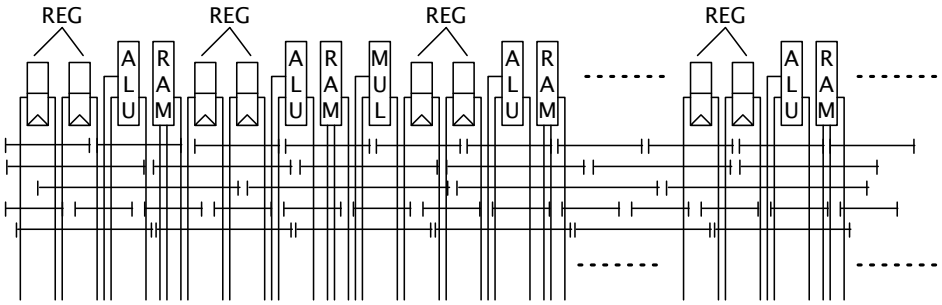


Figure 2.5: A RaPiD processor

PipeRench is an attached reconfigurable co-processor for pipelined applications. PipeRench contains a set of physical pipeline stages called stripes. Each stripe has an interconnection network and a set of PEs. Each PE contains an arithmetic logic unit and a pass register file. Each ALU contains lookup tables (LUTs) and extra circuitry for carry chains, zero detection, and so on. Through the interconnection network, PEs can access operands from registered outputs of the previous stripe, as well as registered or unregistered outputs of the other PEs in the same stripe. Moreover, the PEs access global I/O buses.

Crossbar-based architectures

Some reconfigurable architectures use full crossbar switches to connect PEs. Crossbar switches provide the most powerful communication

network.

PADDI (Programmable arithmetic device for DSP) [25] consists of eight arithmetic execution units (EXUs). Each EXU is 16-bits wide, with a small SRAM, and connected to a central crossbar switch box. A crossbar network interconnects the EXUs to ensure flexible and high-bandwidth data routing. PADDI-2 [115, 134] integrates 48 EXUs. These EXUs are grouped into 12 clusters, and each cluster has four EXUs. A restricted two-level interconnect connect provide intra-cluster and inter-cluster data routing.

To summarize, coarse-grained reconfigurable architectures provide word-level datapaths and area-efficient routing switches. Some of them support multi-granular by grouping several PEs to support wider arithmetic operations. However, most coarse-grained architectures aim at the speed-up of highly regular and computation-intensive applications. Compared with FPGA-based architectures, it is very hard to map arbitrary applications on these architectures.

2.2.3 Characteristics of reconfigurable architectures

These reconfigurable architectures distinguish themselves from traditional computing systems by their parallel computation, distributed control, superior performance, flexibility, and improved reliability.

Reconfigurable architectures consist of an array of processor cores or

an array of configurable logic blocks. Compared with software solutions, an operation can be executed whenever its operands are ready since these processing blocks are configured to arbitrary functions. Operations do not need to be serialized in the instruction queue during execution. All functional units work in parallel. Because of same reason, reconfigurable computer systems are not based on the instruction set architecture. There is no control to issue instructions and write back results. All functional units are controlled by local configuration. In addition, intermediate results are spatially distributed over the systems.

DeHon [32] showed that FPGA-based systems achieve much greater performance per unit of silicon area compared to processors. Filters implemented on Xilinx or Altera components outperform digital signal processing by one or two orders of magnitude. FPGA-based systems execute DNA sequence matching two orders of magnitude faster than supercomputers and three orders of magnitude faster than workstations.

Compared with the application-specific systems, reconfigurable systems have superior flexibility due to their reconfigurability. At the same time, these systems provide great reliability because these processing blocks are redundant. If some failed, applications can still be mapped using available processing blocks and avoiding failed units.

Reconfigurable systems have very important usages in critical systems where it is hard to replace or repair them due to the natures of tasks and the tremendous cost. For example, Xilinx FPGAs are utilized to provide the Australian FedSat satellite and NASA's Mars Exploration Rover pow-

erful reconfigurable computing capabilities [11, 133]. Reconfigurable systems in satellites can be reconfigured by either remote commands or its own internal operations, and may be used in disaster detection systems, satellite autonomous navigation systems, and so forth. It is well known that the entire satellite may be lost through tiny failures in computing systems. Because of its reconfigurability, the satellite system may be reconfigured remotely to avoid catastrophic failures, and it can be reconfigured for new purposes.

To summarize, reconfigurable architectures combine the flexibility of software with the high performance of hardware, which enable them as practical solutions bridging the gap between general-purpose processors and application-specific systems.

2.3 Design Flows

This section presents design flows from system specifications to synthesized reconfigurable computing systems. As shown in Figure 2.6, a typical design flow consists of specification, parallelizing compilation, partitioning, hardware synthesis, technology mapping, software generation, verification, and other related components.

This design flow is an iterative process, which performs transformations and optimizations for the specified target architectures, conducts system partitioning, generates hardware description and software codes, and verifies the synthesized designs, until meeting the required design

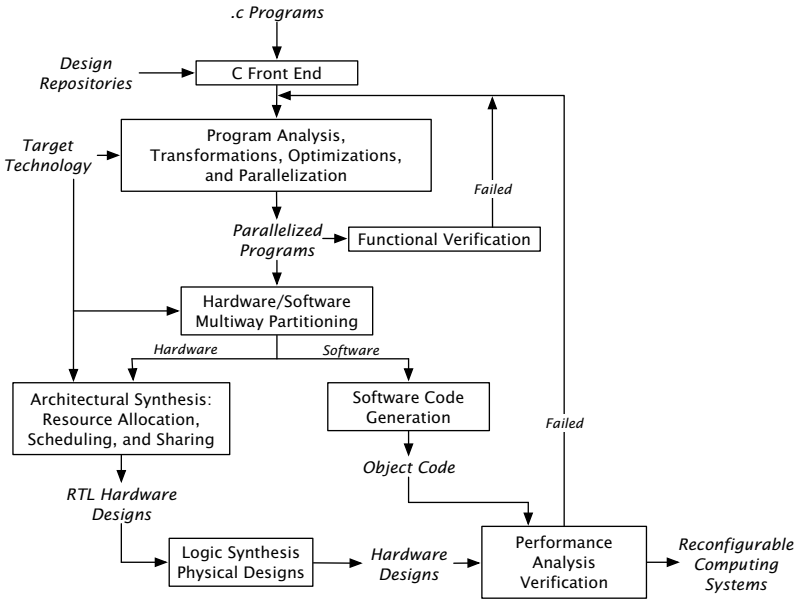


Figure 2.6: A design flow of synthesizing reconfigurable computing systems

goals.

2.3.1 System specification

System specifications are representations that capture all aspects of a reconfigurable computing system, which include *architectural specifications*, *functional specifications*, and *performance*. The architectural specification is the target technology used to implement the system. Architectural specifications normally include what processing abilities those processor cores/configurable logic blocks have, how many cores/blocks are available in the system, how these cores/blocks are interconnected, what kind of control the attached general proposed processors can provide, and how the memory hierarchies are organized.

Functional specifications define the computation tasks conducted on inputs to generate the expected outputs. In current system designs, hierarchical functional specifications are usually adopted. At the higher level, non-executable models are used to decompose the system into subsystems, and represents communications between different components in this system.

While at the lower level, executable models are used to capture more details of the functionality that the system implements. Executable models benefit system designs since these kinds of models enable simulations and early verifications. Most executable models are specified in high-level programming languages or specific languages used in popular industrial tools. For example, the Garp compiler accepts standard ANSI C programs as inputs. Other projects are the RAWCC, the C compiler for RAW processors, and CASH [18], a compiler framework to compile and synthesize C programs into reconfigurable fabrics. Some designs tools start from extended C programming languages with explicit parallelism marks. They are Handel-C, RaPiD-C, and several others. Matlab is another popular language used in functional specifications. Synplicity Synplify DSP starts from Matlab programs. In this work, we mainly focused on the sequential languages, such as the C programming language.

Performance specifications define the expected quality of the synthesized design, which include one or more of the latency, the throughput, and the expected clock frequencies.

2.3.2 Compilation, transformation, and optimization

In this stage, the synthesizer parses and analyzes the input programs, performs common transformations and optimizations, and exploits parallelism at different levels.

Those programming languages specifying functionalities are sequential, but most reconfigurable architectures are parallel computing architectures. Before scheduling and assigning each operation to a functional unit, parallelization compilers are used to exploit as much parallelism as possible. A parallelization compiler accepts intermediate program representations produced in the front end, and generates parallelized program representations. In the literature of parallelizing compilation, a number of tests and transformations [5] are designed to enhance fine-grained parallelism and create coarse-grained parallelism. The parallelization compiler creates coarse-grained parallelism, where the original program is partitioned to a number of threads and these threads can be parallelized and executed on different processing blocks. On the other side, fine-grained parallelism, including instruction-level parallelism, is also explored in order to execute more than one operation in the same program portion on different functional units at the same time. Normally the larger a program portion, the more fine-grained parallelism exists.

In order to accomplish these tasks, a compiler framework is required. The SUIF and Machine SUIF is a very popular choice in academia. The SUIF compiler is a compiler framework dedicated to research in paral-

lelizing compiler. It converts C programs into *abstract syntax trees* (ASTs), and perform transformations to exploit parallelism [54]. Supported analysis and optimizations include array analysis, scalar optimizations, interprocedural analysis, and so forth. Machine SUIF, developed at Harvard [112], conducts further optimizations on results obtained from SUIF compilation. Machine SUIF is a flexible, extensible framework for constructing compile back-ends. This framework performs optimizations orienting particular computer architectures. In most research projects in high-level synthesis, it is used to convert the SUIF syntax trees into *control flow graphs* (CFGs) or the *static single-assignment* (SSA) form [65, 62], and generate object code for embedded microprocessors. Machine SUIF also supports control flow analysis and bit-vector data-flow analysis [64, 63].

2.3.3 System partitioning

The transformed and optimized programs should be partitioned by the synthesizer. More specifically, parallelized programs obtained from the parallelizing compilation are partitioned into small portions, and each piece is scheduled to execute on programmable logic or one of the embedded processors in a specified period. The synthesis tool also constructs a proper memory hierarchy. Those portions assigned to programmable logic are be further synthesized to netlist, and those portions assigned to processors are converted to object code.

This partitioning process can be conducted manually by designers. For example, Celoxica's design tool starts from Handel-C programs, and de-

signers need to specify partitioning and parallelism in their programs.

This partitioning process can be driven automatically by performance, such as latencies, or areas of the synthesized designs. Depending on the target architecture, this partitioning process can be categorized as bi-partitioning or multi-way partitioning. A single program can be partitioned and mapped onto one or more processor cores, a number of processor elements, and sometimes dedicated data processing hardware.

2.3.4 Software generation

If some portions of the system are assigned to processors, software object code should be generated. Software generation is usually performed by utilizing a compiler back-end for the specified processor architectures, such as PowerPC or ARM. The size of the generated object code is constrained by available size of instruction memory.

2.3.5 Hardware synthesis

Hardware synthesis refers to constructing the *macroscopic* structure of a digital circuit [31]. This phase is specifically required by fine-grained architectures in order to implement arbitrary functionalities. Processor elements in coarse-grained architectures can only implement supported operations.

The result of hardware synthesis is usually a control unit, and a structural view of data-paths, including functional units, interconnects, and

storage components. Hardware synthesis starts from tasks discussed in traditional high-level synthesis, including resource allocation, scheduling, sharing, and so forth. Synthesized results are usually specified in register-transfer level (RTL) hardware descriptions.

2.3.6 Technology mapping

In fine-grained architectures, technology mappers generate *netlist* from RTL descriptions, conduct placement and routing, and then generate *configuration data*. Logic synthesis is applied to map the netlists to configurable logic blocks. Placement tries to minimize the number of wires in each channel and the length of wires. Routing connects configurable logic blocks using limited channel resources. These tasks are similar to corresponding tasks in traditional FPGA designs except target architectures may be arrays of reconfigurable processor elements.

Technology mapping is mostly simpler for coarse-grained architectures than for FPGAs. Direct mapping approaches map operators straightforward onto processor elements, with one PE for one operator. Sometimes technology libraries are required for functions not directly implementable by a single PE. Placement and routing for coarse-grained architectures are normally done at the same time. Depending on the structures of processor elements, placement and routing sometimes are integrated to the technology mapping phase.

Technology mapping is highly dependent on structure and granularity of reconfigurable architectures. However, most of them are based on com-

mon optimization techniques, such as simulated annealing and genetic algorithms.

Compared to traditional VLSI designs, hardware synthesis and technology mapping in reconfigurable designs also exploit reconfigurability, such as minimizing the differences between different configuration files, and constrain configuration files in given blocks.

2.3.7 Performance analysis and verification

The feasibility and performance of reconfigurable computing systems are determined by its physical attributes, such as area and power consumption. When these issues are considered in the higher level, there is a larger optimization space, and it is more probable to obtain better designs. Hence, these issues should be addressed from the architectural synthesis stage.

In order to guarantee that the synthesized designs perform similarly as the system specifications and achieve design goals, the synthesized designs should be verified, which could be implemented by applying formal methods to prove the synthesized designs' functionally equal to the specified programs. The synthesized designs could also be verified by simulation. Execution results of the synthesized designs are compared with results from the system specifications and intermediate results.

To summarize, design flows for reconfigurable computing systems involve a number of different research topics, and provide a huge research room. However, our work focuses on parallelizing compilation and archi-

tectural synthesis.

2.4 Challenges in Synthesizing Applications to Reconfigurable Architectures

Reconfigurable computing systems provide flexibility of general-purpose processors and accelerate executions of application-specific systems. Advancement in parallelization compilers and electronic design automation make it possible to design complex reconfigurable computing systems. However, designers still face a number of challenges when mapping applications to reconfigurable architectures. In general, these challenges are mainly on improving the system performance and resource utilization, reducing interferences from designers, and automatically synthesizing complicated designs.

As discussed before, reconfigurable architectures have an array of processor cores or configurable logic blocks. In order to effectively and efficiently utilize these resources, more coarse-grained parallelism should be created, and better fine-grained parallelism, including the instruction-level parallelism, should be explored; then data storage needs to be carefully arranged. Issues in parallelization compilers, especially those transformations and optimizations towards the specific reconfigurable architectures, should be addressed.

According to Moore's law, the number of components per integrated function increases exponentially. Sizes of applications increase dramati-

cally as well. Complexity of computing systems becomes unmanageable. At the same time, design tools become more and more complex. Compilation and synthesis tools take a long time in the order of hours to days. It is harder and harder to generate the optimal designs. Therefore, how to design *heuristic algorithms* for traditional design problems, which consistently generate good results, is another great challenge.

Moreover, one of the most important issues is to reduce the interferences from designers during the process of synthesizing system specifications into reconfigurable computing systems. Because of the huge design space and the complicated design flow, a designer often fails to generate globally better results. If the synthesizer can carefully evaluate candidate solutions, and consider heuristics extracted from existing experiences, a better design is normally generated. Therefore, an automatic synthesizer is very important to the successful adoption of the reconfigurable computing systems.

Chapter 3

Program Representations

A design flow for reconfigurable computing systems conducts parallelizing compilation and reconfigurable hardware synthesis in an integrated framework. The front-end of this framework creates coarse-grained parallelism and exploits fine-grained parallelism in order to utilize limited hardware resources in an effective and efficient manner. The ability to accomplish these tasks are heavily relied upon by the program representations used in the framework.

It is believed that a common application representation is needed to tame the complexity of mapping an application to state-of-the-art reconfigurable systems. This representation must be able to generate code for any microprocessors in the reconfigurable systems. Additionally, it must easily translate into a bitstream to program the configurable logic array. Furthermore, it must allow a variety of transformations and optimizations in order to exploit the performance of the underlying reconfigurable

architecture.

In this chapter, we use the program dependence graph (PDG) with the static single-assignment (SSA) extension as a representation for the synthesis framework. The PDG+SSA representation can be synthesized to software object code or reconfigurable hardware. We begin with an introduction to program representations in the literature. In Section 3.2, we present the basic idea of the PDG, and show how the PDG is extended to a program representation good for hardware synthesis in Section 3.3. In Sections 3.4 and 3.4.2, we describes the synthesis of the PDG+SSA representation to a configurable logic array, and experimental results. Finally, we summarize our work in the program representation and hardware synthesis.

3.1 Common Program Representations

Wide varieties of program representations have been presented in the past two decades. The rest of this section discusses several program representations used in different design environments, including the abstract syntax tree (AST) [2, 88], the control-flow graph (CFG) [2, 59], and the Predicated Static Single-Assignment (PSSA) form [23]. Section 3.3.1 particularly describes the Program Dependence Graph (PDG) [41]. This program representation and its variants promise to better exploit both coarse- and fine-grained parallelism, and optimize memory and communications for complex reconfigurable systems.

3.1.1 Abstract syntax tree

The AST is a high-level IR which is produced by the compiler front end and retained in the original structure. Each AST node represents an operation, and its children represent the operands [2]. Most non-terminal symbols are removed when constructing an AST from the parse tree. AST, along with a symbol table, stores all necessary information for reconstruction, such as variable declarations; types of operations; and controls, like loops and branches. Because AST are sensitive to source code and easy to build, it is widely used in parallelizing compilers.

3.1.2 Control flow graph

The CFG is the traditional program representation used in high-level synthesis. Many research projects perform transformations on CFGs, and generate VHDL programs. As mentioned before, Machine SUIF can generate CFGs from SUIF IR [65].

A CFG is a directed graph that expresses the control flow in a given procedure. Each node in a CFG is a *basic block*. A basic block is a sequential list of instructions. There is only one control-transfer instruction in the instruction list of a basic block. Other instructions are arithmetic/-logic instructions. If control can potentially transfer from block i to block j, there is an edge (i, j) from block i to block j. In a structured program, each CFG contains only one *entry* node, and possibly more than one *exit* node.

The CFG enables some transformations and optimizations, such as *unreachable code elimination*. Any nodes in a CFG that cannot be reached from the *entry* node can be removed from the graph. However, without further flow-analysis and dependence analysis, it is difficult to detect coarse-grained parallelism. Moreover, the basic-block is too small to exploit instruction-level parallelism. Other main drawbacks of the CFG include that the CFG is not a hierarchical structure: when the design grows, it is difficult to handle complexity. It is also difficult to do flow-sensitive interprocedural analysis.

3.1.3 Static single-assignment form

The SSA form [6, 104] is an intermediate representation in the context of data-flow analysis. In the SSA form of a procedure, each variable can have only one assignment, and whenever this variable is used, it is referenced using the same name. Hence, the *def-use* chains are explicitly expressed. At joint points of a CFG, special ϕ nodes need to be inserted.

Using an SSA form, some optimizations can be easily performed, and the compiler can detect more ILP since the SSA form successfully removes the false data dependence in a CFG. Cytron *et al* [28] presented an efficient algorithm to build the SSA form, and Briggs *et al* further improved the construction algorithm [15]. Machine SUIF can translate CFGs into or out of the SSA form [62] based on algorithms by Briggs *et al*

3.1.4 The Predicated Static Single-Assignment Form

The Predicated Static Single-Assignment (PSSA) form, introduced by Carter *et al* [23], is based on the static single-assignment (SSA) form. The PSSA form is a predicate-sensitive implementation of SSA. This program representation is also based on the notion of *hyperblock* [82], in which there are no cyclic control- and data-flow dependencies. In addition to assigning each target of assignment a unique name, PSSA summarizes predicate conditions at points where multiple control paths join together to indicate which value should be committed at these joint points.

After transforming to PSSA form, all basic blocks in a hyperblock are labeled with full-path predicates, which enable aggressive predicated speculation, and reduce control height.

3.1.5 Hyperblock

As Lam and Wilson [77] suggested, executing multiple flows of control and speculative execution are helpful to relaxing limits of control flow on parallelism. To leverage multiple data-paths and functional units in superscalar and VLIW processors, Mahlke *et al* [82] presented their compilation techniques on support predicated execution using the *hyperblock*.

A hyperblock, by definition, is a set of predicated basic blocks in which control can only enter from the top, but may exit from one or more locations. One hyperblock usually contains multiple paths of control, which are formed using *if-conversion* [5] based on their execution frequencies

and sizes. The maximum possible size of a hyperblock usually is the size of the innermost loop body, and outer loops span multiple hyperblocks.

Hyperblock effectively enlarges the optimization unit from basic blocks to hyperblocks, and are suitable for speculative execution. With the support of superscalar and VLIW architectures, it gains speed-up on branch execution. However, this technique may be very slow when taking rare execution paths. In addition, the gain of predicated execution may greatly depend on which basic blocks are selected to form hyperblocks.

In reconfigurable system designs, those design environments using hyperblock techniques are mainly focused on revealing ILP, such as the compiler for the Garp architecture [22].

Early research on reconfigurable systems focused mainly on reconfigurable architectures and did not put too mach efforts in synthesis work, such as RaPiD. Some other projects target particular applications, such as the Cameron project for image processing [55]; those compilers use their own programming languages, and exploit parallelism and reconfigurability.

3.1.6 Summary of common program representations

To summarize, this section described parallelizing compilation techniques, especially those program representations used in different design environments for reconfigurable computing systems. Commonly used program representations are SUIF, CFG, PSSA, PDG, and variants of PDG. Experiences from previous research results taught us that different pro-

gram representations support different transformations in parallelizing compilers, and it is necessary to utilize the right IR in different stages.

- The AST retains program semantics and supports high-level transformations to enhancing fine-grained parallelism. It does not have knowledge on target architecture, and hence cannot support low-level transformations.

- The CFG presents the AST in a directed graph, and expresses control flow between basic blocks. Combined with the SSA form, a number of synthesizing compilers start optimizations from this point. However, the CFG cannot support low-level transformations either.

- Predicted execution is an important technique to exploit ILP in modern architectures. The PSSA form and hyperblocks are medium-level IR for exploiting non-loop parallelism.

- The PDG uniformly expresses both control- and data-dependencies, which enable it for both high-level transformations and low-level transformations. Hence, the PDG can create both fine- and coarse-grained parallelism. The most important point is that architecture constraints can be integrated in the PDG as dependencies, which greatly benefitsf architecture exploration and reconfigurability detection.

3.2 Dependence Graphs

When synthesizing a high-level programming language to reconfigurable devices, dependence analysis and dependence graphs are essential for compilers to exploit both fine- and coarse-grained parallelism. With proper dependence graph representations, more parallelism and optimizations can be achieved.

This section describes a program representation called the Program Dependence Graph (PDG) [41]. Several other similar program representations will also be discussed.

3.2.1 Program dependence graphs

The PDG, developed by Ferrante *et al.*, explicitly expresses both control and data dependencies, and consists of a control dependence subgraph (CDG) and a data-dependence subgraph. The CDG was a novel contribution.

In a PDG, there are four kinds of nodes. They are `ENTRY` , `REGION` , `PREDICTE` , and `STATEMENTS` . The `STATEMENTS` and `PREDICATE` nodes contain arbitrary sequential computations. `PREDICATE` nodes also contain predicate expressions. A `REGION` node summarizes the set of control conditions for a node, and groups all nodes with the same set of control conditions together. An `ENTRY` node is the root node of a PDG. A PDG contains a *unique* `ENTRY` node. This `ENTRY` node can be treated as a special `REGION` node.

Edges in the PDG represent the dependencies. Outgoing edges from a

REGION node group all PREDICATE and STATEMENTS nodes with the same set of control conditions together. An outgoing edge from a PREDICATE node indicates that the STATEMENTS node or the REGION node is control dependent upon the PREDICATE node. The data dependencies are not well defined by Ferrante *et al* [41]. Any data dependence can be put into the PDG.

Ferrante *et al* [41] suggested two possible methods to transform a CFG to a PDG. One is a precise method, and the other is an approximate method based on the notion of hammock. The PDG is built based on control-flow analysis. Using a post-dominator tree, the control dependencies between basic blocks are revealed. Then the compiler inserts REGION nodes to finalize the PDG.

3.2.2 Other dependence graphs

After the PDG was presented, a variety of research was done on it. Horwitz *et al* [66] first showed that the PDG is an adequate structure for representing a program's execution behavior. Wide varieties of different dependence graphs are presented to enhance the PDG, and to incorporate other advanced techniques, like the SSA form.

Horwitz *et al* [67] introduced the system dependence graph (SDG) for interprocedural analysis. The SDG extended the PDG by using edges to express procedure calls and parameter passing. Compared with aggressive in-lining using CFG, the SDG enables more transformations and optimizations since there are not many differences between the SDG and the PDG.

The program dependence web (PDW) [89] is an extension of the PDG and the SSA form. When constructing a PDW, the compiler needs to convert an SSA-form PDG into the gated single assignment (GSA) form, and then convert it into the PDW. The PDW consists of control-, dataflow-, and demand-interpretable program graphs (IPGs). The dataflow IGP was used to generate code for dataflow architectures.

The dependence flow graph [70] extended the SSA form using switch nodes. Single-entry single-exit regions are located in CFG, and then switch nodes and merge nodes are inserted to construct the dependence flow graph. The dependence flow graph is more based on the CFG than it is based on control dependence. Hence, it is difficult to reveal as much parallelism as using the PDG.

The value dependence graph (VDG) [128] is originally a program representation for functional programs. The VDG is very similar to the GSA form, a PDG coupled with the SSA form. The VDG presents value flows. In the code generation stage, this representation is converted to the demand PDG form, which replaces control dependence in the PDG by demand dependence.

There are other variants of the PDG. Most of these variants are natural progressions from the PDG. Compared with the CFG, the PDG replaces control-flow dependence by control dependence. These variants eliminate more control dependence from those nodes whose execution can be determined by data-dependence.

To summarize, benefits gained from the PDG and its variants include

exposition of parallelism, support to reorder the nodes, and simplicity of transformations and optimizations.

3.2.3 Present research activities

Dependence graphs are widely used in parallel compilations. Recently, dependence graphs have been adopted by several projects in high-level synthesis of embedded systems.

Edwards [36, 37] used the PDG as the program representation when compiling Esterel programs into hardware. The Esterel language is an imperative language including concurrency, preemption, and a synchronous model. When compiling an Esterel program into circuits, the compiler first converts a program into an equivalent concurrent control flow graph (CCFG) [79], and generates the CDG from the CCFG. Circuit generation from the CDG is trivial, but generated circuits are compact and better than those generated directly from the CFG [37]. Because this Esterel compiler is mainly focused on large control-dominated systems, it does not need to consider data dependence, and the original CDG can be properly utilized here.

Ramasubramanian, Subramanian, and Pande [101] used the PDG as the program representation to analyze loops in synthesis of reconfigurable systems.

3.2.4 Transformations

The PDG eliminates the artificial linear order in AST or CFG, and exposes only the order specified by control and data dependencies. This inherent advantage promises optimizing transformations and both fine- and coarse-grained parallelism, as well as low-level transformations, which cannot be done based on the AST.

Traditional optimization Ferrante *et al* [41] showed that PDGs support traditional program transformations, such as *common subexpression elimination* and *constant expression folding*. Because PDGs only express data and control dependences, there are more freedom to perform forward/backward code motion.

Fine-grained parallelism Like the AST, the PDG also supports high-level transformations, such as scalar expansion and array renaming, to exploit fine-grained parallelism. [41, 74]. The PDG also enables *node-splitting*, which duplicates the PDG nodes and divides its edges between two copies to break dependence cycles. When looking for opportunities for *loop-interchange* and *vectorization*, the AST requires performing *if-conversion* first, while the PDG can directly perform vectorization *without* doing if-conversion first since the PDG edges express data- and control-dependences uniformly.

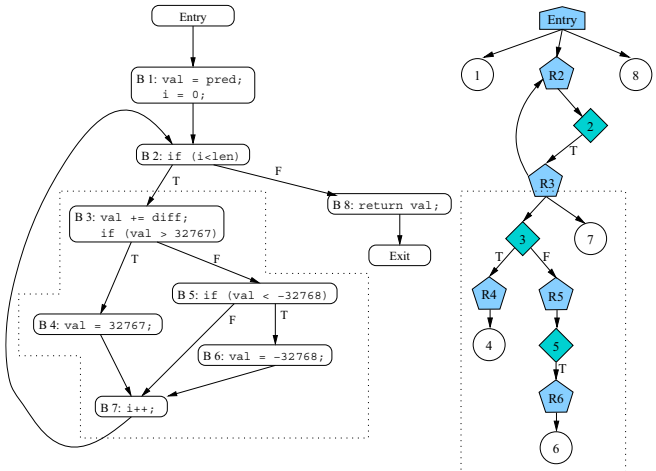
Coarse-grained parallelism When creating coarse-grained parallelism, many trade-offs should be managed upon the target parallel architecture, such as the number of threads/communication and

synchronization overheads [5]. Sarkar [105] presented an automatic partitioning on the PDG, which creates coarse-grained parallelism while eliminating overheads induced by over loop-distribution, and showed that it is particularly important to perform loop transformations in loop nests when the target architecture contains a large number of processors. Gupta and Soffa [50] presented a scheduling technique to redistribute parallelism into the PDG nodes, and obtained better results than trace scheduling used in the CFG.

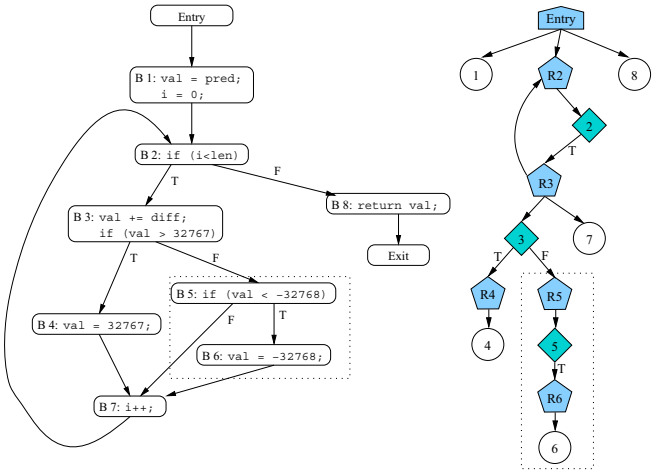
Low-level transformations The PDG distinguishes from the AST and the CFG by its supports to low-level transformations. Low-level transforms are tightly bound with the target architectures, such as computing resources and memory requirements. The PDG `REGION` node can summarize resource usage information as well as control dependence [13]. Though counting machine details induces costly overhead, this attribute is particularly useful when performing architectural exploration and detecting reconfigurability in reconfigurable computing systems.

The dependence graphs are powerful program representations for parallelizing compilers. However, it cannot be utilized solely since the PDG is constructed using dependence analysis based on either AST or CFG. It is necessary to add more low-level dependences to exploit the reconfigurable computing architecture.

Using the PDG for Hyperblocks



(a) A hyperblock containing the inner loop body



(b) A smaller hyperblock

Figure 3.1: The above graphs show that there are multiple ways to form hyperblocks using the PDG

As discussed earlier, the hyperblock is an effective compilation technique to exploit fine-grained parallelism. In the PDG, hyperblocks can be easily represented and manipulated. Figure 3.1 shows that the PDG is flexible enough to represent different hyperblocks.

Theorem: *Given a hyperblock H formed of Blocks* $\{E,N_1,\ldots,N_n\}$*, where Block E is the entry point. In the PDG, all blocks are successors of Node R, which is the immediate* REGION *predecessor of Block E.*

Proof: Following the definition of the hyperblocks, only the entry block has incoming control flows from the outside blocks to the hyperblock. Hence, if the control dependence set of the entry node is CD, then the control dependence set of the other nodes in H is the same as CD or a subset of CD.

Each REGION node in the PDG summarizes control dependence for a PREDICATE/COMPUTE node, and groups together all nodes with the same control conditions. Therefore, the corresponding REGION nodes for Blocks $\{N_1,\ldots,N_n\}$ are either the same one as Node R, the immediate REGION predecessor of Block E, or successors of Node R. Hence, all blocks in H are successors of Node R □

It is also easy to perform optimization of the hyperblock using the PDG. Since the PDG is suitable for both high-level and low-level transformations, it is easier to perform those conventional compiler techniques that the hyperblock supports. Section 3.3.2 also shows that the PDG can perform the speculative execution, hence instruction promotion.

3.3 Generating PDG+SSA from Sequential Programs

This section presents how the PDG is constructed from the CFG, how the PDG is extended with the SSA form, and how the PDG+SSA form is synthesized to reconfigurable hardware.

3.3.1 Constructing the PDG

We use the PDG to represent control dependencies. The PDG uses four kinds of nodes: ENTRY, REGION, PREDICATE, and STATEMENTS. An ENTRY node is the root node of a PDG. A REGION node summarizes a set of control conditions. It is used to group all operations with the same set of control conditions together. The STATEMENTS and PREDICATE nodes contain arbitrary sets of expressions. PREDICATE nodes also contain predicate expressions. Edges in the PDG represent dependencies. An outgoing edge from Node *A* to Node *B* indicates that Node *B* is control dependent on Node *A*.

The PDG can be constructed from the CFG following Ferrante's algorithm [41]. Each node in the PDG has a corresponding node in the CFG. If a node in the CFG produces a predicated value, there is a PREDICATED node in the PDG; otherwise, there is a STATEMENTS node in the PDG.

A post-dominator tree is constructed to determine the control dependencies. Node *A* *postdominates* node *B* when every execution path from *B* to the *exit* includes node *A* [88]. For example, in Figure 3.2, every execu-

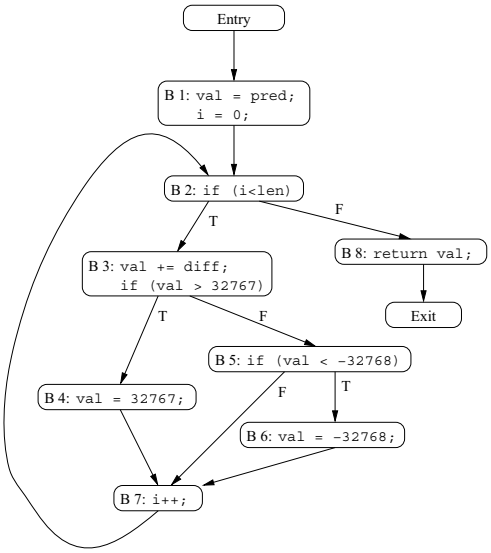
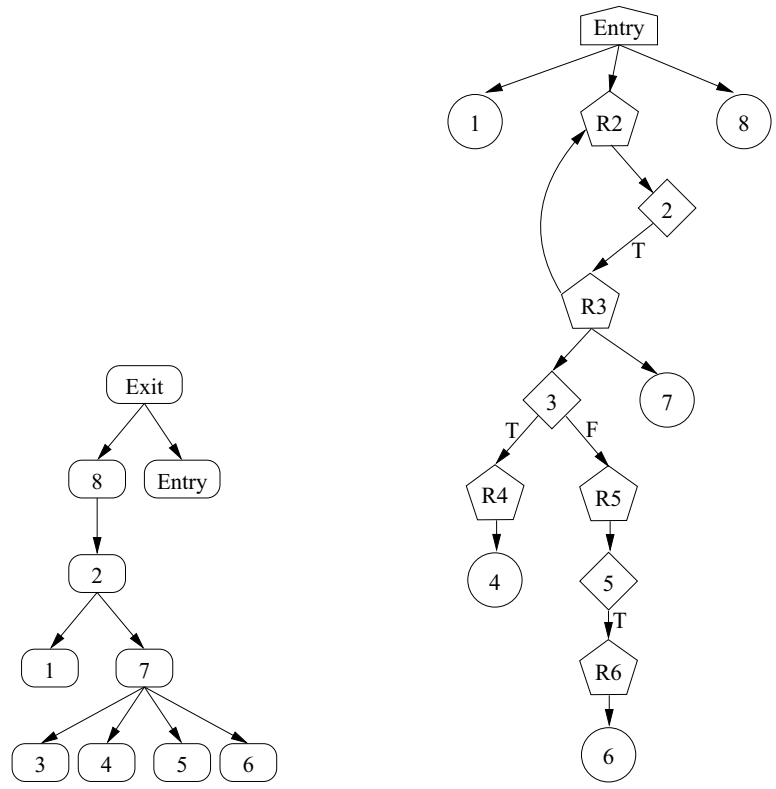


Figure 3.2: The control flow graph of a portion of the ADPCM encoder application.

tion path from $B2$ to the *exit* includes $B8$; therefore, $B8$ post-dominates $B2$, and there is an edge from node 8 to node 2 in the post-dominator tree (see Figure 3.3).

Control dependencies are determined in the following manner: If there is an edge from node S to node T in the CFG, but T does not postdominate S, then the least common ancestor of S and T in the post-dominator tree (node L) is used. L is either S or S's parent. The nodes on the path from L to T are control-dependent on S. For example, there is an edge from node 3 to node 4 in the CFG and node 4 does not postdominate node 3. Thus, node 4 is control-dependent on node 3. Using the same intuition, it can be determined that both nodes 7 and 3 are control-dependent on node 2.

After determining the control dependencies, REGION nodes are inserted into the PDG to group nodes with the same control conditions



(a) Post-dominator tree

(b) Control dependence subgraph

Figure 3.3: The post-dominator tree and the control dependence subgraph of its PDG for the ADPCM encoder example.

together. For example, nodes 3 and 7 are executed under the same control condition $\{2\mathtt{T}\}$. Thus, a node $R3$ is inserted to represent $\{2\mathtt{T}\}$, and both nodes 3 and 7 are children of $R3$. This completes the construction of the *control dependence subgraph* of the PDG (See Figure 3.3).

3.3.2 Incorporating the SSA form

In order to analyze the program and perform optimizations, it is also necessary to determine data dependencies and model them in the repre-

sentation. We incorporate the SSA form into the PDG to represent the data dependencies. We model data dependencies using edges between STATEMENTS and PREDICATE nodes.

```
1 val += diff;
  if (val > 32767)
3       val = 32767;
  else if (val < -32768)
5       val = -32768;
```

(a) Before SSA conversion

```
1 val_2 = val_1 + diff;
  if (val_2 > 32767)
3     val_3 = 32767;
  else if (val_2 < -32768)
5     val_4 = -32768;
  val_5 = phi
7     (val_2,val_3,val_4);
```

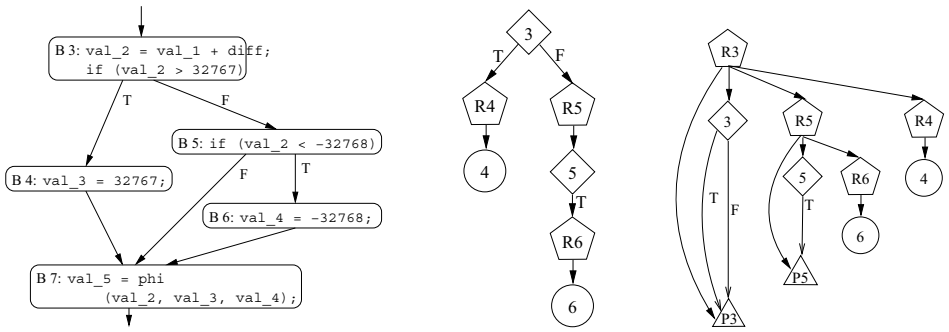
(b) After SSA conversion

Figure 3.4: The ADPCM example before and after SSA conversion

In the SSA form, each variable has exactly one assignment, and it is referenced always using the same name. Thus, it effectively separates the values from the locations where they are stored. At joint points of a CFG, special ϕ nodes are inserted. Figure 3.4 shows an example of the SSA form.

The SSA form is enhanced by summarizing predicate conditions at joint points, and labeling the predicated values for each control edge. This is similar to the PSSA form. In the PSSA form, all operations in a hyper-block are labeled with full-path predicates. This transformation indicates which value should be committed at these join points, enables predicated execution, and reduces control height. For example, in Figure 3.5(a), *val_2* is committed only if the predicate conditions are {3F,5F }.

In order to incorporate the PDG with the SSA form, a ϕ-node is inserted



(a) (b) (c)

Figure 3.5: Extending the PDG with the ϕ-nodes

for each PREDICATE node P in the PDG. Figure 3.5(c) shows that the control dependence subgraph is extended by inserting ϕ-nodes. This ϕ-node has the same control conditions as the PREDICATE node, i.e. this ϕ-node is enabled whenever the PREDICATE node is executed. ϕ-nodes inserted here are not the same as those originally presented in [29]. A ϕ-node contains not only the ϕ-functions to express the possible value, but also the predicated value generated by the PREDICATE node. This determines the definitions that will reach this node. This form is similar to the gated SSA form. However, unlike the gated SSA form, this form does not constrain the number of arguments of the ϕ-nodes. Therefore, we can easily combine two or more such ϕ-nodes together during transformations and optimizations.

After inserting ϕ-nodes, data dependencies are expressed explicitly between STATEMENTS and PREDICATE nodes. Figure 3.6 shows such a graph. Within each node, there is a data-flow graph. Definitions of variables are also connected to ϕ-nodes, if necessary.

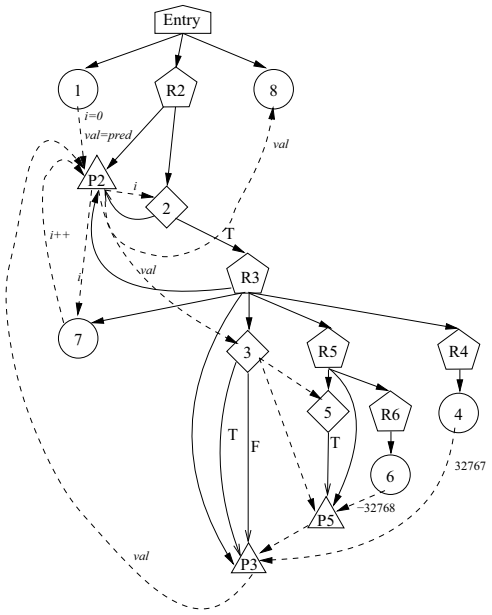


Figure 3.6: A dependence graph, which is converted to benefit speculative execution, shows both control and data dependence. Dashed edges show data-dependence, and solid ones show control-dependence

3.3.3 Loop-independent and loop-carried ϕ-nodes

There are two kinds of ϕ-nodes: *loop-independent* ϕ-nodes, and *loop-carried* ϕ-nodes. A loop-independent ϕ-node takes two or more input values and a predicate value, and, depending on this predicate, commits one of the inputs. These ϕ-nodes remove the predicates from the critical path in some cases, enable speculative execution, and therefore increase parallelism.

A loop-carried ϕ-node takes the initial value and the loop-carried value, and a predicate value. It has two outputs, one to the iteration body, and another to the loop-exit. At the first iteration, it directs the initial values to the iteration body if the predicate value is true. At the following itera-

tions, depending on the predicate, it directs the input values to one of the two outputs. For example, in Figure 3.6, Node $P2$ is a loop-carried ϕ-node. It directs *val* to either $n8$ or $n3$ depending on the predicate value from $n2$. This loop-carried ϕ-node is necessary for implementing loops.

3.3.4 Speculative execution

High-performance representations must support speculative execution. Speculative execution performs operations before the predication is known to execute those operations. In the PDG+SSA representation, this equates to removing control conditions from PREDICATE nodes. Consider the control dependence from Node 3 to R5, i.e. the control path if *val* is less than 32767. This control dependence is substituted by one from Node R3 to R5, which means Node R5 and its successors are executed before the comparison result in Node 3 becomes available.

3.4 Synthesizing Hardware from PDG+SSA

There are two approaches of synthesizing reconfigurable hardware from the PDG+SSA form. One is to conduct a region-by-region synthesis using architectural synthesis, technology mapping, and placement and routing techniques. The other is to conduct directed mapping to synthesis reconfigurable hardware.

3.4.1 Region-by-region synthesis

Region-by-region synthesis is good for any application in the PDG+SSA form. Each region is a data flow graph. With the SSA extension, explicit data dependencies are carried by the edges. More timing constraints are added among nodes to represent interface requirements or performance requirements. Architectural synthesis tools conduct resource allocation, scheduling, resource sharing, and register sharing on the graph, and output structural register-transfer level (RTL) hardware descriptions.

As discussed before, each region is a group of operations with the same control conditions. Therefore, it is easy to generate a finite-state machine (FSM) from the hierarchical region structure to control the execution of the application.

The structural hardware description and the FSM can be further synthesized into configuration files using technology-specific synthesis tools.

Resource allocation and scheduling is one of the most important issues in hardware synthesis. Our approaches based on the max-min ant system optimization are presented in Chapters 5 and 6.

3.4.2 Direct mapping

The PDG+SSA form has naturally direct mapping into structural RTL hardware description, given the target architecture is the FPGA-based fine-grained reconfigurable architectures, and the application is not so complicated that it requires much resource sharing. The RTL hardware

description can be synthesized using commercial tools to a bitstream to program the reconfigurable hardware.

Mapping the PDG+SSA to reconfigurable hardware

The PREDICATE and STATEMENTS nodes represent arbitrary sets of expressions or data-flow graphs (DFGs). In order to synthesize such DFGs into a bitstream, a variety of methods can be utilized. We currently use a one-to-one mapping. It is possible to use a number of different scheduling and binding algorithms and perform hardware sharing to generate smaller circuits; this is out of the scope of this paper, however, we plan on addressing this in future work.

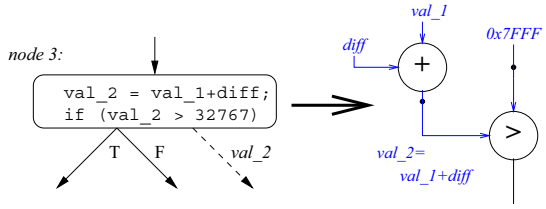


Figure 3.7: Synthesizing the ϕ-node

Figure 3.7 shows the synthesis of data-path elements in node 3 of the previous example (see Figure 3.6). Each operation has an operator and a number of operands. Operands are synthesized directly to wires in the circuit since each variable in the SSA form has only one definition point. Every PREDICATE node contains operations that generate predicate values. These predicate values are synthesized to Boolean logic signals to control next-stage transitions and direct multiplexers to commit the correct value.

A loop-independent ϕ-node is synthesized to a multiplexer. The multiplexer selects input values depending on the predicate values. For example, as shown in Figure 3.8, *P*5 is translated to a two-input multiplexer `MUX_P5`, which uses the predicated value from 5 to determine which result should be committed.

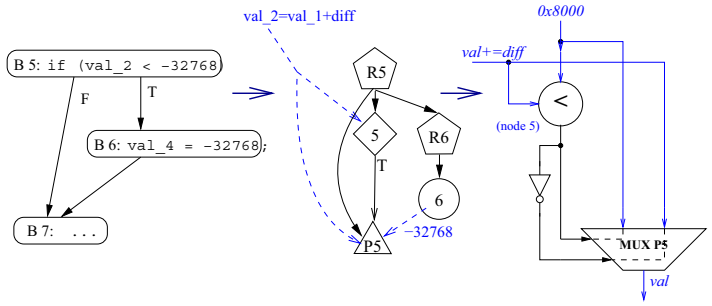


Figure 3.8: Synthesizing the ϕ-node

More work is required to synthesize a loop-carried node since it must select the initial value and the loop-carried value, and direct these values to the iteration exit. Using a two-input multiplexer, the initial value and the loop-carried value can be selected depending on the predicate values. A switch is generated to direct the loop-exiting values.

Before synthesizing the PDG to hardware, some optimizations and simplification should be done. For example, unnecessary control dependencies can be removed. Node R4 and R6 in Figure 3.6 are unnecessary and can be removed. Cascaded ϕ-nodes, such as nodes P3 and P5, can be combined into a bigger ϕ-node with all predicated values. This allows the downstream synthesis tools to choose a proper (possibly cascaded) multiplexor implementation. These ϕ-nodes can also be synthesized directly if

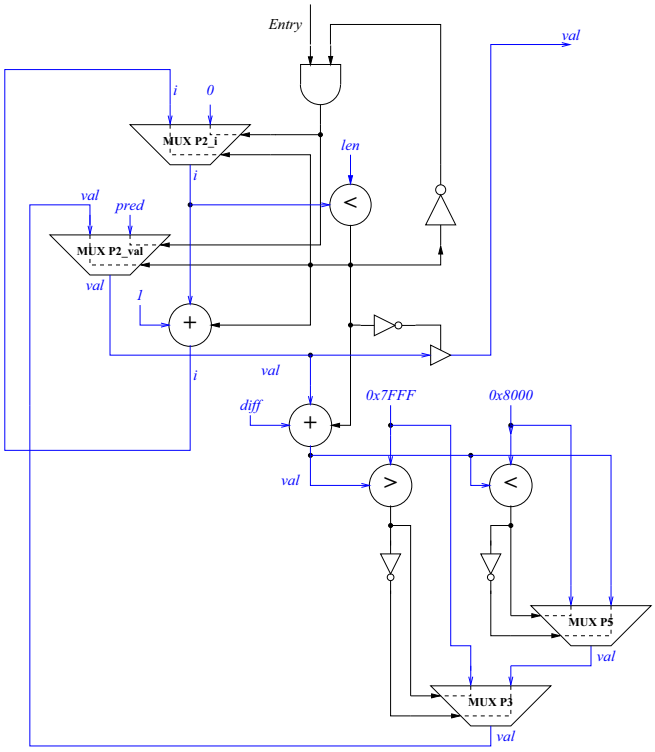


Figure 3.9: FPGA circuitry synthesized from the above PDG (See Figure 3.6)

necessary, i.e. the downstream synthesis tools do not perform multiplexor optimizations.

Synthesizing the PDG removes artificial control dependencies. Only those necessary control signals are transmitted. After synthesis, scheduling should be performed to insert flip-flops to guarantee that correct values are available no matter which execution path is taken.

Experimental results

We conducted experiments on direct mapping the PDG+SSA form to FPGA-based reconfigurable hardware, and collected results. This section

presents the experimental setup and results.

We use MediaBench [78] as our benchmark suite. More than 100 functions in six multimedia applications are tested. Among them, results of 16 functions are reported here. The other non-reported functions exhibited similar behaviors. Table 3.1 shows some statistical information for the reported functions, including the number of operations, the number of logic and arithmetic operations, memory access, and control transfer instructions; the number of CFG nodes and average instructions per CFG node; and the number of REGION nodes, PREDICATE nodes, and STATEMENTS nodes in PDGs.

	Operations				CFG		PDG		
	#Instr	ALM	Mem	CTI	#N	#Instr/N	R	P	C
func_1	233	148	10	18	31	7.52	32	13	18
func_2	188	128	9	14	24	7.83	25	10	14
func_3	73	52	3	2	5	14.60	5	2	3
func_4	79	51	1	7	13	6.08	15	5	8
func_5	22	15	1	1	3	7.33	3	1	2
func_6	68	51	0	6	10	6.80	10	5	5
func_7	81	55	3	8	13	6.23	13	6	7
func_8	326	250	25	1	3	108.67	3	1	2
func_9	391	306	34	1	3	130.33	3	1	2
func_10	52	36	1	6	10	5.20	12	3	7
func_11	140	104	5	12	18	7.78	19	7	11
func_12	104	72	3	11	17	6.12	18	7	10
func_13	118	85	7	9	14	8.43	17	5	9
func_14	142	104	6	6	11	12.91	11	4	7
func_15	95	54	4	5	9	10.56	11	3	6
func_16	491	336	16	49	67	7.33	77	27	40

Table 3.1: Statistical information of CFGs and PDGs

The experiments are performed using the SUIF and Machine SUIF infrastructure [3, 112]. SUIF provides a front-end for the C programming language, and Machine SUIF constructs the CFG from the SUIF IR. Us-

ing the HALT profiling tool included with Machine SUIF, we import profiling results of the MediaBench applications from the representative input data included with the benchmark suite. We created a PDG pass, which currently performs limited analysis and optimizations.

After constructing the PDG, we estimate the execution time and synthesized area on a configurable logic array. The target architecture is the Xilinx Virtex II Platform FPGA [131]. Based on the specification data of the Virtex II FPGA, we get the typical performance characteristics for every operation, which is used estimate the performance of the PDG.

At the same time, we use the similar direct mapping to synthesize the CFGs and the PSSAs to reconfigurable hardware, and collect performance and area data.

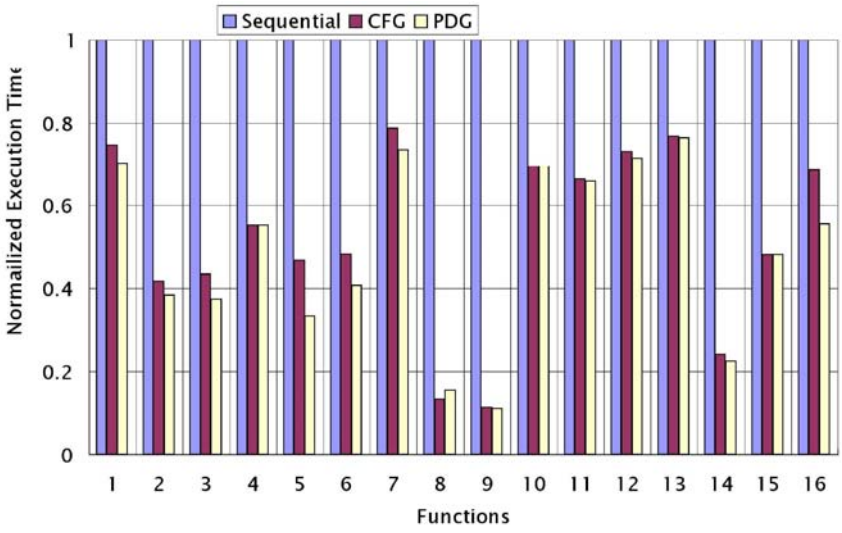


Figure 3.10: Estimated execution time of PDGs and CFGs

Figure 3.10 shows the estimated execution time using the PDG representation compared to the CFG representation and sequential execution.

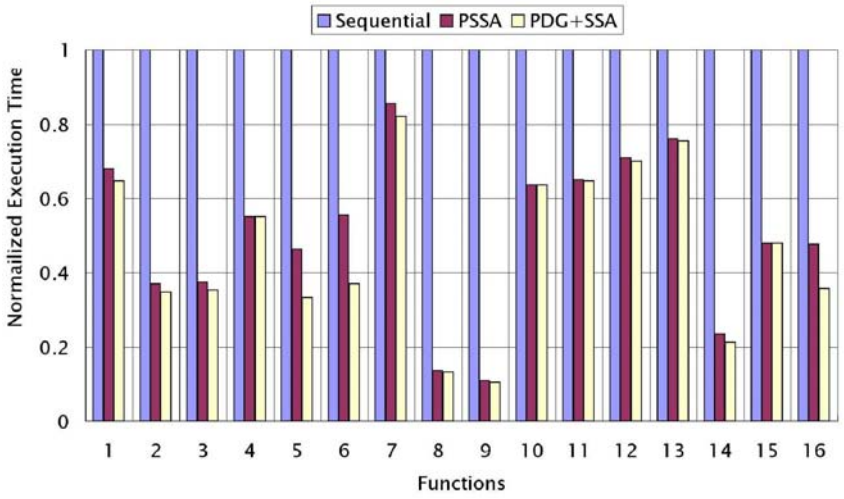


Figure 3.11: Estimated execution time using aggressive speculative execution

The PDG and the CFG are 2 to 3 times faster than sequential execution; the PDG is about 7% faster than the CFG. These results use a simple scheduling scheme for estimation. In the CFG, a basic block can be executed when all its predecessors complete their executions. In the PDG, a node is executed once its control and data dependencies are satisfied.

Figure 3.11 shows the estimated execution time of the PDG+SSA form. Here an aggressive speculative execution is performed. All possible execution paths are taken and the results are committed when the predicated values are available. The results of the PDG+SSA form are on average 8% better when compared to the results of the aggressive speculative execution results of the PSSA form.

It is necessary to note that our experimental results do not use all of the optimizations presented in the original PSSA paper [23]. The projects using the PSSA representation [18, 121] perform other optimizations, in-

cluding operation simplification, constant folding, dead-code removal, and SSA form minimization. Though the PDG+SSA form is capable of performing these optimizations, we did not perform these optimizations in our experiments. It is unclear how these optimizations affect the performance and area results. We intend to look into these optimizations in our future work. Our results simply indicate that when using aggressive speculative execution, the PDG+SSA form executes faster than the PSSA form.

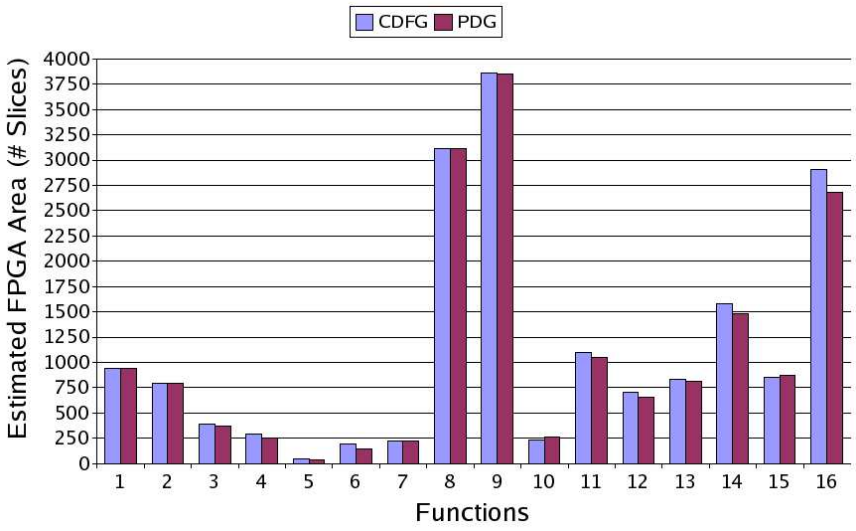


Figure 3.12: Estimated area of the PDG and CFG representations

Figure 3.12 shows the estimated number of FPGA slices. The results are estimated based on the direct mapping. Thus, each operation takes a fixed amount of computational resources. The results of the PDG are better than those of the CFG, but the difference is small. We do not consider resource sharing. These results are similar to those reported by Edwards [37]. His results show that that the PDG generates smaller circuits than

the CFG for control intensive applications, e.g. applications described using the Esterel language.

3.5 Summary

The above work and experimental results show that the PDG is a suitable program representation for synthesizing sequential programs to reconfigurable computing systems. If we apply more parallelizing compilation techniques on the PDG, it is possible to reveal much more parallelism. In addition, it is possible to represent target architectural constraints on the PDG as other kinds of dependence since synthesis from the PDG to circuits is trivial.

Chapter 4

Data Partitioning and Storage Assignment

Typical configurable computing systems are integrated with ample distributed block RAM modules, and exhibit superior computing abilities, storage capacities, and flexibilities over traditional FPGAs. However, designers lack the necessary design tools to effectively and efficiently synthesize applications onto these complex architectures. In particular, there is a pressing need for memory optimization techniques in the early stages of the design flow as modern configurable architectures have a complex memory hierarchy, and earlier architectural-level decisions greatly affect the final design qualities.

This section focuses on seeking a partitioning-based solution to the storage assignment problem at the earliest stages of the design flow, and shows how other memory optimizations can help achieve design goals,

such as reduce latencies and increase throughput.

In this chapter, we begin with an introduction to the target architecture and an example of a bank of filters, and discuss related work in Section 4.3. In Section 4.4, we formally define the data partitioning and storage assignment problem, provide our approach and other memory optimization techniques, and present experimental results. Finally, we summarize in Section 4.6.

4.1 Introduction

Modern reconfigurable architectures incorporate a distributed memory modules, among their configurable logic blocks (CLBs). These architectures can be divided into homogeneous and heterogeneous architectures, according to the capacities and distribution of the RAM blocks.

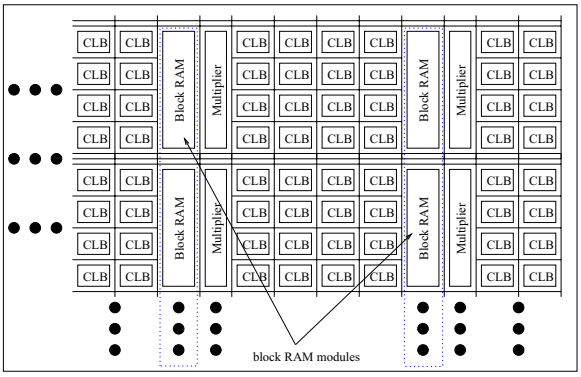


Figure 4.1: FPGA with distributed Block RAM modules

Figure 4.1 represents an example of a *homogeneous* architecture. This roughly corresponds to Xilinx Virtex II FPGA [132]. The block RAM modules are evenly distributed on the chip and connected with CLBs using

reprogrammable interconnects. Every block RAM has the same capacity. Additionally, there is an embedded multiplier located beside each block RAM. A large Virtex II chip contains 168 blocks of 18 Kbits block RAM modules, providing 3,024 Kbits of on-chip memory.

The *heterogeneous* architecture contains a variety of block RAM modules with different capacities. For example, the TriMatrix memory on an Altera Stratix II FPGA chip [7] consists of three types of on-chip block RAM modules: M512, M4K, and M-RAM. Their capacities are 576 bits, 4 Kbits, and 512 Kbits, respectively. A Stratix II chip may contain a large number of M512 and M4K modules, but generally only a few M-RAM modules. Currently our work only considers homogeneous architectures.

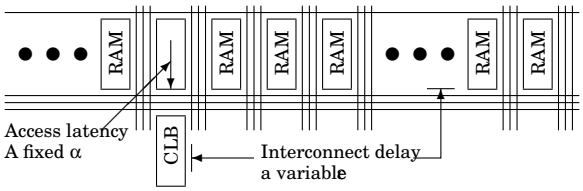


Figure 4.2: Total access latencies = $\alpha + \varepsilon$

The access latency of the on-chip block RAM is equal to the propagation delay to the memory port after the positive edge of the `clock` signal. This delay is usually a fixed number α for a specific FPGA architecture. For example, α is 3.7 ns for Xilinx XC2V3000 FPGA. Additionally it takes an extra ε ns to transfer data from the memory port to the accessing CLB. Hence, a design running at 200MHz could take one clock cycle to retrieve data close to the accessing CLB, but two or even more clock cycles to access data far away from the CLB. On the other hand, it is often difficult to

distinguish whether the data accessed is near or far.

In addition to block RAM modules, CLBs can be configured as local memory, which is convenient for storing intermediate results. When CLBs are configured as distributed memory, the access latency, i.e. the logic access time, is quite small. However, if a data array is assigned to CLBs, an access involves extra delay for MUX selecting the addressed element. For example, the delay for a 512-bit CLB memory is around 3.5 ns for Xilinx XC2V3000 FPGA; the delay for a 16 Kbit CLB memory increases to 6.2 ns.

The FPGA can be complimented by an external, global memory for storing large amounts of data. Access latencies to the external memory depend on the bus protocol and type of memory. The access latencies usually are an order of magnitude slower than those of on-chip block RAM.

This section presents a methodology for partitioning data to distributed block RAM modules. When compared to off-chip global memory, and using CLBs as distributed RAM, this approach is an effective and efficient solution for most applications.

The main contribution of this section is a novel integrated approach of deriving appropriate data partitioning, and synthesizing the program behavior to configurable devices. Through intensive research on the interplay between the data partitions and architectural synthesis decisions, such as scheduling and binding, it is shown that designs that minimize the number of global memory accesses and exhibit local computation can meet the design goals, and minimize the execution time (or maximize the

system throughput) under resource constraints. Other optimization techniques, including scalar replacement, data prefetching, and buffer insertion, are applied to improve the overall performance. In particular, these optimizations further reduce latencies, and improve the achievable clock frequencies.

4.2 Motivating example: correlation

In order to give the reader an understanding of the problem, a motivation example is presented here. Designers synthesize a bank of correlators to an FPGA-based configurable architecture with embedded multipliers and block RAM modules. Such a bank of correlators is a commonly occurring operation in DSP applications, e.g. Kalman filters, matching pursuit (MP), recursive least squares (RLS), and minimum mean-square error estimation (MMSE) [58].

The bank of correlators multiplies each sample of a received vector $\mathbf{r}$ with the corresponding sample of a column in a matrix $\mathbf{S}$, i.e. $C_i = \sum_{j=1}^{l} r_j \times S_{j,i}$, where r is a vector of l complex numbers, and S is a $m \times l$ real numbers. l and m may vary based on the application. For instance, if we wish to perform radiolocation in the ISM band (i.e. 802.11x) using the matching pursuit algorithm, both l and m are equal to 88 [84].

It was assumed that a large enough memory module could be embedded on the chip, and is possible to assign the S matrix on this memory module. The advanced commercial high-level synthesis tool either gener-

ates a design with an extremely slow execution time of about 77,440 ns, or fails to synthesize this design due to the huge S matrix. On the other hand, distributing the data accesses to block RAM modules results in designs up to 80 times faster. Obviously the partitioning of the S matrix to the block RAM modules greatly affects the overall system performance.

The data space is intuitively partitioned by columns or by rows. By simple analysis, column-wise partition results in a communication-free partitioning. Figure 4.3 suggests several candidates for column-wise partitioning solutions. Figure 4.3(a), 4.3(b), and 4.3(c) assign one block RAM module to one column, four columns, and eight columns, respectively.

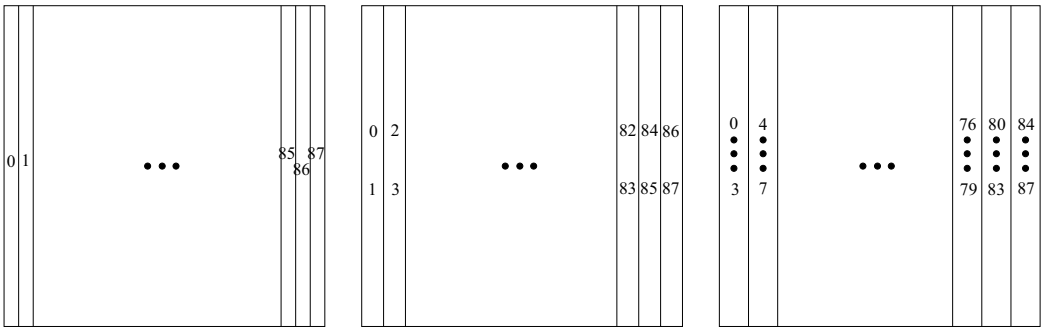
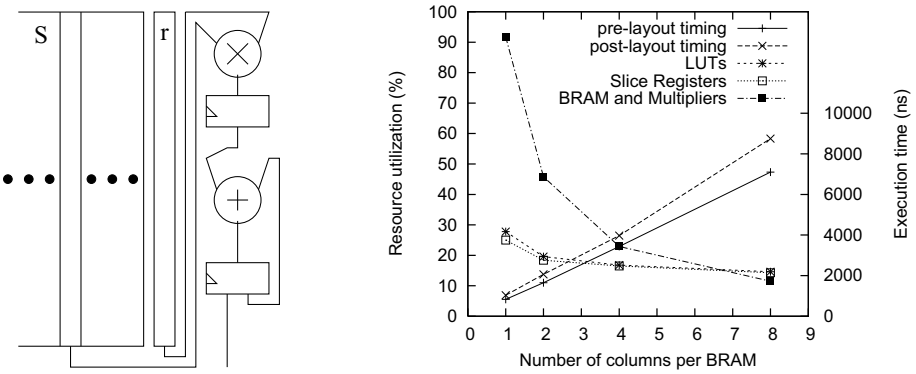


Figure 4.3: Candidates for communication-free data partitioning

Figure 4.4(a) presents the control and computations of the column-wise data partitioning. Computations of each correlator are conducted using the embedded multipliers beside the block RAM in a multiplication and accumulation (MAC) manner. For each correlator, the control logic and computational resources are local to the block RAM module.

Figure 4.4(b) presents area and timing trends of different granular-

ity for the column-wise scheme. When assigning one block RAM to one column, the design takes approximately 1000 ns, but occupies about 90% of available block RAM modules and embedded multipliers, and approximately 25% of available LUTs. When more columns are packed into one block RAM, the hardware requirements decrease. However, the execution time increases linearly to the granularity of partitions. When assigning one block RAM to two columns, the execution time doubled. When assigning one block RAM to eight columns, the executions are approximately 8 times longer than that of one column per block RAM.



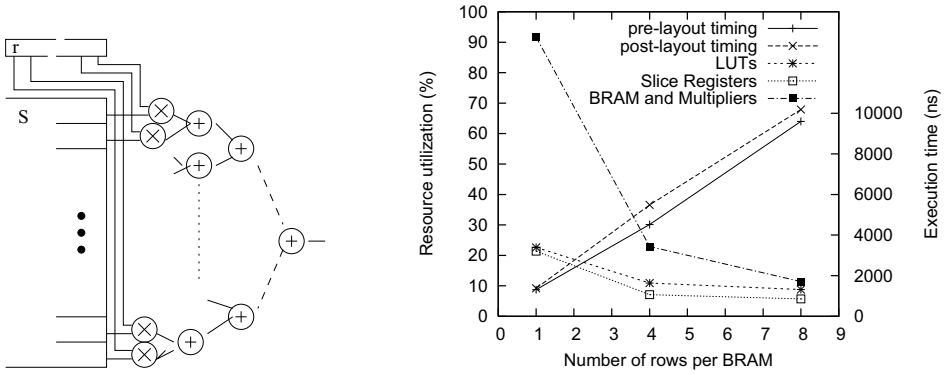
(a) Computational model

(b) Synthesized results

Figure 4.4: Implementations and area/timing trade-offs

To evaluate different partitioning schemes, we also obtained performance results for row-wise partitions.

Figure 4.5(a) illustrates the parallel computation scheme, or the *by-row* scheme, where one block RAM is assigned to one or multiple rows. Data at the same column is read and multiplied using the local fixed multiplier. A pipelined adder-tree is used for the summation of the products.



(a) Computational model (b) Synthesized results

Figure 4.5: Implementation and results of the row-wise partitioning

The adder tree requires global accesses to each of the block RAM modules, hence this is not a communication-free partitioning. This scheme parallelizes each correlator, and therefore requires a global control on the multipliers and the pipelined adder-tree. Figure 4.5(b) presents area and timing trends of different granularity for both schemes, respectively.

Data per BRAM	# of cycles	Pre-layout Timing		Post-layout Timing	
		F(MHz)	L(ns)	F(MHz)	L(ns)
1 column	178	214.7	829	171.6	1037
1 row	184	140.5	1309	133.5	1378
4 columns	706	205.0	3436	178.2	3961
4 rows	710	157.0	4520	129.4	5486
8 columns	1410	198.6	7099	161	8752
8 rows	1413	147.1	9602	138.7	10183

Table 4.1: Comparison between the same granularity

Table 4.1 compares the row-wise and column-wise schemes with the same granularity (i.e. same number of rows/columns). In the term of numbers of clock cycles, the difference is minimal. However, if we check the maximal achieved frequencies, designs of the column-wise partitioning scheme are 30-50% faster than those of the row-wise partitioning scheme.

Performance gaps are mainly due to the increased amount of global communications needed for the control logic and global memory accesses to block RAM modules. Pre-layout timing results are always better than those of post-layout results, which shows that, in both schemes, the RTL tools under-estimate the interconnect delays.

To summarize, different partitions of the array S deliver a wide variety of candidate solutions. Synthesized designs showed that data partitioning and storage assignment not only affect the number of clock cycles, but also affect the achieved clock frequencies. The design with fewer global communications achieves better performance.

4.3 Related work

In traditional design flow of configurable devices, synthesis of block RAM modules is generally handled as a physical problem. They are directly inferred from arrays, or instantiated using vendor macros. They are packed in a single component in placement, and only partitioned when it is difficult to fit into the device. In most situations, the memory bandwidth and storage capacities are not well utilized, and hence the generated designs are not efficient in terms of latencies, throughput, and achieved frequencies.

High-level synthesis can dramatically reduce the design time, and deliver high performance designs, with less clock cycles, higher clock frequencies, less area, and even less power [31, 43]. Most early efforts on

the high-level synthesis were focused on resource allocation, scheduling, and binding. Different approaches were proposed to synthesize memory modules. Early efforts usually mapped data arrays into a single memory module [26, 30]. Thomas,*et al* [119] assigned each data array a memory module. Comprehensive storage exploration and memory optimizations technologies are presented in IMEC's DTSE work [24]. In most of their work, they assumed that the memory module is large enough for those data arrays and did not consider memory capacity constraints.

Panda,*et al* [90] investigated architectural-level exploration techniques for embedded processors with complex hierarchical memory systems. Based on the PICO method [107], Kurdur,*et al* [75] presented an ILP formulation to solve the storage arrangement problem. They assumed every data array can fit into one of the local memories, and they used an extra move operation to access remote data. These works are more like a processor-based data exploration and memory optimization works.

Early efforts on utilizing multiple memory modules on FPGA [46] allocated an entire array to a single memory module rather than partitioning data arrays. Furthermore, they assumed that the latencies differences had little effect on system throughput. As to memory optimization in synthesis to configurable platforms, Budiu,*et al* [17], and Diniz,*et al* [9], respectively, presented some effective techniques to reduce memory accesses and benefit high-level synthesis.

Huang,*et al* [69] presented their work in high-level synthesis with inte-

grated data partitioning for ASIC design flow. Their work is quite similar to our work as they adopted code analysis techniques from the traditional parallelizing compilation field. However, their work is more like an ASIC flow and is not limited by the capacities of available memory modules. They started from a fixed number of partitions. Our proposed work starts from the program cores and the resource constraints, and uses granularity adjustment to find out how many partitions are reasonable for the design.

The data partitioning and storage assignment problem is well studied in the field of parallelizing compilation [5, 92, 130]. Early efforts developed effective analysis techniques and program transformations to reduce global communications and improve system performance. Shih and Sheu [111] and Ramanujam and Sadayappan [100] addressed the methodology to achieve communication-free iteration space and data partitioning problem. Pande [91] presented a communication-efficient data partitioning solution when it is impossible to get communication-free partitioning.

The following differences make it impossible to directly adopt these approaches into a system compiler for configurable architectures with distributed block RAM modules:

- The target architectures are different. Multiprocessor systems have a fixed number of microprocessors. Each microprocessor has its own local memory, and is connected with a different remote memory module that exhibits non-uniform memory access (NUMA) attributes.

- Configurable architectures execute programs using CLBs rather

than microprocessors. The number of block RAM modules is fixed. There is not a fixed number of CLBs associated with a particular block RAM. Hence, the boundaries between local and remote memory are indistinct.

- Programs are executed sequentially or with limited instruction level parallelism (ILP) on each microprocessor, while the parallelizing compiler exploits coarse-grained parallelism. Computing tasks run in a fully parallelized and concurrent manner on configurable architectures.

Our problem is distinguished from the previous studies as follows. First, these differences violate a fundamental assumption held in the previous research. Most of the previous efforts assumed that global communications or latencies to remote memory are an order of magnitude slower than access latencies to local memory. This makes it reasonable to simplify the objective function to simply reduce the amount of global communications.

This assumption is not true in the context of data partitioning for configurable architectures. As previously described, the boundaries between local and remote memory are indistinct. Access latencies to block RAM modules depend on the distance between the accessing CLBs and the memory ports. There is no way to determine the exact delay before performing placement and routing.

Second, data partition and storage assignment have more compound

effects on system performance. In parallelizing compilation for multiprocessor architectures, once computations and data are partitioned, it is relatively easy to estimate the execution time since the clock period is fixed, and the number of clock cycles consists of the communication overheads and computation latencies for each instruction. However, it is extremely difficult to determine the execution time in configurable systems before physical synthesis. Our results in Section 4.5 show that even though the number of clock cycles is almost the same, there can be 30-50% deviations in execution time due to variation in frequency. Therefore, the control logic and computation times are effected, and not just the memory access delays.

Moreover, the flexibility to configure block RAM modules makes this problem even more difficult. Block RAM modules could be configured with a variety of $width \times depth$ schemes, and as described before, even CLBs could be used to store small data arrays.

To summarize, configurable architectures are drastically different from traditional NUMA machines, making it difficult to estimate candidate solutions during the early stages of synthesis. Flexibilities in configuring block RAM modules greatly enlarge the solution space, making the problem even more challenging.

4.4 The data partitioning and storage assignment algorithm

This section formally describes the data partitioning and storage assignment problem, and proposes an approach to computing the number of memory accesses for a given partition. Then, we discuss some of the techniques that we use to reduce memory accesses and improve system performance for FPGA-based configurable architectures with distributed block RAM modules.

4.4.1 Problem formulation

The proposed approach is focused on data-intensive applications in digital signal processing. These applications usually contain nested loops and multiple data arrays.

In order to simplify our problem, we assume that *a*) the input programs are perfectly nested loops; *b*) index expressions of array references are affine functions of loop indices; *c*) there is no indirect array references, or other similar pointer operations; *d*) all data arrays are assigned to block RAM modules; and *e*) each data element is assigned one and only one single block RAM modules, i.e. no duplicate data. Furthermore, we assume that all data types are fixed-point numbers due to the current capability of our system compiler.

The inputs to this data partitioning and storage assignment problem are as follows:

- A program d contains an l-level perfectly nested loop $\mathbf{L} = \{L_1, L_2, \ldots, L_l\}$.
- The program d accesses a set of n data arrays $\mathbf{N} = \{N_1, N_2, \ldots, N_n\}$.
- A specific target architecture, i.e. an FPGA, contains a set of m block RAM modules $\mathbf{M} = \{M_1, M_2, \ldots, M_m\}$. This FPGA also contains A CLBs.
- The desired clock frequency F, and the maximum execution time is L.

The problem of data partitioning and storage assignment is to partition $\mathbf{N}$ into a set of p data portions $\mathbf{P} = \{P_1, P_2, \ldots, P_p\}$, where $p \leq m$, and seek an assignment $\{\mathbf{P} \rightarrow \mathbf{M}\}$ subject to the following constraints:

- $\bigcup_{i=1}^{p} P_i = \mathbf{N}$, and $P_i \bigcap P_j = \emptyset$, i.e. that all data arrays are assigned to block RAM and each data element is assigned to one and only one block RAM module.
- $\forall (P_i, M_j) \in \{\mathbf{P} \rightarrow \mathbf{M}\}$, the memory requirement of P_i is less than the capacity of M_j

After obtaining data partitions and storage assignments, we reconstruct the input program d, and conduct behavioral-level synthesis. After RTL and physical synthesis, the synthesized design must satisfy the following constraint:

- The slices of CLBs occupied by synthesized design d is less than A.

The objective is to minimize the total execution time (or maximize the system throughput) under the resource constraints of specific configurable architectures. The desired frequency F and the maximum execution time T among inputs are used as target metrics during compilation and synthesis.

4.4.2 Overview of the proposed approach

The proposed approach is based on our current efforts of synthesizing C programs into RTL designs. A system compiler takes C programs, and performs necessary transformations and optimizations. By specifying target architecture, and desired performance (throughput), this compiler performs resource allocation, scheduling, and binding tasks, and generates RTL designs in hardware designs, which can then be synthesized or simulated by commercial tools.

As discussed before, in configurable architectures, the boundaries between local and remote accesses are indistinct. In our preliminary experiments, we found that, given the same datapath with memory accesses to block RAM modules with different locations, the lengths of critical paths achieved after placement and routing can have a 30-50% variation. A limited number of functional units could be placed near the block RAM modules, which they access.

Therefore, we could still assume that, once the data space is partitioned, we can obtain a corresponding partitioning of the iteration space, or a partitioning of the computations. Each portion of the data space can

be mapped to one portion of the iteration space. Then we divide all memory accesses into local accesses and remote ones. However, these local and remote memory accesses are different from those in parallel multiprocessor systems in that the access latencies are usually on the same order of magnitude.

Based on this further assumption, we adopt some concepts and analysis techniques in traditional parallelizing compilation. A *communication-free* partitioning refers to a situation where each partition of the iteration space only accesses the associated partition of the data space. If we cannot find a communication-free partition, we look for a *communication-efficient* partition to minimize the execution time.

Our proposed approach integrates traditional program tests and transformation techniques in parallelizing compilation into our system compiler framework. In order to tackle the performance estimation during data space partitioning, we use our behavioral-level synthesis techniques, i.e. resource allocation, scheduling, and binding.

4.4.3 Algorithm formulation

This section discusses our data and iteration space partitioning algorithm in detail. Our approach is illustrated in Algorithm 1. Before line 6, we adopt existing analysis techniques in parallelizing compilation to determine a set of directions to partition. In line 6 and 7, we call our behavioral-synthesis algorithms to synthesize the innermost iteration body. After that, we evaluate every candidate partition, and return the

one with the most likelihood of achieving the short execution time subject to the resource constraints.

Algorithm 1 Partitioning

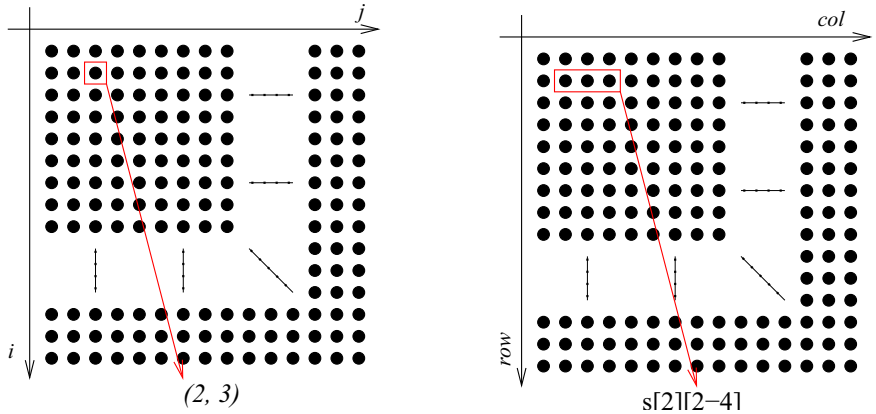
Ensure: $\bigcup_{i=1}^{p} P_i = \mathbf{N}$, and $P_i \bigcap P_j = \emptyset$
Ensure: $|\mathbf{P}| \leq |\mathbf{M}|$
1: Calculate the iteration space $IS(\mathbf{L})$
2: **for** each $N_i \in \mathbf{N}$ calculate the data space $DS(N_i)$
3: $B =$ Innermost iteration body
4: Calculate the reference footprints, F, for B using reference functions
5: Analyze $IS(\mathbf{L})$ and F, and obtain a set of partitioning direction $\mathbf{D}$
6: $a = A/|\mathbf{M}|$ {# of CLBs associated to each RAM}
7: Synthesis($B, 1, 1, a, u_{ram}, u_{mul}, u_a, T, II$)
8: $g_{min} =$ size of$IS(\mathbf{L})/|\mathbf{M}|$ {the finest partition}
9: $g_{max} = \frac{\text{size of } \sum DS(N_i)}{\text{size of each block RAM}}$ {the coarsest partition}
10: $d_{cur} = d_0$, $g_{cur} = g_{min}$
11: $C_{cur} = \infty$
12: **for** each $d_i \in \mathbf{D}$ **do**
13: **for** $g_j = g_{min}, g_{max}$ **do**
14: Partition $DS(\mathbf{N})$ following d_i and g_j
15: Estimate the number of memory accesses using reference functions
16: $m_r =$ # of remote accesses
17: $m_t =$ # of total accesses
18: $\tau = 2^{\frac{m_r}{m_t}}$ {the choice of 2 depends on the chip size}
19: $C = \tau \times (\max\{u_r, u_m, u_a\} \times II \times g_j + (T))$
20: **if** $C < C_{cur}$ **then**
21: $d_{cur} = d_i$, $g_{cur} = g_j$
22: $C_{cur} = C$
23: **end if**
24: **end for**
25: **end for**
26: Output d_{cur} and g_{cur}

Program analysis Given an l-level nested loop, the iteration space is an l-dimensional integer space. The loop bounds of each nested level set the bounds of the iteration space. An integer point in this iteration space

solely refers to an iteration, which includes all statements in the innermost iteration body. Each m-dimension data array has a corresponding m-dimensional integer space. An integer point refers to a data element with that data index.

```
  for (i=1; i<ROW-1; i++)
2   for (j=1; j<COL-1; j++)
      d[i][j]=(s[i][j-1]+(s[i][j]<<1)+s[i][j+1])>>2;
```

Figure 4.6: A 1-dimensional mean filter



(a) Iteration space (b) Data spaces of `d` and `s`

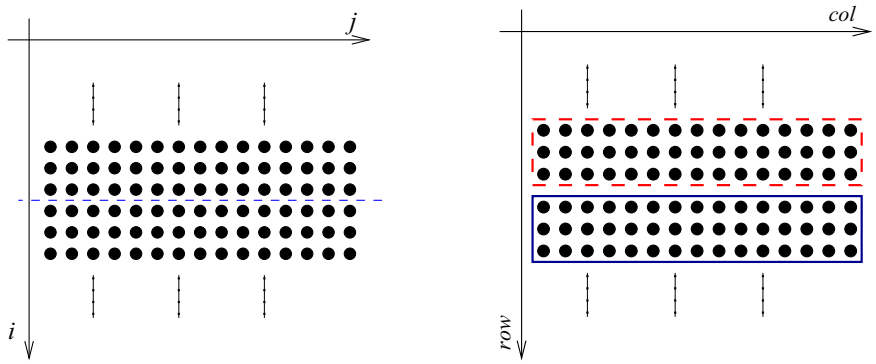
Figure 4.7: Iteration space and data spaces of the 1-dimensional mean filter

For example, Figure 4.6 shows the kernel of a 1-dimensional mean filter. This simplest mean filter blurs the image and removes speckles of high frequency noise in the row direction. The corresponding iteration space is shown in Figure 4.7(a).

During each iteration, data elements in the data space are accessed. Since we assume that index expressions of array references are affine functions of loop indices, a footprint of each iteration can be calculated

using the affine functions, i.e. each iteration is mapped to a set of data points in the data space by means of a specified array reference. In the above mean filter example, given an iteration (2,3), we can easily obtain the access footprints in the $DS((S))$ as $\{(2,2),(2,3),(2,4)\}$ (as shown in the rectanglar box in Figure 4.7).

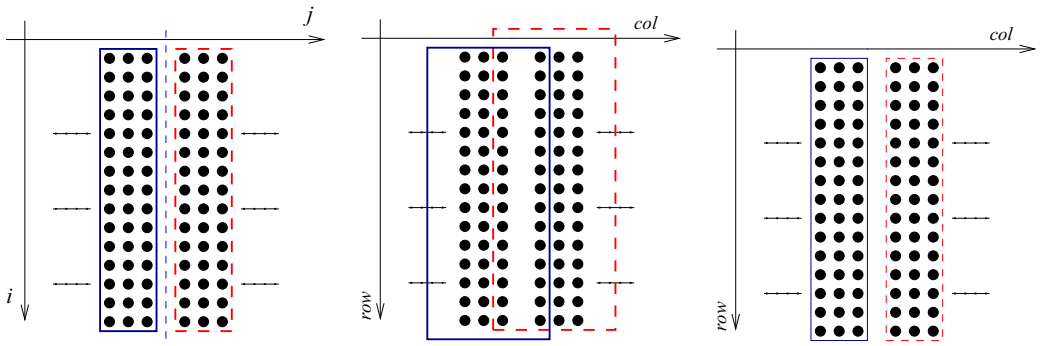
With the iteration space $IS(\mathbf{L})$ and the reference footprints F, we can determine a set of directions to partition the iteration space. The direction can be represented by a multi-dimensional vector. For example, if we have a 2-level nested loop, we usually do row-wise or column-wise partitioning, or in the (col, row) vector form, (0,1) or (1,0), respectively. Figure 4.8(a) shows a row-wise bi-partitioning of the iteration space of the above mean filter example, and the corresponding data space partitioning is shown in Figure 4.8(b).



(a) Iteration space (b) Data spaces of `d` and `s`

Figure 4.8: Data spaces are correspondingly partitioned when the iteration space is partitioned.

In the row-wise partitioning of the mean filter example, the data access footprints of any iteration are in one of the data space portions. This could



(a) Iteration space (b) Data spaces of d and s (c) Iteration space

Figure 4.9: Partitioning of overlapped data access footprints

mean that, after synthesis and physical design, all data accesses can be local memory accesses. However, in some cases, data access footprints may be broken. Hence, some iterations may access data from more than one data space partitions. As shown in Figure 4.9(b) the data in the rectanglar boxes are overlapped with the dashed box, i.e. data are required by iterations in both iteration partitions. This is the reason why we have non-local or remote data accesses. Although we could not achieve communication-free partitioning, we could evenly partition the overlapped data spaces. For instance, this array is partitioned like these boxes shown in Figure 4.9(c).

Synthesis of iteration bodies In order to evaluate our candidate solutions, their performance on target configurable architectures should be determined. Since most design problems in behavior synthesis are *NP*-complete, and time-consuming, it is extremely inefficient to perform synthesis on each candidate solutions.

Algorithm 2 Synthesis

1: Generate DFG g from B
2: Schedule and pipeline g to minimize the initial interval, subject to allocated resources, including r block RAM, m multipliers, and a CLBs.
3: Output resource utilization u_r, u_m, and u_a.
4: Output execution time T, and the initial interval II

In our approach, we first synthesize the innermost iteration body with a proper resource constraint, obtain performance results for the single iteration, and then use them to evaluate our cost function in line 19 of Algorithm 1.

The innermost iteration body is scheduled and pipelined using allocated resources, including 1 block RAM modules, 1 embedded multiplier, and a portion of CLBs, which, by our assumption, are associated with a specific block RAM module. We pipeline our design because, for a large iteration space $IS(\mathbf{L})$, the pipelined iteration body gives the shortest execution time, and the best resource utilization. After synthesis, we return the resource utilization for the block RAM, multiplier, and the CLBs, respectively. We also output the number of total clock cycles, and the initial interval (II), which determines the maximum system throughput.

Granularity adjustment For each partitioning direction, we evaluate every possible partition granularity. Given a specific nested loop and data arrays, and a specific architecture, we can determine the finest and coarsest grain for homogeneous partitioning. As shown in line 8 of Algorithm 1, the finest partition granularity partitions the iteration space (and the data space) into as many portions as possible. It therefore depends on the

number of block RAM modules. The coarsest-grained partition requires that each block RAM store as much data as possible. It depends on the capacity of a block RAM module.

Once we determined the partitioning direction and granularity, we can use reference functions to estimate the total number of memory accesses, and among them, the number of global memory accesses.

Our cost function, as shown in line 19, gives us a good idea of how long the execution time is. It consists of two parts. The first one is the τ, a special factor greater than or equal to 1, as shown in line 15. This τ includes effects of remote memory accesses. When there is no remote memory access, $\tau = 1$, and we can achieve communication-free partitioning; otherwise, we want to minimize it, which reduces the execution time. The second part is an experiential formula estimating the total clock cycles for a pipelined design under resource constraints. Since the iteration body is pipelined, the most utilized components determines the performance (or throughput) when more than one iteration is assigned to this block. For example, after pipelining, $II = 1$, $T = 10$, $u_m = 1$. If there are ten iterations in one partition, then the execution time will be $1 \times II \times 10 + (T - II) = 19$ clock cycles, without considering effects of remote memory accesses. Another example could be, after pipelining, $II = 1$, $T = 10$, $u_m = 0.5$; if there are still ten iterations in one partition, then the execution time is $0.5 \times II \times 10 + (T - II) = 14$ clock cycles, without considering effects of remote memory accesses. The reason why the second one is faster is that there half as many multipliers and other resources are free,

which allow more operations to be scheduled at the same time.

4.4.4 Performance estimation and optimizations

In order to evaluate our data partitioning and storage assignment solutions, we apply architectural-level synthesis techniques to each portion of the partitioned design using sophisticated scheduling and binding algorithms. In addition to the traditional architectural-level synthesis techniques, we apply other optimization techniques, in particular those that take advantage of FPGA-based configurable architectures, such as port vectorization, scalar replacement, and input prefetching. These optimization techniques can be utilized to increase memory bandwidth, reduce memory accesses, and improve overall performance.

Scalar replacement of array elements

Scalar replacement, or register pipelining, is an effective method to reduce the number of memory accesses. This method takes advantage of sequential multiple accesses to array elements by making them available in registers [19]. When executing a program, especially those with nested loops, one array element may be accessed in different iterations. In order to reduce the amount of memory accesses, the array element can be stored in registers after the first memory access, and the following references are replaced by scalar temporaries. This is especially beneficial for configurable systems as registers are essentially free in FPGAs compared to an ASIC implementation.

```
1 for (i=1; i<N-1; i++)
    for (j=1; j<M-1; j++){
3     // ...
      i00=in[i-1][j-1]; i01=in[i-1][j]; i02=in[i-1][j+1];
5     i10=in[i  ][j-1];                 ; i12=in[i
  ][j+1];
      i20=in[i+1][j-1]; i21=in[i+1][j]; i22=in[i+1][j+1];
7     // ...
    }
```

(a) Before scalar replacement

```
  // ... initial two iterations
2 for (i=3; i<N-1; i++)
    for (j=1; j<M-1; j++) {
4     // ...
      i00=i10; i01=i11; i02=i12;
  // scalar replacement
6     i10=i20; i11=i21; i12=i22;
  // scalar replacement
      i20=in[i+1][j-1]; i21=in[i+1][j]; i22=in[i+1][j+1];
8     // ...
    }
```

(b) After scalar replacement

Figure 4.10: Scalar replacement of array elements

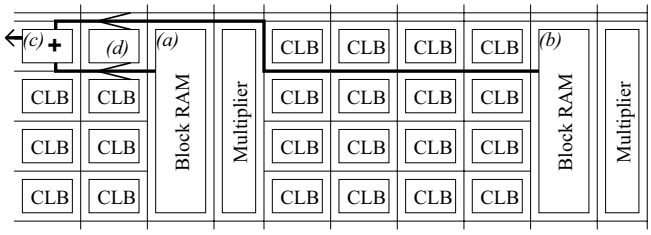
Consider the SOBEL edge detection code given in Figure 4.10. Part of the references to array `in[]` could be replaced by scalar temporaries obtained in the previous iterations. This reduces the number of memory accesses by approximately 62%. If the implementation is pipelined, the design has a better throughput, using the same memory ports configuration.

Data prefetching and buffer insertion

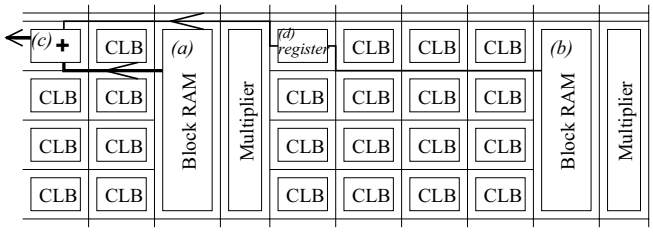
Data prefetching was originally introduced to reduce cache miss latencies [20]. The microprocessor issues a prefetching instruction to load a data block that is accessed in the near future. Prefetching avoids stalling by having the data readily accessible when it is needed. While it is loading data in the main memory, the microprocessor executes other computations that are independent of the data being fetched. Prefetching is most useful in programs that access large arrays sequentially. There are no caches in FPGA-based configurable architectures with block RAM modules. However, we can apply similar prefetching techniques to reduce the delay of the critical path, and improve system performance.

Before placement and routing, it is difficult to accurately estimate clock frequency, and to determine the number of clock cycles that it takes to access a particular block RAM module. An access to a block RAM module far away from the CLB may reduce the system's maximal frequency due to the interconnect delay, especially in high-speed designs. For example, in Figure 4.11(a), it is faster for CLB (c) to access block RAM (a) than to access block RAM (b).

In order to reduce the memory access time, we schedule the memory access one clock cycle earlier, and insert a register on the data path. Thus, the critical path is reduced and the data is available on time. Figure 4.11(b) shows a design in which the data in block RAM (b) is fetched one clock cycle earlier. This is similar to software prefetching. However, our



(a) Before transformation



(b) After transformation

Figure 4.11: Data prefetching and buffer insertion

goal is to reduce the critical path, or maximize the clock frequency.

4.5 Experimental Results

This section presents our experimental setup and results.

4.5.1 Experimental setup

Our benchmark suite consists of several DSP and image processing applications: `SOBEL` edge detection applies horizontal and vertical detection masks to an input image; `Bilinear` filtering is a suitable way to eliminate blocky textures in a 3-D image engine; `2D_Gauss` applies low-pass filtering to 2D arrays a.k.a. blurring 2D images; and `1D_Gauss` is more general low-pass filter. A number of DSP and image applications have the similar control structure and memory access patterns, such as texture smoothing

and convolution [48]. Except the SOBEL ones, all other four algorithm cores have the same input size and resource constraints.

The target architecture is the Xilinx Virtex II FPGA series, which contains evenly distributed block RAM modules. The target frequency was set to 150 MHz for our benchmark suite. This frequency represents a typical clock frequency of *high-speed designs* for the specific target Virtex II FPGA. There is no other special reason for us to select this particular clock frequency.

We partitioned the arrays using the algorithm proposed in Algorithm 1, performed program transformations, then used commercial tools to obtain area and timing results. Experiment results are collected after RTL synthesis and placement and routing.

Table 4.2 presents detailed results of these benchmarks. For each benchmark, there is an *original* design, where the iteration space and data spaces are not partitioned; a *partitioned* design, of which the iteration space and data spaces are partitioned under the resource constraints; and an *optimized* design, on which more memory optimizations, scalar replacements and buffer insertions, are applied. There are timing and area results for both pre-layout and post-layout designs. For each design, the number of clock cycles is reported. With the estimated clock frequencies, we estimate the execution time before actual physical synthesis. The areas are estimated as well. After placement and routing, the achieved clock frequencies and the areas are collected, and the execution times are calculated.

(a) SOBEL (small)

SOBEL (small)	# of cycles	Pre-layout Timing/Area			Post-layout Timing/Area		
		F(MHz)	L(ms)	A(%)	F(MHz)	L(ms)	A(%)
original	12,196	159.52	76.5	2.68	152.21	80.1	3.30
partitioned	2,032	150.60	13.5	11.70	140.85	14.4	12.77
optimized	263	185.19	1.4	9.45	150.83	1.7	13.95

(b) SOBEL (large)

SOBEL (large)	# of cycles	Pre-layout Timing/Area			Post-layout Timing/Area		
		F(MHz)	L(ms)	A(%)	F(MHz)	L(ms)	A(%)
original	29,718	160.9	184.7	3.32	151.19	196.6	4.10
partitioned	2,032	145.92	13.9	41.97	105.37	19.2	52.60
optimized	263	185.19	1.4	44.32	125.94	2.1	53.91

(c) SUSAN principle

SUSAN	# of cycles	Pre-layout Timing/Area			Post-layout Timing/Area		
		F(MHz)	L(ms)	A(%)	F(MHz)	L(ms)	A(%)
original	41,769	145.56	286.9	5.96	137.95	302.8	6.56
partitioned	17,409	173.28	100,5	22.01	143.25	121.5	24.12
optimized	9,293	127.50	72.9	21.35	133.60	69.6	26.17

(d) Bilinear filtering

Bilinear Filtering	# of cycles	Pre-layout Timing/Area			Post-layout Timing/Area		
		F(MHz)	L(ms)	A(%)	F(MHz)	L(ms)	A(%)
original	32,771	188.68	173.9	2.97	158.68	206.5	3.38
partitioned	10,243	204.04	50.2	6.17	146.54	69.9	6.99
optimized	4,608	180.96	25.5	4.94	172.62	26.7	6.48

(e) Gauss blurring

1-D Gauss Blurring	# of cycles	Pre-layout Timing/Area			Post-layout Timing/Area		
		F(MHz)	L(ms)	A(%)	F(MHz)	L(ms)	A(%)
original	32,776	150.47	217.8	3.13	146.16	224.3	3.83
partitioned	12,296	177.53	69.3	10.35	125.58	97.9	11.95
optimized	8,896	150.74	59.0	10.84	129.40	68.7	13.14

(f) 2D Gauss blurring

2-D Gauss Blurring	# of cycles	Pre-layout Timing/Area			Post-layout Timing/Area		
		F(MHz)	L(ms)	A(%)	F(MHz)	L(ms)	A(%)
original	40,963	155.33	263.7	3.13	155.33	263.7	3.61
partitioned	10,243	237.81	43.072	6.29	140.04	73.1	7.01
optimized	6,400	255.95	25.0	5.84	150.78	44.2	7.13

Table 4.2: Experimental Results

4.5.2 Experimental results

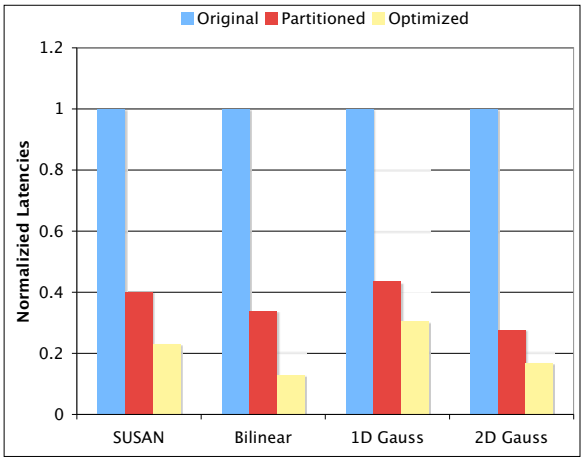


Figure 4.12: Normalized latencies

Figure 4.12 shows latencies of all designs normalized to the original un-partitioned designs. We found that the execution time of the partitioned designs is significantly smaller than that of the original one. (Since the SOBEL applications have different input sizes and resource constraints, their results are discussed in the later sections.)

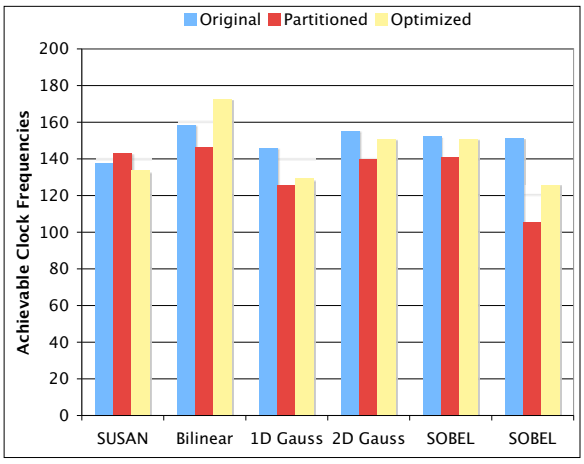


Figure 4.13: Maximum achievable frequencies

Figure 4.13 presents the maximum achievable clock frequencies. In most cases, the partitioned designs are about 10 percent slower than the original ones. However, after applying those optimization techniques, the achievable frequencies are about 7 percent faster than those of partitioned ones. Considering the area of partitioned designs and optimized designs are much larger than the original ones and with more complicated control, these results are quite good.

If we only partition the data arrays, the number of clock cycles is reduced, and the maximal frequencies after placement and routing are slower than our desired frequencies. In order to reduce memory accesses, optimization techniques such as scalar replacement for array elements and buffer insertion for data prefetching are utilized.

After partitioning, the average speed-up over the original is 2.75 times faster, and after further optimizations, the average speed-up is 4.80 times faster. Among half of those designs, the optimized designs could finally achieve the 150 MHz design goal.

Performance effects of partitioning decisions

Tables 4.1(a) and 4.1(b) show timing results for `SOBEL` edge detection with two different input image sizes. In the smaller design, we achieve the 150 MHz design goal, with a 46x speed-up compared to the original design. However, we could not achieve the design goal in the larger `SOBEL` design. The constraints on the block RAM modules result in the original design being partitioned into up to 16 portions, which is hard for later

stages of placement and routing.

It is interesting to note that after applying prefetching, both designs achieved a 185.0 MHz maximal frequency after RTL synthesis. After placement and routing, the frequency was drastically reduced to 125.9 MHz. This shows that as the number of partitions increases, the effects of physical designs on performance also increases. This result is also consistent with our motivation examples. Therefore, it is extremely important to consider physical attributes of the problem at the early stage of the design flow since these kinds of effects greatly influence the performance of the entire system.

Summary of experimental results

Architectural-level decisions on data partition and storage assignment in the early stage could affect the final result greatly. In general, a partitioned design decreases execution time, but occupies more memory and hardware resources. Different optimization techniques can be utilized to reduce memory access, and improve the overall performance. When the size of designs increase, it becomes more difficult to achieve design goals since it lacks the support from down-stream tools, especially physical design tools.

4.6 Summary

Modern configurable computing systems offer enormous computing capacities, and continue to integrate on-chip computation and storage components. Advanced synthesis tools are required to map large applications to these increasingly complicated chips. More importantly, these tools must be powerful and smart enough to conduct memory optimizations to effectively utilize on-chip distributed block RAM modules.

The proposed data and iteration space partitioning approach can be integrated with existing architectural-level synthesis techniques, parallelized input designs, and dramatically improved system performance. Experimental results indicated that partitioned designs achieve much better performance.

Chapter 5

Operation Scheduling

With the parallelized programs, the next step is to synthesize reconfigurable hardware from these graph-based representations. Resource allocation and scheduling, one of the most important problems in hardware synthesis, determines the start time of operations and minimizes the silicon area or latencies subject to timing or resource constraints. The quality of scheduling results greatly affects the quality of completed designs.

In this chapter, we present our work using the ant colony optimization (ACO) to solve the scheduling problem. We begin with an introduction to the timing-constraint scheduling and resource-constraint scheduling problems, and review representative scheduling algorithms in the literature. We then introduce the fundamental principles of the ACO algorithms, and the max-min ant system (MMAS) extensions. In Section 5.4, we present our work on resource-constraint scheduling using the MMAS optimization, and present experimental results. In Section 5.5, we present

the MMAS algorithm for the timing-constraint scheduling problem and experimental results. Finally, we summarize our work and lessons and observations for future algorithms design in Section 5.6.

5.1 Introduction

This section introduces the data-flow graph, the graph-model used in scheduling, and the problem formulation of the timing-constraint scheduling and resource-constraint scheduling problems.

5.1.1 Data-flow graph

Most research work in the literature uses the data-flow graph (DFG). A DFG is derived from a basic block, which is a sequence of operations $\mathbf{O} = \{o_1, \ldots, o_N\}$. A basic block usually contains no control structures, especially loops and backwards jumps. After conducting data-flow analysis on such a basic block, a DFG is constructed.

A DFG, denoted as $G(\mathbf{V}, \mathbf{E})$, is a directed acyclic graph. The vertices $\mathbf{V} = \{v_1, \ldots, v_N\}$ represent those operations $\mathbf{O}$.

The edges $\mathbf{E}$ describe timing constraints of the hardware behavior. Each edge $e(v_i, v_j)$ shows a *chained dependency* from operation o_i to operation o_j, denoted as $v_i \preccurlyeq v_j$. Such a chained dependency is defined as

$$f_i \leq s_i, \tag{5.1}$$

i.e. operation o_j can only start after the completion of operation o_i. In

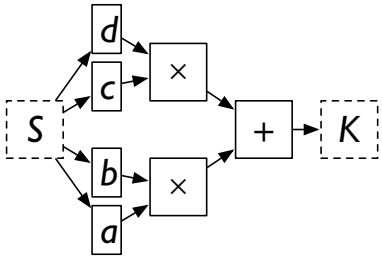
other words, an operation can only start when all its predecessors have finished. This work assumes that edges do not carry any delays.

In order to clarify this model, two *virtual* vertices, v_S and v_K, are added to the DFG. These two vertices are associated with null operations. Hence, the delays of these two virtual vertices are *zero*. It is further assumed that, for any vertex $v_i \in \mathbf{V}$, $v_S \preccurlyeq v_i$ and $v_i \preccurlyeq v_K$ are defined, i.e. v_S is the only source vertex in the DFG, and v_K is the only sink vertex. v_S starts before the start of any other vertex $v_i \in \mathbf{V}$ and v_K finishes after the completion of any other vertex v_i.

For example, a simple piece of C program and the corresponding DFG is shown in Figure 5.1, where the program reads four integer numbers and writes to the direct output the sum of the product of `a` and `b` and the product of `c` and `d`, and two virtual nodes v_s and v_k are added.

```
1 int foo (int a, int b,
           int c, int d)
3 {
    return a*b + c*d;
5 }
```

(a) A simple C program



(b) The corresponding DFG

Figure 5.1: A DFG example

The main limit of the DFG is that the DFG is an acyclic graph. Some complicated timing constraints cannot be represented in the DFG without breaking this rule. For example, it is impossible to show feedback constraints for pipelined hardware designs. In addition, it is hard to show

some specific schedule arrangement, for example two operations are required to schedule at the same clock cycle.

5.1.2 Resource allocation

Traditionally, scheduling is a separate phase after resource allocation. Resource allocation determines how many of a particular type of hardware resources are available.

A technology library consists of various hardware resource types, denoted by $\mathbf{Q} = \{q_0, \ldots, q_M\}$. Each component $q_i(A_i, T_i, M_i, \mathbf{O}_{q_i})$ has its area A_i and timing information T_i, and a set of operations $\mathbf{O}_{q_i}$ supported by this component, where $\mathbf{O}_{q_i} \subset \mathbf{O}$ and $\bigcup_i \mathbf{O}_{q_i} = \mathbf{O}$.

When each of the operations, O_i, is uniquely associated with one resource type q_j, this is called *homogenous* scheduling. If an operation can be performed by more than one resource type, this is called *heterogeneous* scheduling [120].

Most of resource constraints are introduced from the target architectures and technology libraries. For example, if an integer array is mapped to a single-port memory block RAM, the number of available memory ports is 1, and only one memory access operation to this array is scheduled in the same clock cycle. If the same array is mapped to a dual-port memory, then two memory accesses are allowed in one clock cycle.

In order to achieve particular design goals, designers specify resource constraints. For example, integrated multipliers or DSP blocks are considered precious in FPGA architectures. It is normal to limit the number

of available multipliers less than a specific number.

It is not recommended for designers to specify other resource constraints if the synthesis tool is powerful enough to generate designs using as few hardware resources as possible.

5.1.3 Problem formulations

The scheduling problem is to determine the start time of each operation in the DFG. Much of the research work in the literature uses one clock cycle as the minimum time unit in scheduling. It takes one or multiple cycles for a hardware component to complete an operation. Therefore, the start time of an operation o_i, denoted as s_i, states that this operation should start in the beginning of the clock cycle s_i. If this operation is assigned to a resource type $q_j(A_j, D_j, M_j, O_{q_j})$, the finish time of this operation, denoted as f_i, is the end of clock cycle $s_i + D_j - 1$.

The objective of this problem is to minimize the total number of required hardware resources, $\sum_i a_i where q_i \in \mathbf{Q}$, subject to the specified maximum control-steps. This is called timing constraint scheduling (TCS).

In some designs where the latency is a more important design goal, the objective is to minimize the number of control-steps given the resource allocation results, i.e. the available number of each resource type is specified. This is called resource constraint scheduling (RCS), which is a dual problem of the TCS problem. The TCS differs from the RCS on the objective to generate a schedule as short as possible.

Depending on different priority of hardware designs, there are other

objectives in the resource allocation and scheduling problem, and this could be further formulated as a multiple objective optimization problem. However, our research work is focused on the fundamental RCS/TCS problems.

5.2 Related Work

The scheduling problems are *NP*-hard [12]. Exact solutions with feasible complexities are available only for a very limited subset of this problem, such as Hu's algorithm [68]. Although it is possible to formulate and solve the problem using Integer Linear Programming (ILP)[80, 129], the feasible solution space quickly becomes intractable for larger problem instances.

In order to address these problems, researchers proposed varieties of heuristic methods with polynomial complexity. A number of algorithms for the RCS problem exist, including list scheduling [120, 1], forced-directed scheduling [96], genetic algorithm [49], tabu search [10], and simulated annealing [116]. Among these methods, list scheduling is the most common due to its simplicity of implementation and capability of generating reasonably good results for small-sized problems.

Many TCS algorithms used in high-level synthesis are derived from the force-directed scheduling (FDS) algorithm presented by Paulin and Knight [96, 97]. Verhaegh,*et al* [122, 123] and enhanced and extended this algorithm. Park and Kyung [93] addressed the issue of the FDS lacking a

look-ahead scheme by applying iterative approaches based on Kernighan and Lin's heuristic [71], solving the graph-bisection problem. More recently, Heijligers,*et al* [60] and InSyn [110] use evolutionary techniques like genetic algorithms and simulated evolution.

This section presents the most fundamental work, as soon as possible (ASAP) scheduling, as late as possible (ALAP) scheduling, and discusses the concept of *mobility* in Section 5.2.1. Section 5.2.2 presents the list scheduler. Section 5.2.3 presents the force-directed scheduling (FDS) algorithm. Solutions for optimal solutions based on ILP and other approaches are presented in Section 5.2.4.

5.2.1 ASAP/ALAP scheduling

The simplest scheduling problem is the *unconstraint* scheduling problem. The unconstraint scheduling problem is to exploit a schedule of a number of data operations with unlimited hardware resources without any timing constraints. The as soon as possible (ASAP) scheduling is a simple and fast solution to this problem. As presented in Algorithm 3, each operation is scheduled on the fastest functional units in the earliest possible clock cycle. Because of its earliest possible schedule, it is closely related with finding the longest path from the virtual source vertex v_s to an operation.

Correspondingly, there is the so-called as late as possible (ALAP) scheduling, where each operation is scheduled to the latest opportunity. As shown in Algorithm 4, this can be done by calculating the longest

Algorithm 3 ASAP scheduling

Require: vertices **V** is sorted by the partial order ($\preccurlyeq$) relationship

1: $s_s = 0$; $f_s = 0$;
2: **for all** $v_i \in \mathbf{V}$ **do**
3: $s_i = 0$;
4: **for all** v_j, where $v_j \preccurlyeq v_i$ **do**
5: $s_i = max(f_j, s_i)$;
6: **end for**
7: update f_i;
8: **end for**
9: return f_k;

path from an operation to the virtual sink vertex v_k. The ALAP schedule provides the upper bound for the starting time of each operation in order to finish the computation task before the returned shortest latency f_k.

Algorithm 4 ALAP scheduling

Require: vertices **V** is sorted by the *reversed* partial order ($\succcurlyeq$) relationship

1: $s_k = 0$; $f_k = 0$;
2: **for all** $v_i \in \mathbf{V}$ **do**
3: $s_i = 0$;
4: **for all** v_j, where $v_j \succcurlyeq v_i$ **do**
5: $f_i = min(s_j, f_i)$;
6: **end for**
7: update s_i
8: **end for**
9: **for all** $v_i \in \mathbf{V}$ **do**
10: $s_i -= s_s$;
11: **end for**
12: return f_k;

Because the ASAP and ALAP scheduling are conducted as unconstraint scheduling, they are not used to generate the scheduling results but to act as critical parts of advanced scheduling algorithms to exploit the characteristics of the program behavior.

The *mobility* m_i of an operation o_i is one of the most important attributes of an operation, which describes the range of moving an operation subject to the latency constraint. Therefore, mobility is defined by the ASAP and ALAP scheduling result $[s_i^S, s_i^L]$.

The ASAP schedule provides the lower bound for the starting time of each operation, together with the lower bound of the overall application latency. This lower bound of the application latency can be derived from the ALAP scheduling results as well.

The upper bound of the application latency (under a given technology mapping) can be obtained by serializing the DFG; that is, to perform the operations sequentially based on a topologically sorted sequence of the operations. This is equivalent to having only one unit for each type of operation.

5.2.2 List scheduling

List scheduling is a commonly used heuristic for solving a variety of RCS problems [108, 99]. It is a generalization of the ASAP algorithm with the inclusion of resource constraints [72].

The list scheduling algorithm iteratively constructs a schedule using a *prioritized ready list*, as shown in Algorithm 5. Initially, the prioritized ready list L is empty and the virtual source vertex is scheduled at time 0. During each iteration, the list scheduler updates the priority ready list. If an operation whose preceding operations are all scheduled, then this operation is ready and it is inserted into the list by its priority. The pri-

ority can be the mobility, the number of succeeding operations, the depth from the virtual source vertex in the DFG, and so forth. If more than one ready operations share the same priority, ties are broken randomly. After that, the list scheduler checks whether it is possible for a ready operation to assign an available hardware resource in this control step. If all operations in the priority list are checked, this iteration is done. Scheduling an operator to a control step makes its successor operations ready, which is added to the ready list in the next iteration. This process is carried out until all of the operations have been scheduled.

Algorithm 5 List scheduling

```
1: initialize the empty priority ready list L;
2: cycle = 0; s_s = 0; f_s = 0;
3: repeat
4:   for all v_i ∈ V and v_i ∉ L do
5:     if v_i is not scheduled and ready now then
6:       insert v_i to the right position of L;
7:     end if
8:   end for
9:   for each v_i ∈ L do
10:    if an idle component q exists then
11:      schedule v_i on q at time cycle;
12:    end if
13:  end for
14:  cycle = cycle + 1;
15: until the virtual sink vertex is scheduled
16: return f_k
```

The success of the list scheduler is highly dependent on the priority function and the structure of the input application (DFG) [72, 116, 86]. One of the commonly used priority functions is the priority inversely proportional to the mobility, which ensures that operations with large mo-

bility are scheduled later because they have more flexibility as to when they can be scheduled. Many other priority functions have been proposed [1, 8, 49, 72]. However, it is commonly agreed that there is no single good heuristic for prioritizing the DFG nodes across a range of applications using list scheduling. Our results in Section 5.4 confirm this.

Given that the DFG is a directed acyclic graph, it is easy to prove that the list scheduler always generates feasible schedules. However, the list scheduler often fails at generating pipelined designs because of the lack of look-ahead abilities.

5.2.3 Force-directed scheduling

The force-directed scheduling (FDS) algorithm [96] selects candidate operations and schedules them in proper control steps by calculating *force*, which attract operations into a specific control step on proper resource types or repel them from those control steps. The objective is to distribute operations uniformly onto available resource units subject to timing constraints. This distribution ensures that hardware resources that are allocated to perform operations in one control step are used efficiently in other control steps, which leads to a high utilization rate.

As discussed in Section 5.2.1, the ASAP and ALAP scheduling results define the mobility $[s_i^S, s_i^L]$ of an operation o_i. Therefore, given a specific resource type, the *operation probability*, which is the probability of that

operation o_i is active at time step j, can be calculated as follows.

$$p(i,j) = \begin{cases} \sum_{l=0}^{D_i} H_i(j-l)/(s_i^L - s_i^S + 1) & \text{if } s_i^S \leqslant j \leqslant s_i^L + D_i, \\ 0 & \text{otherwise.} \end{cases} \tag{5.2}$$

where $H(.)$ is a unit window function defined on $[s_i^S, s_i^L + D_i]$, and D_i is the delay in time steps to perform operation o_i.

One specific resource type may be suitable for more than one data operations. The *type distribution* for type k resource is the summation of probabilities of all these operations for each time step j.

$$q(k,j) = \sum_i p(i,j), \tag{5.3}$$

where the type k resource type is able to implement operation o_i. It is obvious that $q(k,j)$ is an estimation on the number of type k resources that are required at time step l.

The FDS algorithm tries to minimize the overall concurrency under a fixed latency by scheduling operations one by one. The larger the concurrency, the larger the forces evenly distribute operations among time steps. Forces are comprised of two portions, *self-force* and *predecessor/successor forces*. The self-force of scheduling operation o_i on time step j, denoted as $sf(i,j)$, represents the direct effect of this scheduling on the overall concurrency of type k resource.

$$sf(i,j) = \sum_{l=s_i^S}^{s_i^L + D_i} q(k,l) \cdot (H_i(l) - p(i,j)) \tag{5.4}$$

where $s_i^S \leqslant j \leqslant s_i^L + D_i$, k is the resource type of operation o_i scheduled on, and $H_i(\cdot)$ is the unit window function defined on $[j, j + D_i]$.

The predecessor/successor forces derived from the effects of scheduling an operation to a time step affecting the mobility of preceding and succeeding operations. When assigning operation o_i to time step j, the mobility of a predecessor or successor operation o_l may change from $[s_l^S, s_l^L]$ to $[\widetilde{s}_l^S, \widetilde{s}_l^S]$.

$$psf(i,j,l) = \sum_{m=\widetilde{s}_i^S}^{\widetilde{s}_i^L+D_l} q(k,m) \cdot \widetilde{p}(l,m) - \sum_{m=s_i^S}^{s_i^L+D_l} q(k,m) \cdot p(m,l) \tag{5.5}$$

where $\widetilde{p}(l,m)$ is computed in the same way as the operation probability above, except the updated mobility information $[\widetilde{s}_l^S, \widetilde{s}_l^S]$ is used.

Therefore, the *total force* of the candidate schedule for operation o_i on time step j is the self-force and the summation of all the predecessor/successor forces.

$$f(i,j) = sf(i,j) + \sum_l psf(i,j,l) \tag{5.6}$$

where o_l is a predecessor or successor of op_i.

The FDS algorithm starts from the virtual source vertex. The total forces are calculated for each unscheduled operations at every possible time step. The operation and time step with the best force reduction is chosen and the partial scheduling result is incremented until all the operations have been scheduled. The algorithm is shown in Algorithm 6.

The FDS method is *constructive* because the solution is computed without any backtracking. Every decision is made in a greedy manner. If there are two possible assignments sharing the same cost, the above algorithm cannot accurately estimate the best choice. Moreover, FDS does not take into account future assignments of operators to the same control step. Consequently, it is likely that the resulting solutions are not optimal, due

Algorithm 6 Force-directed scheduling

```
conduct the ASAP and ALAP scheduling;
initialize mobility range [s^S, s^L];
calculate operation/type probabilities;
while exists unscheduled instruction do
  for each unscheduled instruction o_i do
    for each j that s_i^S ⩽ j ⩽ s_i^L do
      calculate sf(i,j); f(i,j) = sf(i,j);
      for each predecessor/sucessor o_l of o_i do
        calculate psf(i,j,l);
        f(i,j) += psf(i,j,l);
      end for
      update the smallest force f;
      update the candidate operation o and time step t;
    end for
  end for
  update the mobility of predecessors and successors of operation o;
  Update the operation/type probabilities
end while
```

to the lack of a look-ahead scheme and the lack of compromises between early and late decisions.

5.2.4 Integer linear programming

Both time and resource constrained problems can be formulated as integer linear programming (ILP) problems. The ILP solvers try to find an optimal solution using a branch-and-bound search algorithm.

An ILP model is provided for the heterogeneous RCS problem. Though this is focused on RCS, it is possible to utilize the similar model to solve other scheduling problems. The need for formulating this is supported by the lack of references for the same problem in existing research literatures. Most of the ILP formulations for scheduling problems that can be

found are done for homogenous resources, i.e. the execution time for a certain type of operation is a constant. The scheduling program is formally described by the following integer linear program.

The inputs of this ILP problem are as follows.

- A set of vertices $\mathbf{V} = v_1, v_2, ...$, representing the operations in the program.
- Associated with each vertex $v_i \in \mathbf{V}$, are non-negative integers $D_{i,j}$, where $j = 1, 2, \ldots, |q_i|$, representing the delays of different implementations.
- A directed acyclic graph (DAG) $G(\mathbf{V}, \mathbf{E})$. $\mathbf{E}$ is a set of edges $e(i, j)$, where $v_i, v_j \in \mathbf{V}$. An edge $e(i, j) \in \mathbf{E}$ implies that $o_i \preccurlyeq o_j$. The virtual source and sink nodes in the graph G are identified as v_S and v_K respectively.
- One non-negative integer valued parameter D is specified. D is the deadline constraint, i.e. the time between the start time of the source v_S and the finish time of the sink v_K should be at most D. D could be easily obtained by serializing the graph G.

Some variables are defined as follows.

- For each $v_i \in \mathbf{V}$, define a set of binary variables m_{ij} such that $m_{ij} = 1$ if and only if operation i is mapped to implementation q_j; otherwise, $m_{ij} = 0$. In general, there are at most I implementations per operation. ($N \times I$ variables)

- For each $v_i \in \mathbf{V}$, define a non-negative integer s_i, the starting time of operation o_i. (N variables)

- For each $v_i, v_j \in \mathbf{V}$, define a binary variable p_{ij} such that $p_{ij} = 1$ if $s_i \leq s_j$; otherwise, $p_{ij} = 0$. ($N \times (N-1)$ variables)

The objective function is to minimize the execution time

$$min(s_K). \tag{5.7}$$

This process is subject to the following constraints.

- Implementation constraints ensure that only one implementation is selected for every operation. (N constraints)

$$\sum_{j \in I_i} m_{ij} = 1 \tag{5.8}$$

- Precedence constraints ensure the dependencies defined in G are satisfied. ($2E + N \times (N-1)$ constraints)

$$s_i + \sum_{k \in I_i} D_{ik} m_{ik} \leq s_j, \text{ where } (i,j) \in \mathbf{E} \tag{5.9}$$

$$p_{ij} = 1, \text{ where } (i,j) \in \mathbf{E} \tag{5.10}$$

and

$$p_{ij} + p_{ji} = 1, \text{where} i \neq j, \text{ and } i, j = 1, \ldots, N. \tag{5.11}$$

- Functional units overlapping constraints ensure that no two operations can be scheduled simultaneously on the same functional units. (much less than $I \times (N \times (N-1))$ constraints)

$$s_i + L_{ip} - s_j \leq D(3 - p_{ij} - m_{ip} - m_{jp}), \tag{5.12}$$

where $i \neq j$, and $i, j = 1, \ldots, N$. The above inequality is restrictive only when both $m_{ip} = 1$ and $m_{jp} = 1$, and $p_{ij} = 1$, i.e. both operations i and j can be implemented on functional units p, and operation i are scheduled first. In this case, it guarantees that the finish time of operation i should be less than the start time of operation j.

- Bounds limit all variables in a small range. (N constraints)

$$s_S = 0 \tag{5.13}$$

$$0 \leq s_i \leq D, \text{ where } i = 1, \ldots, N \tag{5.14}$$

A solution to the scheduling problem is specified completely by m_{ij} (mapping) and s_i (schedule). The formulation has at most $N^2 + I \times N$ variables and at most $N^2 + N + 2E + I(N^2 - N)$ constraints, in addition to the integrality constraints on the variables.

Other scheduling problems, such the TCS problem and the pipelining problem, can be formulated in a similar way.

It is obvious that the ILP formulation increases rapidly with the number of operations, dependencies, control steps, and feasible choices of hardware resources. Therefore the time of execution of the algorithm also increases rapidly. In practice, the ILP approach is applicable only to rather small designs.

5.3 The ant colony optimization

The section briefly introduces the ant colony optimization (ACO) metaheuristic and the max-min ant system (MMAS) optimization.

5.3.1 The ACO algorithm

The ACO algorithm, originally introduced by Dorigo *et al* [34], is a cooperative heuristic searching algorithm inspired by ethological studies on the behavior of ants.

It was observed [33] that ants – who lack sophisticated vision – manage to establish the optimal path between their colony and a food source within a very short period. This is done via indirect communication known as *stigmergy* via the chemical substance, or *pheromone*, left by the ants on the paths. Each individual ant makes a decision on its direction biased on the *strength* of the pheromone trails that lie before it, where a higher amount of pheromone hints a better path. As an ant traverses a path, it reinforces that path with its own pheromone. A collective autocatalytic behavior emerges as more ants choose the shortest trails, which in turn creates an even larger amount of pheromones on those short trails, making those short trails more likely to be chosen by future ants.

The ACO algorithm is inspired by this observation. It is a population-based approach where a collection of ants cooperates to explore the search space. They communicate via mechanisms that imitate the pheromone trails.

One of the first problems to which ACO was successfully applied was the Traveling Salesman Problem (TSP) [34], for which it gave competitive results compared to traditional methods. The TSP can be modeled as a complete weighted directed graph $G(\mathbf{V},\mathbf{E},d)$, where $V = \{v_1,\ldots,v_N\}$ is a set of vertices or cities, $\mathbf{E}$ is a set of edges, and d is a function that associates a numeric weight $d(i,j)$ for each edge $e(v_i,v_j) \in \mathbf{E}$. This weight is naturally determined as the distance between vertices v_i and v_j in the TSP. The objective is to find the shortest Hamiltonian path of the graph G.

In order to solve the TSP, the ACO algorithm associates a pheromone trail $\tau(i,j)$ for each edge $e(v_i,v_j) \in \mathbf{E}$. The pheromone indicates the attractiveness of the edge and serves as a global distributed heuristic. Initially, $\tau(i,j,0)$ is set with some fixed value T_0. During each iteration, M agents (ants) are released randomly on the cities, and each starts to construct a tour. Every agent has memory about the cities it has visited so far in order to guarantee the constructed tour is a Hamiltonian path. If at step t the agent is at city i, the agent chooses the next city j probabilistically using the following:

$$p(i,j) = \begin{cases} \frac{\tau^{\alpha}(i,j,t)\cdot\eta^{\beta}(i,j)}{\sum_k(\tau^{\alpha}(i,k,t)\cdot\eta^{\beta}(i,k))} & \text{if is } j \text{ not visited} \\ 0 & \text{otherwise} \end{cases} \tag{5.15}$$

where edges $e(v_i,v_k)$ are all the allowed moves from v_i, $\eta(i,k)$ is a local heuristic which is defined as the inverse of $d(i,j)$, α and β are parameters to control the relative influence of the distributed global heuristic $\tau(i,k,t)$ and local heuristic $\eta(i,k,t)$.

Intuitively, the ant favors a decision on an edge that possesses a higher

volume of pheromones and a better local distance. At the completion of each iteration, amounts of the pheromones are updated to favor the edges from the solutions found by the ants in the current iteration.

At the end of every iteration, certain amounts of new pheromones are released on tours those agents constructed.

$$\Delta\tau_a(i,j) = \begin{cases} Q/l_a & \text{if edge } e(v_i,v_j) \text{ in the tour ant } a \text{ constructed} \\ 0 & \text{otherwise} \end{cases} \tag{5.16}$$

where Q is a fixed constant to control the delivery rate of the pheromone, and l_a is the tour length for ant a.

In the meantime, a certain amount of pheromone evaporates from every edge. More specifically, it is $\rho \cdot \tau(i,j,t-1)$, where ρ is the evaporation ratio and $0 < \rho < 1$.

Therefore, the updated pheromone trail on edge $e(v_i,v_j)$ at iteration t is defined as

$$\tau(i,j,t) = \rho \cdot \tau(i,j,t-1) + \sum_{a=1}^{M} \Delta\tau_a(i,j) \tag{5.17}$$

Two important operations are taken in this pheromone trail updating process. The evaporation operation is necessary for the ACO to be effective in exploring different parts of the search space, while the reinforcement operation ensures that frequently used edges and edges contained in the better tours receive a higher volume of pheromones and have a better chance to be selected in the future iterations of the algorithm. The above process is repeated multiple times until a certain ending condition is reached. The best result found by the algorithm is reported.

Researchers have since formulated ACO methods for a variety of traditional $\mathcal{NP}$-hard problems. These problems include the maximum clique problem [40], the quadratic assignment problem [45], the graph coloring problem [27], the shortest common super-sequence problem [81, 85], and the multiple knapsack problem [42]. ACO also has been applied to practical problems such as the vehicle routing problem [44], data mining [94], the network routing problem [106], and the system level task partitioning problem [125, 126].

The convergence of the ACO was investigated by Gutjahr [52]. It was shown that ACO with a time-dependent evaporation factor or a time-dependent lower pheromone bound converges to an optimal solution with probability of exactly one. The result enhanced the work presented by Gutjahr [53, 51, 113] for the ACO algorithm. It turns out that a convergence guarantee can be obtained by a suitable speed of cooling (i.e., reduction of the influence of randomness). This is similar to the optimality proof for the simulated annealing meta-heuristic. A geometric decrease in pheromones on non-reinforced arcs is too fast and may lead to premature convergence to suboptimal solutions. On the other hand, introducing a fixed lower pheromone bound stops the cooling at some point and leads to random-search-like behavior without convergence. In between lies a compromise of allowing pheromone trails to move towards zero, but at a slower than geometric rate. This can be achieved either by decreasing the evaporation factors, or by decreasing the lower pheromone bounds.

5.3.2 The max-min ant system (MMAS) optimization

Premature convergence to local minima is a critical algorithmic issue that can be experienced by all evolutionary algorithms. Balancing exploration and exploitation is not trivial in these algorithms, especially for algorithms that use positive feedback such as ACO [34]. The max-min ant system (MMAS) is specifically designed to address this problem.

The MMAS [114] is built upon the original ant system algorithm. It improves the original algorithm by providing dynamically evolving bounds on the pheromone trails such that the heuristic is always within a limit compared with that of the best path. As a result, all possible paths have a non-trivial probability of being selected and thus it encourages broader exploration of the search space.

The MMAS forces the pheromone trails to be limited within evolving bounds, that is for iteration t, $\tau_{min}(t) \leqslant \tau(i,j,t) \leqslant \tau_{max}(t)$. If we use f to denote the cost function of a specific solution S, the upper bound τ_{max} [114] is defined as follows.

$$\tau_{max}(t) = \frac{1}{1-\rho}\frac{1}{f(\mathbf{s}(t-1))} \tag{5.18}$$

where $\mathbf{s}(\cdot)$ is the global best solution found so far. The lower bound is defined as follows.

$$\tau_{min}(t) = \frac{\tau_{max}(t)(1-\sqrt[n]{p_{best}})}{(avg-1)\sqrt[n]{p_{best}}} \tag{5.19}$$

where $p_{best} \in (0,1]$ is a controlling parameter to dynamically adjust the bounds of the pheromone trails. The physical meaning of p_{best} is that it indicates the conditional probability of the current global best solution

$\mathbf{s}(t)$ being selected, given that all edges not belonging to the global best solution have a pheromone level of $\tau_{min}(t)$ and all edges in the global best solution have $\tau_{max}(t)$. Here avg is the average size of the decision choices over all the iterations. For a TSP problem of n cities, $avg = N/2$. It is noticed from (5.19) that lowering p_{best} results in a tighter range for the pheromone heuristic. As $p_{best} \to 0$, $\tau_{min}(t) \to \tau_{max}(t)$, which means more emphasis is given to search space exploration.

Theoretical treatments of using the pheromone bounds and other modifications on the original ant system algorithm are proposed in [114]. These include a pheromone updating policy that only utilizes the best performing ant, initializing pheromone with τ_{max}, and combining local search with the algorithm. It was reported by the authors that MMAS was the best performing AS approach and provided very high quality solutions.

5.4 Resource constraint scheduling

In this section, we present our algorithm of applying the ant system, or more specifically, the max-min ant system (MMAS) [114] optimization, to solve the resource constraint scheduling (RCS) problem.

5.4.1 Algorithm formulation

The MMAS resource-constrained scheduling algorithm, as shown in Algorithm 7, combines the MMAS approach with the traditional list scheduling algorithm, and formulates the problem as an iterative

searching process over the design space.

Algorithm 7 MMAS resource-constraint scheduling

```
1: initialize parameter ρ, τ_ij, p_best, τ_max, τ_min;
2: construct M ants;
3: BestSolution = ∅
4: while ending condition is not met do
5:   for each m that 1 ⩽ m ⩽ M do
6:     ant m constructs a list L_m of vertices using global heuristic τ; and
       local heuristic η
7:     conduct list scheduling on G(V,E) using the list L_m;
8:     update BestSolution
9:   end for
10:  update heuristic boundaries τ_max and τ_min;
11:  update local heuristics η if needed;
12:  update τ(i, j,t) based on (5.21);
13: end while
14: return BestSolution;
```

Each iteration consists of two stages. First, a collection of ants traverse the DFG to construct individual operation lists using global and local heuristics associated with the DFG vertices. Then, these results are evaluated in a list scheduler. Based on the evaluation, the heuristics are updated to favor better solutions. The hope is that further iterations benefit from the updates and come up with better priority list.

Similar to the algorithm presented in Section 5.3, each DFG vertex v_i is associated with a set of pheromone trails $\tau(i, j)$. Each trail indicates the global favorableness of assigning the i-th vertex to the j-th position in the priority list, where $j = 1, \ldots, N$. Since it is valid for an operation to be assigned to any position in the priority list, every possible pheromone trail is valid. Initially, $\tau(i, j)$ is set with some fixed value T_0.

During each iteration, M ants are released and each starts to construct

an individual priority list by filling the list with an operation every step. Every ant has memory of those operations it already selected. Upon starting step j, the ant already selected $j-1$ operations of the DFG. To fill the j-th position of the list, the ant chooses the next operation o_i probabilistically according to the probability as follows.

$$p(i,j) = \begin{cases} \frac{\tau^{\alpha}(i,j,t)\cdot\eta^{\beta}(i,j)}{\Sigma_k(\tau^{\alpha}(i,k,t)\cdot\eta^{\beta}(i,k))} & \text{if operation } o_j \text{ is not scheduled yet} \\ 0 & \text{otherwise} \end{cases} \tag{5.20}$$

where $\eta(i,k)$ is a local heuristic of selection operation v_k, and α and β are parameters to control the relative influence of the distributed global heuristic $\tau(i,k,t)$ and local heuristic $\eta(i,k,t)$.

The local heuristic η gives the local favorableness of scheduling the i-th operation at the j-th position of the priority list. Different well-known heuristics [86] are tested here.

1. **Instruction mobility** (IM): The mobility of an operation is determined by the difference between the ALAP and ASAP schedules. The smaller the mobility, the more urgent the operation is. When the mobility is zero, the operation is on the critical path.

2. **Instruction depth** (ID): Instruction depth is the length of the longest path in the DFG from the operation to the sink. It is an obvious measure of the priority for an operation as it gives the number of operations that must be scheduled after.

3. **Latency-weighted instruction depth** (LWID): This is computed

in a similar manner as ID, except vertices along the path to the virtual sink vertex are weighted using the latency of the operation.

4. **Successor number** (SN): This is to benefit vertices with many successors, which is more likely to make other vertices be scheduled earlier.

The second stage is the result quality assessment and pheromone trail updating step.

$$\tau(i,j,t) = \rho \cdot \tau(i,j,t-1) + \sum_{h=1}^{M} \Delta\tau_h(i,j) \tag{5.21}$$

$$\Delta\tau_h(i,j) = \begin{cases} Q/l_h & \text{if } op_i \text{ is scheduled at } j \text{ by ant } h \\ 0 & \text{otherwise} \end{cases} \tag{5.22}$$

where l_h is the total latency of the scheduling result generated by ant h, and ρ is the evaporation ratio and $0 \leqslant \rho \leqslant 1$.

5.4.2 Complexity analysis

List scheduling is a two-step process. In the first step, a priority list is built. The second step takes n steps to solve the scheduling problem since it is a constructive method without backtracking. For different heuristics, the complexity of the first step is different. When operation mobility, operation depth, and latency weight operation depth are used, it takes $O(n^2)$ steps to build the priority list since depth-first or breadth-first graph transverses are involved. When the successor vertex number is adopted as the list construction heuristic, it only takes n steps to do so. Therefore, the complexities for these methods are either $O(n^2)$ or $O(n)$ accordingly.

The force-directed resource constrained operation scheduling method is different. Though it is also a constructive method without backtracking, we need to compute the force of each operation at every step since the total latency is dynamically increased, based on whether there are enough resources to handle the ready operations. Thus the FDS method has $O(n^3)$ complexity.

The complexity of the proposed MMAS solution is determined mainly by the number of ants m and the total iteration N in every run. It also depends on the list scheduler that is utilized. If mN is proportional to n, we have one order higher complexity than the corresponding list scheduling approach. However, based on our experience, it is possible to fix such factor for a large set of practical cases such that the complexity of the MMAS solution is the same as the list scheduling approach.

5.4.3 Experimental results

Benchmarks

In order to test and evaluate our algorithms, we have constructed a comprehensive set of benchmarks. These benchmarks are taken from one of two sources. One source is from popular benchmarks used in previous literature. The benefit of having classic samples is that they provide a direct comparison between results generated by our algorithm and results from previously published methods. This is especially helpful when some of the benchmarks have known optimal solutions. In our final testing

benchmark set, seven samples widely used in operation scheduling studies are included. These samples focus mainly on frequently used numeric calculations performed by different applications.

In addition to these classic benchmarks, test cases from real-life applications in the MediaBench suite [78] are selected. The MediaBench suite contains a wide range of complete applications for image processing, communications, and DSP applications. These applications are analyzed using the SUIF [4] and Machine SUIF [112] tools. Thirteen DFGs are selected from core algorithms of these MediaBench applications.

Benchmark Name	# Nodes	# Edges	Depth
HAL	11	8	4
horner_bezier	18	16	8
AWR	28	30	8
motion_vectors	32	29	6
EWF	34	47	14
FIR2	40	39	11
FIR1	44	43	11
h2v2_smooth_downsample	51	52	16
feedback_points	53	50	7
collapse_pyr	56	73	7
COSINE1	66	76	8
COSINE2	82	91	8
write_bmp_header	106	88	7
interpolate_aux	108	104	8
matmul	109	116	9
idctcol	114	164	16
jpeg_idct_ifast	122	162	14
jpeg_fdct_islow	134	169	13
smooth_color_z_triangle	197	196	11
invert_matrix_general	333	354	11

Table 5.1: Benchmark node and edge count with the instruction depth assuming unit delay.

Table 5.1 lists all twenty benchmarks that were included in the benchmark set. Together with the names of the various functions where the basic blocks originated are the number of vertices, number of edges, and

operation depth (assuming unit delay for every operation) of the DFG.

Experimental results

The proposed MMAS scheduling algorithm was implemented and the quality of results is compared with the popularly used list scheduling and the known optimal solutions.

There are a set of different local heuristics available. For each local heuristic, five runs are conducted to obtain enough statistics for evaluating the stability of the algorithm. The number of ants per iteration, M, is set to 10. In each run, the scheduling algorithm stops after 100 iterations. The shortest latency is reported at the end of each run. The average value is reported here as the quality-of-results for the corresponding setting.

Experiments are conducted to solve two kinds of the RCS problems. They are the *homogenous* scheduling and the *heterogeneous* scheduling. The homogenous RCS problem allows only a single choice for each data operation type, and resource allocation is conducted prior to the scheduling. In this experiment, two types of functional units are allowed. They are multipliers and ALU, respectively. The ALU can implement most data operations other than multiplication. The number of each resource type is determined in the resource allocation stage and are less than the concurrency showing in the ASAP/ALAP scheduling, which guarantees that the test cases are not simplified to ASAP/ALAP scheduling.

Table 5.2 shows the testing results for the homogenous case. The best results for each case are shown in bold. Compared with a variety of list

	Size	Resources (MUL/ALU)	FDS	List Scheduling				MMAS (average over 5 runs)			
				IM	ID	LWID	SN	IM	ID	LWID	SN
HAL	(8/11)	(2 1)	**8**	10	**8**	**8**	**8**	**8.0**	**8.0**	**8.0**	**8.0**
horner_bezier_surf	(16/18)	(2 1)	**12**	16	**12**	13	13	**12.0**	**12.0**	**12.0**	**12.0**
ARF	(30/28)	(3 1)	18	19	**16**	18	18	**16.0**	**16.0**	**16.0**	**16.0**
motion_vectors	(29/32)	(3 4)	**12**	15	**12**	**12**	14	**12.0**	**12.0**	**12.0**	**12.0**
EWF	(47/34)	(1 2)	**21**	22	**21**	**21**	22	**21.0**	**21.0**	**21.0**	**21.0**
FIR2	(39/40)	(2 3)	17	19	18	17	15	17.0	**16.8**	17.0	17.0
FIR1	(43/44)	(2 3)	**16**	22	22	21	**16**	**16.0**	**16.0**	**16.0**	**16.0**
h2v2_smooth_downsample	(52/51)	(1 3)	23	28	23	23	**22**	22.4	22.8	22.8	22.8
feedback_points	(50/53)	(3 3)	16	20	**14**	19	**14**	14.4	14.2	14.6	14.6
collapse_pyr	(73/56)	(3 5)	**11**	12	**11**	**11**	**11**	**11.0**	**11.0**	**11.0**	**11.0**
COSINE1	(76/66)	(4 5)	16	18	16	17	16	**14.0**	**14.0**	**14.0**	**14.0**
COSINE2	(91/82)	(5 8)	14	18	14	17	13	**12.4**	**12.4**	12.6	12.8
write_bmp_header	(88/106)	(1 9)	**12**	17	**12**	**12**	**12**	12.8	12.6	12.8	12.4
interpolate_aux	(104/108)	(9 8)	13	16	12	16	16	**11.0**	11.8	**11.0**	11.8
matmul	(116/109)	(9 8)	15	14	**13**	14	14	13.6	13.8	13.8	13.8
idctcol	(164/114)	(5 6)	21	26	21	21	21	20.6	**19.8**	20.2	20.0
jpeg_idct_ifast	(162/122)	(10 9)	**19**	21	20	**19**	**19**	**19.0**	**19.0**	**19.0**	**19.0**
jpeg_fdct_islow	(169/134)	(5 7)	**21**	28	22	22	**21**	22.0	22.0	21.8	21.8
smooth_color_z_triangle	(196/197)	(8 9)	**24**	25	25	23	**24**	**24.0**	**24.0**	**24.0**	**24.0**
invert_matrix_general	(354/333)	(15 11)	26	28	28	25	25	**24.0**	**24.2**	**24.2**	**24.2**

Table 5.2: Result summary for the homogeneous resource constrained scheduling

scheduling approaches and the force-directed scheduling method, the proposed algorithm generates better results consistently over all test cases. This can be demonstrated by the number of times that it provides the best results for the tested cases. Compared with the list scheduling, the FDS generates more hits (10 times) for the best results, which is less than the worst case of the MMAS. For some of the test cases, our method provides significant improvement on the schedule latency. The greatest savings achieved is 22%. This is obtained for the COSINE2 when the instruction mobility is used as the local heuristic and as the heuristic for constructing the priority list for the traditional list scheduler. For test cases where this heuristic does not provide the best solution, the quality of results is much closer to the best than other methods.

It is important that a scheduling algorithm generate consistently good results over different input applications, besides the shortest absolute latency. As indicated in Section 5.2.2, the performance of traditional list scheduling heavily depends on the input. This is shown by the results of the list scheduling in Table 5.2. However, it is obvious that the proposed MMAS algorithm is much less sensitive to the choice of different local heuristics and input applications. This is evidenced by the fact that the standard deviation of the results achieved by the new algorithm is much smaller than that of the traditional list scheduler. Based on the data shown in Table 5.2, the average standard deviation for the list scheduler over all the benchmarks and different heuristic choices is 1.2, while that for the MMAS algorithm is only 0.19. In other words, we can expect

to achieve much more stable scheduling results on different application DFGs regardless of the choice of local heuristic. This is a great attribute desired in practice.

The second experiment, the heterogeneous RCS, allows more than one resource type qualified for a data operation type. For example, a multiplication can be implemented using a faster multiplier or a regular one.

The heterogeneous RCS experiments are conducted with the same configuration as that for the homogenous RCS ones. In order to better assess the quality of results, the same heterogeneous RCS tasks are formulated as an ILP problem, as described in Section 5.2.4, and exploiting the optima using CPLEX, a commercial ILP solver. Because solving the ILP problem is a time consuming process, the heterogeneous RCS experiments are only conducted on those several classic scheduling benchmarks.

Table 5.3 summarizes the heterogeneous RCS experiment results. Compared to a variety of list scheduling approaches and the force-directed scheduling method, the proposed algorithm generates better results consistently over all test cases. The greatest savings achieved is 23%. This is obtained for the FIR2 benchmark when the LWID is used as the local heuristic. Similar to the homogenous scheduling, the proposed algorithm outperforms other methods regarding consistently generating high-quality results. The average standard deviation for the list scheduler over all the benchmarks and different heuristic choices is 0.8128, while that for the MMAS algorithm is only 0.1673.

Though the results of the force-directed list scheduler are generally

	Size	Resources (A/FM/M/I/O)	CPLEX (lat./min.)	FDS	List Scheduling				MMAS (average over 5 runs)			
					IM	ID	LWID	SN	IM	ID	LWID	SN
HAL	(21/25)	1/1/1/3/3	8 / 32	**8**	**8**	**8**	9	**8**	**8**	**8**	**8**	**8**
ARF	(28/30)	2/1/2/0/0	**11**/22	**11**	**11**	13	13	13	**11**	**11**	**11**	**11**
EWF	(34/47)	1/1/1/0/0	27 / 24000	28	28	31	31	28	27.2	27.2	**27**	27.2
FIR1	(40/39)	2/0/2/3/3	13 / 232	19	19	19	19	18	17.2	17.2	**17**	17.8
FIR2	(44/43)	1/1/1/3/3	14 / 11560	19	19	21	21	21	**16.2**	16.4	**16.2**	17
COSINE1	(66/76)	2/1/23/3	†	18	19	20	18	18	**17.4**	18.2	17.6	17.6
COSINE2	(82/91)	2/1/2/3/3	†	23	23	23	23	23	**21.2**	**21.2**	**21.2**	**21.2**

Table 5.3: Result summary of the heterogeneous resource constraint scheduling

superior to that of the list scheduler, our algorithm achieves even better results. On average, compared to the force-directed approach, our algorithm provides a 6.2% performance enhancement for the test cases, while performance improvement for individual test samples can be as much as 14.7%.

Finally, compared to the optimal scheduling results computed by the ILP model, the results generated by the proposed algorithm are much closer to the optimal than those from the list scheduling and the force-directed approach. For all the benchmarks with known optima, our algorithm improves the average schedule latency by 44% compared to the list scheduling heuristics. For the larger size DFGs such as COSINE1 and COSINE2, CPLEX fails to generate optimal results after more than 10 hours of execution on a SPARC workstation with a 440MHz CPU and 384MByte memory. In fact, CPLEX crashes for these two cases because of running out of memory. For COSINE1, CPLEX does provide an intermediate sub-optimal solution of 18 cycles before it crashes. This result is worse than the best result found by our proposed algorithm.

The evolutionary effect on the global heuristics τ_{ij} is illustrated in Figure 5.2. It plots the pheromone values for the ARF test case after 100 iterations of the proposed algorithm. The x-axis is the index of the vertex in the DFG, and the y-axis is the order index in the priority list passed to the list scheduler. There are a total of 30 vertices with vertex 1 and vertex 30 as the virtual source and sink vertices of the DFG, respectively. Each dot in the diagram indicates the strength of the pheromone trails for

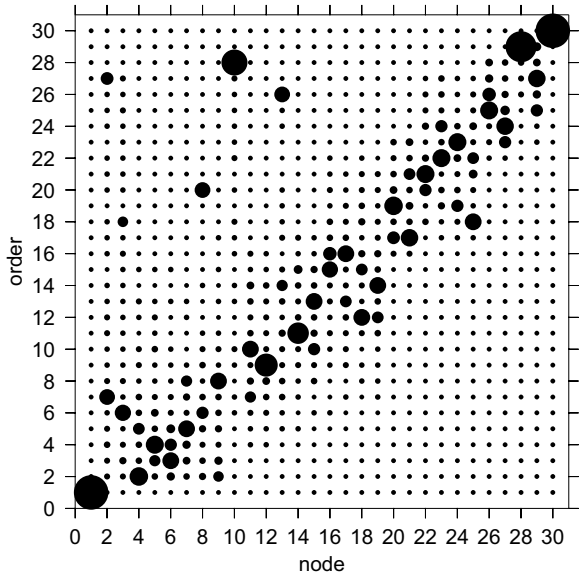


Figure 5.2: Pheromone Heuristic Distribution for ARF

assigning corresponding order to a certain operation – the larger the size of the dot, the stronger the value of the pheromone.

It is clearly seen from Figure 5.2 that there are a few strong pheromone trails while the rest are rather weak. It is interesting that, although a good amount of operations have a limited few alternative *good* positions, such as operation 6 and 26, the pheromone heuristics of most operations are strong enough to lock their positions. For example, according to its pheromone distribution, operation 10 shall be placed as the 28-th item in the list and there is no other competitive position for its placement. More important, this ordering preference cannot be trivially obtained by constructing priority lists with any of the popularly used heuristics discussed above. This shows that the proposed algorithm has the ability to discover a better priority ready list, which is hard to achieve intuitively.

5.5 Timing constraint scheduling

In this section, the proposed algorithm applying the ant system, or more specifically the max-min ant system (MMAS) [114] optimization, to solve the timing constraint scheduling (TCS) problem, is presented..

5.5.1 Algorithm formulation

The TCS problem is addressed here in an evolutionary manner. The proposed algorithm is built upon the MMAS optimization and is formulated as an iterative searching process, as shown in Algorithm 8. Each iteration consists of two stages. First, a collection of agents (ants) traverses the DFG to construct individual operation schedules subject to the specified deadline using global and local heuristics. Second, these results are evaluated concerning the resource cost. The heuristics are updated based on the characteristics of the best candidate solutions found in the current iteration. The hope is that future iterations benefit from these updates and result in better schedules.

In order to solve the TCS scheduling problem, each operation o_i is associated with L pheromone trails $\tau(i,j)$, where $j \in 1,\ldots,L$ and L is the specified deadline. These pheromone trails indicate the global favorableness of assigning the i-th operation at the j-th control step in order to minimize the resource cost subject to the latency constraint.

Initially, based on the ASAP/ALAP scheduling results, or more specifically the mobility range $[s^S, s^L]$, $\tau(i,j)$ is set with a fixed initial value T_0 if j

Algorithm 8 MMAS timing constraint scheduling

```
initialize parameter ρ, τ_ij, p_best, τ_max, τ_min
construct M ants
BestSolution = ∅
while ending condition is not met do
  for each m that 1 ⩽ m ⩽ M do
    ant (m) constructs a feasible schedule S^current subject to the timing
    constraints using Algorithm 9;
    update BestSolution;
  end for
  update heuristic boundaries τ_max and τ_min;
  update local heuristics η if needed;
  update τ(i, j, t) based on (5.24);
end while
return BestSolution
```

is a valid control step for op_i; otherwise, it is set to be 0.

During each iteration, m ants are released and each ant individually starts to construct a schedule by picking an unscheduled operation and determining its desired control step, as shown in Algorithm 9.

However, unlike the greedy approach used in the FDS method, each ant probabilistically picks up the next operation to be scheduled. The simplest way is to select an operation uniformly among all unscheduled operations. Once an operation o_i is selected, the ant needs to make decision onto which control step it should be assigned. This decision is also made probabilistically as illustrated in Equation (5.23).

$$p(i,j) = \begin{cases} \frac{\tau^{\alpha}(i,j,t)\cdot\eta\beta(i,j)}{\sum_l \tau^{\alpha}(i,l,t)\cdot\eta(i,l)} & \text{if } o_i \text{ can be scheduled at } j \text{ and } l \\ 0 & \text{otherwise} \end{cases} \tag{5.23}$$

where j is a candidate time step, which is between o_i's mobility range

Algorithm 9 MMAS constructing individual timing constraint schedule

```
1: S_current = ∅;
2: conduct the ASAP and ALAP scheduling;
3: while exists unscheduled operation do
4:   for each unscheduled operation o_i do
5:     update the mobility range [s_i^S, s_i^L];
6:     update the operation probability r(i, j);
7:   end for
8:   for each resource type k do
9:     update the type distribution q(k);
10:  end for
11:  probabilistically select candidate operation o_i;
12:  for s_i^S ⩽ j ⩽ s_i^L do
13:    local heuristic η(i, j) = 1/q(k, j) where o_i is of type k;
14:  end for
15:  select time step j using the p(i, j) as in Equation (5.23).
16:  s_i^current = j;
17: end while
```

$[s_i^S, s_i^L]$. The item $\eta(i,j)$ is the local heuristic for scheduling operation o_i at control step j, and α and β are parameters to control the relative influence of the distributed global heuristic $\tau(i,j)$ and local heuristic $\eta(i,j)$. In this proposed algorithm, it is assumed that, if operation o_i is of type k, then local heuristic $\eta(i,j)$ is the inverse of $q(k,j)$, the type distribution defined in Equation 5.3; that is the distribution graph value of resource type k at control step j (calculated exactly that same as in FDS). Recalling that q_k is an indication of the number of type k functional units required at control step j. The ant intuitively favors a decision that possesses a higher volume of pheromones and a better local heuristic, i.e. a lower q_k. In other words, an ant is more likely to make a decision that is globally *good* and uses the fewest number of resources under the current partially scheduled result.

The second stage of the algorithm evaluates generated results and up-

dates the pheromone trial. The quality of results from ant m is judged by the total number of resources, i.e. $a_m = \sum_k r_k$. At the end of the iteration, the pheromone trail is updated according to the quality of individual schedules. Additionally, a certain amount of pheromones evaporate.

$$\tau(i,j,t) = \rho \cdot \tau(i,j,t-1) + \sum_{m=1}^{M} \Delta\tau_m(i,j,t) \tag{5.24}$$

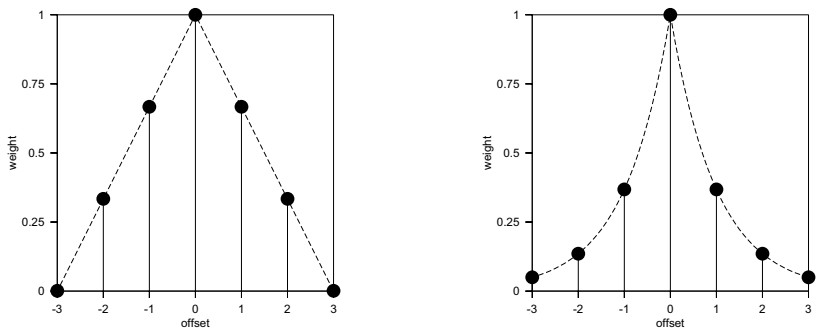
$$\Delta\tau_m(i,j) = \begin{cases} Q/a_m & \text{if } o_i \text{ is scheduled at } j \text{ by ant } m \\ 0 & \text{otherwise} \end{cases} \tag{5.25}$$

where ρ is the evaporation ratio and $0 < \rho < 1$, and Q is a fixed constant to control the delivery rate of the pheromone. Two important actions are performed in the pheromone trail updating process. Evaporation is necessary for the MMAS optimization to effectively exploit the search space to avoid sub-optimal solutions, while reinforcement ensures that the favorable operation orderings receive a higher amount of pheromones and have a better chance of being selected in the future iterations.

The above process is repeated multiple times until an ending condition is reached. The best result found by the algorithm is reported.

Updating neighboring pheromones

In many test cases, a better solution can often be achieved based on a good known schedule by simply adjusting a few operations' schedule within their mobility range. Based on this observation, the pheromone update policy is refined to exploit neighbor positions. More specifically, in the pheromone reinforcement step indicated by Equation 5.24, the amount



(a) (b)

Figure 5.3: Pheromone update windows

of pheromone trials on the control steps adjacent position j subject to a weighted function window. Two such windowing functions are shown in Figure 5.3 and subject to the operation's mobility $[s_i^S, s_i^L]$.

Operation selection

When an individual ant constructs a schedule for the given DFG, the next candidate operation is selected probabilistically. The simplest approach to select the next operation is to randomly pick one amongst all the unscheduled operations. Although it is simple and computationally effective, it does not appreciate the information accumulated from the pheromone trials, and it ignores the dynamic mobility range information. A possible refinement is to make the selected operation proportional to the pheromone and inversely proportional to the size of its mobility at that instance. More precisely, the probability of picking the next operation o_i is defined as follows.

$$p(i) = \frac{\frac{\sum_j \tau(i,j)}{(s_i^L - s_i^S + 1)}}{\sum_l \frac{\sum_k \tau(i,k)}{(s_l^L - s_l^S + 1)}} \tag{5.26}$$

where the numerator can be viewed as the average pheromone value over all possible positions in the current mobility for operation o_i. The denominator is a normalization factor to bring the result to be a valid probability value between 0 and 1. It is the addition of the average of pheromones for all the unscheduled operations o_l. Notice that as the mobility of the operations change dynamically depending on the partial schedule, the average pheromone trail is not a constant during the schedule construction process. In other words, we only consider a pheromone $\tau(i,j)$ when $s_i^S \leqslant j \leqslant s_i^L$.

Intuitively, this formulation favors an operation with stronger pheromones and fewer possible scheduling possibilities. In the extreme case, $t_i^L = t_i^S$, which means operation op_i is on the critical path, we have only one choice for op_i. If the pheromone for op_i at this position happens to be very strong, we will have a better chance to pick op_i at the next step compared to other operations. Our experiments show that applying this operation selection policy makes the algorithm faster in identifying high quality results. We could reduce the runtime of the algorithm by about 23% while achieving almost the same quality in the testing results.

5.5.2 Complexity analysis

The process of constructing an individual schedule by the ants, or the body of the inner loop in the proposed algorithm, is of the complexity $O(N^2)$, where N is the number of vertices in the DFG. Thus, the total complexity of the algorithm is determined by the number of ants M and the iteration number I. Theoretically, the production of M and I shall be pro-

portional to the production of N and the deadline L. In this case, we have a total complexity of $O(L \cdot N^3)$ which is the same as the normal version of the FDS. However, in practice, it is possible to fix M and I for a large range of applications, which means that in practical use, the algorithm can be expected to work with $O(N^2)$ complexity for most of the cases.

5.5.3 Experimental results

In order to evaluate the quality of the proposed TCS algorithm, the experimental results are compared to those from the FDS. For all test cases, operations are allocated on two types of computing resources, namely MUL and ALU, where MUL is capable of handling multiplication, while ALU is used for other operations such as addition and subtraction. Furthermore, we define that each operation running on MUL takes two clock cycles and every other operation on ALU takes one. This definitely is a simplified case from reality. However, it is a close enough approximation and does not change the generality of the results. Other operations to resource mappings can easily be implemented within the framework.

The implementation of FDS is based on [98] and has all the applicable refinements proposed in the paper, including multi-cycle operation support, resource preference control, and look-ahead using second order of displacement in force computation.

With the assigned resource/operation mapping, ASAP is first performed to find the lengths of critical paths L_c. Then the predefined deadline range is set to be $[L_c, 2L_c]$, i.e. from the critical path delay to

2 times of this delay. This results collected are from 263 test cases in total. For each delay, we run FDS first to obtain its scheduling result. Following this, the proposed algorithm is executed 5 times to obtain enough data to evaluate the quality of results. The average results, the best, and standard deviation of the FDS are reported. The execution time information for both algorithms is also discussed.

The MMAS TCS algorithm with refinements is implemented in C. The evaporation rate ρ is configured to be 0.98. The scaling parameters for global and local heuristics are set to be $\alpha = \beta = 1$ and the delivery rate $Q = 1$. These parameters are not changed over the tests. We also experimented with a different ant number M and the allowed iteration count I. For example, we set M to be proportional to the average branching factor of the DFG understudy and I to be proportional to the total operation number. However, it is found that a fixed value pair for M and I may work well across the wide range of test cases. In the final settings, we set M to be 10, and I to be 150 for all the TCS test cases.

Due to the large amount of data, it is hard to report test results for all 263 cases in detail. Table 5.4 compares the results for `idctcol` , one of the biggest samples. A side-by-side comparison between the FDS and the proposed method is presented here. The scheduling results are reported as a MUL/ALU number pair required by the obtained scheduling. For the latter one, we report both the average performance and the best performance in the 5 runs for each testing case, together with the savings percentage. The savings is measured by the reduction of computing resources. In or-

Deadline	FDS	Average	Save	Best	Save	σ
19	(6 8)	(5.0 6.0)	-21.43%	(5 6)	-21.43%	0.000
20	(5 7)	(4.4 6.0)	-13.33%	(4 6)	-16.67%	0.219
21	(4 7)	(4.2 5.8)	-9.09%	(4 6)	-9.09%	0.000
22	(4 7)	(4.2 5.4)	-12.73%	(4 5)	-18.18%	0.219
23	(4 7)	(4.0 5.4)	-14.55%	(4 5)	-18.18%	0.219
24	(4 7)	(3.6 5.2)	-20.00%	(3 5)	-27.27%	0.335
25	**(8 8)**	(3.8 5.0)	-45.00%	(3 5)	-50.00%	0.179
26	**(8 8)**	(3.4 5.0)	-47.50%	(3 5)	-50.00%	0.219
27	**(8 8)**	(3.0 5.0)	-50.00%	(3 5)	-50.00%	0.00
28	**(8 8)**	(3.0 4.6)	-52.50%	(3 4)	-56.25%	0.219
29	**(8 8)**	(3.0 4.4)	-53.75%	(3 4)	-56.25%	0.219
30	**(8 8)**	(3.0 4.6)	-52.50%	(3 4)	-56.25%	0.219
31	(4 6)	(3.0 4.6)	-24.00%	(3 4)	-30.00%	0.219
32	(4 5)	(3.0 4.0)	-22.22%	(3 4)	-22.22%	0.000
33	(4 5)	(2.8 4.0)	-24.44%	(2 4)	-33.33%	0.179
34	(4 5)	(3.0 4.0)	-22.22%	(3 4)	-22.22%	0.000
35	(4 5)	(3.0 4.0)	-22.22%	(3 4)	-22.22%	0.000
36	**(4 6)**	(3.0 3.8)	-32.00%	(3 3)	-40.00%	0.179
37	**(4 6)**	(2.6 3.8)	-36.00%	(3 3)	-40.00%	0.219
38	**(4 6)**	(2.8 3.4)	-38.00%	(3 3)	-40.00%	0.179

Table 5.4: Detailed results of the timing constrained scheduling of `idctcol`

der to keep things more general and simpler, the total count of resources is used as the quality metrics without considering their individual cost factors.

Besides the quality of results, one difference between the FDS and the proposed method is that the method is relatively more stable. When the deadline is relaxed, the FDS performance often gets worse, as indicated by the lines in bold. One possible reason is that as the deadline is extended, the mobility of each operation is also extended, which makes the force computation more likely to clash with similar values. Due to the lack of backtracking and good look-ahead capability, an early mistake would leads to inferior results. On the other hand, the proposed algorithm stably generates monotonically non-increasing results with fewer resource

requirements as the deadline decreases.

Name	Size	Deadline	Avg. Save	Best Save	Avg. σ
HAL	11/8	(6 - 12)	-7.1%	-7.1%	0.000
horner_bezier_surf	18/16	(11 - 22)	-9.9%	-13.2%	0.015
ARF	28/30	(11 - 22)	-12.4%	-16.9%	0.093
motion_vectors	32/29	(7 - 14)	-13.1%	-16.0%	0.072
EWF	34/47	(17 - 34)	-11.5%	-18.1%	0.081
FIR2	40/39	(12 - 24)	-16.8%	-22.8%	0.106
FIR1	44/43	(12 - 24)	-15.2%	-18.0%	0.047
h2v2_smooth_downsample	51/52	(17 - 34)	-19.3%	-21.3%	0.042
feedback_points	53/50	(11 - 22)	-5.9%	-9.2%	0.103
collapse_pyr	56/73	(8 - 16)	-18.3%	-18.9%	0.044
COSINE1	66/76	(10 - 20)	-21.5%	-25.9%	0.150
COSINE2	82/91	(10 - 20)	-5.6%	-12.0%	0.232
write_bmp_header	106/88	(8 - 16)	-0.9%	-1.0%	0.064
interpolate_aux	108/104	(10 - 20)	-0.2%	-2.0%	0.109
matmul	109/116	(11 - 22)	-3.7%	-5.1%	0.088
idctcol	114/164	(19 - 38)	-30.7%	-34.0%	0.151
jpeg_idct_ifast	122/162	(17 - 34)	-50.3%	-52.1%	0.147
jpeg_fdct_islow	134/169	(16 - 32)	-31.4%	-34.2%	0.171
smooth_color_z_triangle	197/196	(15 - 30)	-7.3%	-9.2%	0.136
invert_matrix_general	333/354	(15 - 30)	-11.2%	-13.2%	0.237
Total Avg.			-16.4%	-19.5%	0.104

Table 5.5: Result summary for the TCS algorithms

Table 5.5 summarizes the testing results for all of the benchmarks. We present the average and the best results for each testing benchmark, its tested deadline range, and the average standard deviations. The table is arranged in the increasing order of the complexity of the DFGs. The average result is that the quality of the algorithm is better than or equal to the FDS results in 258 out of 263 cases. Among them, for 192 test cases (or 73% of the cases) the MMAS based formulation outperforms the FDS method. There are only five cases where the approach has worse average quality results. They all happened on the *invert_matrix_general* benchmark. On average, as shown in Table 5.5, we are expecting a 16.4%

performance improvement over FDS. If only considering the best results among the 5 runs for each testing case, we achieve a 19.5% resource reduction averaged over all tested samples. The most outstanding results provided by the proposed method achieve a 75% resource reduction compared with FDS. These results are obtained on a few deadlines for the *jpeg_idct_ifast* benchmark.

From Table 5.5, it is easy to see that for all the examples, MMAS-based operation scheduling achieves better to much better results. Our approach seems to have much stronger capabilities in robustly finding better results for different test cases. Furthermore, it scales very well over different DFG sizes and complexities. Another aspect of scalability is the pre-defined deadline; based on the results presented in Table 5.4 and Table 5.5, the proposed algorithm demonstrates better scalability over this parameter.

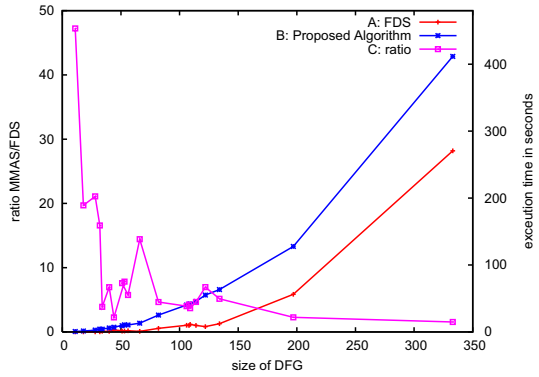


Figure 5.4: Run-time comparisons of the TCS algorithms

All of the experiment results are obtained on a Linux box with a 2GHz CPU. Figure 5.4 diagrams the execution time comparison between the presented algorithm and FDS. Curve A and B shows the run time for FDS

and the proposed method (respectively), where we take the average run-time for the MMAS solution. As discussed earlier, when we use fixed ant number m and iteration limit N in the experiments, there exists a large gap between the execution times for the smaller-sized cases. For example, for the HAL example, which only has 11 operations, the execution time of FDS is 0.014 seconds while the method takes 0.66 seconds. This translates into a ratio of 47. However, as the size of the problem gets larger, this ratio drops quickly. For the biggest cases *invert_matrix_genera*, FDS takes 270.6 seconds while the method spends about 411.7 seconds, which makes the ratio 1.5. To summarize, for smaller cases, the algorithm does have a relatively larger execution times but the absolute run time is still very short. For the HAL example, it only takes a fraction of a second. For larger cases, the proposed method runs at the same scale as FDS. This makes the algorithm practical in reality.

In Figure 5.4, there are some spikes in the ratio curve, which may be caused by two main reasons. First, the recorded execution time is based on system time and it is relatively more unreliable when the execution time is short. Secondly, but perhaps more important, the timing performance of both algorithms is determined by not only the number of the DFG vertices but also the predefined deadline L. This may introduce variances when the curves are drawn against the vertex count.

5.6 Summary

In this chapter, novel scheduling algorithms using the MMAS optimization for the TCS and RCS problems are presented. These algorithms use a unique hybrid approach by combining the ant system meta-heuristic with traditional scheduling heuristics. The proposed algorithms dynamically adjust a number of local and global heuristics in an iterative manner. Experiments are conducted using a comprehensive testing benchmark set from real-world applications in order to verify the effectiveness and efficiency of the proposed algorithms.

For the TCS problem, the proposed algorithm achieves equal or better results compared to the FDS for almost all the test cases, with a maximum 19.5% area reduction. For the RCS problem, the proposed algorithm outperforms a number of different list scheduling heuristics with better stability, and generates better results with up to 14.7% improvement (on average 6.2% better). Furthermore, by solving the test samples optimally using ILP formulation, it was shown that our algorithm consistently achieves a near optimal solution.

Chapter 6

Concurrent Resource Allocation and Scheduling

The solution space of the resource allocation and scheduling problem is shown in Figure 6.1. The top curve shows solutions of minimum silicon area subject to the timing constraints. The bottom curve shows solutions minimizing functional units' area subject to the timing constraints. Much of the existing work in resource allocation and scheduling uses the functional units' area or even the total number of hardware components as cost functions. Therefore, these works are not able to generate real area-efficient solutions considering functional units and register/multiplexor/-logic costs.

In addition, our MMAS scheduling work presented in the previous chapter has too many assumptions and limitations. For example, the MMAS scheduling uses one clock cycle as the minimum time unit, and

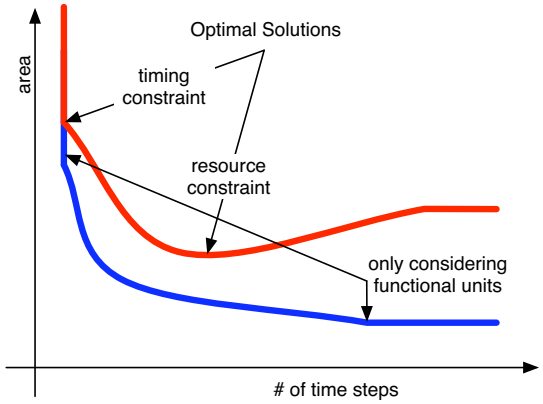


Figure 6.1: The design goal of resource allocation and scheduling

assumes homogeneous scheduling in the timing-constraint scheduling. In addition, the MMAS scheduling uses the DFG representing program behavior, which is a directed acyclic graph. Therefore, the MMAS scheduling algorithm cannot handle complicated timing constraints and loops.

This chapter presents our work on developing an MMAS-based concurrent resource allocation and scheduling (CRAAS) algorithm subject to timing/resource constraints for actual hardware designs. We begin with a simple example, a pipelined FIR filter. In Sections 6.2 and 6.3, we introduce the hardware resources used in scheduling and complicated factors in designing scheduling algorithms. In Section 6.4, we introduce the constraint graph, which is a hierarchical graph model capable of representing most timing constraints and support speculative execution. We describe a generalized model of the resource allocation and scheduling problem in Section 6.5. In Section 6.6, we present our MMAS CRAAS algorithm, and show experimental results in Section 6.7. Finally, we summarize in Section 6.8.

6.1 Motivating Example: a Pipelined FIR Filter

The functionality of a high throughput 64-tap FIR filter design is specified using the C programming language as shown in Figure 6.2. It is required to produces an output each clock cycle. The latency before the first output does not matter. This FIR filter will be synthesized using a 180 nm technology process. The target clock frequency is 300 MHz. The design goal is to generate a design as small as possible given that the throughput requirement is satisfied.

```
1 void fir_filter ( short *input,
                    short coef[64],
3                   short *output) {
    static short regs[64];
5   int temp = 0;

7 loop_shift:   for (int i = 63; i>=0; i--)
    regs[i] = (i == 0) ? *input : regs[i-1];
9
  loop_mac:  for (int i = 63; i>=0; i--)
11  temp += regs[i]*coef[i];

13  *output = temp>>16;
  }
```

Figure 6.2: A 64-tap FIR filter

A high-level synthesis tool is selected to accomplish this synthesis task. First, in order to achieve the specified high throughput, both `loop_shift` and `loop_mac` are fully unrolled and pipelined with an initial interval of one clock cycle. After control and data dependence analysis, a control/-

data flow graph (CDFG) is constructed, as briefly shown in Figure 6.3. Then the synthesis tool conducts resource allocation and scheduling upon the CDFG. Each operation is assigned to a compatible component selected from the technology library, and the start time is determined. In the technology library, there are a number of adders with different widths and different speed grades. Some of them are very fast, and some of them are slower but smaller. There are a number of different multipliers as well. It is possible that more than one operations share the same component if they are scheduled in different clock cycles. Once the schedule is determined, control and interconnects are synthesized and a hardware design is output using a structural register-transfer level (RTL) hardware description language (HDL), and can be further synthesized with downstream design tools.

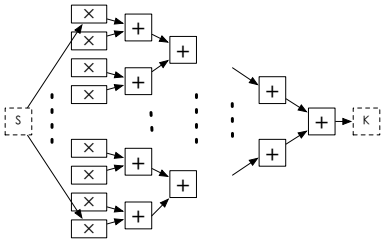


Figure 6.3: The control/data flow graph

	Fast		Normal		Small	
	D (ns)	Area	D (ns)	Area	D (ns)	Area
Adder (32)	0.51	3974.61	1.17	1565.81	2.62	1219.85
Multiplier (16x16)	2.21	9039.54	2.64	6659.56	3.96	5802.89
MUX (1,1,2)	0.08	40.45	0.20	31.48	0.39	23.23
Register (32)	-	685.72	-	685.72	-	685.72

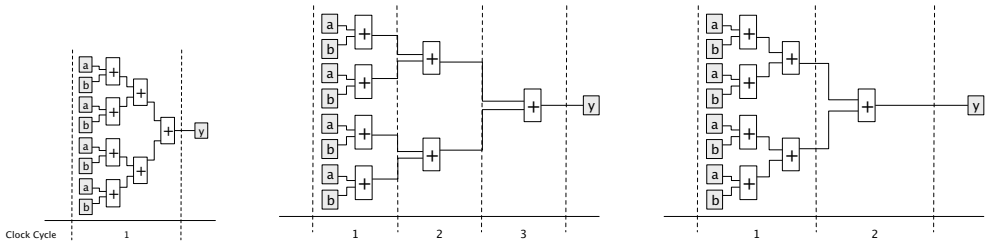
Table 6.1: A sample technology library

It is worthy to notice that even such a simple technology library of the

actual hardware design is much more complicated than any technology libraries ever published in technical papers. In this technology library, an operation can be implemented by multiple components in the library. One component can implement more than one kind of operations. Whether a component should be shared among operations depends on the ratio of its area and the area of required multiplexors. For example, given the areas of different 1-bit 2-to-1 multiplexors, it costs more to share a slower 32-bit adder because this requires 64 1-bit 2-to-1 multiplexors, and the area ranges from 1400 to 2600 area units(au, $1au = 54\mu m^2$). It may save area to share a normal adder depending on which kind of multiplexors are selected. It is always good to share a multiplier or a fast adder.

Figure 6.4 shows three different resource allocation and scheduling results of a part of the CDFG of the FIR filter, which is a balanced adder tree accumulating eight numbers. As specified above, the design goal is to minimize the total area given that the throughput constraint is satisfied. A trivial method to satisfy the throughput constraint is to allocate the fastest components for all operations, and build a schedule. This is shown in 6.4(a), where seven fast adders are used. A schedule minimizing the functional units' area, as shown in 6.4(b), uses seven small adders. In addition, this schedule requires seven registers. However, this is not the most area-efficient solution yet. If the register cost and the functional units' area are considered during the resource allocation and scheduling, a better schedule, as shown in Figure 6.4(c), uses six normal adders and a small adder. This solution requires three registers. The total area is about

6 percent smaller than that of the second schedule.



(a) The latency is 1 cycle and the area is 28508. 1 Registers are required.

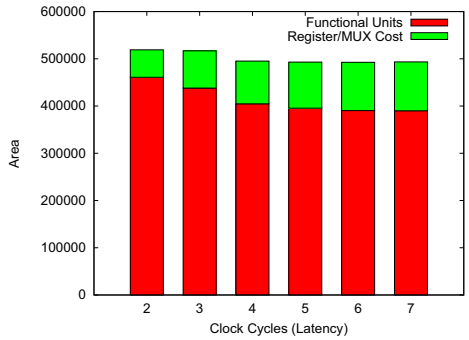
(b) The latency is 3 cycles and the area is 13334. 7 Registers are required.

(c) The latency is 2 cycles and the area is 12612. 3 Registers are required.

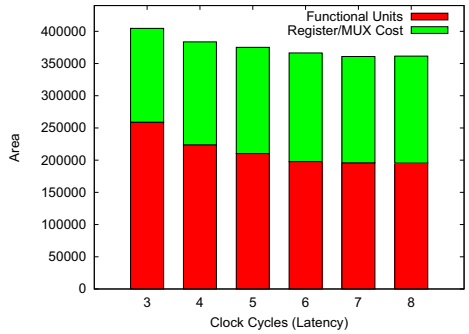
Figure 6.4: Three feasible schedules of a balanced adder tree

In order to investigate the relationship between areas and latencies globally, a number of implementations of the pipelined FIR filter are synthesized with extra latency constraints. For each given latency, the smallest design is reported. All of these designs fulfill the throughput constraints. Their areas are shown in Figure 6.5(a). The total area consists of two parts. One part is the area of functional units, i.e. adders and multipliers in the FIR filter, shown in the bottom of each bar. The other is the area of registers and multiplexors, as shown in the top of each bar. If only considering the area of functional units, the smallest design is the design with a latency of 7 clock cycles. The total area score is 493671 and the area score of functional units is 389938. If considering the total area, the smallest design is the one with a latency of 6 clock cycles. This design has a slightly larger area of functional units, but the total score is 492812, which is smaller than that of the previous one.

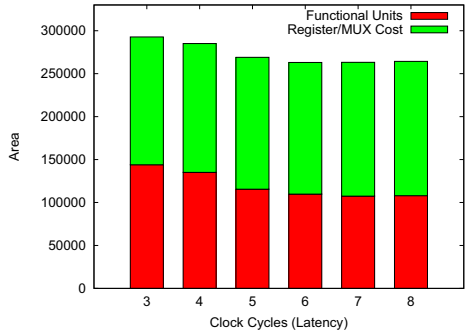
This is true when synthesizing the pipelined FIR filters with slower



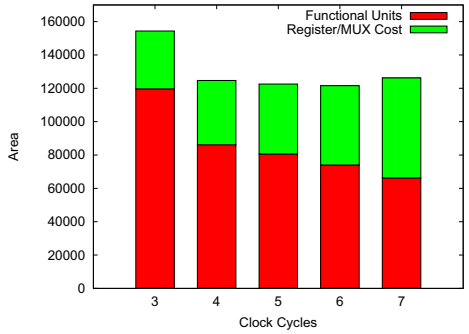
(a) Initial interval is 1 clock cycle



(b) Initial interval is 2 clock cycles



(c) Initial interval is 4 clock cycles



(d) Simplified FIR filter with all coefficients are 1

Figure 6.5: The area and latency trade-offs of synthesized FIR filter

throughputs. Figure 6.5(b) shows the area/latency tradeoff of a design with an initial interval of two clock cycles. Figure 6.5(c) shows the area/latency tradeoff of a design with an initial interval of four clock cycles. Figure 6.5(d) is a simplified FIR filter where all coefficients are 1. In other words, this adder tree adds 64 inputs together.

To summarize, these four designs shows that, when the design goal is to minimize the total silicon area, it is necessary to consider the area of registers and multiplexors. The real area-efficient schedules are different from the solutions minimizing functional units' area.

Some lessons and observations learned in these simple examples are:

1. There are many choices of resource types for an operation type. It is necessary for the scheduler to exploit a large solution space.

2. It is necessary to precisely estimate timing. Chaining more than one operations shows great benefits of saving register and multiplexors' cost.

3. It is hard to precisely estimate the total silicon area in resource allocation and scheduling. Different solutions of resource sharing and register sharing generate different results given the same schedule. In addition, placement and routing greatly affect the area of synthesized designs.

6.2 Hardware Resources

The data-path in the synthesized hardware designs consist of three types of elements: functional units, storage components, and interconnect logic. Functional units implement arithmetic operations or compound operations. Storage components store temporary intermediate results or data specified in the program. Interconnect logic steers appropriate signals between functional units and storage components.

This section describes the timing and other important attributes required in resource allocation and scheduling.

6.2.1 Functional units

Functional units implement actual data operations, such as accumulation, multiplication, comparison, and so forth. Based on whether these

hardware components contain states and storage inside, functional units could be categorized as combinational components and sequential components, and sequential components can be further categorized as non-pipelined sequential components and pipelined components.

Combinational components

A combinational component is a digital circuit performing a specific data operation, fully specified logically by a set of Boolean functions [83]. The value of its output is determined directly from and only from the present input combination. There are no memory elements in combinational components.

The timing attributes $T_i(D_i)$ of a combinational component q_i contain only one element, which is the absolute time measuring the output delay D_i from the input data available to the available output data.

Given a specific length of the clock cycle, some combinational components in a technology library are not fast enough to fit in a single clock cycle. Operations scheduled on these components are across two or more clock cycles. These components remain active during these clock cycles. At the same time, because combinational components do not have any memory elements, the input data should be kept stable before the output is registered by a storage component. This implicitly requires registers for the input data and multiplexors to maintain the lifetime of the input data.

Non-pipelined sequential components

A sequential component is a digital circuit that employs memory elements in addition to combinational logic gates. The value of its output is determined from not only the present input combination but also the content in the memory elements, or the current state.

If a sequential component cannot take another input data when it is processing the current input, this component is a *non-pipelined* sequential component; otherwise, it is a *pipelined* component, which is discussed below. It remains active until the output is available and registered by another component.

The timing attributes $T_i(L_i, D_i, K_i)$ of a non-pipelined sequential component is characterized by its latency L_i, output delay D_i, and the minimum clock period K_i. The *latency* specifies how many clock cycles it takes to generate the output after the input is available. The *output delay* is the length of the critical path to the output. A critical path is defined as the longest logic path containing no other memory elements. The *minimum clock period* is the length of the critical path in this sequential component. It can be from the input to the output, from the input to a memory element, from a memory element to a memory element, or from a memory element to the output. The length of the target clock period is normally longer than the minimum clock period of this sequential component. Otherwise, the latency or the number of clock cycles should be re-calculated, and input registers and multiplexors may be required.

Pipelined components

Some sequential components can start a new computation prior to the completion of the current computation. These components are called *pipelined* components. A pipelined component is normally designed by dividing a combinational component into a number of stages and inserting memory elements between stages.

The timing attributes $T_i(I_i, L_i, D_i, K_i)$ of a pipelined component is characterized by its initial interval I_i, latency L_i, output delay D_i, and minimum clock period K_i [38]. The latency, the output delay, and the minimum clock period are defined similarly as those of a non-pipelined sequential component. The *initial interval* is the number of clock cycles required to start a new computation task after starting the prior one, or in other words, is the number of clock cycles on which the result becomes available after the prior result is available.

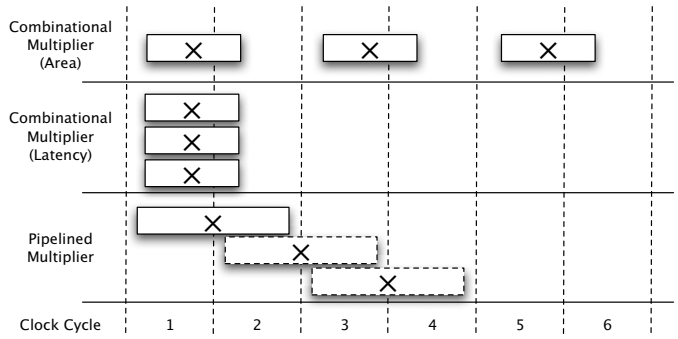


Figure 6.6: Multiplications scheduled on pipelined multipliers

For example, a design contains three multiplications. There are two options available. One is to use combinational multipliers, denoted as $m_1(6706.72a.u., 9.28ns)$, where $1a.u. = 54\mu m^2$. The other is to use pipelined

multipliers with 2-cycle latency and 1-cycle initial interval, denoted as $m_2(6220.77a.u., (1, 2, 5.73ns, 6.84ns))$. The target clock frequency is 125MHz, i.e. the length of clock period is 8ns. Figure 6.6 shows the scheduling results. The first design uses only one combinational multiplier m_1. The second uses three m_1. The third uses one pipelined multiplier m_2. It is obvious that designs using pipelined components benefit throughput and area.

6.2.2 Storage components

Storage components are found in digital designs to store inputs, outputs, and intermediate results. Two kinds of storage components are used in the resource allocation and scheduling process. They are registers and on-chip memory blocks. Registers are suitable for scalable variables, small-size data arrays and implicit intermediate results. Memory blocks are used for large size data arrays.

Registers

If an operation is dependent on another operation, and those two operations are not chained in the same clock cycle, i.e. if the data dependence crosses one or more clock cycle boundaries after scheduling, the intermediate results carried by the data dependence need to be stored in a register.

The timing attributes of a register are determined by the setup time and the ready time. The *setup* time is the amount of time required for the data to arrive at the register prior to the rising edge of the clock signal.

The *ready* time is the amount of time required for the data to become stable at the output of a register after the rising edge of the clock signal.

Two or more intermediate results can share one register if their lifetimes are not overlapping. If two or more intermediate results share the same register, or the lifetime of a variable mapped to a register is longer than one clock cycle, a multiplexor is required. It is necessary to consider area and delay of such a multiplexor.

Normally, register allocation and sharing is not the task of the resource allocation and scheduling problem. However, it is necessary to estimate the number of registers as early as possible. After scheduling, the lifetime of an intermediate result is fixed by the scheduling results. There is not such a large space left for optimizations to reduce the number of registers. Design tools conduct lifetime analysis and allocate and share registers to significantly reduce the area of generated hardware designs.

Memory blocks

A sequential program may have large data arrays as input, output, or just intermediate results. These data arrays should be stored in embedded on-chip memory blocks. A large amount of scalar variables can also be assigned to these memory blocks to save the area of register files. An optimized storage assignment greatly affects the overall performance of synthesized hardware designs.

The timing attributes of memory accesses are characterized by the *setup* time and the *ready* time as well. It typically takes one clock cycle to

perform a memory access. The setup time is required for the address to become stable prior to the rising edge of the clock signal, and so does the data input if this is a memory write operation. The data is available after the ready time, following the clock rising edge.

If two or more data arrays/scalar variables are assigned to the same memory blocks, then multiplexors are required on the address and data-in ports to access the right data on the right clock cycle. It is necessary to count delays on multiplexors to get a close timing estimation.

More memory and storage related transformations and optimizations are discussed in Chapter 4, Data Partitioning and Storage Assignment.

6.2.3 Interconnect logic

As discussed above, functional units can be shared by more than one data operation. Data-in and address ports for a storage component can be used in more than one places. Interconnect logic steers the data to the right place, which are mainly implemented using multiplexors.

Multiplexors can be modeled as simple combinational components. They have areas and their timing attributes $T_i(D_i)$ are characterized by the output delay D_i. Sometimes when the target clock frequency is too high, the delay is longer than the available clock period. Then input registers are required.

Sharing a functional unit or a register is determined by the ratio of multiplexors' area and how mach area of functional units or registers could be saved by sharing them. For some technology and target archi-

tectures, it is not worth to share registers or simple functional units, such as adder, which is especially obvious for designs mapping to FPGAs.

It is necessary for the resource allocation and scheduling algorithm to evaluate different strategies to generate high-quality designs.

6.3 Complicated Scheduling Factors

Scheduling and resource allocation algorithms addressing actual design problems are generally very complicated. Why these problems are so complicated is discussed in this section.

6.3.1 Chained operations

When two or more data-dependent operations are scheduled in the same clock period, these operations are *chained*. It is very effective to reduce latency using operation chaining. However, this may come at the cost of the increased number of hardware resources and the area of faster but much larger functional units fitting into a single clock period.

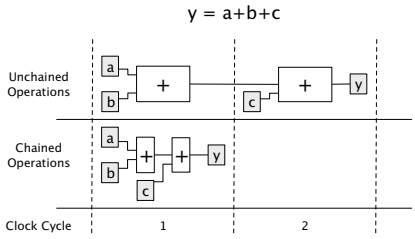


Figure 6.7: Chained operations

For example, as shown in Figure 6.7, those two add operations could

be scheduled in two clock periods or be chained in one clock cycles. If these operations are chained, two adders are required since these two add operations are active concurrently. Compared with the first schedule, the chained operations may be required to be implemented on faster adders to fit with one clock cycles.

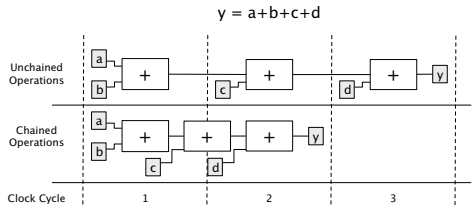


Figure 6.8: Three add operations chained in two clock cycles

It is assumed that chained operations must fit in one clock cycle. In Figure 6.8, the delays of adders are $6ns$, and the length of the clock period is $10ns$. Those three add operations could be scheduled in three clock cycles, or be chained in two clock cycles. However, this greatly increases the complexity of the presented scheduling algorithm and the complexity of the generated designs.

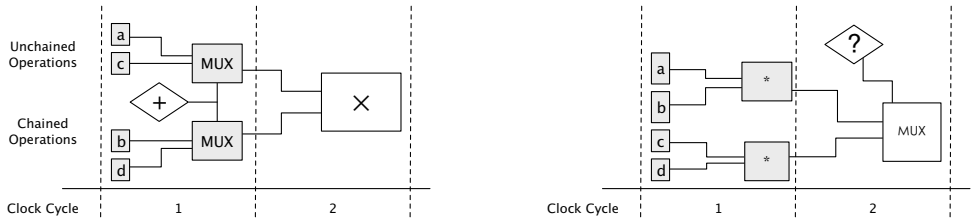
6.3.2 Multiple possible bindings

A technology library typically defines multiple implementations for a particular operation. For example, an add operation can be implemented with a ripple adder or a carry-look-ahead adder. Some data operations can be implemented using wider bit-width functional units due to these operations' characteristics. For example, both an 18-bit multiplication and a 20-bit multiplication can be implemented with a 20-bit multiplier.

In order to effectively utilize the allocated hardware resources, the scheduling and resource allocation algorithm must exploit multiple binding options of a data operation and trade-off between area and latency.

6.3.3 Mutually exclusive sharing

Mutually exclusive sharing occurs when two operations in different branches of a program can be scheduled in the same clock cycle and assigned to the same hardware resource. This happens in `if-then-else` and `select-case` statements in high-level programming languages. For example, as shown in Figure 6.9, given such a piece of C programs, those two multiplications are in different branches. In the first schedule, both two multiplications are scheduled after the predicate condition is available. The two multiplications can mutual-exclusively share one multiplier. In the second schedule, those two multiplications are scheduled before the predicate condition is available, i.e. they are speculatively executed. They cannot share the same multiplier.



(a) Shared (b) Non-shared

Figure 6.9: Mutually exclusive sharing

Mutually exclusive sharing can better utilize available hardware re-

sources and decrease the latency of the generated design. It is important to note that, when some operations are speculatively executed, they cannot share the same hardware resource.

6.3.4 Pipelining loops

In order to improve the throughput of the computing system, it is normally required to pipeline the synthesized hardware designs, especially streaming data processing designs.

Pipelining is usually applied to portions of a program that are executed multiple times. The iteration bodies of loops are good candidates to be pipelined. For example, if the iteration body is scheduled with 10 clock cycles, then the design starts to process new data every 10 clock cycles. If this design could be pipelined with an initial interval equal to 10, then the design can start processing new data every clock cycle, which may improve the overall performance 10 times.

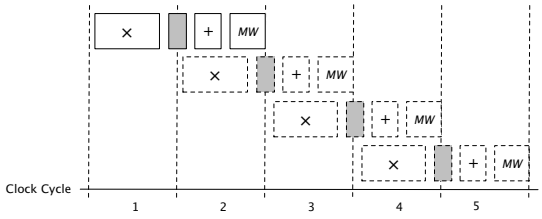
The timing attributes of a pipelined hardware design are characterized by its *latency* and *initial interval*. The latency refers to the running time from processing the first input data to writing out the last output data. Some designs run indefinitely. Hence, the latency may also refer to the running time from processing one input to writing out processed results of this input. The initial interval specifies the throughput of the synthesized hardware designs, i.e. the number of clock cycles to start processing a new input after taking the prior one, or the number of clock cycles in which the next result becomes available after the prior result is available. This is

quite similar to the latency and initial interval of a pipelined component.

For example, the iteration body of a loop is synthesized to a pipelined design with a 5-cycle latency and a 2-cycle initial interval. After every two clock cycles, an input is read, the hardware processes the data, and the output is available in five clock cycles. Because the initial interval is two clock cycles, functional units and registers could be shared. If two multiplications are scheduled in clock cycle 1 and 4, then they can share the same multiplier, and this multiplier is always fully utilized.

```
  for (i = 0; i<N; ++i)
2 {
      b = a[i];
4     a[i+1] = b*s;
  }
```

(a) The iteration body of a loop.



(b) A simple pipelined schedule

Figure 6.10: A pipelined design

The throughput is limited by the program behavior. Not all loops can be pipelined with a very high throughput. When there is a *read-after-write* dependency across different iterations, it may be impossible to achieve the desired throughput. For example, Figure 6.10 shows a piece of code in an iteration body, and a pipelined schedule. If the multiplication cannot be finished in one clock cycle, this design cannot be pipelined with a 1-cycle iteration interval.

There are different approaches to generate pipelined designs. One way is to schedule designs first and then try to pipeline the scheduled designs. A better one is an integrated approach. The throughput timing con-

straints are specified on the graph model, and the resource allocation and scheduling algorithm generates a design optimized to the desired goals, subject to the specified timing constraints. The latter one is more effective and efficient. However, this requires the graph model used in the scheduling algorithm represent the throughput constraints, which are normally feedback paths from later operations to an earlier operation. While the DFG does not have such abilities, scheduling for pipelined designs is covered only in Section 6.6.

6.4 Constraint graph

This section presents the constraint graph, which is a graph-based model describing hardware behavior in the resource allocation and scheduling algorithm. In order to generate schedules for actual hardware designs, the traditional CDFG should be enhanced.

The constraint graph is a polar and hierarchical directed graph, denoted by $G(\mathbf{V},\mathbf{E})$. The vertices $\mathbf{V} = \{v_0,\ldots,v_N\}$ represents operations to be scheduled. The directed edges $\mathbf{E}$ connect vertices and represent timing constraints among these vertices.

Vertices $\mathbf{V}$ in the constraint graph are classified into two categories: data operations and compound operations. Data operations represent arithmetic operations, data I/O operations, memory access operations, logic operations, and so forth. Each operation has one or more compatible components in the technology library. The resource allocation and

scheduling algorithm associates a proper implementation with this operation and determines the start time.

A compound operation is a child constraint graph consisting of a set of operations. These operations can be either data operations or other compound operations. Each compound operation represents a loop, a branch, or a function call in high-level programming languages. A constraint graph has one or more compound operations. Delays of child compound operations are treated as zero. The contained constraint graph is scheduled separately. Although there are optimizations across different constraint graphs, it is not discussed in the methodology presented here.

In order to clarify this model, two *virtual* vertices, v_S and v_K, are added to the constraint graph. These two vertices are associated with null operations. Hence, the delays of these two virtual vertices are *zero*. It is further assumed that, for any vertex $v_i \in \mathbf{V}$, $v_S \preccurlyeq v_i$ and $v_i \preccurlyeq v_K$ are defined. v_S will begin before the start of any other vertex $v_i \in \mathbf{V}$ and v_K will finish after the completion of any other vertex v_i. As a polar graph, v_S is the only source vertex in the constraint graph, and v_K is the only sink vertex.

Timing constraints

A directed and weighted edge $e(v_a, v_b, T)$, describing the timing constraint T between vertices v_a and v_b, is denoted as

$$t_a + t \leq t_b. \tag{6.1}$$

There are three kinds of timing constraints. The first one is only on the control steps of a pair of operations, where t_a is the beginning *time step* of operation o_a, t_b is the beginning *time step* of operation o_b, and t is a fixed value of integers, denoted by the number of control steps c, and c is an arbitrary integer. The other is a chained constraints from the finish time of operation o_a to the start time of operation o_b, where t is a fixed value of integers, denoted by the number of control steps c and an offset o, and $0 \leq o < C$, where C is the length of the target clock period. The beginning and completion of operations o_a and o_b are denoted as s_a, f_a, s_b and f_b, respectively. More specifically, the following constraints can be represented on edge $e(v_a, v_b, T)$.

- If operation o_b starts at the time equal to or greater than t after the completion of operation o_a, this timing constraint is denoted as

$$f_a + (c, o) \leq s_b \tag{6.2}$$

- If operation o_b starts at the time equal to or greater than t after the beginning of operation o_a, this timing constraint is denoted as

$$s_a.c + c \leq s_b.c \tag{6.3}$$

  This timing constraint specifies that o_a should be scheduled at least a certain number of control steps later. However, c is an arbitrary integer. When c is negative, this constraint referred to o_b can be scheduled at most c cycles earlier than o_a.

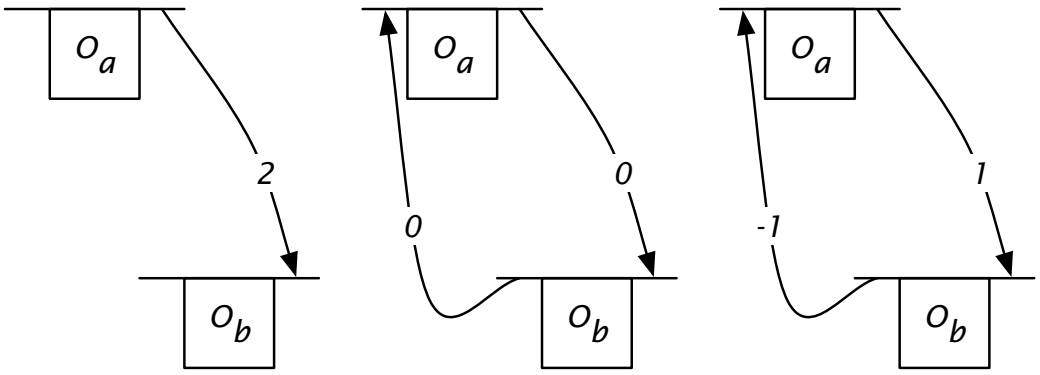
- If operation o_b finishes at the time equal to or greater than c control steps after the finish of operation o_a, this timing constraint is denoted as

$$f_a.c + c \leq f_b.c \tag{6.4}$$

All known timing constraints could be specified as one of the above situations or a combination of several constraints.

Constraint graph examples

This section presents several typical timing constraints represented using constraint graphs.



(a) Operation o_a should start two clock cycles earlier than operations o_b

(b) Two operations should start in the same cycle

(c) Two operations should be scheduled in two consecutive clock cycles

Figure 6.11: Constraint graph examples

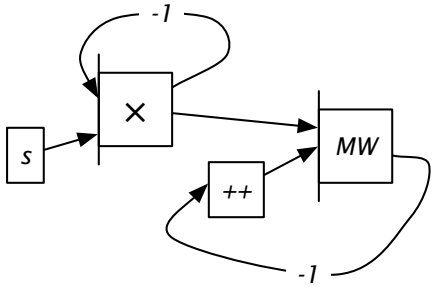
Figure 6.11 shows three constraint graph examples. Figure 6.11(a) represents operation o_b that should be scheduled at least two clock cycles later than operation o_a. Figure 6.11(b) represents two operations that should start at the same clock cycle. It is also possible to allow the con-

straint graph to represent that those two operations should start in the same clock cycle. Figure 6.11(c) shows that two operations have the specific order. Operation o_a should start exactly one clock cycle earlier than operation o_b. This gives the constraint graph the ability to represent specific schedules required by some interfaces and protocols.

The constraint graphs representing pipelined designs and speculative execution will be further discussed below.

```
1 for (int i = 0; i < n; ++i)
      m[i+1] = m[i] * s;
```

(a) A simple C program, where the `for` loop should be pipelined with an initial interval of four clock cycles.



(b) The backward edges show the throughput constraint.

Figure 6.12: A constraint graph showing a pipelined loop

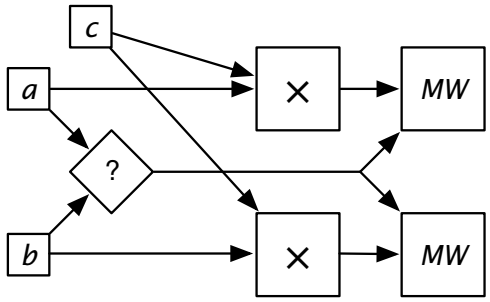
Figure 6.12(a) shows a simple piece of C code. This is a `for` loop. There is loop-carried data dependence from the memory read and memory write operations. This `for` loop should be pipelined with an initial interval of one clock cycle. Figure 6.12(b) shows the corresponding constraint graph, where the edge e from vertex v_b to vertex v_a has the timing constraint

$$f_c + (-1) \geq s_a.$$

This clearly shows that the memory write operation should be completed every clock cycle, as does the multiplication.

```
  if (a > b)
2      m[i] = a*c;
  else
4      m[i] = b*c;
```

(a) An `if`-branch. The two multiplications can be speculatively executed. The two memory accesses cannot be speculative executed.

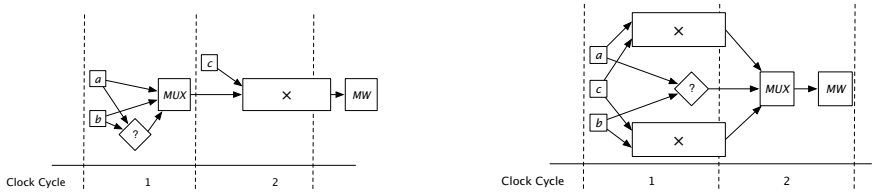


(b) The constraint graph showing the `if`-branch.

Figure 6.13: A constraint graph showing a branch structure

Figure 6.13(a) shows another piece of C code. This is an `if`-branch containing a multiplication and a memory access in each branch. Figure 6.13(b) shows the corresponding constraint graph. The edges here show the precedence dependencies of these operations. This is quite different from the normal basic-block based graph representations where each branch is represented as a single control/data flow graph. The constraint graph is capable of representing this in a similar way since the constraint graph is hierarchical. When two branches are not balanced, i.e. the delays and hardware resource requirements are quite different, these two branches can be represented in two constraint graphs and scheduled separately.

The advantage is that this constraint graph is flexible enough to support both speculative execution and non-speculative execution. In the above example, those two multiplications in the different branches can be speculatively executed. However, which of those two memory accesses



(a) A schedule showing non-speculative execution.

(b) A schedule showing speculative execution.

Figure 6.14: Two feasible schedule of the above branch structure

should be executed must wait until the predicate condition is available. A feasible schedule is shown in Figure 6.14(a). The comparison is scheduled before the two multiplications start. The two multiplications are hence non-speculatively executed, and the allocated multipliers are mutually shareable. Another schedule is shown in Figure 6.14(b). The comparison is completed after those two multiplications. Therefore, two multipliers are required but this schedule is slightly faster. In both schedules, the comparison is scheduled prior to those two memory accesses because of the precedence specified by the timing constraints.

Summary of constraint graphs

To summarize, the constraint graph is the underlying representation of hardware behavior in the resource allocation and scheduling stage. It can be easily derived from CDFG or PDG.

- In a constraint graph, compound operations and associated constraint graphs presents a hierarchy. This hierarchy describes loops, branches, and function calls.

- The execution delay is determined by the resource allocation and scheduling results. The delay of a compound operation is the latency of the associated constraint graph. The delay of a data operation is determined by the assigned component.

- Detailed timing constraints associated with edges in the constraint graph are able to present different design goals, including the maximum latency, the throughput of a pipelined design, interface protocols, and so forth.

In the following sections, detailed descriptions of the scheduling algorithm and experimental results will be presented. It is assumed that during resource allocation and scheduling, the constraint graph will not be transformed and optimized. There are some known optimizations of the constraint graphs, such as balancing adder trees to reduce the execution latency. However, these transformations and optimizations are out of the range of research works presented here.

6.5 A General Model of the Resource Allocation and Scheduling Problem

This section presents the general model of the resource allocation and scheduling problem. The inputs are as follows:

1. A constraint graph, denoted by $G(\mathbf{V},\mathbf{E})$, describes hardware behavior. The vertices $\mathbf{V} = \{v_0,\ldots,v_N\}$ represent operations $\mathbf{O} = \{o_0,\ldots,v_N\}$ to

be scheduled. The directed edges **E** connect vertices and represent timing between these vertices.

2. A specific technology library, which is derived from the target architecture, is a set of hardware resource types, denoted by $\mathbf{Q} = \{q_0, \ldots, q_M\}$. Each component $q_i(A_i, T_i, M_i, \mathbf{O}_{q_i})$ has its area A_i and timing information T_i, and a set of operations $\mathbf{O}_{q_i}$ supported by this component, where $\mathbf{O}_{q_i} \subset \mathbf{O}$ and $\bigcup_i \mathbf{O}_{q_i} = \mathbf{O}$. The target architecture and designers can specify resource constraints, which is the maximum available number M_i for each resource type q_i.

3. The desired length of clock period C, in the unit of seconds or nanoseconds, which specifies the target clock frequency of generated hardware designs.

The problem of resource allocation and scheduling is allocating a set of hardware components, i.e. determining the allocated number a_i for each resource type $q_i \in \mathbf{Q}$, seeking an assignment $\{\mathbf{O} \rightarrow \mathbf{Q}\}$, and determining the start time of each operation o subject to the timing constraints specified in the constraint graph and the resource constraints of the given technology library.

The start time of an operation o_i, denoted as $s_i(c_s, d_s)$, states that this operation should start in the cth clock cycle with an offset of time d from the beginning of this clock cycle. If the start time s_i is determined, and this operation is assigned to a resource $q_j(A_j, D_j, M_j, O_{q_j})$, the finish time of this operation, denoted as $f_i(c_f, d_s)$, is determined by increasing s_i by T_j.

The objective of this problem is to minimize the total area of the synthesized hardware design subject to given timing constraints and resource constraints, i.e. to minimize $\sum_i a_i A_i$. This is called timing constraint scheduling (TCS). For pipelined designs, if the throughput constraints are satisfied, normally, the latency or the number of control steps is not so important compared to the area, and it is reasonable to minimize area to reduce the product cost.

In some designs where the latency is a more important design goal, the objective is to minimize the total area of the synthesized hardware design given that the total number of control steps (or clock cycles) is equal to or lesser than that of the shortest schedule that is achievable with the specified resource constraints. This is called resource constraint scheduling (RCS), which is a dual problem of the TCS problem. The difference from the TCS is that the first priority of RCS is to generate a schedule as short as possible.

Depending on different priorities of hardware designs, there are other objectives in the resource allocation and scheduling problem, and this could be further formulated as a multiple objective optimization problem. However, our research work is focused on the fundamental RCS/TCS problems.

6.6 Concurrent Scheduling and Resource Allocation

This section presents an MMAS-based algorithm to solve the generalized resource allocation and scheduling problem. As described before, this problem is to allocate proper hardware resources from the given technology library and determine the start time of each operation subject to specified timing constraints and resource constraints. The objective is to minimize the total hardware area, including functional units, interconnects, and registers.

In order to clearly describe our methodology, more assumptions are made here besides those assumptions discussed in Section 6.3. As discussed earlier, the constraint graph is hierarchical. Each compound operation is associated with a constraint graph. Transformations and optimizations, especially those on resource allocation, could be conducted across those constraint graphs. These optimizations are out of the range of research work presented here.

We assume that all timing constraints and resource constraints are feasible, i.e. there is an allocation and scheduling solution available for the given constraint graph to satisfy those constraints. We further assume that during resource allocation and scheduling, the constraint graph is not transformed.

Although these assumptions may affect the quality of results, the presented algorithm is practical in actual hardware designs.

The proposed algorithm conducts resource allocation and scheduling in two stages: the first stage constructs an initial schedule satisfying the timing and resource constraints, and, based on the initial results, the second stage searches for a better schedule using the ant colony optimization.

This remainder of this section is organized as follows: Section 6.6.1 describes algorithms generating an initial schedule satisfying both timing and resource constraints, and Section 6.6.2 presents the MMAS CRAAS to utilize the local heuristics and global update schemes.

6.6.1 Generating initial schedules

The algorithm to generate an initial schedule iteratively performs two tasks. The first is conducting as soon as possible (ASAP) and as late as possible (ALAP) scheduling using unlimited fastest compatible hardware resources. The second is resolving hardware resource conflicts by incrementally scheduling based on the ASAP/ALAP scheduling results.

The initial schedule derived by this algorithm is not guaranteed to be the shortest schedule satisfying both resource and timing constraints. The goal is to lay down the groundwork for further optimizations.

Satisfying timing constraints The constraint graph specifies detailed timing constraints among operations. The ASAP scheduling determines the minimum value of the start times subject to these timing constraints, and the ALAP scheduling determines the maximum value of the start time when the start time of the sink vertex is fixed. During ASAP and

ALAP scheduling, the resource constraints are ignored. The objective of conducting ASAP and ALAP scheduling is to determine the *mobility* of an operation, which is the difference between the ASAP and ALAP scheduling results.

During ASAP and ALAP scheduling, the fastest compatible component of each data operation is assigned. Under the assumptions discussed in previous sections, it is easy to prove that the fastest compatible component guarantees the ASAP schedules are the earliest possible start times.

Algorithm 10 ASAP scheduling

1: **for all** $v_j \in \mathbf{V}$ **do**
2: allocate the fastest compatible component to v_j
3: $s_j^0 = 0$
4: **end for**
5: **repeat**
6: **for** each vertex $v_j \in \mathbf{V}$ **do**
7: incrementally calculate $s_j^{\gamma+1}$
8: **end for**
9: **until** the schedule of all vertices are not changed

The ASAP scheduling algorithm consists of two steps, as shown in Algorithm 10. Initially, each data operation is assigned to the fastest compatible component in the given technology library, and each operation is scheduled to start at time zero. Then increment scheduling is iteratively applied on each vertex. A constraint graph is directed and may be cyclic, which is quite different from the directed and acyclic CDFG. Edges carrying backward timing constraints affect schedules of those succeeding vertices. Therefore, the algorithm should iteratively calculate the ASAP schedules.

Algorithm 11 Adjusting ASAP schedule

1: **for all** $e(v_i, v_j, T_{i,j}) \in \mathbf{E}$ **do**
2: calculate $s_{i,j}^{\gamma+1}$ satisfying $T_{i,j}$
3: **end for**
4: $s_j^{\gamma+1} = \max(s_j^{\gamma}, s_{i_1,j}^{\gamma+1}, \ldots, s_{i_N,j}^{\gamma+1})$

For each vertex, the earliest start time is determined by inspecting all timing constraints on edges from precedent vertices. Specifically, given a directed edge $e(v_i, v_j, T_{i,j}) \in \mathbf{E}$, operations o_i and o_j are associated with vertices v_i and v_j, respectively, if the start time of operation o_j is known as s_j, and the delay of o_j is d_j, the finish time of operation o_j is denoted as f_j, where

$$s_j + d_j = f_j.$$

If the ASAP schedule of vertex v_j in the γ-th iteration is s_j^{γ}, then the new schedule $s_j^{\gamma+1}$ can be calculated as the following Algorithm 11. The last statement shows that

$$s_j^{\gamma+1} \geq s_j^{\gamma}.$$

This guarantees that the ASAP scheduling algorithm converges if all timing constraints are feasible.

The ALAP scheduling algorithm is shown in Algorithm 12. Assuming the virtual sink operation should be finished at time L, this scheduling algorithm calculates the latest start time s_i for each operation o_i. It is obvious that $s_i \leq L$. After resolving resource conflicts, s_i should be adjusted against the actual shortest latency.

The ASAP and ALAP schedules of each operation o_i, s_i^{ASAP} and s_i^{ALAP},

Algorithm 12 ALAP scheduling

1: assume the virtual sink operation complete at time L
2: **for all** $v_j \in \mathbf{V}$ **do**
3: allocate the fastest compatible component to v_j
4: $s_j^0 = L$
5: **end for**
6: **repeat**
7: **for** each vertex $v_j \in \mathbf{V}$ **do**
8: **for** each directed edge $e(v_j, v_k, T_{j,k}) \in \mathbf{E}$ **do**
9: incrementally calculate the schedule $s_{j,k}^{\gamma+1}$ satisfying timing constraint $T_{j,k}$
10: **end for**
11: $s_j^{\gamma+1} = \max(s_j^{\gamma}, s_{j,k_1}^{\gamma+1}, \ldots, s_{j,k_N}^{\gamma+1})$
12: **end for**
13: **until** the schedule of all vertices are not changed

respectively, define the *mobility* of this operation.

It is easy to prove that the complexity of this scheduling algorithm is polynomial and converges very quickly given that vertices and edges are sorted.

Resolving resource conflicts Resource constraints specify the quantities of available hardware resources, such as the number of memory ports, and the number of available multipliers. If the number of hardware resources is not enough for all data operations assigned on them, then it is possible that there are resource conflicts. For example, assume a number of memory operations access the same *dual*-port block RAM; if more than two accesses are scheduled in the same clock cycle, and they are not mutually shareable, then there are resource conflicts on these two memory ports.

During the ASAP and ALAP scheduling, the fastest compatible component is assigned and all resource constraints are ignored. There may be resource conflicts in the scheduling results. Therefore, ASAP and ALAP scheduling results are not feasible, and hardware resource conflicts must be resolved.

Algorithm 13 Resolving resource conflicts

```
1: repeat
2:   for each component q violated resource constraints do
3:     clear schedules s_i^γ of all operations assigned on q
4:     for each operation o_i assigned on q do
5:       s_i^{γ+1} = the earliest time from s_i^γ without violating resource constraints
6:     end for
7:   end for
8:   for each vertex v_j ∈ V do
9:     adjusting s_j^{γ+1} using Algorithm 11
10:  end for
11: until schedules are not changed and no more resource conflicts
```

Algorithm 13 shows the algorithm resolving resource conflicts based on the ASAP scheduling. Initially, for each data operation o_i, the schedule s_i^o is the same as the ASAP schedule s_i^{ASAP}.

The scheduling algorithm then iteratively conducts the following two tasks. First, the algorithm inspects whether a resource constraint is violated. If this resource constraint is violated, i.e. more than available components q are required, all data operations using this hardware resource are collected, denoted as $\mathbf{O}_q = \{o_1, \ldots, o_n\}$. They are sorted by the order of the data dependencies. For each operation o_i, the new start time $s_i^{\gamma+1}$ is determined by checking every clock cycle from s_i^γ to see whether a

free component exists. If a component is free in this clock cycle, this operation is scheduled to start in that cycle. If there is no available component, the scheduler tries the next cycle.

It is likely to violate timing constraints during resolving resource conflicts. After operations are re-scheduled, timing constraints of the constraint graph are inspected. If there are violated timing constraints, adjustments similar to Algorithm 11 are applied to correct these violations.

Experientially, this algorithm converges fast. During resolving resource conflicts and correcting timing violations, data operations are not scheduled earlier than previous scheduling results. This effectively avoids infinite iterations during generating initial scheduling results. In a few difficult cases, data operations are pushed forward infinitely. There are effective and efficient approaches to detect these situations. However, they are not the main topics presented here.

A schedule satisfying both timing and resource constraints can be derived from the ALAP scheduling results by a similar algorithm.

6.6.2 The MMAS CRAAS algorithm

This section presents our evolutionary approach that addresses the concurrent scheduling and resource allocation (CRAAS) problem, which is similar to the MMAS approach that solves the timing-constrained scheduling problem, previously discussed in Section 5.5.

The proposed algorithm, as shown in Algorithm 14, is formulated as a searching process iteratively applying two tasks. The first is that a collec-

tion of M agents (ants) constructs individual schedules with local heuristics subject to both timing and resource constraints. This is followed by globally evaluating intermediate results to update local heuristics. The best solution achieved in these iterations is reported at the end of the searching process.

Algorithm 14 The MMAS CRAAS framework

```
1: construct τ(i, j, k) using results from Section 6.6.1;
2: initialize M ants
3: repeat
4:   for each single agent a_i such that 1 ≤ i ≤ M do
5:     individually construct a schedule S_i as Algorithm 15
6:     if schedule S_i is feasible then
7:       evaluate schedule S_i
8:       update S^Best
9:     end if
10:  end for
11:  update heuristic boundaries τ_max and τ_min;
12:  update global heuristics τ(i, j, k);
13: until no better solution found in the recent I iterations
14: report S^Best;
```

Constructing schedule using global/local heuristics

An individual ant a_m constructs a feasible schedule, as shown in Algorithm 15, starts from the ASAP/ALAP scheduling results, and iteratively conducts three tasks. The first task is to analyze the current scheduling results and check whether all resource constraints are satisfied. If not, the mobility range $[s^S, s^L]$ is updated. At the same time, the operation probability and the type distribution are updated. The second task is to determine which operation o_i should be scheduled in this iteration and which ones

should be deferred due to resource conflicts. The third is to schedule this candidate operation o_i on a type k resource at time step j, and update the ASAP/ALAP results.

Algorithm 15 MMAS constructing individual timing constraint schedule

```
1: load the ASAP/ALAP results
2: while exists unfulfilled resource constraints do
3:    for each operation o_i who violates timing/resource constraint do
4:       update the mobility range [s_i^S, s_i^L];
5:       update the operation probability r(i, j);
6:    end for
7:    for each resource type k do
8:       update the type distribution q(k);
9:    end for
10:   probabilistically defer operations that competes critical resources;
11:   probabilistically select candidate operation o_i;
12:   for s_i^S ⩽ j ⩽ s_i^L and all qualified resource type k do
13:      update local heuristic η(i, j, k);
14:   end for
15:   select time step j and type k resource using the p(i, j, k) as in Equa-
      tion (6.11);
16:   s_i^current = (j, k);
17:   update ASAP/ALAP schedules
18: end while
```

It is possible to find that there are no valid choices in step 15, or that the scheduler cannot successfully finish step 17. When this happens, this ant a_m quits the current searching process, analyzes the obtained partial schedule, and updates related heuristics with the hope that future iterations avoid similar failures.

Operation probabilities and type distribution The operation probability $p_o(i,j,k)$ shows that operation o_i is active during the control step j on type k resources, as shown in Equation 6.5. As discussed before, one of

the main limits of the FDS algorithm is that the FDS algorithm does not support more than one candidate resource types. Delays of different compatible resource types are considered. It is assumed that the probability of assigning operation o_i to type k resource is uniformly distributed.

$$p_{op}(i,j,k) = \begin{cases} \frac{1}{|K|}\sum_{l=0}^{D(i,k)} H(i,k)\dot{(}j-l)/(f_i^L - s_i^S + 1) & \text{if } s_i^S \leqslant j \leqslant f_i^L, \\ 0 & \text{otherwise.} \end{cases} \tag{6.5}$$

where $D(i,k)$ is the delay of performing operation o_i on a type k resource, $H(i,k)$ is a unit window function defined on $[j, j+D(i,k)]$, and K is the size of compatible resource types.

Therefore, the type distribution $q_{FU}k,j)$, which shows the concurrency of a type k resource at time step j, is defined as in Equation (6.6).

$$q_{FU}(k,j) = \sum_i p_{op}(i,j,k), \tag{6.6}$$

where the type k resource type is able to implement operation o_i. It is obvious that $q(k,j)$ estimates the number of type k resources that are required at time step l.

In order to consider the register and multiplexor cost, $p_l(i,j,k)$ is defined as the probability that the output from operation o_i is alive at time step j if o_i is assigned to a type k resource, where different resource types represent different latencies, hence different mobility ranges of successors. Therefore, a similar register distribution $q_R(b,j)$ showing the requirements of the b bits registers, can be defined in a similar manner as the type distribution, where b is the bit-width of o_i's output.

Local and global heuristics With type distribution and register distribution, it is possible to define the local heuristics $\eta(i,j,k)$ as follows.

$$\eta(i,j,k) = \frac{1}{q_{FU}(k,j)\cdot A_k + q_R(b,j)\cdot b\cdot A_R} \tag{6.7}$$

where A_k is the area of a type k resource and A_R is the area of a 1-bit register. Naturally, the local heuristic benefits decisions using less hardware resources.

The global heuristics $\tau(i,j,k)$ is similar to the global heuristics in the MMAS TCS algorithm.

$$\tau^T(i,j,k) = \rho\cdot\tau^{T-1}(i,j,k) + \sum_{m=1}^{M}\Delta\tau_m^T(i,j,k) \tag{6.8}$$

and

$$\Delta\tau_m(i,j,k) = \begin{cases} Q/(a^T_{average} - a_m) & \text{if } o_i \text{ is scheduled at } j \text{ on } k \text{ by ant } m \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

where ρ is the evaporation ratio and $0 < \rho < 1$, and Q is a fixed constant to control the delivery rate of the pheromone. Just like the previous work on the MMAS TCS algorithm, two important actions are performed in the global pheromone trail updating process. Evaporation is necessary for the MMAS optimization to effectively exploit the search space to avoid being caught by local optima, while reinforcement ensures that the favorable operation orderings receive a higher amount of pheromones and will have a better chance of being selected in the future iterations.

The difference is the comparisons of the current best solution and the

results generated by this individual agent. This comparison greatly benefits better results and discourages worse results.

Moreover, because a feasible schedule is not guaranteed by an individual agent, the related pheromone trails should be decreased properly to avoid being trapped there again. This is done by heuristics, such as decreasing the pheromone on the last scheduling decisions, or decreasing pheromones on highly competed resources and slower resources.

Deferring operations During each iteration, there could be more than enough operations ready to be scheduled or adjusted to meet timing constraints. Sometimes, operations compete for limited hardware resources. Some of these operations are dependent on each other. Because an operation cannot be scheduled before its ancestors, this operation should be deferred. It is possible that that a number of operations, which are not dependent on each other, are competing for the same resource. Which operation should be deferred is probabilistically determined, which is similar to that of the force-directed list scheduling. Deferring an operation in this iteration does not necessarily mean that this operation is scheduled in a later clock cycle, but it excludes this operation from the operations scheduled in this iteration. The probability of deferring the operation o_i is defined as follows.

$$p_d(i) = 1 - \frac{\frac{\sum_j \tau(i,j,k)}{(f_i^L - s_i^S - D(i,k))}}{\sum_l \frac{\sum_j \tau(l,j,k)}{(f_l^L - s_l^S - D(l,k))}} \tag{6.10}$$

where o_l are operations competing for the same resource. This intuitively defers operations with loose timing constraints. This formulation favors

an operation with much weaker pheromones and more possible schedules.

Depending on how coarse the optimization is, the scheduler could defer all but one operation in one iteration, schedule the only operation left, update ASAP/ALAP results, and schedule others in the following iterations. Alternatively, the scheduler could keep a number of candidates, which could be scheduled at the type k resource at the same time.

Scheduling operations When a candidate operation o_i is probabilistically picked up by an individual ant a_m as the next one to be scheduled, the ant needs to make decision on which resource type this operation should be assigned to and which time step this operation should start. This decision is made probabilistically as illustrated in Equation (6.11).

$$p(i,j,k) = \begin{cases} \frac{\tau^T(i,j,k)\cdot\eta(i,j,k)}{\sum_r\sum_l \tau^T(i,l,k)\cdot\eta(i,l,k)} & \text{if } (j,k) \text{ and } (l,r) \text{ are valid for } o_i \\ 0 & \text{otherwise} \end{cases} \tag{6.11}$$

where j is a candidate time step, which is between o_i's mobility range $[s_i^S, f_i^L - D(i,k)]$. Intuitively, those individual agents favor decisions that possess higher volumes of pheromone and a better local heuristic, i.e. a smaller area.

Update ASAP/ALAP scheduling Because an operation a_i is scheduled, schedules of its successors should be adjusted. This could be done by applying algorithms presented in Section 6.6.1. This ASAP algorithm always pushes start time forward and never looks backward when constructing individual scheduling results. If proper pruning and sorting are applied, this algorithm will converge rather quickly.

If the update process fails, which means the last deferring decision or the scheduling decision is not that good, related heuristics should be weakened.

Evaluating results The process of evaluating candidate results is straightforward. The area is calculated as follows.

$$a_m = \sum_k u_k \cdot A_k + \sum_b = 1^{max} u_{R_b} \cdot b \cdot A_R \tag{6.12}$$

Multiplexors implicitly implied by sharing functional units and registers should be counted as well.

The length of the schedule is defined as

$$l_m = f_K^m - s_S^m \tag{6.13}$$

where f_K is the finish time of the virtual sink vertex and s_S is the start time of the virtual source vertex.

If the target scheduling problem is the RCS problem, which optimizes area subject to the minimum number of control steps, the length of the schedule should be treated as a switch to update ASAP/ALAP schedules and global heuristics during the iterative searching process. If longer schedules are reported by individual ants, the results should be analyzed and related pheromone trials should be decreased.

6.7 Experimental Setup and Results

In order to evaluate the quality of the proposed MMAS CRAAS algorithm and collect results from actual synthesized hardware designs,

the proposed algorithm is implemented in a leading architectural synthesis framework, and compared with the existing resource allocation and scheduling algorithm.

The existing algorithm works on the constraint graph and conducts scheduling using allocated resources. If the target is to minimize latency or it is a pipelined design, the fastest components are allocated. The synthesis tool uses a scheduling algorithm based on force-directed scheduling [98]. This scheduling algorithm is refined to support multi-cycle operation, operation chaining, resource preference control, local timing constraints, and pipelined designs. This scheduling algorithm applies force-directed operation deferring to resolve resource conflicts. With the initial scheduling results, this synthesis tool conducts resource re-allocation to further minimize areas or latencies of generated designs.

The MMAS CRAAS algorithm with refinements is implemented in C/C++. The evaporation rate ρ is configured to be 0.98. The delivery rate $Q = 1$. These parameters are not changed over the tests. M is set to 10 for all the MMAS CRAAS test cases.

6.7.1 Summary of results

The benchmark suite of FPGA-based designs consists of 260 non-pipelined designs, 160 low-throughput pipelined designs, 173 middle-throughput pipelined designs, and 71 high-throughput pipelined designs. They cover almost all known designs using architectural synthesis tools. Their sizes range from small to huge. Most of them are computation

intensive designs, and some others are control dominant designs.

Results are collected at different stages of the design flow. Latencies and separate areas of functional units, logic and register/multiplexors, are collected from the architectural synthesis tool. The RTL areas are reported by Mentor Graphics Precision after technology mapping. The PnR areas are reported by Xilinx ISE after placement and routing.

The results show the *percentages* compared with the existing solution. If a percentage number is positive, then the result from our MMAS-based algorithm is less than the results from the existing algorithm, which is good when the algorithm is optimizing for area and latency. The greater the positive number is, the better the results from the MMAS CRAAS algorithm. For example, Table 6.1(a) shows an average of 3.11% smaller areas after place and routing but an average of 0.47% shorter latencies.

The average saving is the average of all test cases in each test suite with uniform weights on each design. The weighted average is the average of all test cases in each test suite with weights corresponding to their area. The larger the design, the more significant the savings in the weighted average.

Table 6.1(a) presents the results summary of 260 non-pipelined designs and the target is to minimize the number of control steps. The proposed algorithm generates schedules that are slightly faster than the existing algorithm, but the area is 3.25% smaller on average. Table 6.1(a) presents the results of non-pipelined designs and the target is to minimize the number of configurable logic blocks. The area is 3.4% smaller on average. How-

(a) 260 non-pipelined FPGA designs optimized for latency

	# Control Steps	RTL Area	PnR Area	Savings of Area		
				Func	Logic	MUX
Average	0.47	3.25	3.11	-4.69	-3.73	5.12
Weighted Average	0.04	6.66	6.10	3.33	-6.07	12.86

(b) 260 non-pipelined FPGA designs optimized for area

	# Control Steps	RTL Area	PnR Area	Savings of Area		
				Func	Logic	MUX
Average	-9.66	3.51	3.40	0.69	-4.29	0.18
Weighted Average	-0.02	6.64	6.20	21.04	-3.74	4.59

Table 6.2: Summary of the quality-of-results of non-pipelined FPGA designs designs

ever, the latencies are 9% longer on average, but this could be dominated by some small designs because the weighted average is only 0.02% longer on average.

(a) 160 low-throughput FPGA designs(59/55/36)

	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
Average	-9.45	6.54	11.62	1.03	-8.31
Weighted Average	-0.05	14.90	25.23	7.27	-12.48

(b) 173 mid-throughput FPGA designs(64/88/21)

	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
Average	-7.69	6.59	11.43	2.16	-5.89
Weighted Average	-0.04	13.90	21.35	2.37	-4.82

(c) 71 high-throughput FPGA designs(18/47/6)

Test case Id	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
Average	-1.15	0.25	0.05	9.56	6.70
Weighted Average	-0.03	1.42	1.41	2.27	0.01

Table 6.3: Summary of the quality-of-results of FPGA pipelined designs

Table 6.3 presents the results summary of all pipelined designs. The target here is to minimize the number of configurable logic blocks. The average area savings range from 0.25% to 6.54%.

To summarize, the proposed algorithm shows stronger abilities to exploit the opportunities of sharing resources among operations. This is more obvious in larger test cases, as shown by the weighted average results. The existing algorithm may converge to pre-mature results. However, due to different situations occurring in hardware synthesis, especially those transformation and optimizations happening after resource allocation and scheduling, some designs received worse results against the expected design goals.

6.7.2 Case-by-case comparisons

This section presents case-by-case comparisons of the quality of results generated by the proposed algorithm and the existing approach. The example suite is the low-throughput pipelined FPGA designs. Table 6.4, 6.5, and 6.6 shows detailed results of many test cases.

There are 160 test cases in this benchmark suite. These two algorithms tie each other on 55 designs. The proposed algorithm generates smaller designs in 59 test cases and the existing algorithm generates smaller designs in 36 test cases. The average area savings over all test cases is 6.54%. The weighted average area savings of all test cases is 14.90%.

Table 6.6 presents losing test cases in this test suite.

Analysis of some typical results are presented here to help understand the behavior of the proposed algorithm and generated hardware.

1. **Id 1** In this test case, the proposed approach gained 54% savings in

Test case Id	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
1	-33.33	54.29	58.40	5.77	-30.43
2	-22.22	54.26	61.48	-51.49	-8.42
3	-20.00	52.71	64.00	3.35	-26.93
4	-33.33	50.06	55.97	-25.71	-58.05
5	-33.33	49.07	58.89	-0.23	-37.53
6	-16.67	48.65	56.15	-5.42	-54.89
7	-16.00	45.77	52.55	-0.88	-39.48
8	-7.69	43.07	56.14	-1.21	-17.14
9	0.00	40.77	44.54	0.00	-0.48
10	-33.33	39.43	50.00	100.00	-33.33
11	-66.67	37.94	44.96	8.33	-68.54
12	-66.67	34.97	61.24	-6.99	-32.15
13	-0.66	28.50	45.47	-6.95	-12.79
14	-25.00	26.15	28.49	-10.45	-68.04
15	0.00	24.06	46.48	0.00	-0.94
16	-80.00	23.97	25.47	63.07	-32.89
17	-28.57	22.88	41.33	-16.55	-18.27
18	-60.00	21.61	25.34	-4.24	-33.88
19	-20.00	21.37	31.46	-9.36	-13.02
20	-13.04	21.26	29.64	-2.64	-26.19
21	-50.00	21.13	24.58	-44.12	-23.81
22	-0.03	20.78	34.67	-0.33	-15.00
23	-0.09	20.64	24.32	-4.36	-19.27
24	-0.47	19.64	25.67	-2.47	-3.85
25	0.00	19.39	0.00	77.31	-23.90
26	-33.33	18.37	48.13	-8.11	-31.01
27	-14.29	18.35	42.57	0.66	-14.05
28	-42.86	17.86	47.14	-4.24	-25.08
29	-33.33	17.52	24.60	100.00	-35.11

Table 6.4: Details of mid-low throughput designs (Winning test part 1)

Test case Id	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
30	-33.33	16.90	21.90	0.64	-10.02
31	-33.33	15.29	32.75	-3.73	-5.14
32	-66.67	15.01	44.18	-8.83	-47.26
33	-16.00	14.97	17.38	-3.54	-14.31
34	-14.29	14.75	48.16	-37.23	-11.10
35	-0.05	14.00	48.71	-4.79	-11.17
36	-57.14	13.94	28.85	-13.79	-22.80
37	-50.00	13.21	42.99	-4.51	-30.57
38	-40.00	12.34	30.83	2.60	-24.93
39	-75.00	11.55	49.67	-50.23	1.13
40	-33.33	9.10	20.00	100.00	-54.74
41	-0.06	7.92	9.71	-8.97	-7.56
42	-0.06	7.92	9.71	-8.97	-7.56
43	-54.55	7.17	15.43	-42.08	-33.16
44	0.00	6.54	27.27	-26.58	-6.27
45	-0.37	5.86	12.82	-2.19	-22.30
46	-27.27	4.58	21.89	-2.90	-14.31
47	-12.50	1.97	5.26	-7.63	-0.43
48	-1.43	1.05	5.90	-4.32	-15.48
49	-1.43	1.05	5.90	-4.32	-15.48
50	-20.00	0.98	0.00	-3.38	5.95
51	0.00	0.98	0.00	-34.69	3.51
52	0.00	0.68	-1.32	1.74	1.37
53	-42.86	0.51	22.40	-16.02	-9.93
54	-42.86	0.51	22.40	-16.02	-9.93
55	0.00	0.43	3.38	-1.23	-0.55
56	0.00	0.21	0.00	1.14	0.36
57	0.00	0.14	0.00	2.71	0.00
58	0.00	0.02	-0.03	0.00	0.30

Table 6.5: Details of mid-low throughput designs (Winning test part 2)

Test case Id	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
140	-0.05	-0.03	-0.12	5.10	-0.88
141	0.00	-0.07	-0.15	0.00	0.00
142	-14.29	-0.10	0.00	-9.81	-0.08
143	0.00	-0.29	-1.20	0.00	0.00
144	0.00	-0.30	0.00	-2.95	0.00
145	0.00	-0.33	-0.45	0.00	0.00
146	0.00	-0.40	0.00	1.40	-2.66
147	0.00	-0.43	0.00	0.04	-1.68
148	-0.03	-0.75	0.00	-2.99	-1.07
149	0.00	-0.88	-0.81	10.97	-6.89
150	0.00	-0.91	0.00	-24.00	-1.03
151	-33.33	-1.50	0.00	0.00	-50.00
152	-66.67	-2.66	0.00	-45.17	0.05
153	0.00	-2.67	0.00	-0.29	-4.58
154	0.00	-2.70	1.27	-21.73	-0.48
155	0.00	-3.25	0.00	-43.57	4.62
156	0.00	-4.02	0.30	-3.71	-5.26
157	-4.00	-10.96	1.17	-2.66	-44.43
158	-20.00	-11.03	0.00	-25.80	-9.75
159	0.00	-11.79	0.00	0.18	-25.54
160	0.00	-12.20	4.07	-25.12	-13.85

Table 6.6: Details of mid-low throughput designs (Losing test cases)

total, but the latency increased 33%. The latency increase is fine given this is a pipelined design. Once the throughput goal is met, the rest of the job is to optimize area. This design shows more sharing compared to the existing approach. Hence, the functional units saved a lot, but the register and multiplexor's area goes up 30%.

This is the representative behavior of the proposed approach: the functional units and multiplexors' areas are carefully evaluated, more sharing is obtained. Although more multiplexors and registers are found, they are just enough to keep the savings from the functional units. Similar patterns can be found in test cases 3–8.

2. **Ids 9 and 15** These two test cases show better results on functional units only and there are no changes on latencies. Smaller components or more sharing is obtained from the proposed approach.

3. **Id 53 and 54** These two test cases show good savings on functional units but spend more on registers and multiplexors, which is normal. However, these two test cases also contain a large amount of control logic, which cannot be well estimated in high-level synthesis stage. Therefore, we did not obtain very good results here.

4. **Id 160** This is another typical results from the proposed algorithm, but this is generally bad. It is quite hard to estimate the area of multiplexors when constructing individual schedules. This may actually cause larger designs when the algorithm thought it is saving functional area.

6.7.3 Experimental results of ASIC designs

We also conduct experiments of solving the resource allocation and scheduling problems for ASIC designs using the proposed MMAS CRAAS algorithm. This is compared with the same existing solution with different target architecture.

The benchmark suite consists of 260 non-pipelined designs, 250 low-throughput pipelined designs, 160 middle-throughput pipelined designs, and 120 high-throughput pipelined designs. Latencies and separate areas of functional units, logic and register/multiplexors, are collected from the architectural synthesis tool. The estimated total areas are collected from the Synopsys Design Compiler after RTL synthesis. As with the results of FPGA-based designs, these numbers are an improvement in percentages compared with the existing solution.

(a) 260 Non-pipelined ASIC designs optimized for latency

	# Control Steps	DC Area	Savings of Area		
			Func	Logic	MUX
Average	3.61	1.77	0.33	-3.26	8.75
Weighted Average	5.15	4.05	9.74	0.23	11.29

(b) 260 Non-pipelined ASIC designs optimized for area

	# Control Steps	DC Area	Savings of Area		
			Func	Logic	MUX
Average	-17.92	6.17	13.51	-3.73	-0.99
Weighted Average	-26.00	10.96	35.60	-3.31	-0.54

Table 6.7: Summary of the quality of results of non-pipelined designs

Tables 6.7 presents the results of non-pipelined ASIC designs. The proposed algorithm achieves 6.07% smaller designs compared with the existing solution for the TCS problem. For the RCS problem, the achieved

designs are 1.77% smaller but 3.61% faster compared with the existing solution.

(a) 250 low-throughput ASIC designs(113/68/68)

	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
Average	-22.70	8.09	18.91	2.47	-5.81
Weighted Average	-0.42	22.86	53.07	-1.02	-20.47

(b) 160 mid-throughput ASIC designs(67/53/40)

	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
Average	-12.56	6.05	14.16	-1.33	-5.09
Weighted Averagez1	-0.09	11.06	31.47	-5.61	-24.62

(c) 120 high-throughput ASIC designs(37/60/23)

	# Control Steps	Savings of Area			
		Total	Func	Logic	MUX
Average	-12.63	3.61	11.13	4.51	8.34
Weighted Average	-0.59	10.12	21.07	0.58	1.59

Table 6.8: Summary of the quality of results of ASIC pipelined designs

Table 6.8 presents the summary of results of pipelined designs. As the pipelined designs, the area savings ranges from 3.61% to 8.09%.

The resource allocation and scheduling problem is more complicated than the FPGA-based reconfigurable architecture. Because the ASIC designs are based on standard cells, it is possible to implement a data operation in multiple ways with different delay and areas. To summarize, the empirical data shows that the proposed algorithm performs well on different design goals.

6.8 Summary

A concurrent resource allocation and scheduling problem and its solution are presented in this section. This problem is generalized for actual architectural-level hardware synthesis. It is required to find a good design subject to specified timing and resource constraints but few other constraints, which leaves the proposed solution a lot of freedom but also a huge solution space.

The proposed algorithm combines the MMAS evolutionary approach and the distribution graphs from the FDS, and multiple agents iteratively search the design space and generate resource allocation and scheduling results.

Experiments are conducted on about 1250 industrial test cases, ranging from very small to very large designs. The average results are rather good, which is especially good for pipelined ASIC designs, which have more choices for mapping data operations, sharing, and so forth.

Future work is mainly focused on better timing estimation of control-dominated design and further utilizing regularities of the graph to reduce the searching space.

Chapter 7

Conclusions and Future Work

Reconfigurable computing combines the flexibility of software with the high performance of hardware, bridges the gap between general-purpose processors and application-specific systems, and enables higher productivity and shorter time to market.

Design flows for reconfigurable computing systems conduct parallelizing compilation and reconfigurable hardware synthesis in an integrated framework. A successful synthesizer starts from the system specifications in high-level programming languages, conducts parallelizing transformations and optimizations to exploit parallelism at different levels, generates software object code, and synthesizes reconfigurable hardware using architectural synthesis, technology mapping, and physical design technologies.

Advancements in parallelization compilers and electronic design automation make it possible to design complex reconfigurable computing

systems. However, when synthesizing these systems, designers face great challenges in improving the system performance and resource utilization, developing effective and efficient optimization algorithms, and reducing interferences from designers. This dissertation presents novel synthesis techniques and optimization algorithms. The major highlights are summarized in the following section.

7.1 Summary of Major Results

Program representation We propose a novel program representation as the basis of the compiler framework synthesizing sequential programs into reconfigurable systems. This program representation is derived from extending the program dependence graph (PDG) with the static single assignment (SSA) form.

This PDG+SSA form enables the synthesizer to explore more parallelism at not only the instruction level but also at higher levels. A number of loop transformations can be easily conducted using this form. With the extension of the SSA form, it is possible to conduct data-flow analysis, create large synthesis blocks, exploit instruction level parallelism, and therefore generate more area-efficient designs compared with the widely adopted control/data-flow graph model.

Operation scheduling Scheduling data operations on allocated resources is always one of the most important problems in architectural

synthesis. The quality of the scheduling results determines the quality of synthesized hardware. Because the size of the design and the complexity of design problems keep increasing, it is impossible to apply exact algorithms to obtain the optimal solutions.

To generate quantitatively close to optimal solutions during scheduling, the MMAS scheduling algorithm is designed to exploit the solution spaces effectively and efficiently. The MMAS scheduling is a probabilistic optimization algorithm based on the ant system meta-heuristics. Our experimental results show that the MMAS scheduling algorithm outperforms published scheduling algorithms, such as the list scheduler, and the force directed scheduling algorithm. Compared with results from the integer linear programming, our results are closer to the known optima. The proposed algorithm obtains the optima for some test cases.

Concurrent resource allocation and scheduling Realistic hardware design presents much more complicated optimization problems. This is especially true during resource allocation and scheduling. Different complicated design factors should be considered. Timing constraints and resource constraints are normally mixed together. All of these make it impossible to model these problems using existing RCS/TCS model, or solve them using existing algorithms.

We present a general model of the resource allocation and scheduling problem, redefine the RCS/TCS problems, and propose a concurrent resource allocation and scheduling algorithm based on the MMAS opti-

mization. Experimental results show that our work outperforms existing algorithms from 5% to 20%, depending on specific design goals.

Data space partitioning and storage arrangement Modern FPGA-based reconfigurable architectures normally integrate a rather complicated memory hierarchy. In order to create more coarse-grained parallelism, and fully utilize available hardware resources, especially those storage components, algorithms analyzing the loop structures and exploiting reasonable storage plan are proposed. Results show that a good partition of the iteration space and the data space can effectively parallelize the input program, and create great parallelism among those program portions.

7.2 Future Work

Extract regularity Regularity comes from the program behavior and different transformations, such as loop unrolling, loop merging, etc. Two or more portions of the program show the same or very similar structures between each other. If the compiler framework can effectively extract these regularities, the architectural synthesizer can consider these regularities, then generate higher performance area-efficient reconfigurable hardware. Regularity is also very important to reduce the reconfiguration cost.

Optimized heuristic algorithms Design automation problems can be modeled to mathematical optimization problems. However, these problems can never be solved as pure mathematical problems, and heuristics from realistic designs and practical work should be carefully considered in order to effectively and efficiently solve these problems. Those proposed algorithms based on MMAS and other heuristics need to be further refined to reflect the design problems they are working on.

Loop transformations Our work on loop transformations presented in this dissertation is limited to certain loop structures. More generalized approaches that create coarse-grained parallelism should be explored. These techniques also benefit current efforts on parallelizing programs from tera-scale computer architectures.

Bibliography

[1] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 1986.

[3] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. *An Overview of the SUIF2 Compiler Infrastructure*. Computer Systems Laboratory, Stanford University, 1999.

[4] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. *The Basic SUIF Programming Guide*. Computer Systems Laboratory, Stanford University, August 2000.

[5] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[6] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988.

[7] Altera Corporation. *Stratix II Device Handbook*, January 2005.

[8] A. Auyeung, I. Gondra, and H. K. Dai. *Advances in Soft Computing: Intelligent Systems Design and Applications*, chapter Integrating random ordering into multi-heuristic list scheduling genetic algorithm. Springer-Verlag, 2003.

[9] Nastaran Baradaran and Pedro C. Diniz. A register allocation algorithm in the presence of scalar replacement for fine-grain configurable architectures. In *Proceedings of the 2005 Conferenc on Design Automation and Testing in Europe (DATE05)*, 2005.

[10] Steve J. Beaty. Genetic algorithms versus tabu search for instruction scheduling. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, 1993.

[11] Peter Bergsman. Xilinx FPGA Blasted into Orbit. *Xcell Journal*, (46):86–88, Summer 2003.

[12] David Bernstein, Michael Rodeh, and Izidor Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. *IEEE Transactions on Computers*, 38(9):1308–13, September 1989.

[13] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. GURRR: a Global Unified Resource Requirements Representation. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, 1995.

[14] Kiran Bondalapati and Viktor K. Prasanna. Reconfigurable Computing Systems. *Proc. of the IEEE*, 90(7):1201–17, July 2002.

[15] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software: Practice and Experience*, 28(8):859–81, July 1998.

[16] Stephen Brown and Jonathan Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, Summer 1996.

[17] Mihai Budiu and Seth C. Goldstein. Optimizing Memory Accesses For Spatial Computation. In *International Symposium on Code Generation and Optimization*, 2003.

[18] Mihai Budiu and Seth Copen Goldstein. Compiling Application-Specific Hardware. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, 2002.

[19] David Callahan, Steve Carr, and Ken Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the SIGPLAN '90 Symposium of Programming Language Design and Implementation*, 1990.

[20] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, 1991.

[21] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69, April 2000.

[22] Timothy J. Callahan and John Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, 1998.

[23] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated Static Single Assignment. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 1999.

[24] Francky Catthoor, Koen Danckart, Chidamber Kulkarni, Eric Brockmeyer, Per Gunnar Kjeldsberg, Tanja Van Achteren, and Thierry Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, Norwell, MA, 2002.

[25] D. Chen and J. Rabaey. Paddi : Programmable arithmetic devices for digital signal processing. In *Proceedings of the IEEE Workshop on VLSI Signal Processing*, pages 240–249, November 1990.

[26] Richard J. Cloutier and Donald E. Thomas. The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[27] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the Operational Research Society*, 48:295–305, 1996.

[28] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–90, October 1991.

[29] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of dag parallelism. In *Proceedings fo the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[30] Hugo De Man, Francky Catthoor, Gert Goossens, Jan Vanhoof, Jef Van Meerbergen, Stefaan Note, and Jef Huisken. Architecture-driven synthesis techniques for VLSl implementation of DSP algorithms. *Proc. of the IEEE*, 78(2):319–35, February 1990.

[31] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., Hightstown, NJ, 1994.

[32] Andrè DeHon. The Density Advantage of Configrable Computing. *Computer*, 33(4):41–49, April 2000.

[33] J. L. Deneubourg and S. Goss. Collective Patterns and Decision Making. *Ethology, Ecology & Evolution*, 1:295–311, 1989.

[34] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man and Cybernetics, Part-B*, 26(1):29–41, February 1996.

[35] Carl Ebeling, Darren C. Cronquist, Paul Franklin, and Chris Fisher. RaPiD - A Configurable Computing Architecture for Compute-Intensive Applications. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications*, 1996.

[36] Stephen A. Edwards. An Esterel Compiler for Large Control-Dominated Systems. *IEEE Transactions on Computer-Aided Design of Integrated Citcuits and Systems*, 21(2):169–83, February 2002.

[37] Stephen A. Edwards. High-Level Synthesis from the Synchronous Language Esterel. In *Proceedings of the IEEE/ACM 11th International Workshop on Logic and Synthesis*, 2002.

[38] John P. Elliott. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Kluwer Academic Publishers, Norwell, MA, 1999.

[39] G. Estrin and C. R. Viswanathan. Organization of a "fixed-plus-variable" structure computer for computation of eigenvalues and eigenvectors of real symmetric matrices. *J. ACM*, 9(1):41–60, 1962.

[40] Serge Fenet and Christine Solnon. Searching for maximum cliques with ant colony optimization. *3rd European Workshop on Evolutionary Computation in Combinatorial Optimization*, April 2003.

[41] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–49, July 1987.

[42] S. Fidanova. Evolutionary Algorithm for Multiple Knapsack Problem. In *Proceedings of PPSN-VII, Seventh International Conference on Parallel Problem Solving from Nature*, Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany, 2002.

[43] Daniel D. Gajski and Loganath Ramachandran. Introduction to High-Level synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, Winter 1994.

[44] L. M. Gambardella, E. D. Taillard, and G. Agazzi. *New Ideas in Optimization*, chapter A multiple ant colony system for vehicle routing problems with time windows, pages 51–61. McGraw Hill, London, UK, 1999.

[45] L. M. Gambardella, E. D. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment. *Journal of the Operational Research Society*, 50(2):167–176, 1996.

[46] Maya B. Gokhale and Janice M. Stone. Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[47] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[48] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing, 2nd Edition*. Prentice Hall, Englewood Cliffs, NJ, 2002.

[49] Martin Grajcar. Genetic List Scheduling Algorithm for Scheduling and Allocation on a Loosely Coupled Heterogeneous Multiprocessor System. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation Conference*, 1999.

[50] Rajiv Gupta and Mary Lou Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–31, April 1990.

[51] Walter J. Gutjahr. A graph-based ant system and its convergence. *Future Gener. Comput. Syst.*, 16(9):873–888, 2000.

[52] Walter J. Gutjahr. Aco algorithms with guaranteed convergence to the optimal solution. *Inf. Process. Lett.*, 82(3):145–153, 2002.

[53] Walter J. Gutjahr. A generalized convergence result for the graph-based ant system metaheuristic. *Probability in the Engineering and Informational Sciences*, 17:545 – 569, 2003.

[54] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *Computer*, 29(12):84–89, December 1996.

[55] Jeffrey Hammes, Bob Rinker, A. P. Wim Bohm, Walid A. Najjar, Bruce A. Draper, and J. Ross Beveridge. Cameron: High Level Language Compilation for Reconfigurable Systems. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[56] Reiner W. Hartenstein and Rainer Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *ASP-DAC '95: Proceedings of the 1995 conference on Asia Pacific design automation (CD-ROM)*, page 77, New York, NY, USA, 1995. ACM.

[57] John R. Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

[58] Simon Haykin. *Adaptive Filter Theory, Fourth Edition*. Prentice Hall, Englewood Cliffs, NJ, 2001.

[59] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, New York, NY, 1977.

[60] M. Heijligers and J. Jess. High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques. In *International Conference on Evolutionary Computation*, pages 56–61, Perth, Australia, 1995.

[61] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[62] Glenn Holloway. *The Machine-SUIF Static Single Assignment Library*. Division of Engineering and Applied Sciences, Harvard University, July 2002.

[63] Glenn Holloway and Allyn Dimock. *The Machine-SUIF Bit-Vector Data-Flow-Analysis Library*. Division of Engineering and Applied Sciences, Harvard University, July 2002.

[64] Glenn Holloway and Michael D. Smith. *The Machine-SUIF Control Flow Analysis Library*. Division of Engineering and Applied Sciences, Harvard University, July 2002.

[65] Glenn Holloway and Michael D. Smith. *The Machine-SUIF Control Flow Graph Library*. Division of Engineering and Applied Sciences, Harvard University, July 2002.

[66] Susan Horwitz, Jan Prins, and Thomas Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[67] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990.

[68] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–48, 1961.

[69] Zhining Huang and Sharad Malik. Exploiting Operation Level Parallelism through Dynamically Reconfigurable Datapaths. In *Proceedings of the 39th Conference on Design Automation*, 2002.

[70] Richard Johnson and Keshav Pingali. Dependence-Based Program Analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1993.

[71] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[72] Rainer Kolisch and Sonke Hartmann. *Project Scheduling: Recent models, algorithms and applications*, chapter Heuristic Algorithms for Solving the Resource-Constrained Project Scheduling problem: Classification and Computational Analysis. Kluwer Academic Publishers, 1999.

[73] Rainer Kress. *A fast reconfigurable ALU for Xputers*. PhD thesis, University of Kaiserslautern, 1996.

[74] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1981.

[75] Manjunath Kudlur, Kevin Fan, Michael Chu, and Scott Mahlke. Automatic synthesis of customized local memories for multicluster application accelerators. In *Proceedings of IEEE 15th International Conference on Application-Specific Systems, Architectures and Processors*, 2004.

[76] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–15, February 2007.

[77] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[78] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[79] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, October 1999.

[80] Jiahn-Hung Lee, Yu-Chin Hsu, and Youn-Long Lin. A new integer linear programming formulation for the schedulingproblem in data path synthesis. In *Proceedings of ICCAD-89*, pages 20–23, Santa Clara, CA, USA, Nov 1989.

[81] G. Leguizamon and Z. Michalewicz. A new version of ant system for subset problems. In *Proceedings of the 1999 Congress of Evolutionary Computation*, pages 1459–1464. IEEE Press, 1999.

[82] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, 1992.

[83] M. Morris Mano and Charles Kime. *Logic and Computer Design Fundamentals (2nd edition)*. Prentice Hall, Englewood Cliffs, NJ, 1999.

[84] Yan Meng, Andrew P. Brown, Ronald A. Iltis, Timothy S herwood, Hua Lee, and Ryan Kastner. Mp core: Algorithm and design techniques for efficient channel estimation in wireless applications. In *Proceedings of the 42nd Design Automation Conference (DAC)*, Anaheim, California, USA, June 2005.

[85] R. Michel and M. Middendorf. *New Ideas in Optimization*, chapter An ACO algorithm for the shortest supersequence problem, pages 51–61. McGraw Hill, London, UK, 1999.

[86] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[87] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[88] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[89] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990.

[90] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Exploiting Off-Chip Memory Access Modes in High-Level Synthesis. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design*, 1997.

[91] Santosh Pande. A Compile Time Partitioning Method for DOALL Loops on Distributed Memory Systems. In *Proceedings of 1996 International Conference on Parallel Processing*, 1996.

[92] Santosh Pande and Dharma P. Agrawal, editors. *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation*

Techniques, and Run Time Systems. Springer, Heidelberg, Germany, 2001.

[93] In-Cheol Park and Chong-Min Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 680–685, New York, NY, USA, 1991. ACM Press.

[94] Rafael S. Parpinelli, Heitor S. Lopes, and Alex A. Freitas. Data mining with an ant colony optimization algorithm. *IEEE Transaction on Evolutionary Computation*, 6(4):321–332, August 2002.

[95] David Patterson and John Hennessy. *Computer Organization and Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[96] P. G. Paulin and J. P. Knight. Force-directed scheduling in automatic data path synthesis. In *24th ACM/IEEE Conference Proceedings on Design Automation Conference*, 1987.

[97] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Trans. Computer-Aided Design*, 8:661–679, 1989.

[98] Pierre G. Paulin and John P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Citcuits and Systems*, 8(6):661–79, June 1989.

[99] P. Poplavko, C.A.J. van Eijk, and T. Basten. Constraint analysis and heuristic scheduling methods. In *Proceedings of 11th. Workshop on Circuits, Systems and Signal Processing (ProRISC2000)*, pages 447–453, 2000.

[100] J. Ramanujam and P. Sadayappan. Compile-time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–82, October 1991.

[101] Narasimhan Ramasubramanian, Ram Subramanian, and Santosh Pande. Automatic Analysis of Loops to Exploit Operator Parallelism on Reconfigurable Systems. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, 1998.

[102] Ronny Ronen, Avi Mendelson, Konrad Lai, Shih-Lien Lu, Fred Pollack, and John P. Shen. Coming Challenges in Microarchitecture and Architecture. *Proc. of the IEEE*, 89(3):325–40, March 2001.

[103] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *Proc. of the IEEE*, 81(7):1010–29, July 1993.

[104] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988.

[105] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[106] Ruud Schoonderwoerd, Owen Holland, Janet Bruten, and Leon Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, 5:169–207, 1996.

[107] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. Piconpa: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing Systems*, 31(2):127–42, June 2002.

[108] J. M. J. Schutten. List scheduling revisited. *Operation Research Letter*, 18:167–170, 1996.

[109] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors, 2002 Update*, 2002.

[110] Alok Sharma and Rajiv Jain. Insyn: Integrated scheduling for dsp applications. In *DAC*, pages 349–354, 1993.

[111] Kuei-Ping Shih, Jang-Ping Sheu, and Chua-Huang Huang. Statement-Level Communication-Free Partitioning Techniques for Parallelizing Compilers. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, 1996.

[112] Michael D. Smith and Glenn Holloway. *An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization*. Division of Engineering and Applied Sciences, Harvard University, July 2002.

[113] T. Stützle and M. Dorigo. A short convergence proof for a class of ACO algorithms. *IEEE Transactions on Evolutionary Computation*, 6(4):358–365, 2002.

[114] Thomas Stützle and Holger H. Hoos. MAX-MIN Ant System. *Future Generation Comput. Systems*, 16(9):889–914, September 2000.

[115] Roy A. Sutton, Vason P. Srini, and Jan M. Rabaey. A multiprocessor dsp system using paddi-2. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 62–65, New York, NY, USA, 1998. ACM.

[116] Philip H. Sweany and Steve J. Beaty. Instruction scheduling using simulated annealing. In *Proceedings of 3rd International Conference on Massively Parallel Computing Systems*, 1998.

[117] Xinan Tang, Manning Aalsma, and Raymond Jou. A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors. In *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications*, 2000.

[118] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: a Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, March/Arpil 2002.

[119] Donald E. Thomas, Elizabeth D. Lagnese, John A. Nestor, Jayanth V. Rajan, Robert L. Blackburn, and Robert A. Walker. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, Norwell, MA, 1989.

[120] Haluk Topcuouglu, Salim Hariri, and Min you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[121] Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings. Sea Cucumber: A Synthesizing Compiler for FPGAs. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, 2002.

[122] W. F. J. Verhaegh, E. H. L. Aarts, J. H. M. Korst, and P. E. R. Lippens. Improved force-directed scheduling. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 430–435, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[123] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, A. van der Werf, and J. L. van Meerbergen. Efficiency improvements for force-directed scheduling. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 286–291, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[124] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *Computer*, 30(9):86–93, September 1999.

[125] Gang Wang, Wenrui Gong, and Ryan Kastner. A New Approach for Task Level Computational Resource Bi-partitioning. *15th International Conference on Parallel and Distributed Computing and Systems*, 1(1):439–444, November 2003.

[126] Gang Wang, Wenrui Gong, and Ryan Kastner. System level partitioning for programmable platforms using the ant colony optimization. *13th International Workshop on Logic and Synthesis, IWLS'04*, June 2004.

[127] Gang Wang, Wenrui Gong, and Ryan Kastner. Instruction scheduling using MAX-MIN ant optimization. In *15th ACM Great Lakes Symposium on VLSI, GLSVLSI'2005*, April 2005.

[128] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: Representation Without Taxation. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languag*, 1994.

[129] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000.

[130] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[131] Xilinx, Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, October 2003.

[132] Xilinx, Inc. *Virtex-II Pro Platform FPGA Data Sheet*, January 2003.

[133] Xilinx, Inc. *Xilinx FPGAs Aboard Mars 2003 Exploration Mission*, July 2003.

[134] A. K. W. Yeung. *PADDI-2 Architecture and Implementation*. PhD thesis, University of California, Berkeley, 1995.