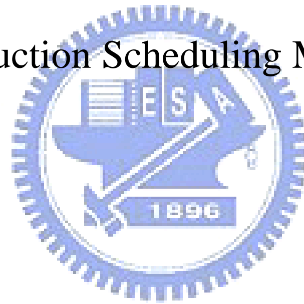
# 國立交通大學

## 資訊工程學系

## 博 士 論 文

在數位訊號處理器架構下有效指令排程法之研究

A Study on Effective Instruction Scheduling Methods for DSP Architecture




研 究 生：李宜軒

指導教授：陳　正　教授

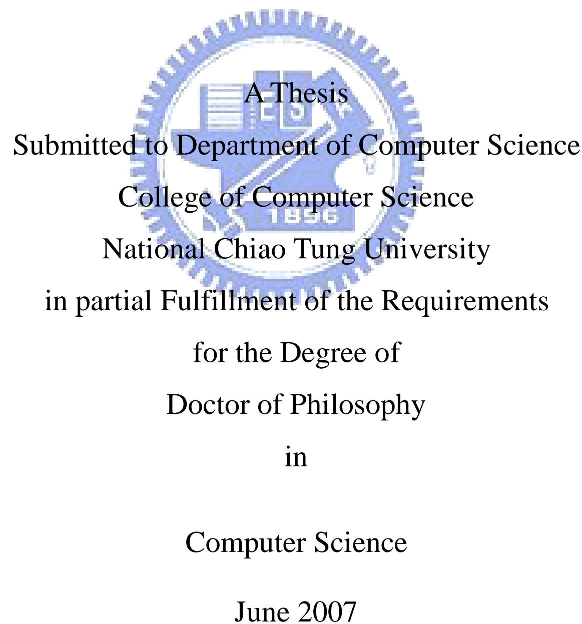
中 華 民 國 九 十 六 年 六 月

# 在數位訊號處理器架構下有效指令排程法之研究
# A Study on Effective Instruction Scheduling Methods for DSP Architecture

研 究 生：李宜軒　　Student：Yi-Hsuan Lee

指導教授：陳　正　　Advisor：Cheng Chen

國 立 交 通 大 學
資 訊 工 程 學 系
博 士 論 文



A Thesis

Submitted to Department of Computer Science

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

# 在數位訊號處理器架構下有效指令排程法之研究

學生：李宜軒　　　　　　　　　　　　　　　　指導教授：陳　正 教授

國立交通大學資訊工程學系博士班

摘要

隨著多媒體通訊日益劇增的需求，陸續發展出許多關於科學計算及數位訊號處理的方法。這些應用程式以規則相依迴圈為主，計算複雜度很高，大部分時間都是執行 ALU 運算指令。數位訊號處理器 (digital signal processor, DSP) 是一種為特殊目的設計的微處理器，通常包含多個獨立的資料記憶體模組 (multiple data memory banks)，並採用 heterogeneous register sets 方式；而要有效利用這些架構特性，顯然需要充分的編譯技術支援。為了提高數位訊號處理應用程式的執行效能，在編譯過程中必須開發迴圈之間潛在的平行度，並盡量減少額外指令 (spill codes) 的產生。同時，由於攜帶型電子裝置的逐漸普及，功率消耗也成為另一個重要的設計議題；若是能從高階合成 (high-level synthesis) 的觀點來考慮功率消耗，通常能以較低的代價 (cost)，來有效降低功率消耗。

在本論文中我們將針對包含多重資料記憶體模組的數位訊號處理器以及規則相依迴圈，設計有效的指令排程法。對於這種系統架構，完整的編譯過程必須涵蓋多個步驟，由於這些步驟彼此之間有複雜的資料相依性，同時考慮多個步驟將有助於得到較佳的排程結果。本論文主要分為三個研究議題，首先我們設計三個簡單的變數分割機制 (variable partition mechanism)，以及三個對應的指令排程法 rotation scheduling with unfolding (RSF)、rotation scheduling with tiling (RST) 和 rotation scheduling with parallelization (RSP)，不考慮暫存器 (accumulator/register) 的配置。第二個研究議題我們先針對 Motorola DSP56000 這顆數位訊號處理器的架構特性，提出指令排程法 rotation scheduling with spill codes predicting (RSSP)，涵蓋編譯過程的所有步驟。RSSP 的特色是在實際排程指令之前事先預測暫存器

滿溢 (accumulator spill) 發生的時機，並產生對應的 spill codes。接著我們提出指令排程法 rotation scheduling with spill codes avoiding (RSSA)，它是 RSSP 的延伸，可以適用於多種特性類似的架構。RSSA 同時將縮短排程長度和減少 spill codes 列為排程目標，也使用其他的機制解決 accumulator spill，不再預測其發生的時機。除此之外，我們定義一個虛擬架構模組 (hypothetical machine model)，配合 RSSA 深入探討不同系統資源數量改變時對排程結果造成的影響。最後在第三個研究議題中我們進一步延伸 RSSA，利用運算元分享 (operand sharing) 的方式，提出二個低功耗指令排程法 rotation scheduling with operand reutilization (RSOR) 和 rotation scheduling with exploiting operand reutilization (RSER)。RSOR 只在單一迴圈元素 (iteration) 內令運算元重複使用。RSER 則是設計一個迴圈轉換 (loop transformation) 機制，尋找在不同迴圈元素內指令共用運算元的情形，以增加運算元重複使用的機會。

在描述所有提出方法的特性之後，我們選擇數個數位訊號處理的應用程式來評估執行效能，分別使用排程長度、指令個數以及運算元重複使用次數等三個標準。另外我們也定義數學模組，用來計算迴圈轉換之後整體排程長度和運算元重複使用次數。初步評估，所有提出的指令排程法都能達到預期的效能。

# A Study on Effective Instruction Scheduling Methods for DSP Architecture

**Student: Yi-Hsuan Lee    Advisor: Prof. Cheng Chen**

**Department of Computer Science, National Chiao Tung University**

**Abstract**

As the multimedia and communication flourishing, many scientific and digital signal processing applications are developed. These applications are iterative and data-dominated, which are usually represented by uniform loops and characterized by a predominance of arithmetic instructions. A *digital signal processor* (*DSP*) is a special-purpose microprocessor designed to achieve high performance in digital signal processing applications, and commonly employs architecture with irregular data paths, multiple data memory banks, and heterogeneous register sets. Sufficient compiler support is apparently important to harvest benefits of this architecture. To optimize the throughput of such digital signal processing applications, we need to explore the embedded parallelism of a loop and generate fewer spill codes. As the portable system market grows rapidly, power becomes another critical constraint in the design specification. If we consider low power design at high-level synthesis, we can obtain much more effective power reduction with less cost and effort.

In this thesis we will focus on designing code generation methods to schedule uniform loops on DSP architecture with multiple data memory banks. The complete code generation process for this architecture must include several phases, and to consider more phases at the same time may lead more effective results due to their extremely data dependent. Our research contains three main issues. For this first issue

we design three efficient variable partition mechanisms and three corresponded methods *rotation scheduling with unfolding* (*RSF*), *rotation scheduling with tiling* (*RST*), and *rotation scheduling with parallelization* (*RSP*) without considering the accumulator/register assignment. In the second issue we first present method *rotation scheduling with spill codes predicting* (*RSSP*) focus on Motorola DSP5600 covering all code generation phases. The main feature of RSSP is to predict the occurrence of accumulator spills, and generate corresponding spill codes in advance. After that, we generalize RSSP to *rotation scheduling with spill codes avoiding* (*RSSA*), which can suit various DSPs with similar architectural features. The scheduling goal of RSSA is to achieve both shorter schedule length and fewer spill codes. Meanwhile, new mechanisms are designed for resolving accumulator spills instead of predicting their occurrences. Besides, we also evaluate RSSA on a defined hypothetical machine model, and deep study the influence of differing number of resources on the scheduling result. Finally, two energy-efficient code generation methods *rotation scheduling with operand reutilization* (*RSOR*) and *rotation scheduling with exploiting operand reutilization* (*RSER*) are proposed in our third issue. These two methods are extended from RSSA to further consider the *operand sharing* technique. In RSOR only the potential operand sharing within an original iteration is considered. In RSER we design a novel loop transformation mechanism to reconstruct the given loop, to find instructions with common operands hidden in different original iterations.

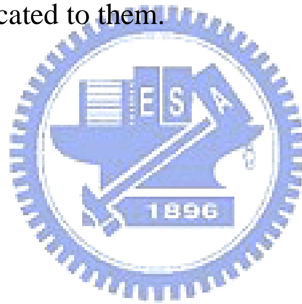
In addition to present detailed principles of all proposed methods, we select some MDFGs to evaluate their performances based on metrics schedule length, instruction count, and the number of operand reutilizations. We also design analytic models for every proposed method, which can calculate the overall scheduling length and the number of operand reutilizations of a reconstructed loop. Preliminary evaluations show that all proposed methods can achieve desirable results.

# Acknowledgements

I would like to express my sincere thanks to my advisor, Prof. Cheng Chen, for his supervision and advice. I appreciate the other members of my thesis committee for their time, support, and valuable comments.

There are many friends whom I wish to thank. My thanks to Dr. Der-Lin Pean, senior Lan-Mao Chung, senior Jiang-Long Wu, Ming-Lung Tsai, and Yi-Siou Lin for their encouragement during my initial process of doctor's degree. I also thank many delightful fellows, I feel happy and relaxed because of your presence. They help me in different ways during my stay at National Chiao Tung University.

Finally, I am grateful to my dearest family. Without their support, I can not finish this thesis. This thesis is dedicated to them.



# Contents

**Chinese Abstract……………………………………………………………..….i**

**English Abstract……………………………………………………………..iii**

**Acknowledgements………………………………………..………………….v**

**Contents………………………………………………………………………vi**

**List of Tables…………………………………………………………………ix**

**List of Figures……………………………………………………………..….xi**

**1 Introduction………………………………………………………………….1**

1.1 The Practicability of DSP……………………………………………………...1

1.2 The Power Constraint of DSP…………………………………………………2

1.3 Our Studies in this Thesis……………………………………………………...4

1.3.1 Variable Partition Mechanisms…………………………………………………4

1.3.2 Code Generation Methods for DSP with Multiple Data Memory Banks………6

1.3.3 Energy-efficient Code Generation Methods……………………………………7

1.4 Thesis Organization…………………………………………..……………..9

**2 Fundamental Background…………………………………………………10**

2.1 Program Model…………………………………………………………………10

2.2 Retiming Technique……………………………………………………………..11

2.3 Unimodular Transformations……………………………………………………14

2.4 Related Work……………………………………………………………………15

2.4.1 Retiming-based Instruction Scheduling Methods……………………………..15

2.4.2 Variable Partition Mechanisms………………………………………………..16

2.4.3 Code Generation Methods for DSP with Multiple Data Memory Banks……..17

2.4.4 Energy-efficient Code Generation Methods…………………………………..19

**3 Variable Partition Mechanisms……………………………………………22**

3.1 Flaws of RSVR……………………………………………………………….22
3.2 Rotation Scheduling with Unfolding (RSF) and Rotation Scheduling with Tiling (RST)………………………………………………………………………23
3.3 Rotation Scheduling with Parallelization (RSP)……………………………...28
3.4 Performance Evaluations……………………………………………………..31
3.4.1 Performance Studies of a Single Iteration……………………………………..31
3.4.2 Performance Studies of the Entire Retimed Loop……………………………33
3.4.3 Comparisons among RSF, RST, and RSP……………………………………..34
**4 Effective Code Generation Method for Motorola DSP56000…………………39**
4.1 Motorola DSP56000 Architecture……………………………………………39
4.2 Design Motivations…………………………………………………………...41
4.3 Rotation Scheduling with Spill Codes Predicting (RSSP)……………………...42
4.3.1 MDFG Construction…………………………………………………………..43
4.3.2 TDAG Construction…………………………………………………………...43
4.3.3 TDAG Modification…………………………………………………………...46
4.3.4 ALU Instruction Scheduling…………………………………………………..49
4.3.5 Other Instruction Scheduling…………………………………………………50
4.3.6 Initial Schedule Retiming……………………………………………………..52
4.4 Performance Evaluations……………………………………………………..53
**5 Effective Generalized Code Generation Method……………………………57**
5.1 Hypothetical Machine Model………………………………………………...57
5.2 Design Motivations…………………………………………………………...60
5.3 Rotation Scheduling with Spill Codes Avoiding (RSSA)………………………62
5.3.1 Instruction Scheduling (I)……………………………………………………..62
5.3.2 Instruction Scheduling (II)…………………………………………………….67
5.3.3 Initial Schedule Retiming……………………………………………………..69

5.4 Applying to Real DSP Families……………………………………………………69
5.4.1 Data Memory Bank………………………………………………………………70
5.4.2 Function Unit……………………………………………………………………...70
5.4.3 Register Set………………………………………………………………………71
5.5 Performance Evaluations……………………………………………………………..72
5.5.1 Comparison with Previous Work……………………………………………...72
5.5.2 The Influence of Resources…………………………………………………...74
5.5.3 Brief Summaries…………………………………………………………………80
**6 Energy-efficient Code Generation Methods…………………………………...83**
6.1 Brief Analyses of RSSA……………………………………………………………...83
6.2 Rotation Scheduling with Operand Reutilization (RSOR)……………………...84
6.2.1 Detailed Algorithms of RSOR…………………………………………………...84
6.2.2 Comparisons between RSSA and RSOR……………………………………...87
6.3 Rotation Scheduling with Exploiting Operand Reutilization (RSER)…………..89
6.3.1 MDFG Reconstruction Mechanism…………………………………………...90
6.3.2 Detailed Algorithms of RSER…………………………………………………95
6.3.3 The Difference between Proposed Methods and Other Methods……………..96
6.4 Performance Evaluations……………………………………………………………..98
**7 Conclusions and Future Work…………………………………………………105**
7.1 Conclusions………………………………………………………………………105
7.2 Future Work………………………………………………………………………109
**Reference…………………………………………………………………...…111**
**Appendix A The Analytic Model for RSVR, RSF, RST, and RSP………….....117**
**Appendix B The Analytic Model for RSOR and RSER…………………….....122**
**Author's Publication List………………………………………………………131**
**Vita……………………………………………………………………………133**

# List of Tables

Table 3.1 Experimental results (1 function unit)(schedule length, retiming depth)..32

Table 3.2 Experimental results (2 function unit)(schedule length, retiming depth)..32

Table 3.3 Variables defined in the analytic model……………………………….34

Table 4.1 Experimental results for a single iteration in the repetitive pattern……..53

Table 5.1 Architectural features of some popular DSPs…………………………...59

Table 5.2 Variables defined for solving accumulator/register spills……………….65

Table 5.3 Schedule lengths obtained by different code generation algorithms…….73

Table 5.4 Number of operations really executed in an iteration obtained by different code generation algorithms……………………………………………...74

Table 5.5 Characteristics of selected TDAGs……………………………………...75

Table 5.6 Experimental results, with target architectures contains different number of accumulators……………………………………………………………76

Table 5.7 Experimental results, with target architectures contains different number of registers………………………………………………………………76

Table 5.8 Experimental results, with target architectures contains different number of function units…………………………………………………………77

Table 5.9 Experimental results, with target architectures contains different number of function units…………………………………………………………78

Table 5.10 Experimental results, with target architectures contains different number of data memory banks……………………………………………………...79

Table 6.1 Average current required for each instruction [20]……………………87

Table 6.2 The comparison between RSOR and RSSA (the number of OPRs)…….88

Table 6.3 The comparison between RSOR and RSSA (under Motorola DSP56000 architecture)……………………………………………………………...89

Table 6.4 The number of OPRs obtained by different scheduling methods……......98

Table 6.5 The comparison among four methods (under Motorola DSP56000)…100

Table 6.6 Definitions of variables used in the analytic model……………………102

Table A.1 Variables defined in the analytic model………………………………...118

Table B.1 Definitions of variables used in the analytic model……………………123

# Lists of Figures


Figure 2.1 The MDFG example. (a) Nested loops in C code, (b) corresponding MDFG, (c) node types……………………………………….…………11

Figure 2.2 Retiming example. (a) Retimed MDFG of Figure 2.1(a), (b) schedule before retiming, (c) schedule after retiming…………………………...12

Figure 2.3 (a) Original iteration space, (b)(c) changed iteration spaces…………..13

Figure 3.1 (a) MDFG fragment, (b) scheduling result of RSVR…………………23

Figure 3.2 Variable partition results of MDFG in Figure 2.1(b). (a) Based on rightmost indices, (b) based on leftmost indices……………………....24

Figure 3.3 Two consecutive iterations of nested loop in Figure 2.1(a)…………....25

Figure 3.4 The entire scheduling steps of RSF…………………………………....25

Figure 3.5 The entire scheduling steps of RST…………………………………....26

Figure 3.6 (a) Unfolding MDFG of Figure 2.1(b), (b) tiled MDFG of Figure 2.1(b)……………………………………………………………………...26

Figure 3.7 Scheduling results of Figure 2.1(b). (a) RSVR, (b) RSF, (c) RST….…26

Figure 3.8 (a) Unfolded nested loop in canonical form, (b) two consecutive iterations…………………………………………………………………27

Figure 3.9 (a) Tiled nested loop in canonical form, (b) two consecutive iterations 28

Figure 3.10 The entire scheduling steps of RSP……………………………………29

Figure 3.11 Loop parallelization algorithm……………………………………...29

Figure 3.12 (a) The parallelized MDFG of Figure 2.1(b), (b) scheduling result of Figure 2.1(b) using RSP………………………………………………30

Figure 3.13 The unfolded MDFG of Figure 3.12(b)……………………………..31

Figure 3.14 Overall schedule lengths of DSP applications (1 function unit, 2 memory banks)……………………………………………………………..35

Figure 3.15 Overall schedule lengths of DSP applications (1 function unit, 2 memory banks)……………………………………………………………..36
Figure 4.1 Data ALU block diagram. (a) DSP56000/DSP56001, (b) DSP56300 family……………………………………………………………..40
Figure 4.2 Motorola DSP56000 architecture…………………………………...41
Figure 4.3 The entire scheduling steps of RSSP………………………………..43
Figure 4.4 The TDAG constructing algorithm…………………………………44
Figure 4.5 (a) Two cases of removing memory accesses, (b) TDAG of MDFG in Figure 2.1(b)………………………………………………………...45
Figure 4.6 (a) A TDAG fragment, (b) after inserting the register transfer $v_k$……...46
Figure 4.7 The register transfer inserting algorithm……………………………46
Figure 4.8 The $G_{op}$ and $G_{pr}$ constructing algorithm……………………………47
Figure 4.9 The *Mark_Edge* algorithm…………………………………………..47
Figure 4.10 The *Check_Cycle* algorithm………………………………………...48
Figure 4.11 Two $G_{op}$ fragments with accumulator spill…………………………48
Figure 4.12 The memory access inserting algorithm……………………………49
Figure 4.13 Scheduling steps of RSSA. (a) An TDAG example, (b) ALU instruction only, (c) initial scheduling result, (d) retimed scheduling result………50
Figure 4.14 Overall schedule lengths of DSP applications………………………55
Figure 4.15 Overall schedule lengths of DSP applications………………………56
Figure 5.1 An example of code compaction. (a) Uncompacted code, (b) compacted code, (c)(d) two scheduling results after resource assignment………...61
Figure 5.2 The entire scheduling steps of RSSA……………………………….63
Figure 5.3 The $G_{op}$ example. (a) TDAG, (b) corresponding $G_{op}$…………………..64
Figure 5.4(a) $G_{op}$ nodes only scheduling result of Figure 5.3(a), unlimited resource 64
Figure 5.4(b) $G_{op}$ nodes only scheduling result of Figure 5.3(a), with unlimited

number of registers……………………………………………………….66
Figure 5.4(c) $G_{op}$ nodes only scheduling result of Figure 5.3(a), without accumulator spills…………………………………………………………………67
Figure 5.4(d) The initial scheduling result of $G_t$ of Figure 5.3(a)……………………68
Figure 5.4(e) The retimed scheduling result of $G_t$ of Figure 5.3(a)………………….70
Figure 6.1 An example of TDAG and sharing sets………………………………..85
Figure 6.2 Scheduling results of Figure 6.1. (a) RSSA, (b) RSOR……………….85
Figure 6.3 The overall scheduling algorithm of RSOR…………………………...86
Figure 6.4 The MDFG example…………………………………………………...91
Figure 6.5 The MDFG reconstructing algorithm………………………………….92
Figure 6.6 An example used to illustrate steps of RSER………………………….94
Figure 6.7 The overall scheduling algorithm of RSER……………………………95
Figure 6.8 The corresponding TDAG of (a) Figure 6.6(a), (b) Figure 6.6(d)……..96
Figure 6.9 Scheduling results of Figure 6.6(a). (a) RSOR, (b) RSER…………….96
Figure 6.10 Experimental results of DSP applications (1 function unit, overall schedule length)……………………………………………………103
Figure 6.11 Experimental results of DSP applications (2 function unit, overall schedule length)……………………………………………………104
Figure A.1 Iteration spaces of a loop with depth one. (a) Original, (b) applying RSVR, (c) applying RSF………………………………………………..118
Figure A.2 Iteration spaces of a loop with depth two. (a) Original, (b) applying RSVR, (c) applying RSF, (d) applying RST…………………………119
Figure A.3 Iteration spaces of a loop with depth two. (a) Original, (b)(c) applying RSP…………………………………………………………………...121
Figure B.1 Iteration spaces of a loop with depth one. (a) Original, (b) applying RSOR(RSVR), (c) applying RSOR(RSF)……………………………124

Figure B.2 Iteration spaces of a loop with depth one. (a) Original, (b) applying RSER(RSVR), (c) applying RSER(RSF)…………………………….124

Figure B.3 Iteration spaces of a loop with depth two. (a) Original, (b) applying RSOR(RSVR), (c) applying RSOR(RSF), (d) applying RSOR (RST)126

Figure B.4 Iteration spaces of a loop with depth two. (a) Applying RSER(RSVR), formula (B.8), (b) applying RSER(RSVR), formula (B.9)…………..127

Figure B.5 Iteration spaces of a loop with depth two. (a) Applying RSER(RSF), formula (B.10), (b) applying RSER(RSF), formula (B.11)…………..129

Figure B.6 Iteration spaces of a loop with depth two. (a) Applying RSER(RST), formula (B.12), (b) applying RSER(RST), formula (B.13)…………..130

# Chapter1. Introduction

## 1.1 The Practicability of DSP

Most scientific and digital signal processing applications, such as fluid dynamics, weather forecasting, image processing, video compression, and speech recognition are iterative and usually represented by uniform nested loops [1-5]. All these applications belong to data-dominated category, which are characterized by a predominance of arithmetic instructions and an absence of control-flow within the data path [5]. A *digital signal processor* (*DSP*) is a special-purpose microprocessor that is designed and produced to better match DSP applications [3, 6-8]. Unlike general-purpose microprocessors, the DSP design is based on the Harvard architecture, and often includes several independent function units those are capable of operating in parallel [3, 7-8]. In order to meet ever-increasing demands for higher performance and stringent power requirement, such DSPs commonly employ architectures with irregular data paths, heterogeneous register sets, and multiple data memory banks [9]. For the data path, this architecture has multiple small register files dedicated to different sets of function unit instead of a large number of centralized homogeneous registers. In addition, because multiple data memory banks are connected through independent data buses, variables can be partitioned into separate banks and accessed simultaneously. These architectural features are supported by some embedded DSP families, such as Motorola DSP56000 [10], Analog Device ADSP2100 [11], NEC uPD77016 [12], and Texas Instruments TMS320C6000 [13].

Although parallel access, which is enabled by multi-bank memory, is useful to explore the potential of higher memory bandwidth, it gives rise to the problem of how to partition the variables into multiple data memory banks [6, 9, 14-20]. Similarly, using heterogeneous register sets can decrease the architectural complexity but

increase the difficulty of deciding which register set to use for a certain instruction [9, 17, 21-23]. It is well known that compilation techniques for general-purpose microprocessor do not adapt well to the irregularities of DSP. Therefore, to harvest the benefits provided by DSPs with irregular architectural features, adequate compiler support is obviously essential [3, 8].

Many researches seek to design code generation methods for specific DSP architectures to fully use their features. The complete code generation process for DSP with multiple data memory banks must include five phases: *intermediate representation*, *code compaction*, *instruction scheduling*, *memory bank assignment* (or *variable partition*), and *accumulator/register assignment* [17]. These phases can be performed individually in various sequences because they are logically independent. Meanwhile, because they are extremely data dependent, considering more phases at the same time may lead more effective results. Since nested loops are the most time-critical section in such DSP applications, their execution time will dominate the entire computational performance. To optimize the execution rate of such applications we need to explore the embedded parallelism of a loop. Moreover, due to strict resource constraints of the DSP architecture, accumulator/register spills will supposedly occur very often compared to general-purpose microprocessor. If more spill codes are added, not only the schedule length may be lengthened, but also consumes more power to execute those additional instructions. That is, in addition to increase the instruction-level parallelism, how to avoid generating too many spill codes is also an important issue of designing the code generation method for DSP architecture.

## 1.2 The Power Constraint of DSP

Until 1980's, throughput and latency are two important factors used to determine the quality of an embedded system. In 1990's, as the portable system such as cellular

phone or portable electronic devices grows rapidly, power consumption becomes another important constraint in the design specification [24]. Because low power is now one of the major concerns in system design, this has forced to analyze and optimize power in all components of a system [25]. Most research to date on power minimization in DSPs is focused on hardware solution. However, if we consider low power design at higher levels of abstractions, we can apply various transformation techniques to system design with wider view and obtain much more effective power reduction with less cost and effort [24, 26].

High-level synthesis techniques for low power have mostly targeted data-dominated designs [5]. In a data-dominated application specific circuit such as DSP, it is the power consumed in the data path, including function units, registers, and interconnections, that accounts for a large fraction of the overall power budget [24]. Power consumption is mainly considered in the function units, among units that compose a data path [4, 27]. As shown in [4], authors present that function units account for over 80% of the total data path power, if the data path contains $n$ function units, $4n$ registers, and $8n$ multiplexers. Authors of [24, 28] further show that if the overall system is divided into components including data path, clock, and controller, function units will contribute about 40%~60% power to the overall system. Therefore, if we can reduce power consumed by function units, the entire power consumption of the system can be reasonably decreased.

In most cases, the power consumed by a resource mainly depends on the input switching activity induced by the data being stored or processed [29]. For a function unit, the power consumption will be reduced by reducing the switching activity involving its input signals [28]. Many researches on power minimization in high-level synthesis attempt to reduce the input activity of function units. *Operand sharing* is one of these techniques, which binds one identical function unit to more than two

instructions containing at least one common operand, and any instruction without a common operand does not intervene between these instructions [28]. As presented in [27], the average power consumption of a multiplication (or an addition) when one of its operands remains unchanged with respect to the previous instruction is 35% (or 25%) less than when both operands change. Therefore, to increase the potential for a function unit to reuse an operand, the average power consumption of the function unit is dramatically lower. Operand sharing also assists in reducing the number of memory accesses, which tends to prevent the limited number of memory ports from increasing system latency. Furthermore, as shown in [28], because the power consumed by components other than function units are little increase or no increase at all after applying operand sharing, operand sharing is obviously an appropriate technique in low power design.

## 1.3 Our Studies in this Thesis

Many DSP applications usually contain repetitive groups of operations, which are easily represented by uniform loops and modeled by *multi-dimensional data flow graph* (*MDFG*) [2-3]. From above descriptions, clearly that the code generation plays an important role to harvest benefits provided by irregular DSP architectures. With appropriate instruction ordering sequences, we can obtain scheduling results with shorter schedule length, smaller codes size, or less power consumption. In this thesis we focus on designing code generation methods to schedule uniform loops on DSP with multiple data memory banks. Our three study issues are presented as follows.

### 1.3.1 Variable Partition Mechanisms

For the architecture with multiple data memory banks, the performance gain strongly depends on variable partition and instruction scheduling techniques. Hence,

our first issue is about variable partition. At first we analyze a related method *rotation scheduling with variable repartitioning* (*RSVR*) [30] in some detail. We claim that although RSVR is effective, it uses complex mechanisms to partition variables initially and repartition them during instruction scheduling. Note that a variable in MDFG indicates an array not just a single scalar. Therefore, we present three efficient variable partition mechanisms directly according to their array indices. After transforming the given MDFG by appropriate techniques such as *unfolding* [31], *tiling* [32], and *unimodular transformations* [33], we apply the *multi-dimensional rotation scheduling* [34-35] to schedule instructions. Three code generation methods named *rotation scheduling with unfolding* (*RSF*), *rotation scheduling with tiling* (*RST*), and *rotation scheduling with parallelization* (*RSP*) are proposed corresponded to different variable partition mechanisms [14, 36]. Without repartitioning variables during instruction scheduling, our three methods are obviously efficient compared to RSVR. Moreover, we also define an analytic model to calculate the overall schedule length of an entire retimed loop. Several MDFGs represented DSP applications are selected for performance evaluations. From evaluation results, our methods RSF, RST, and RSP can achieve effective results compared to RSVR, for both a single repetitive iteration and the entire retimed loop. Moreover, the enlarged graph gives a more global view of the data dependencies, which is beneficial for exploring the instruction-level parallelism between different iterations. As for the effectiveness among methods RSF, RST, and RSP, the answer will depend on the topology and loop-carried dependencies of the given nested loop. We also list comparisons among three proposed methods and suggest which method is suitable based on loop-carried dependencies the given MDFG has. Variable partition mechanisms proposed in RSVR and our three methods will be applied in subsequent several studies.

### 1.3.2 Code Generation Methods for DSP with Multiple Data Memory Banks

In section 1.1 we have introduced that the complete code generation process for DSP with multiple data memory banks must include five phases. RSF, RST, and RSP have covered all except the accumulator/register assignment phase. Besides, above methods directly use data memory to store and reload operands, so many unnecessary memory accesses may be generated to degrade the performance. Since considering more phases in a code generation method may lead more effective results, we will design a new method to include accumulator/register assignment and further improve overall performance. The proposed method *rotation scheduling with spill codes predicting* (*RSSP*) is focus on Motorola DSP56000 [37]. Its main feature is to predict the occurrence of accumulator/register spills in advance, and schedule corresponding spill codes in parallel with other instructions. In addition, we also define a *translated data acyclic graph* (*TDAG*) constructed from the given MDFG, in order to remove possible unnecessary memory accesses. We still use selected MDFGs and the analytic model proposed above to evaluate RSSP. Apparently that RSSP outperforms all of RSF, RST, and RSP, because RSSP schedules instructions based on the TDAG which contains less instructions than the MDFG. Comparing to other related studies, RSSP still has advantages of shorter schedule length, for both a single repetitive iteration and the entire retimed loop.

RSSP looks quite effective and efficient, but it is not scalable and specifically designed for Motorola DSP56000. Hence, we will generalize it to suit various DSPs with similar architectural features, and propose another method *rotation scheduling with spill codes avoiding* (*RSSA*) [38]. The scheduling goal of RSSA is to achieve shorter schedule length and fewer spill codes. In RSSA we design another mechanism to resolve accumulator spills instead of to predict their occurrences, because the predicting results become inaccurate easily when the target architecture is no longer

specific. Moreover, we also integrate these mechanisms into instruction scheduling phase to make RSSA more efficient. We evaluate RSSA according to two metrics schedule length and instruction count at the same time. Suppose the target architecture equals to the Motorola DSP56000, our RSSA usually achieves the shortest schedule length and considerably fewer spill codes compared to other related studies. The main reason is that RSSA can fully utilize system resources and insert spill codes only when required. On the other hand, in addition to design effective code generation method, increasing the number of resources is essentially a more direct way to achieve effective scheduling results. Hence, we also define a parameterized machine model to simulate architectures with different number of resources. After evaluating MDFGs using RSSA on this hypothetical machine model, the influence of differing number of resources on the scheduling results is further deep studied in this thesis. Finally, we describe that with minor modifications, our hypothetical machine model and RSSA is capable for applying to DSP families such as Motorola DSP56000 [10], Analog Device ADSP2100 [11], NEC uPD77016 [12], and Texas Instruments TMS320C6000 [13]. This indicates that the proposed machine model and code generation method have enough flexibility, which are suitable to DSPs with various architectural features.

### 1.3.3 Energy-efficient Code Generation Methods

As mentioned in section 1.2, low power consumption becomes another important constraint in the DSP design specification in addition to shorter schedule length and less spill codes. To increase the potential for a function unit to reuse an operand is an appropriate way, because the power consumed by function units will be dramatically lower. Therefore, in the third issue of this thesis, we will propose energy-efficient code generation methods based on the operand sharing technique. At first we analyze

RSSA in view of low power consumption. Then, *rotation scheduling with operand reutilization* (*RSOR*) is proposed by integrating the operand sharing technique into RSSA, where the original features of RSSA are all retained. In RSOR we add a mechanism to group ALU instructions sharing the same operand into a *sharing set*. Then, the same scheduling steps used in RSSA are applied, and instructions belong to a sharing set are restrictively scheduled at consecutive time steps to reuse operands. According to preliminary evaluations, because we restrict the execution sequence of some ALU instructions to achieve operand reusing, schedules generated by RSOR may be slightly longer and with more instruction count compared to RSSA. Unfortunately, common operands are not encountered very frequently in real designs, resulting in few opportunities of operand sharing, and hence insignificant power reduction [29]. Nevertheless, instructions with common operands may be hidden inside the original MDFG, which can be generated using some loop transformation techniques. Thus, we proposed another method *rotation scheduling with exploiting operand reutilization* (*RSER*), which is extended from RSOR and aimed to further explore potential operand sharing between different iterations. In RSER we define an *exploitable sharing set* to group load variable instructions reference the same array element in different iterations. An MDFG reconstruction algorithm is also designed based on the *retiming* [39] technique, to concentrate instructions in a same exploitable sharing set into the same iteration. Then, RSOR is applied to schedule the reconstructed MDFG, so operand sharing within an iteration and existing in different iterations can be both explored in RSER. Metrics including schedule length, instruction count, the number of operands been reused, and information provided from [20] are used to evaluate RSOR and RSER. Besides, we extend the analytic model defined before, to calculate the overall schedule length and the number of operands been reused for the entire retimed loop. From evaluation results, we find that

both RSOR and RSER can successfully explore operand sharing within an iteration. Using RSER further can achieve more number of operands been reused, which indicates that exploiting the operand sharing in different iterations is beneficial for energy-efficient instruction scheduling. On the other hand, because some ALU instructions are restrictively scheduled at consecutive time steps to achieve operand reusing, using RSOR and RSER may generate longer schedules for a single repetitive iteration. However, the overall schedule lengths obtained by RSOR and RSER are still better compared to related studies, because they can effectively explore the instruction-level parallelism between successive iterations. As for the instruction count, the proposed two methods require quite fewer spill codes for a repetitive iteration, but RSER will generate considerable prologue and epilogue codes. That is, if the instruction count is taken as the evaluation metric, RSER will perform poorly compared to related methods.

## 1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 surveys the fundamental background and related work. In chapter 3 we focus on variable partition mechanisms, and introduce three proposed methods RSF, RST, and RSP. Chapter 4 contains an overview of the Motorola DSP56000 architecture, and principles and algorithms of proposed method RSSP are also included. In chapter 5, we present our hypothetical machine model and the general method RSSA, and describe their flexibilities to apply to other real DSP families. Two energy-efficient code generation methods RSOR and RSER extended from RSSA are introduced in chapter 6. Finally, in chapter 7 we list conclusions and plans for future work.

# Chapter 2. Fundamental Background

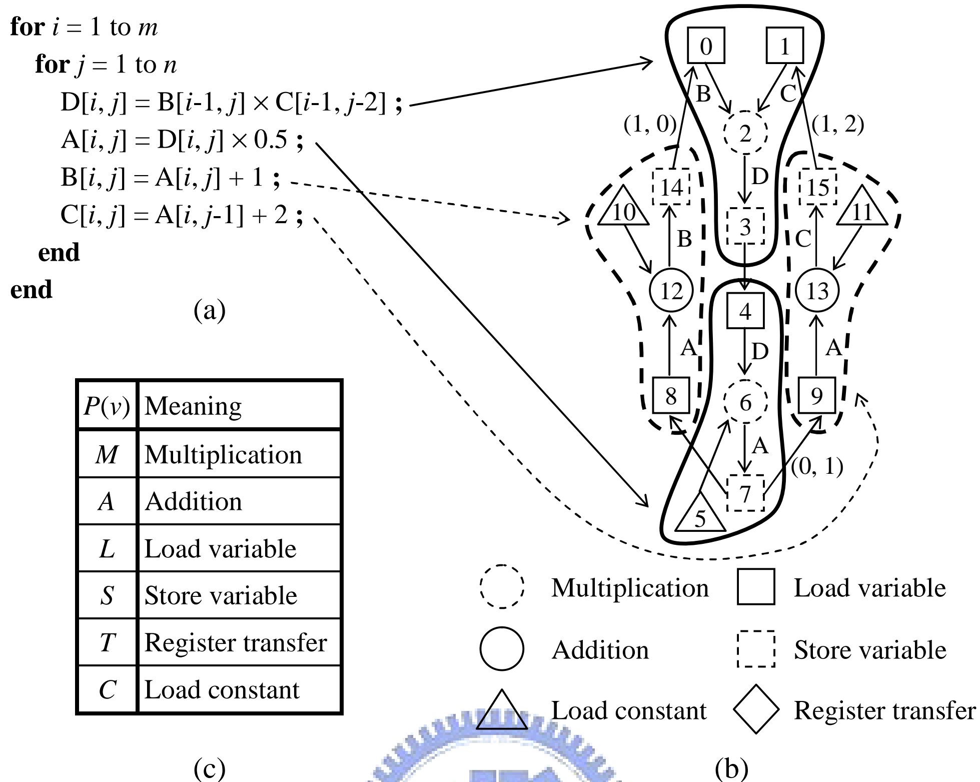
In this chapter, we first model the given problem and survey some fundamentals. Then, we introduce two basic techniques retiming and unimodular transformations widely used in instruction scheduling. After that, related work of our studies in this thesis is presented.

## 2.1 Program Model [37-38]

Because most scientific and digital signal processing applications usually contain repetitive groups of operations, they can be easily represented by uniform nested loops. A *multi-dimensional data flow graph* (*MDFG*) is commonly used to model uniform nested loops. We define the MDFG to be the same as in [37-38], which is slightly different from previous studies [14, 30].

**Definition 2.1** A MDFG $G = (V, E, X, d, P)$ is a node-weighted and edge-weighted direct graph, where $V$ is the set of computation nodes; $E \subseteq V \times V$ is the edge set that defines the precedence relations; $X(e)$ represents the variable accessed by an edge $e$; $d(e)$ is a function from $E$ to $Z^n$ representing the multi-dimensional delays between two nodes, where $n$ is the number of dimensions; and $P(v)$ represents the node type (see Figure 2.1(c)).

Figure 2.1 shows an example of a nested loop and its corresponding MDFG. Nodes in the MDFG include ALU instructions (multiplications and additions), memory accesses (load/store variables and load constants), and register transfers. Note that an edge, $e$, that does not involve a memory access does not have a label $X(e)$. An MDFG is *realizable* if there exists a schedule vector $s$, such that $s \cdot d \geq 0$, where $d$ are loop-carried dependencies. A *schedule vector* $s$ is the normal vector for a set of parallel equitemporal hyperplanes that define a sequence of execution [40]. An



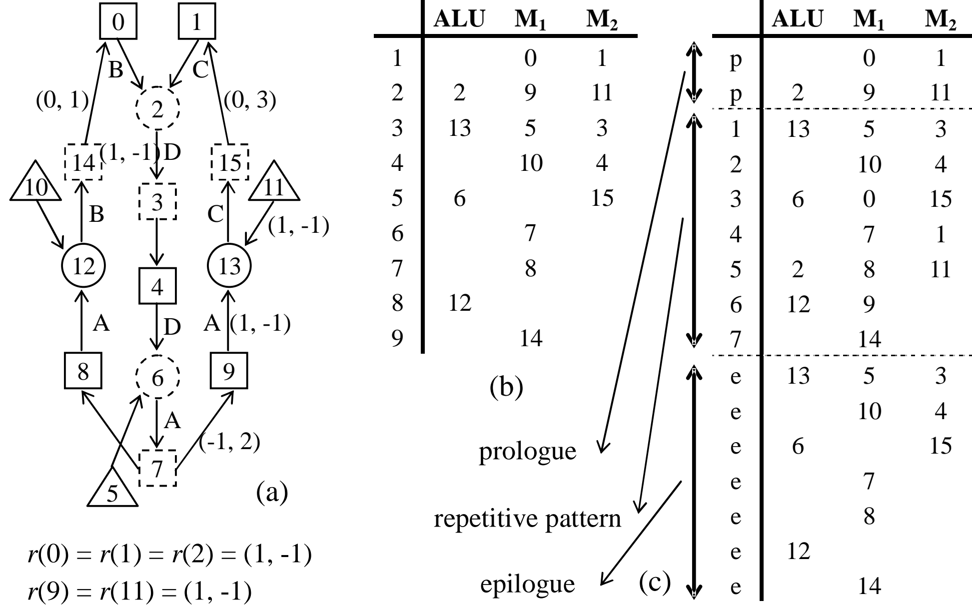

| $P(v)$ | Meaning |
|---|---|
| $M$ | Multiplication |
| $A$ | Addition |
| $L$ | Load variable |
| $S$ | Store variable |
| $T$ | Register transfer |
| $C$ | Load constant |

Figure 2.1. The MDFG example. (a) Nested loop in C code, (b) corresponding MDFG, (c) node types.

*iteration* is equivalent to the execution of each node in $V$ exactly once. The period during which all nodes in an iteration are executed, according to data dependencies and without resource constraints, is called a *cycle period.* It is also the maximum execution time among paths that have no delay, which will dominate the entire execution time of a nested loop. Note that many MDFGs can represent a single DSP application, depending on its representation by nested loops.

## 2.2 Retiming Technique [39]

*Retiming* is a popular technique that reassigns delays to enhance execution performance for a circuit. For a loop, retiming is a loop transformation technique that can be used to increase the throughput and improve the utilization of resources, by introducing partial overlap between the execution time of successive iterations. The




(b)

| | ALU | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | | 0 | 1 |
| 2 | 2 | 9 | 11 |
| 3 | 13 | 5 | 3 |
| 4 | | 10 | 4 |
| 5 | 6 | | 15 |
| 6 | | 7 | |
| 7 | | 8 | |
| 8 | 12 | | |
| 9 | | 14 | |

(c)

| | ALU | $M_1$ | $M_2$ |
|---|---|---|---|
| p | | 0 | 1 |
| p | 2 | 9 | 11 |
| 1 | 13 | 5 | 3 |
| 2 | | 10 | 4 |
| 3 | 6 | 0 | 15 |
| 4 | | 7 | 1 |
| 5 | 2 | 8 | 11 |
| 6 | 12 | 9 | |
| 7 | | 14 | |
| e | 13 | 5 | 3 |
| e | | 10 | 4 |
| e | 6 | | 15 |
| e | | 7 | |
| e | | 8 | |
| e | 12 | | |
| e | | 14 | |

Figure 2.2. Retiming example. (a) Retimed MDFG of Figure 2.1(a), (b) schedule before retiming, (c) schedule after retiming.

*retiming vector* $r(u)$, a function from $V$ to $Z^n$, represents the offset between the original iteration and that after retiming. A new MDFG $G_r = (V, E, X, d_r, P)$ is created after applying $r$, such that each iteration still has one execution of each node. Delay vectors will be changed accordingly to preserve the original data dependencies. Definitions and properties of retiming are shown below.

**Definition 2.2** Given any MDFG $G = (V, E, X, d, P)$, retiming function $r$, and retimed MDFG $G_r = (V, E, X, d_r, P)$, we define the retimed delay vector for every edge, path, and cycle, respectively, by:

(a) $d_r(e) = d(e) + r(u) - r(v)$ for every edge $u \xrightarrow{e} v$, $u, v \in V$ and $e \in E$.

(b) $d_r(p) = d(p) + r(u) - r(v)$ for every path $u \xrightarrow{p} v$, $u, v \in V$ and $p \in G$.

(c) $d_r(l) = d(l)$ for any cycle $l \in G$.

Based on above definition, MDFGs $G$ and $G_r$ are logically equivalent, and the only difference between them is the delay vectors. Figure 2.2 shows an example of retiming technique. Figure 2.2(a) is the retimed MDFG of Figure 2.1(b), and Figure 2.2(b)(c) list schedules before and after retiming respectively. A *prologue* is the

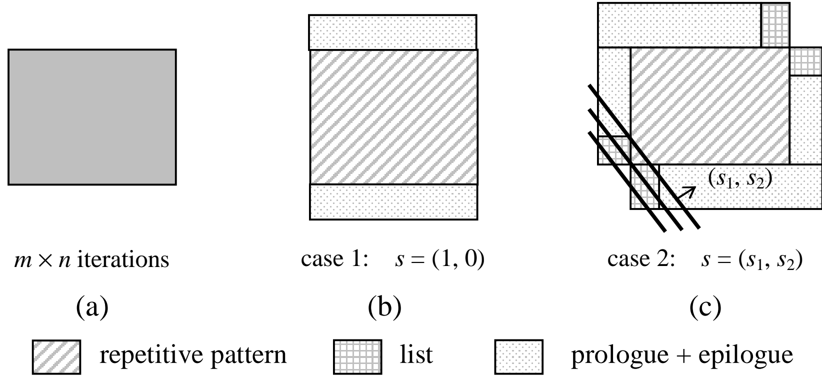



Figure 2.3. (a) Original iteration space, (b)(c) changed iteration spaces.

instruction set that must be executed to provide necessary data for the iterative process. An *epilogue* is the complementary set that will be executed to complete the process. If the nested loop contains sufficient iterations, the time required for prologue and epilogue are negligible.

Because applying retiming technique will change delay vectors of a realizable MDFG $G$, we must guarantee the retimed MDFG $G_r$ is still realizable. As mentioned in chapter 2.1, a realizable MDFG $G$ must have a feasible schedule vector $s$. In [34], it indicates that if we retime MDFG $G$ with a *retiming base r orthogonal* to $s$ ($s \perp r$), the retimed MDFG $G_r$ is definitely realizable. The feasible retiming base is not unique for a given MDFG, but in [34] it doesn't propose how to select a best one. In our early study [41], we analyze the relationship between the selection of schedule vector and the change of iteration space in some detail. From analyzing results we find that the overall schedule length is strongly dependent on which feasible schedule vector been selected, especially for nested loops with depth greater than one. If an unsuitable schedule vector is used, the time required to execute prologue and epilogue will occupy considerable part of the overall schedule length. We take a nested loop with depth two as an example. Figure 2.3 shows two cases of modified iteration space after applying retiming technique using different retiming bases. In [41] we prove that the overall schedule length of case 1 is always shorter than or equal to that of case 2,

which implies that $s = (1, 0)$ will be the best selection if it is feasible. Therefore, in [41] we propose a simple algorithm to select the best schedule vector for a given MDFG, which will achieve minimum overall schedule length after applying the retiming technique. We also list a formula to calculate the overall schedule length of a retimed nested loop. This formula will be used to evaluate code generation methods proposed in this thesis.

## 2.3 Unimodular Transformations [33]

Loop transformation is one of basic techniques for parallel compiler design. It changes the execution sequence of iterations to achieve higher degree of parallelism. *Unimodular transformations* technique unifies loop *permutation*, *skewing*, and *reversal*, and models them as elementary matrix transformations. All combinations of these loop transformations can simply be represented as products of the elementary transformation matrices.

Although unimodular transformations technique is one of the most important techniques used to parallelize uniform nested loops, it doesn't explain how to use its transformations. In [42] we propose a simple algorithm to parallelize the inner loop of a uniform nested loop with depth two. Note that the transformation matrix to parallelize a nested loop is not unique, and our algorithm can obtain one with minimum skew factor. For a given MDFG $G$, a new MDFG $G_p$ is created after applying loop parallelization. $G$ and $G_p$ are still logically equivalent, and the only difference between them is the delay vectors, just like applying retiming technique. Some formulas are listed in [42] to calculate the overall schedule length of a parallelized nested loop with depth two. These formulas will be modified further to evaluate one of code generation method proposed in this thesis.

### 2.4 Related Work

In this section we survey some related work of our studies. The content of this section is divided into four parts: retiming-based instruction scheduling methods, variable partition mechanisms, code generation methods for DSP architecture with multiple data memory banks, and energy-efficient code generation methods. Some related studies of each part are introduced in the following subsections.

#### 2.4.1 Retiming-based Instruction Scheduling Methods [34-35, 43]

Since retiming technique is useful for generating compact schedules, many instruction scheduling methods are designed based on it to achieve shorter schedule length. Among them, *rotation scheduling* [43] and *multi-dimensional rotation scheduling* [34-35] are two effective methods used to schedule MDFG with one or more than one dimensions, respectively. Both these methods contain two main steps. First they simply generate an initial schedule using the list scheduling method under resource constraints. Then instructions scheduled at the first time step are moved to the prologue, and their copies originally resided in the next iteration are rescheduled without violating resource constraints and data dependencies. This step is usually called *rotation phase*. Corresponding to the given MDFG, the action of rotation is essentially equivalent to retime nodes scheduled at the first time step. For an one-dimensional MDFG nodes are always retimed with retiming vector $r(u) = 1$. As for multi-dimensional MDFG, it must select a feasible retiming base $r$ as the retiming vector. After iteratively applying the rotation phase, a more compact schedule, also with higher throughput, can be obtained. Because these two instruction scheduling methods are really effective and efficient, in this thesis we choose them as the basis to design our own methods.

### 2.4.2 Variable Partition Mechanisms [15-16, 30]

As mentioned in chapter 1, appropriately partition and allocate variables is facilitated to generate more compact schedule in DSP architecture with multiple data memory banks. If two variables may be accessed in parallel, they should be allocated to different data memory banks. Some researches focus on designing variable partition mechanisms which try to evenly distribute memory accesses and explore the potential of higher memory bandwidth. Authors of [15] construct an *interference graph* (*IG*) to represent the parallelism available in load instructions for every basic block, and then partition it to determine the allocation of global variables. Two different IG partition heuristics proposed in [15] are based on the same idea: variables will be given higher priority to be stored to different data memory banks if they may be accessed in parallel in a deeper loop. Strictly speaking this mechanism is not accurate enough, because the IG cannot exploit the potential parallelism of memory accesses that reference values produced in different iterations [16]. Therefore, authors of [16] propose another mechanism to recover this flaw by globally constructing the IG for entire functions, and use an *integer linear programming* approach instead of a heuristic to partition variables.

Unlike above methods only focus on variable partition, *rotation scheduling with variable repartitioning* (*RSVR*) is designed to resolve both instruction scheduling and variable partition problems [30]. RSVR is modified from rotation scheduling, which considers multiple memory modules while generating a schedule. For a given MDFG, RSVR constructs a corresponding *variable independence graph* (*VIG*) to expose all parallel memory accesses. Basically the purpose of constructing VIG is similar as constructing IG in [15-16]. But in RSVR it uses more accurate information to assign edge weights of VIG, so it can achieve better variable partition results. After allocating variables, RSVR applies the same steps as rotation scheduling to schedule instructions.

Besides, when the schedule length cannot be improved in a rotation phase, RSVR will try to repartition variables to shorten the schedule length. In this thesis we will take variable partition as our first study issue. Detailed descriptions and our proposed mechanisms will be presented in chapter 3.

### 2.4.3 Code Generation Methods for DSP with Multiple Data Memory Banks [9, 17-23, 44]

A complete code generation process for DSP with multiple data memory banks must include five phases: *intermediate representation*, *code compaction*, *instruction scheduling*, *memory bank assignment* (or *variable partition*), and *accumulator/ register* assignment [17]. These five phases can be performed in various sequences due to their logically independent, or be simultaneously considered because they are extreme data dependences. In previous subsection we have listed some methods focus on the variable partition phase. For heterogeneous register sets, authors of [21-23] present specific register allocation algorithms to fit their irregularity. In addition to RSVR introduced above, methods proposed in [18-19] also resolve both instruction scheduling and memory bank assignment problems without considering the limitation of accumulators/registers. Furthermore, methods [9, 17, 20, 44] contain all five phases, and all expect [44] select Motorola DSP56000 as the target architecture. We describe methods [9, 17] in some detail in the following.

In the method proposed in [9], its main idea is applying the *graph coloring* approach to treat variable partition and accumulator/register assignment. For register/ accumulator assignment, authors of [9] specially decouple this phase into two steps. They first classify physical registers into a set of register classes, and allocate each temporary variable to one of the register classes. Next, the graph coloring algorithm is applied to assign each temporary variable a physical register within the register class

previously allocated to it. After generating compacted codes, a weighted undirected graph is constructed based on the sequence of variables referenced in these codes. Then, a *maximum spanning tree* (MST) of this graph is identified, and variables are assigned also using the graph coloring algorithm. Moreover, authors of this method also propose a heuristic to resolve graph coloring problem. We think the method proposed in [9] is efficient. But it does not present the mechanism to determine and resolve accumulator/register spills, which is definitely required.

The method proposed in [17] is an example that simultaneously considers two code generation phases. The Motorola DSP56000 has heterogeneous register sets, so variables referenced from each data memory bank must be loaded in a restricted set of locations. Thus, authors of [17] claim that variable partition and accumulator/register assignment should be performed simultaneously to maximally explore available parallelism among move operations. After generating compacted codes, an undirected graph is constructed representing constrained conditions on the register and memory bank assignments. Then, an algorithm based on *graph labeling* is used to both memory bank and accumulator/register assignments. Similar as in [9], mechanisms used to insert spill codes are not present in [17]. In addition, authors of [17] suggest applying *simulated annealing* to resolve the graph labeling problem, which is a time-consuming algorithm and makes the entire method much more complicated.

In this thesis, we will study compiler design issues for DSP architecture with multiple data memory banks and heterogeneous register sets. At first we will design a method particularly for Motorola DSP56000. Then, we extend it to a more general method suitable for various DSPs with similar architectural features. Furthermore, this general method is evaluated on various architectures to study the influence of differing number of resources on the scheduling result. Detailed descriptions and our proposed methods will be presented in chapters 4 and 5.

### 2.4.4 Energy-efficient Code Generation Methods [4, 20, 24-25, 28, 45-52]

In section 1.2, we have introduced the importance of considering low power design at high-level synthesis. Authors of [45] use an experiment setup to physically measure the current being drawn by the CPU during the execution for three architecturally different processors. Based on physical measurements, they develop an instruction-level power analysis technique and an instruction-level power model. The power model consists of three main components: *instruction base costs*, *effect of circuit state*, and *other inter-instruction effects*. The base cost of an instruction is the cost associated with the basic processing required to execute the instruction. The circuit state overhead for a pair of consecutive instructions is used to deal with the switching activity changed between their circuit states. As for the power cost of other inter-instruction effects, it can occur in real programs due to prefetch buffer and write buffer stalls, pipeline stalls, and cache misses. For the DSP architecture, the effect of circuit state change is more marked in terms of power consumption, because its instruction control and data path constitute a larger portion of the silicon. Besides, this instruction-level power analysis technique also provides fundamental information that can guide the development of energy-efficient software. Several ideas in this regard motivated by this analysis are: reduction of memory accesses, energy cost driven code generation, and instruction reordering for low power. For the DSP with multiple data memory banks, instruction packing, parallel memory loads, and swapping operands for multiplications are other possible processor-specific optimizations [25]. Therefore, to reduce the power consumption from software is actually an appropriate way.

A number of studies have investigated appropriate scheduling of instructions to reduce the circuit state overhead due to its significant impact on DSP architecture. Methods proposed in [20, 25, 46] are directly based on current measurement technique. They first record base costs of all instructions and circuit state overheads

for different instruction pairs. Then, a ready instruction, which will cost less power after being appended to the current schedule according to measured data, will be selected and scheduled first. Methods proposed in [47-49] attempt low-power schedules with similar mechanisms as previous three methods. But they gather base cost and circuit state overhead information using cycle-accurate simulators *SimplePower* and *SimpleScalar*, instead of experimental measurement. Apparently, using above methods can generate schedules with low circuit state overheads. However, the measured data are only dedicated for the selected processor, so these methods are obviously less general.

On the other hand, there are lots of researches on power optimization in high-level synthesis by means of input activity reduction of function units. Authors of [50] reduce the switched capacitance of modules using an iterative improvement technique for scheduling and module allocating. Authors of [51-52] propose similar techniques, to reduce the power by preserving correlation of data inputs to function units through careful binding of instructions to function units. As for methods designed based on operand sharing technique, authors of [4] present *list-scheduling algorithm for low power* (*LPLS*), to reduce the activity of the function units by minimizing the switching activity of their input operands. LPLS obviously trades off latency for operand reuse, because instructions with common operands have to be scheduled consecutively and some instruction-level parallelism cannot be successfully explored. However, LPLS performs well only in the cases where common input operands can be identified, but it is not easy to find common input operands in real designs. Therefore, to increase the number of instructions with common operands, a high-level loop transformation technique *power-conscious loop folding* is presented in [24]. Its main idea is to find instructions sharing an operand in consecutive iterations. Then, a loop folding technique is applied to concentrate these instructions in the same iteration and execute

them consecutively. Alternatively, the method proposed in [28] contains a *force-directed retiming* to determine which instruction must be retimed. This technique aims to make as many instructions as possible take common operands as their inputs, and use a list scheduling to perform operand sharing under resource constraints.

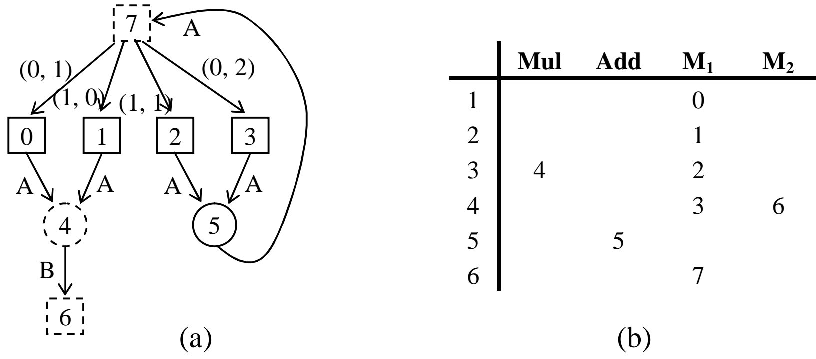
Comparing instruction scheduling methods listed above, methods designed based on operand sharing are apparently more practical. This is because these methods are not only machine-independent, but also do not require additional memory space to store measured information. In this thesis, we will focus on increasing the potential of operand sharing to design energy-efficient code generation methods. Detailed descriptions and proposed methods will be presented in chapter 6.

# Chapter 3. Variable Partition Mechanisms

In this chapter we present our first issue about variable partition mechanism. We target on DSP architecture with multiple data memory banks, and the goal is to evenly distribute memory accesses. At first we summarize some flaws of RSVR in section 3.1. Three proposed variable partition mechanisms and corresponding instruction scheduling algorithms are introduced in section 3.2 and 3.3. In section 3.4 some performance evaluations are shown.

## 3.1 Flaws of RSVR [14]

As introduced in section 2.4.2, RSVR resolves both instruction scheduling and variable partition problems. It mainly consists of four phases: constructing VIG, partitioning VIG, generating the initial schedule, and repartitioning variables during applying rotation scheduling. From our observations, RSVR may contain three flaws as follows. First, the variable partition result is not always optimal, so RSVR will repartition variables during rotation phases. This repartitioning phase obviously increases the scheduling complexity of RSVR. Second, the VIG is constructed to expose parallel memory accesses for a loop, so the variable partition result is only well suited to that loop. When the given program contains more than one loop, it is difficult to find an appropriate variable partition result fitted for all loops. Third, the parallelism between ALU instructions may be restricted by memory accesses in special cases. Consider the MDFG fragment shown in Figure 3.1(a). Actually, nodes 4 and 5 access different operands and can be executed in parallel. But in RSVR they will be scheduled in serial as shown in Figure 3.1(b), because they both access the same variable **A**. This case is similar as the *column major* problem in parallel processing system. Some operations are data independent, but must be executed in




| | Mul | Add | $M_1$ | $M_2$ |
|---|---|---|---|---|
| 1 | | | 0 | |
| 2 | | | 1 | |
| 3 | 4 | | 2 | |
| 4 | | | 3 | 6 |
| 5 | | 5 | | |
| 6 | | | 7 | |

Figure 3.1. (a) MDFG fragment, (b) scheduling result of RSVR.

serial due to unsuitable data allocation. In addition, since variable **A** is accessed many times compared with variable **B**, the memory bank stored **A** will become the schedule bottleneck. The reason caused above flaws is due to the variable partition mechanism. Thus, we will propose some mechanisms to partition variables more effectively.

## 3.2 Rotation Scheduling with Unfolding (RSF) and Rotation Scheduling with Tiling (RST) [14]

Note that a variable in MDFG indicates an array not just a single scalar. Unlike RSVR stores the entire array to a single data memory bank, we propose two mechanisms to partition array elements according to their *rightmost indices* and *leftmost indices*, respectively. For example, suppose there are *N* data memory banks, Figure 3.2 shows two variable partition results of MDFG in Figure 2.1(b). These two mechanisms are clearly more simple and efficient than that of used in RSVR, because they avoid the heavy overhead caused by constructing and partitioning the VIG.

In addition to array variables, operands of ALU instructions may be constants in DSP applications. Intuitively these constants can be loaded using immediate load instructions. But in our studies we let constants be stored in data memory at specific locations in advance, and use *load constant* instead of immediate load. Essentially, the load constant is equivalent to the original load variable instruction, but will directly

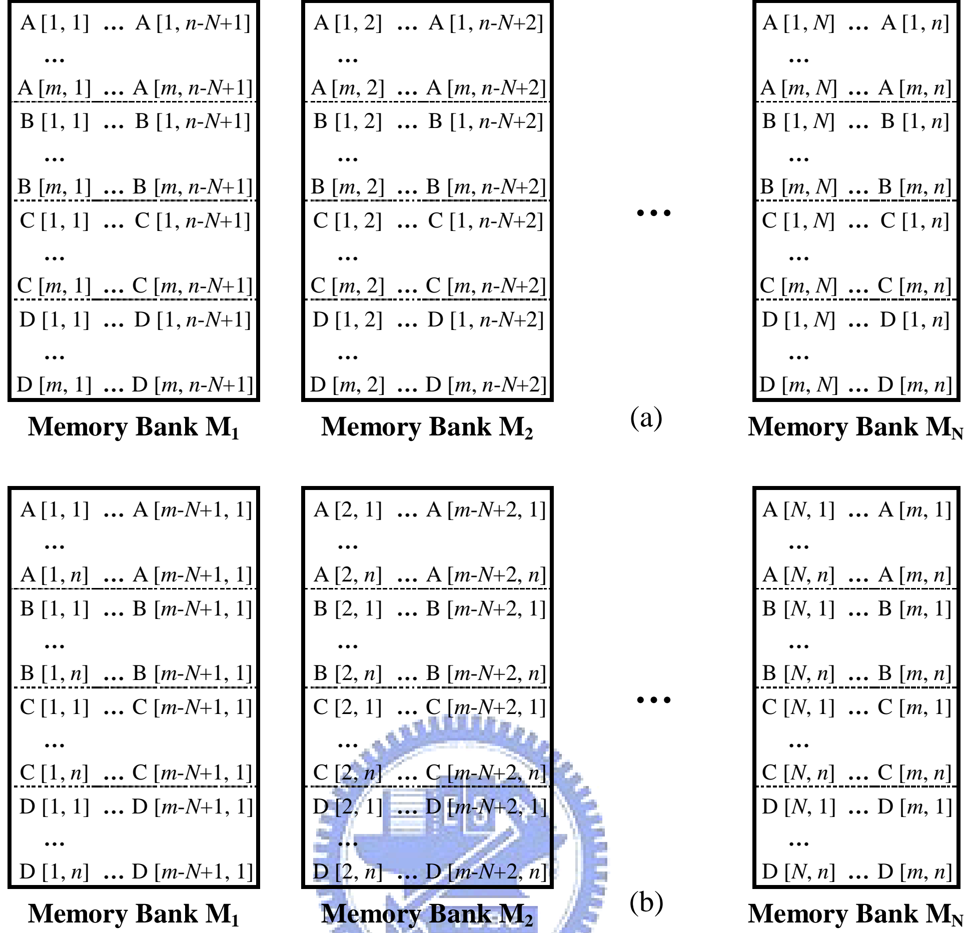



Figure 3.2. Variable partition results of MDFG in Figure 2.1(b). (a) Based on rightmost indices, (b) based on leftmost indices.

load constants from specific address. We also assume that constants are stored in all data memory banks. This feature makes load constant instructions can be scheduled at any data memory bank to increase performance.

After partitioning variable, we plan to apply the concept of multi-dimensional rotation scheduling to schedule instructions. However, it cannot be directly applied, and we illustrate the reason using the following example. Figure 3.3 lists two consecutive iterations of nested loop in Figure 2.1(a), and both instructions with mark “#” correspond to nodes 4~7 in Figure 2.1(b). If we partition variables based on their rightmost indices, operands accessed by two marked instructions will be resided in different data memory banks. That is, nodes 4 in consecutive iterations must be

| | |
|---|---|
| D[$l$, $k$] = B[$l$-1, $k$] × C[$l$-1, $k$-2] **;** | D[$l$, $k$+1] = B[$l$-1, $k$+1] × C[$l$-1, $k$-1] **;** |
| A[$l$, $k$] = D[$l$, $k$] × 0.5 **;** (#) | A[$l$, $k$+1] = D[$l$, $k$+1] × 0.5 **;** (#) |
| B[$l$, $k$] = A[$l$, $k$] + 1 **;** | B[$l$, $k$+1] = A[$l$, $k$+1] + 1 **;** |
| C[$l$, $k$] = A[$l$, $k$-1] + 2 **;** | C[$l$, $k$+1] = A[$l$, $k$] + 2 **;** |

Figure 3.3. Two consecutive iterations of nested loop in Figure 2.1(a).

1. $G_c$ = Construct MDFG;
2. Partition variables to $N$ memory banks according to rightmost indices;
3. $G_N$ = unfold $G_c$ with factor $N$;
4. Select the retiming base $r$;
5. $S$ = schedule $G_N$ using list scheduling;
6. $S'$ = compact $S$ using multi-dimensional rotation scheduling;

Figure 3.4. The entire scheduling steps of RSF.

scheduled to different data memory bank, and so are nodes 7. This situation makes traditional scheduling algorithms unusable. Partitioning variables based on their leftmost indices will also cause similar problem.

*Unfolding* [31] (also called *unrolling*) and *tiling* [32] techniques can be used to resolve this problem. Their feasibilities are proven in Theorems 3.1 and 3.2 listed below. Note that not every nested loop can be tiled directly, so we need to *skew* [32] the nested loop before tiling if necessary. After unfolding or tiling the given nested loop, multi- dimensional rotation scheduling can be successfully applied to generate a compact schedule. Thus, based on two variable partition mechanisms, we propose instruction scheduling algorithms named *rotation scheduling with unfolding* (*RSF*) and *rotation scheduling with tiling* (*RST*) as listed in Figure 3.4 and 3.5, respectively. Suppose the target architecture consists of one function unit and two data memory banks. Figure 3.6 shows unfolded and tiled MDFGs of Figure 2.1(a), and scheduling results generated by different methods are shown in Figure 3.7. From this example, we find that using RSF and RST may obtain more compact schedules compared to using RSVR. Moreover, because RSF and RST never repartition variables during rotation phases, their scheduling complexities are obviously less than that of RSVR.

1. $G_c$ = Construct MDFG;
2. Partition variables to $N$ memory banks according to leftmost indices;
3. $G_N$ = tiled $G_c$ with tile size $N\times1\times\ldots\times1$ (skewed the nested loop before tiling if necessary);
4. Select the retiming base $r$;
5. $S$ = schedule $G_N$ using list scheduling;
6. $S'$ = compact $S$ using multi-dimensional rotation scheduling;

Figure 3.5. The entire scheduling steps of RST.

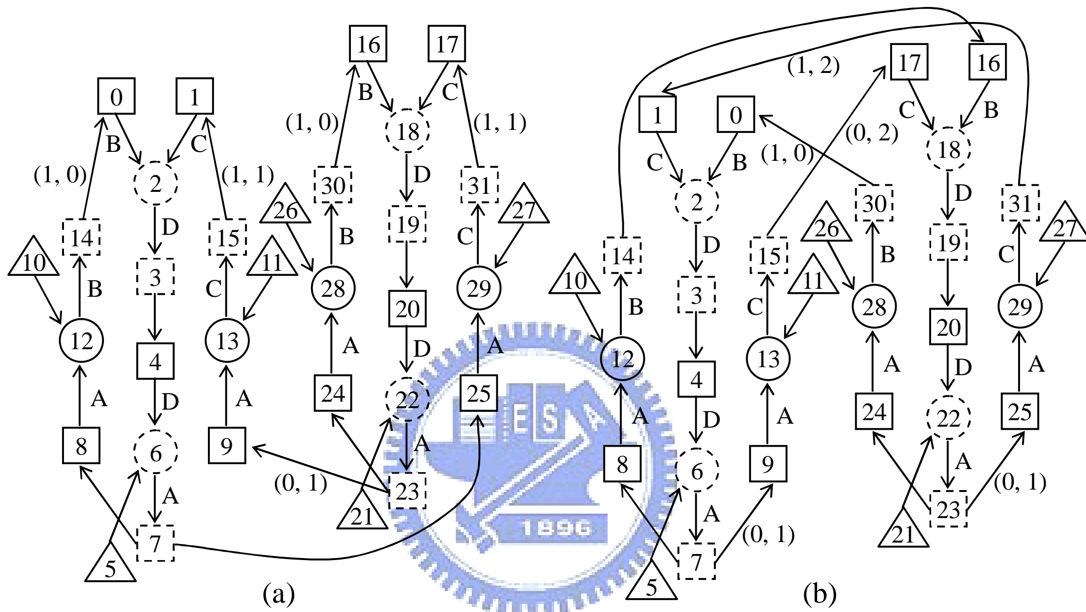



Figure 3.6. (a) Unfolded MDFG of Figure 2.1(b), (b) tiled MDFG of Figure 2.1(b).

| | FU | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | 13 | 5 | 3 |
| 2 | | 10 | 4 |
| 3 | 6 | 0 | 15 |
| 4 | | 7 | 1 |
| 5 | 2 | 8 | 11 |
| 6 | 12 | 9 | |
| 7 | | 14 | |

schedule length = 7

(a)

| | FU | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | 2 | 17 | 16 |
| 2 | 18 | 3 | 5 |
| 3 | | 4 | 19 |
| 4 | 6 | 21 | 20 |
| 5 | 22 | 7 | 10 |
| 6 | | 8 | 23 |
| 7 | 12 | 25 | 27 |
| 8 | 29 | 14 | 24 |
| 9 | | 26 | 31 |
| 10 | 28 | 11 | 9 |
| 11 | 13 | 0 | 30 |
| 12 | | 15 | 1 |

schedule length = 6 (b)

| | FU | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | 13 | 7 | 27 |
| 2 | 19 | 8 | 0 |
| 3 | 12 | 15 | 31 |
| 4 | | 14 | 1 |
| 5 | 2 | 16 | 21 |
| 6 | 18 | 5 | 26 |
| 7 | | 10 | 19 |
| 8 | | 17 | 20 |
| 9 | 22 | 3 | 11 |
| 10 | | 4 | 23 |
| 11 | 6 | 9 | 24 |
| 12 | 28 | | 25 |
| 13 | | | 30 |

(c) schedule length = 6.5

Figure 3.7. Scheduling results of Figure 2.1(b). (a) RSVR, (b) RSF, (c) RST.

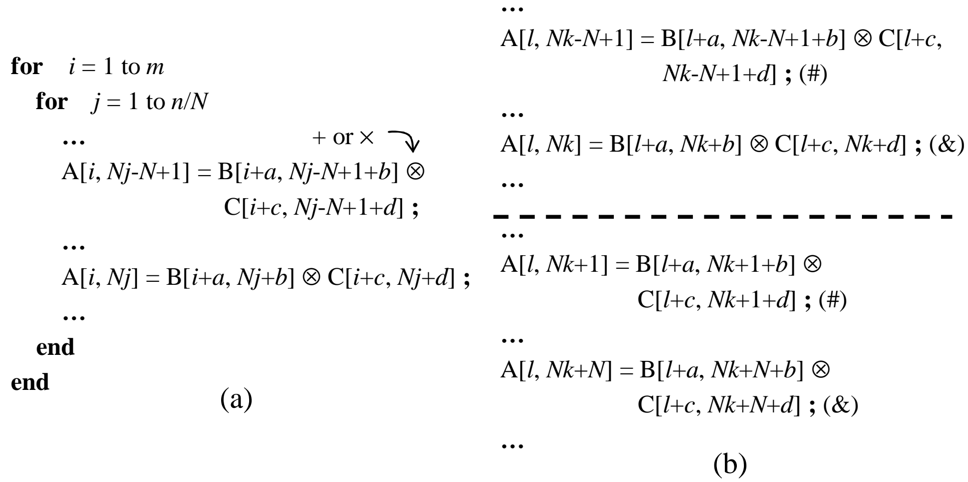



Figure 3.8. (a) Unfolded nested loop in canonical form, (b) two consecutive iterations.

**Theorem 3.1** Given a nested loop and its corresponding MDFG *G*. Variables are partitioned to *N* data memory banks based on their rightmost indices. After unfolding the innermost loop with factor *N*, every memory access in the unfolded MDFG $G_N$ can be scheduled to specific data memory bank.

*Proof*: We use a nested loop with depth two as an example. Figure 3.8 shows the unfolded nested loop with factor *N* and two consecutive iterations in canonical form. In Figure 3.8(b), instructions with the same mark will correspond to the same nodes in $G_N$. From Figure 3.2(a), clearly that operands accessed by instructions with the same mark are stored in the same data memory bank. Thus, every node in $G_N$ can be scheduled to specific data memory bank. Nested loop with depth more than two can be proven using the same way.

**Theorem 3.2** Given a nested loop and its corresponding MDFG *G*. Variables are partitioned to *N* data memory banks based on their leftmost indices. After tiling the nested loop with tile size $N$×1×…×1, every memory access in the transformed MDFG $G_N$ can be scheduled to specific data memory bank.

*Proof*: We still use a nested loop with depth two as an example. Figure 3.9 shows the transformed nested loop with tile size $N$×1 and two consecutive iterations in canonical

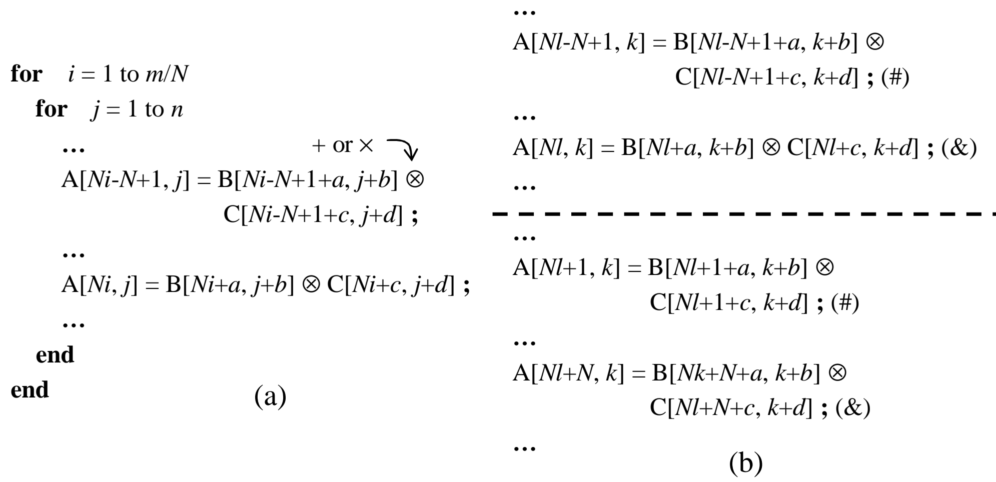
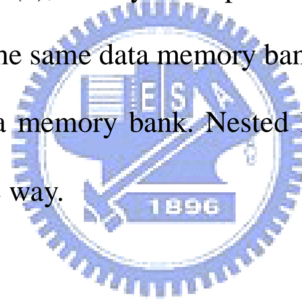



Figure 3.9. (a) Tiled nested loop in canonical form, (b) two consecutive iterations.

form. In Figure 3.9(b), instructions with the same mark will correspond to the same nodes in $G_N$. From Figure 3.2(b), clearly that operands accessed by instructions with the same mark are stored in the same data memory bank. Hence, every node in $G_N$ can be scheduled to specific data memory bank. Nested loop with depth more than two can be proven using the same way.



## 3.3 Rotation Scheduling with Parallelization (RSP) [36]

Because loop unfolding and tiling techniques are applied in RSF and RST to fit variable partition results, their enlarged iterations are composed of several original iterations. If original iterations composed of an enlarged iteration are data independent, RSF and RST are actually effective. However, if critical paths of those original iterations are *cascaded* after applying unfolding or tiling, using RSF or RST will obtain schedules with very long schedule lengths. Therefore, we apply the unimodular transformations to parallelize the inner loop before unfolding, which can ensure that original iterations composed of an unfolded iteration will not depend on each other. This method is named *rotation scheduling with parallelization* (*RSP*). Since RSP avoids the drawback of RSF and RST, we believe it can achieve better results.

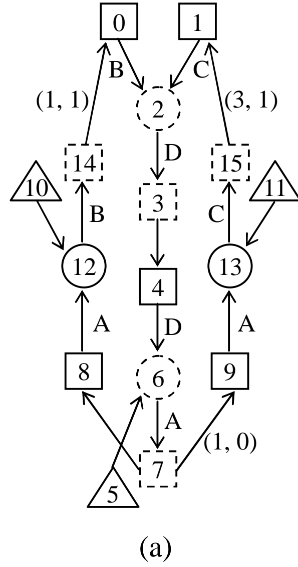
1. $G_c$ = Construct MDFG;
2. Partition variables to $N$ memory banks according to specific mechanism;
3. $G_p$ = parallelize $G_c$ that the inner loop is parallelizable;
4. $G_N$ = unfold $G_p$ with factor $N$;
5. Select the retiming base $r = (0, 1)$;
6. $S$ = schedule $G_N$ using list scheduling;
7. $S'$ = compact $S$ using multi-dimensional rotation scheduling;

Figure 3.10. The entire scheduling steps of RSP.

1. **Input:** MDFG $G = (V, E, X, d, t)$, $N$
2. **Output:** MDFG $G' = (V, E, X, d', t)$;
3. $G' = G$; $w = 0$;
4. **while** ($\exists$ (0, $a$) and ($b$, 0) in $d'$, for $a, b > 0$)
   $w = 1; \forall d'(e) \in d, d'(e) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times d'(e);$
5. **if** ($\exists$ ($b$, -$c$) in $d'$, for $b, c > 0$)
   $w = w + \lceil (c+1)/b \rceil; \forall d'(e) \in d, d'(e) = \begin{bmatrix} 1 & 0 \\ \lceil (c+1)/b \rceil & 1 \end{bmatrix} \times d'(e);$
6. **if** ($N$ is odd)
   **if** (($w$ **mod** $N$) = = 2)
   $w = w + 1; \forall d'(e) \in d, d'(e) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times d'(e);$
7. $\forall d'(e) \in d, d'(e) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times d'(e);$
8. **Return** $G' = (V, E, X, d', t)$;

Figure 3.11. Loop parallelization algorithm.

Figure 3.10 lists the entire scheduling steps of RSP. Note that RSP is designed only for nested loop with depth two, and the number of data memory banks must be odd or power of two. Nevertheless, it can be further extended to cover MDFG with higher dimensions. Figure 3.11 is the algorithm designed to parallelize the inner loop by unimodular transformations, which is slightly modified from the algorithm proposed in our early study [42]. Figure 3.12(a) contains the parallelized MDFG of Figure 2.1(b). Particular variable partition mechanism used in RSP is presented as follows. Although this mechanism seems irregular, it still partition variables according to array indices and is quite simple and efficient.




| | Mul | Add | $M_1$ | $M_2$ | $M_3$ |
|---|---|---|---|---|---|
| 1 | 18 | | 3 | 5 | 21 |
| 2 | 34 | | 4 | 19 | 37 |
| 3 | 6 | | 25 | 20 | 35 |
| 4 | 22 | | 7 | 27 | 36 |
| 5 | 38 | 29 | 11 | 23 | 9 |
| 6 | | 13 | 43 | 41 | 39 |
| 7 | | 45 | 15 | 31 | 10 |
| 8 | | | 8 | 24 | 47 |
| 9 | | 12 | 26 | 42 | 40 |
| 10 | | 28 | 14 | 0 | 1 |
| 11 | 2 | 44 | 17 | 30 | 16 |
| 12 | | | 32 | 33 | 46 |

schedule length = 4

(a) (b)

Figure 3.12. (a) The parallelized MDFG of Figure 2.1(b), (b) scheduling result of Figure 2.1(b) using RSP.

**2 data memory banks**

Bank $i$: $[m, 2k + i]$ $\quad k \in \mathbf{Z}$

***N* data memory banks (*N* is odd)**

Bank $i$: $[m, kN + ((2 - 2m)\ \mathbf{mod}\ N) + i]$ $\quad k \in \mathbf{Z}$

***N* data memory banks ($N = 2^n$, $n \geq 2$)**

Bank $i$: $[m, \frac{kN}{2} + ((\frac{1-m}{2})\ \mathbf{mod}\ \frac{N}{2}) + i]$ $\quad 1 \leq i \leq \frac{N}{2}$, $m$ is odd, $k \in \mathbf{Z}$

Bank $i$: $[m, \frac{kN}{2} + ((\frac{4-m}{2})\ \mathbf{mod}\ \frac{N}{2}) - i]$ $\quad \frac{N}{2}+1 \leq i \leq N$, $m$ is even, $k \in \mathbf{Z}$

With the similar reason of RSF and RST, the parallelized MDFG must be unfolded before applying multi-dimensional rotation scheduling. Moreover, schedule vector (1, 0) can be always selected for applying retiming technique in RSP, which is beneficial to achieve shorter overall schedule length of the retimed loop [41]. Suppose the target architecture consists of one multiplier, one adder, and three data memory banks. The unfolded MDFG of Figure 3.12 (a) is shown in Figure 3.13, and Figure 3.12(b) is its scheduling result generated using RSP. Finally, because variables are never repartitioned during rotation phases, the scheduling complexity of RSP is also less than that of RSVR.

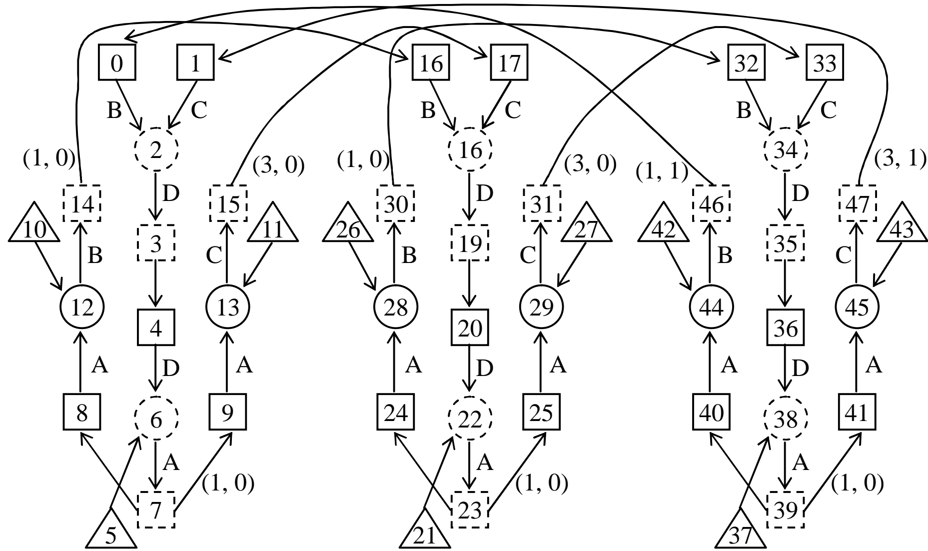



Figure 3.13. The unfolded MDFG of Figure 3.12(a).

## 3.4 Performance Evaluations

### 3.4.1 Performance Studies of a Single Iteration

In the following, we select several MDFGs represented DSP applications to evaluate methods including *list scheduling*, RSVR, RSF, RST, and RSP. The execution process for a retimed loop will consist of three parts: prologue, repetitive pattern, and epilogue. The prologue and epilogue are instruction sets that must be executed before and after the repetitive pattern. The repetitive pattern will be iterated many times and will dominate the entire computation performance of the given loop. Therefore, in this subsection we first focus on a single iteration in the repetitive pattern to compare different methods. Suppose that the system contains 1~2 function units and 2~4 data memory banks, and both ALU and memory access instructions take one time step to execute. Tables 3.1 and 3.2 list schedule lengths and retiming depths obtained from different methods for a single iteration in the repetitive pattern. Note that some schedule lengths in these tables are fractional. This is because the MDFG may be unfolded or tiled before being scheduled, and we show the average schedule length of an original iteration.

Table 3.1. Experimental results (1 function unit)(*schedule length*, *retiming depth*).

| | 2 data memory banks | | | | | 3 data memory banks | | | | | 4 data memory banks | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | List | RSVR | RSF | RST | RSP | List | RSVR | RSF | RST | RSP | List | RSVR | RSF | RST | RSP |
| [1] | 9 | 6, 1 | 6, 0 | 6, 1 | 6, 0 | 9 | 6, 2 | 4, 1 | 4.3, 2 | 4, 1 | 9 | 4, 4 | 4, 1 | 4, 3 | 4, 1 |
| [2] | 13 | 6, 2 | 6.5, 2 | 6.5, 2 | 6, 1 | 13 | 5, 3 | 4.3, 5 | 4.3, 3 | 4.3, 1 | 13 | 4, 3 | 4, 6 | 4, 4 | 4, 1 |
| [3] | 27 | 25, 1 | 24, 1 | 24, 1 | 24, 1 | 21 | 17, 1 | 17, 1 | 17, 1 | 16.3, 1 | 21 | 16, 1 | 16, 1 | 16, 1 | 16.5, 1 |
| [4] | 12 | 7, 2 | 7, 1 | 6.5, 2 | 6.5, 1 | 12 | 5, 4 | 5, 3 | 5, 4 | 5, 1 | 12 | 5, 6 | 5, 4 | 5, 5 | 5, 1 |
| [5] | 7 | 5, 1 | 5, 1 | 5, 1 | 5, 1 | 7 | 5, 2 | 4, 1 | 4, 1 | 4, 1 | 7 | 4, 3 | 4, 1 | 4, 1 | 4, 1 |
| [6] | 19 | 16, 1 | 17.5, 1 | 17, 0 | 16, 1 | 17 | 12, 2 | 14.3, 4 | 12.3, 1 | 12, 1 | 15 | 12, 2 | 13, 2 | 12, 1 | 12, 1 |
| [7] | 31 | 24, 1 | 23.5, 1 | 22, 1 | 22, 1 | 31 | 18, 2 | 18, 2 | 17.3, 2 | 17.3, 1 | 31 | 17, 3 | 17, 3 | 17, 2 | 17, 1 |
| [8] | 20 | 15, 1 | 15, 1 | 17, 2 | 15, 1 | 20 | 12, 2 | 12, 1 | 12, 4 | 12, 1 | 20 | 12, 3 | 12, 1 | 12, 5 | 12, 1 |
| [9] | 14 | 12, 1 | 11.5, 1 | ** | ** | 14 | 12, 1 | 10.7, 1 | ** | ** | 13 | 11, 1 | 10.3, 1 | ** | ** |
| [10] | 20 | 19, 1 | 21, 1 | ** | ** | 19 | 19, 0 | 20, 1 | ** | ** | 19 | 19, 0 | 19.5, 1 | ** | ** |
| [11] | 37 | 32, 1 | 27, 1 | ** | ** | 37 | 34, 1 | 25.7, 1 | ** | ** | 37 | 35, 1 | 24.5, 1 | ** | ** |
| [12] | 50 | 49, 1 | 66, 0 | ** | ** | 49 | 48, 1 | 65.7, 0 | ** | ** | 49 | 49, 0 | 65.5, 0 | ** | ** |

[1] Wave Digital Filter
[2] Filter
[3] Infinite Impulse Response Filter 2D
[4] Forward-substitution
[5] Toeplitz Hyperbolic Cholesky Solver
[6] Discrete Fourier Transform
[7] Floyd-Steinberg
[8] Transmission Line
[9] Infinite Impulse Response Filter 1D
[10] Differential Equation Solver
[11] All-pole Lattice Filter
[12] Elliptic Filter

Table 3.2. Experimental results (2 function units)(*schedule length*, *retiming depth*).

| | 2 data memory banks | | | | | 3 data memory banks | | | | | 4 data memory banks | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | List | RSVR | RSF | RST | RSP | List | RSVR | RSF | RST | RSP | List | RSVR | RSF | RST | RSP |
| [1] | 9 | 6, 1 | 6, 0 | 6, 1 | 6, 0 | 9 | 6, 2 | 4, 0 | 4, 2 | 4, 0 | 9 | 4, 4 | 3, 0 | 3.5, 3 | 3.3, 1 |
| [2] | 13 | 6, 2 | 6.5, 2 | 6.5, 2 | 6, 1 | 13 | 5, 3 | 4, 5 | 4.7, 3 | 4, 1 | 13 | 4, 3 | 3.5, 7 | 3, 4 | 3.3, 1 |
| [3] | 27 | 25, 1 | 24, 1 | 24, 1 | 24, 1 | 20 | 17, 1 | 17, 1 | 17, 1 | 16, 1 | 21 | 13, 1 | 12.8, 1 | 12.8, 1 | 12, 1 |
| [4] | 12 | 7, 2 | 7, 1 | 6.5, 2 | 6.5, 1 | 12 | 5, 4 | 5.3, 2 | 5.3, 3 | 4.3, 1 | 12 | 4, 6 | 4.3, 3 | 3.8, 6 | 3.3, 1 |
| [5] | 7 | 5, 1 | 5, 1 | 5, 0 | 5, 0 | 7 | 5, 2 | 3.7, 1 | 3.3, 1 | 3.3, 1 | 7 | 3, 4 | 2.8, 2 | 2.8, 0 | 2.8, 1 |
| [6] | 18 | 16, 1 | 17.5, 1 | 17, 0 | 16, 0 | 16 | 11, 2 | 12, 2 | 11, 1 | 10.7, 1 | 13 | 8, 2 | 12, 2 | 8, 1 | 8, 1 |
| [7] | 28 | 21, 1 | 23, 1 | 22, 1 | 21, 1 | 28 | 14, 2 | 15.3, 2 | 16.3, 2 | 14, 1 | 28 | 11, 3 | 12.3, 4 | 12.5, 3 | 11, 1 |
| [8] | 18 | 15, 1 | 15, 1 | 15, 2 | 15, 1 | 18 | 11, 2 | 10, 1 | 13.7, 4 | 10.3, 1 | 18 | 8, 3 | 7.8, 1 | 11.5, 5 | 7.5, 1 |
| [9] | 14 | 12, 1 | 11.5, 1 | ** | ** | 14 | 12, 1 | 10.7, 1 | ** | ** | 13 | 11, 1 | 10.3, 1 | ** | ** |
| [10] | 19 | 18, 1 | 21, 1 | ** | ** | 18 | 18, 0 | 20, 1 | ** | ** | 18 | 18, 0 | 19.5, 1 | ** | ** |
| [11] | 36 | 33, 1 | 27, 1 | ** | ** | 36 | 34, 1 | 25.7, 1 | ** | ** | 36 | 35, 1 | 24.5, 1 | ** | ** |
| [12] | 48 | 47, 1 | 66, 0 | ** | ** | 44 | 43, 1 | 65.7, 0 | ** | ** | 43 | 43, 0 | 65.5, 0 | ** | ** |

From these results, RSVR obviously outperforms list scheduling in all cases like evaluations shown in [30]. Three proposed methods RSF, RST, and RSP also achieve effective results, but not always better than that of RSVR. This is because the enlarged MDFG gives a more global view of the data dependencies, which is usually beneficial for compacting schedules. However, based on our variable partition mechanisms, most memory accesses in the same original iteration will be scheduled to the same data memory bank. If an iteration of RSF or RST is cascaded by original iterations, its results will be inferior to RSVR. Besides, if memory accesses will gather at some data memory banks in RSVR, our methods can obtain better results. As for RSP, it usually achieves similar schedule lengths to other methods for a single iteration in the repetitive pattern, but apparently gets smaller retiming depths. This is because an iteration in RSP is composed of independent original iterations and memory accesses will be evenly scheduled. Thus, schedules generated by list scheduling will be already very compact, which can decrease times applying rotation phases and retiming depth.

### 3.4.2 Performance Studies of the Entire Retimed Loop [14, 36]

In addition to the repetitive pattern, prologue and epilogue are also generated for a retimed loop as described in the previous subsection. Strictly speaking, prologue and epilogue are part of the overhead, not only for the execution time but also for the instruction count. Many previous studies have stated that the time required to run the prologue and epilogue are negligible if the given loop contains sufficient iterations. However, as shown in section 2.2, the prologue and epilogue may still constitute a considerable portion of the overall schedule length if an unsuitable schedule vector is selected. Thus, we design an analytic model to calculate the overall schedule length of a retimed one or two-dimensional MDFG. Nevertheless, this analytic model can be easily extended to cover nested loop with depths greater than two.

Table 3.3. Variables defined in the analytic model.

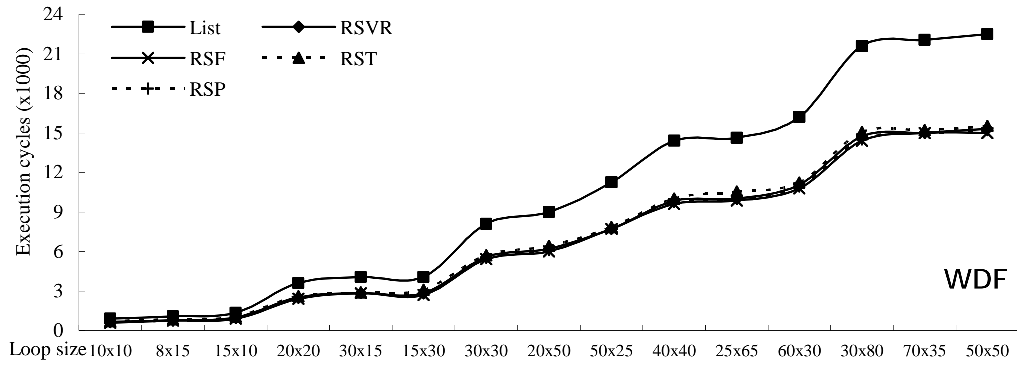
| Variable | Definition |
|---|---|
| $N$ | Number of memory banks |
| $m$ | Loop bound of the outer loop for a two-dimensional nested loop<br>Loop bound for an one-dimensional loop |
| $n$ | Loop bound of the inner loop for a two-dimensional nested loop |
| *prologue* | Schedule length of the prologue part of a retimed loop |
| *epilogue* | Schedule length of the epilogue part of a retimed loop |
| *length* | Schedule length of a single iteration in the repetitive pattern of a retimed loop |
| *list* | Schedule length of a single iteration produced by list scheduling |
| $d$ | Retiming depth, the number of iterations that must be moved into the prologue and epilogue |
| $w$ | Skew factor used to parallelize the inner loop |
| *half* ($k$, $N$) | Schedule length of $k$ original iterations under $N$ memory banks |

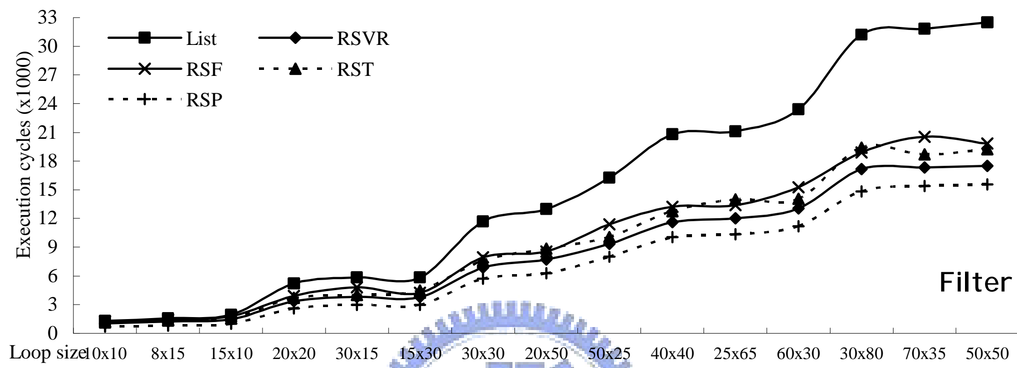
Table 3.3 lists variables used in our analytic model. For a one-dimensional loop the retiming base $r = 1$ is always feasible. Schedule vector ($s_1$, $s_2$) is selected for a two-dimensional nested loop, where $s_1$ and $s_2$ are both positive integers. Several formulas are defined to calculate the overall schedule length of a retimed loop. Detailed derivations of these formulas are listed in appendix A [14, 36].

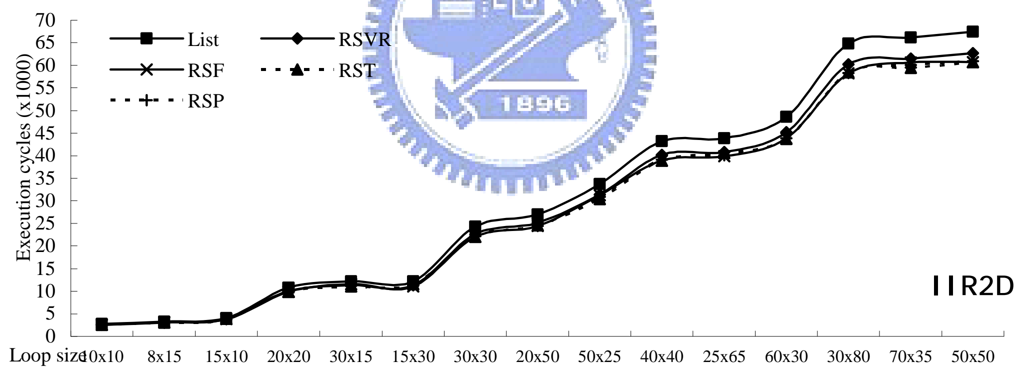
Suppose the system contains one function unit and three data memory banks, Figures 3.14 and 3.15 show the overall schedule length calculated by above formulas. In view of the entire retimed loop, the overall schedule lengths of our methods perform similar to even outperform RSVR. Hence, our RSF, RST, and RSP are not only efficient but also as effective as RSVR.

### 3.4.3 Comparisons among RSF, RST, and RSP [14]

After evaluating RSF, RST, and RSP, we analyze the effectiveness among them. Actually the answer will depend on the topology and loop-carried dependencies of the nested loop. From formulas (A.6) and (A.7) listed in appendix A, we find that after



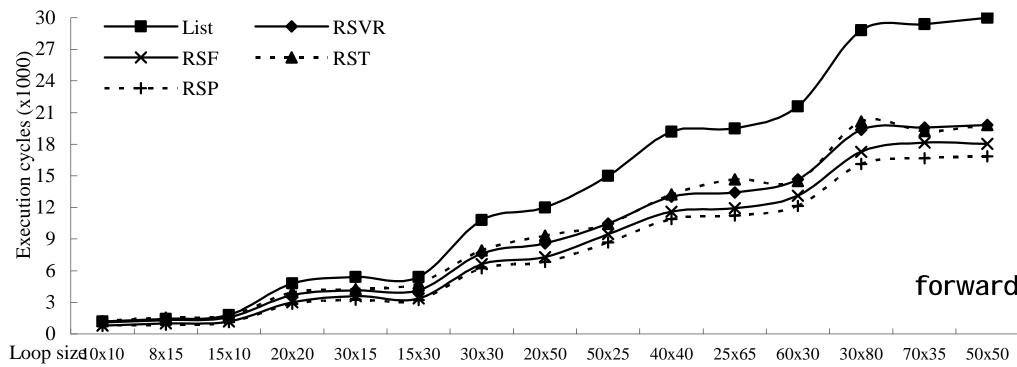




Figure 3.14. Overall schedule lengths of DSP applications (1 function unit, 2 memory banks).

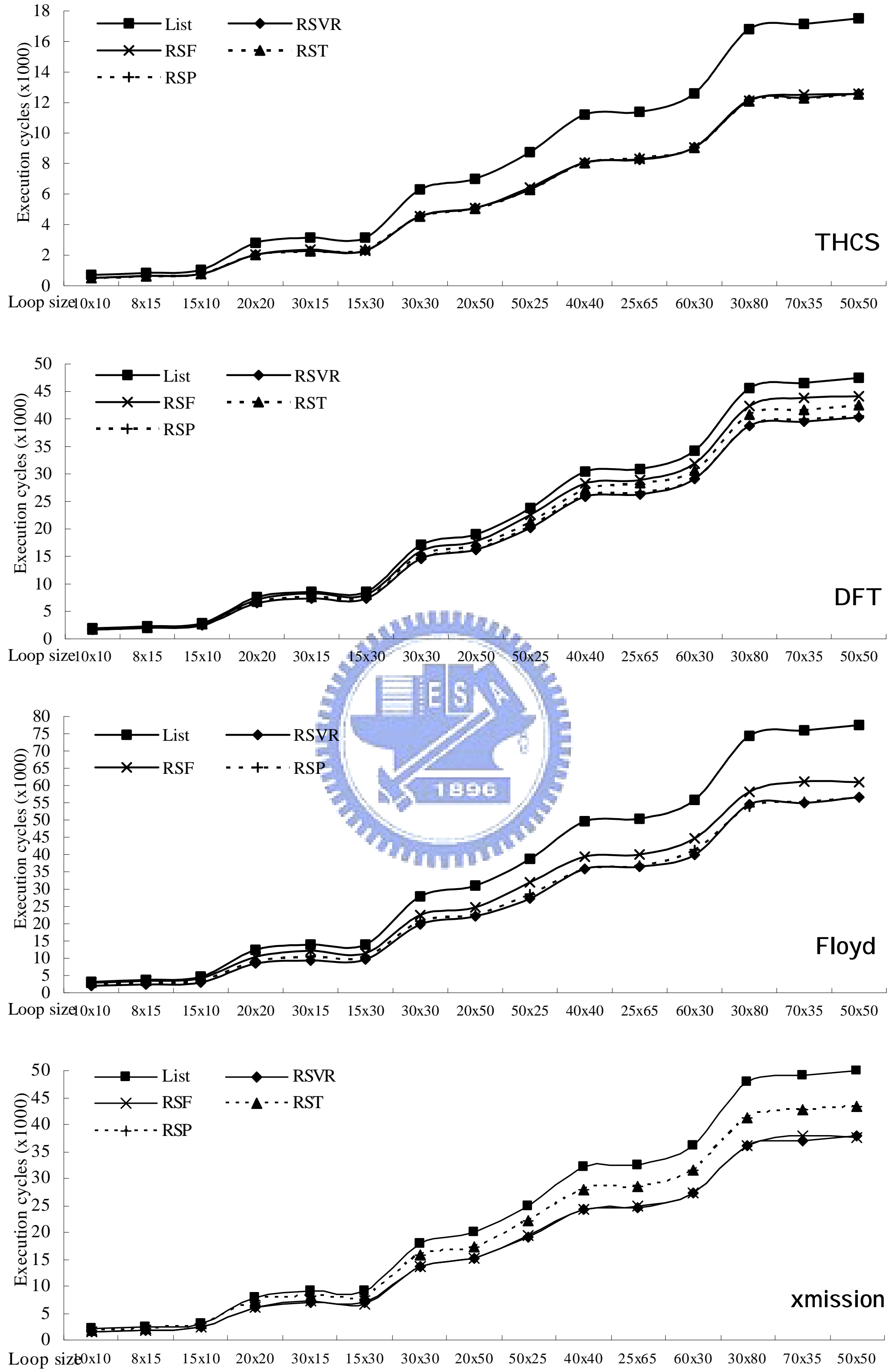



Figure 3.15. Overall schedule lengths of DSP applications (1 function unit, 2 memory banks).

applying RSP, *prologue*, *epilogue*, and *half* really occupy considerable portion of the overall schedule length. Recall that the prologue, epilogue, and half are part of the overhead. That is, although RSP performs as well as other methods from Figures 3.14 and 3.15, it costs more overheads on both execution time and instruction count compared to RSF and RST, especially when the architecture contains more than two data memory banks. Therefore, we suggest using RSP only for the DSP architecture with two data memory banks.

In the following, we conclude some principles of RSF, RST, and RSP. If the nested loop only contains dependencies with distances (1, 0, …, 0) but (0, ..., 0, 1), RSF should obtain better results because original iterations in a single unfolded iteration are data independent. On the contrary, using RST should be better if the nested loop only contains dependencies with distances (0, …, 0, 1) but (1, 0, …, 0), and this nested loop can be tiled directly. These two conclusions are made directly based on the principle of their variable partition mechanisms. Then, if the nested loop contains dependencies with distances (0, …, 0, 1) and (1, 0, …, 0), an enlarged iterations in neither RSF nor RST is combined with original iterations which are data independent. At this time RSP is suited when the architecture contains two data memory banks. As for architectures with more than two data memory banks, we suggest using list scheduling to schedule an iteration of RSF and RST separately and choose the shorter one.

In parallel processing system, column major is one of common problems to prevent the parallelism. If the target DSP architecture contains multiple data memory banks and more than one function unit, the similar problem will also occur. When we design methods RSF, RST, and RSP, we simply assume the given nested loop is executed in row major sequence. After enlarging the given loop, the variable partition mechanism will be selected based on distances of loop-carried data dependencies.

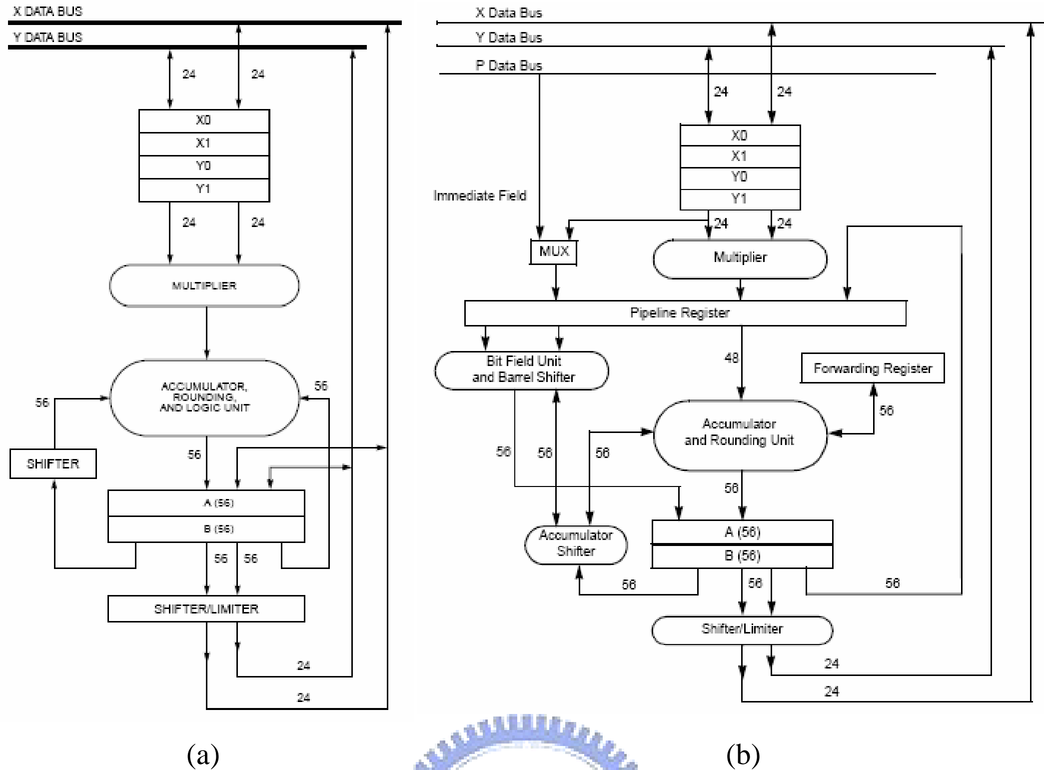
From descriptions listed in above paragraph, our goal is to make original iterations in an enlarged iteration be data independent. That is, for elements of the same array variable which will be accessed in an enlarged iteration, we will separate them into different memory banks as far as possible. This mechanism works well when there is only one DSP core with one or more function units. However, we never consider the memory access sequence between different enlarged iterations. Therefore, if there are two or more DSP cores in the target architecture, the potential column major problem may still occur using our proposed methods.

# Chapter 4. Effective Code Generation Method for Motorola DSP56000

In this and next chapter, we will present our second issue about code generation methods for DSPs with multiple data memory banks and heterogeneous register sets. As mentioned in section 2.4.3, a complete code generation process for DSP with multiple data memory banks must include five phases: *intermediate representation*, *code compaction*, *instruction scheduling*, *memory bank assignment* (or *variable partition*), and *register/accumulator assignment* [17]. Our three methods RSF, RST, and RSP presented above directly use data memory to store temporary variables, so they have covered all except the accumulator/register assignment phase. In this chapter, we introduce a new method focus on Motorola DSP56000 to consider the accumulator/register assignment and further improve overall execution performance. Section 4.1 we briefly give an overview of the Motorola DSP56000. Section 4.2 lists our design motivations, and detailed steps of proposed method are described in section 4.3. Finally, in section 4.4 some performance evaluations are shown.

## 4.1 Motorola DSP56000 Architecture [10]

In our studies we target on the DSP architecture which consists of multiple data memory banks and heterogeneous register sets. Associated with each data memory bank is an independent set of address bus, data bus, and independent unit to calculate address. Motorola DSP56000/DSP56001 and DSP56300 family members are examples of this architecture, and are commonly used in practice and in previous researches. Many members belong to DSP56300 family, which have various memory sizes and peripheral interfaces. However, data ALU circuits of all members of DSP56300 family are the same, and are almost identical to those of DSP56000/




(a) (b)

Figure 4.1. Data ALU block diagram. (a) DSP56000/DSP56001, (b) DSP56300 family.

DSP56001 as shown in Figure 4.1. The main difference is that all ALU instructions are completed in one clock cycle in DSP56000/DSP56001, and performed in two clock cycles in pipeline fashion in DSP56300 family. Therefore, in the following we briefly introduce the Motorola DSP56000 architecture, and design our code generation method based on it.

As shown in Figure 4.2, the DSP56000 architectural units of interest are the data ALU, Address Generation Unit (AGU), and X/Y memory banks. The data ALU consists of four input registers called *X*0, *X*1, *Y*0, and *Y*1, and two accumulators, *A* and *B*. The source operands for all ALU instructions, except multiplication, must be registers or accumulators and the destination operand must always be an accumulator. Source operands of multiplication must always be input registers. Two buses *XDB* and *YDB* permit two input registers or accumulators to be read or written in conjunction during execution of an ALU instruction. Thus, up to two *move* operations (including

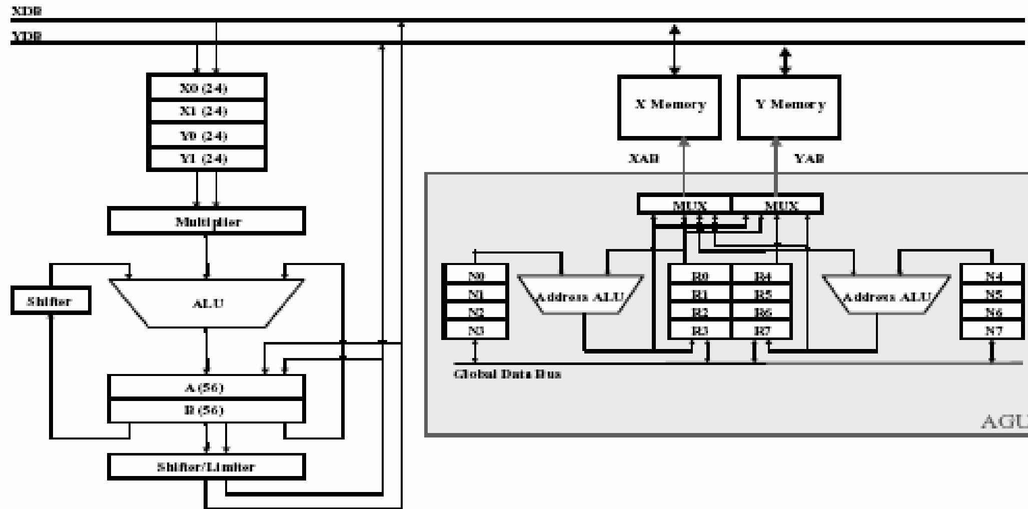



Figure 4.2. Motorola DSP56000 architecture.

memory access, register transfer, and immediate load) and one data ALU instruction may be executed simultaneously in one cycle.

Two independent move operations executed in the same cycle are called *parallel moves*. However, due to the nature of the DSP56000 architecture, not all pairs of move operations can be performed in parallel. Detailed *parallel move conditions* can be found in [10]. In our studies we especially consider the following conditions: (1) the two move operations reference data in different data memory banks; (2) the two destination registers are different; (3) the *X/Y* memory access loads into restricted locations *X*0/*Y*0, *X*1/*Y*1, *A*, or *B*.

## 4.2 Design Motivations [37]

In section 2.4.3 we have surveyed some code generation methods for DSP with multiple data memory banks. Among them, both methods proposed in [9, 17] focus on Motorola DSP56000 and consider all five phases of the code generation process. In the following we summarize them and introduce our design motivations. First, these two methods perform variable partition after code compaction, which means memory accesses are scheduled without information of memory bank assignment. However, in

the DSP56000 architecture, memory accesses involved in a parallel move must reference variables in different banks [10]. That is, memory accesses may be assumed to be executed in parallel, but in fact their reference variables are stored in the same data memory bank. In this situation, an extra cycle (*spill code*) will be required to access them separately. If spill codes occur frequently, the computational performance is clearly degraded. On the other hand, if variables are partitioned before code compaction, this kind of spill codes will not occur. In our design we will use the later mechanism to avoid the occurrence of spill codes.

Apart from location conflict for parallel moves, spill codes also possibly occur in the accumulator/register assignment phase. In methods proposed in [9, 17], they store variables in unlimited *symbolic* accumulator/register during code compaction, and consider accumulator/register assignment at last. But in DSP numbers of accumulators and registers are usually strictly limited. When accumulator and register spills occur, spill codes are required and their *spill costs* may be more than one extra cycle. Therefore, we will design mechanisms to predict the occurrence of register and accumulator spills in advance and generate corresponding spill codes. Then, these spill codes can be scheduled in parallel with other instructions, which is beneficial to decrease the spill costs.

### 4.3 Rotation Scheduling with Spill Codes Predicting (RSSP) [37]

In this section we introduce code generation method named *rotation scheduling with spill codes predicting* (*RSSP*) proposed for Motorola DSP56000. As listed in Figure 4.3 RSSP contains six parts: MDFG construction, TDAG construction, TDAG modification, ALU instruction scheduling, other instruction scheduling, and initial schedule retiming. Detailed description of each part is presented as follows.

1. $G_c$ = Construct MDFG;
    1.1. Partition variables to X and Y memory banks;
    1.2. Unfold or tile $G_c$ if necessary;
2. $G_t$ = Construct TDAG;
3. Modify TDAG $G_t$;
    5.1. $G_t$ = Insert register transfer nodes ($G_t$);
    5.2. ($G_{op}$, $G_{pr}$) = Construct DAG $G_{op}$ and $G_{pr}$ ($G_t$);
    5.3. $G_{op}$ = Mark_Edge ($G_{op}$, $E_{op}$);
    5.4. $G_{op}$ = Mark_Edge ($G_{op}$, $E_{pr}$);
    5.5. $G_{op}$ = Check_Cycle ($G_{op}$, $G_t$);
    5.6. $G_t$ = Insert memory access nodes ($G_{op}$, $G_t$);
4. $S$ = Schedule ALU instructions ($G_{op}$);
5. $S$ = Schedule other instructions ($S$, $G_t$);
6. $S$ = Retime the initial scheduling result ($S$);

Figure 4.3. The entire scheduling steps of RSSP.

### 4.3.1 MDFG Construction

In the first part we construct the MDFG from the high-level language using the same mechanism as in RSF, RST, and RSP. During the MDFG construction operands are stored in memory, and reloaded into registers only when they are required for use. This mechanism appears burdensome but is really used in some DSP compilers, because the number of registers is limited in DSP and memory is the only safe repository. In addition to constructing the MDFG, variables are also partitioned by four mechanisms proposed in RSVR, RSF, RST, and RSP in the first part. Constants are stored in both X and Y memory banks at specific locations in advance.

### 4.3.2 TDAG Construction

If all instructions in the MDFG are scheduled, apparently that accumulator and register spills will not occur. But scheduling according to this complicated MDFG will degrade the computational performance, because ALU results can be temporarily stored in accumulators or registers instead of directly written back to memory. Hence, in RSSP we define a *translated data acyclic graph* (*TDAG*) constructed from the

1. **Input:** $G_c = (V_c, E_c, X_c, d, P_c)$;
2. **Output:** $G_t = (V_t, E_t, X_t, P_t)$;
3. $V_t = V_c$; $E_t = \{e \mid e \in E_c, d(e) = (0,\ldots, 0)\}$;
4. Assume that $v_i, v_j, v_k, v_l, v_m, v_n \in V_c$, and their types are $M$, $A$, $S$, $L$, $M$, and $A$ respectively;
   - 4.1. **If** ($\exists$ a path $v_i \rightarrow v_k \rightarrow v_l \rightarrow v_m \in G_t$) // $M \rightarrow M$
     - Insert node $v_x$ into $V_t$ (set $P_t(v_x) = T$); Insert edge $e_{ix}$ into $E_t$;
     - $\forall\ e_{lm} \in E_t$ delete edges $e_{lm}$ from $E_t$, insert edges $e_{xm}$ into $E_t$;
     - Delete node $v_l$ from $V_t$; Delete edge $e_{kl}$ from $E_t$;
     - **If** ($\exists\ e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain $v_k$, $e_{ik}$
     - **Else** delete node $v_k$ from $V_t$, delete edge $e_{ik}$ from $E_t$;
   - 4.2. **If** ($\exists$ a path $v_j \rightarrow v_k \rightarrow v_l \rightarrow v_m \in G_t$) // $A \rightarrow M$
     - Insert node $v_x$ into $V_t$ (set $P_t(v_x) = T$); Insert edge $e_{jx}$ into $E_t$;
     - $\forall\ e_{lm} \in E_t$ delete edges $e_{lm}$ from $E_t$, insert edges $e_{xm}$ into $E_t$;
     - Delete node $v_l$ from $V_t$; Delete edge $e_{kl}$ from $E_t$;
     - **If** ($\exists\ e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain $v_k$, $e_{jk}$
     - **Else** delete node $v_k$ from $V_t$, delete edge $e_{jk}$ from $E_t$;
   - 4.3. **If** ($\exists$ a path $v_i \rightarrow v_k \rightarrow v_l \rightarrow v_n \in G_t$) // $M \rightarrow A$
     - $\forall\ e_{ln} \in E_t$ delete edges $e_{ln}$ from $E_t$, insert edges $e_{in}$ into $E_t$;
     - Delete node $v_l$ from $V_t$; Delete edge $e_{kl}$ from $E_t$;
     - **If** ($\exists\ e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain $v_k$, $e_{ik}$
     - **Else** delete node $v_k$ from $V_t$, delete edge $e_{ik}$ from $E_t$;
   - 4.4. **If** ($\exists$ a path $v_j \rightarrow v_k \rightarrow v_l \rightarrow v_n \in G_t$) // $A \rightarrow A$
     - $\forall\ e_{ln} \in E_t$ delete edges $e_{ln}$ from $E_t$, insert edges $e_{jn}$ into $E_t$;
     - Delete node $v_l$ from $V_t$; Delete edge $e_{kl}$ from $E_t$;
     - **If** ($\exists\ e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain $v_k$, $e_{jk}$
     - **Else** delete node $v_k$ from $V_t$, delete edge $e_{jk}$ from $E_t$;
5. $X_t(e) = X_c(e)$, if $e$ is remained in $E_t$;
6. $P_t(v) = P_c(v)$, if $v$ is remained in $V_t$;
7. **Return** $G_t$;

Figure 4.4. The TDAG constructing algorithm.

original MDFG, which is aimed at removing possible unnecessary memory accesses. The formal definition of the TDAG is given below.

**Definition 4.1** A *translated data acyclic graph* (*TDAG*) $G = (V, E, X, P)$ is a node-weighted and edge-weighted direct graph, where $V$ is the set of computation nodes; $E \subseteq V \times V$ is the edge set that defines the precedence relations over the nodes in $V$; $X(e)$ represents the variable accessed by an edge $e$; $P(v)$ represents the type of node $v$ (see Figure 2.1(c)).

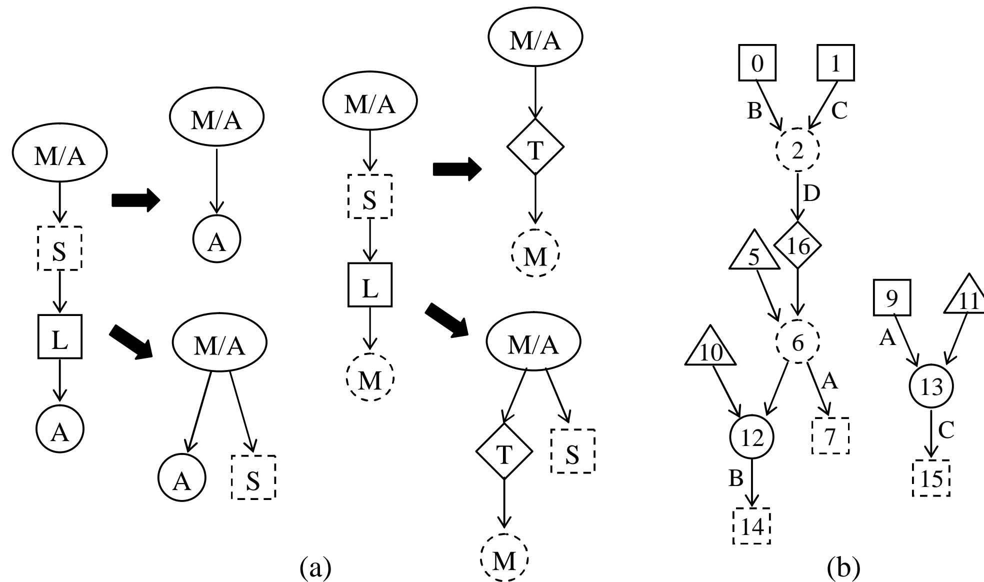



Figure 4.5. (a) Two cases of removing memory accesses, (b) TDAG of MDFG in Figure 2.1(b).

Figure 4.4 shows the TDAG construction algorithm. For a given MDFG, the first step is to remove edges with non-zero delays. Then, for an ALU result written back and reloaded in the same iteration, it can be temporarily stored in an accumulator to remove the corresponding instructions with types *S* and *L*. If an ALU result will be used in latter iteration, its corresponding store variable instruction must be retained. In addition, in Motorola DSP56000 both source operands of a multiplication must always be registers. Hence, a register transfer instruction is added if necessary to ensure all source operands are stored in registers. Figure 4.5(a) shows two cases of removing memory accesses, and Figure 4.5(b) is the TDAG transferred from the MDFG shown in Figure 2.1(b). Note that during constructing TDAG we simply assume unlimited numbers of accumulators and registers. That is, the TDAG only contains absolutely necessary memory accesses, which is beneficial to decrease the instruction count.

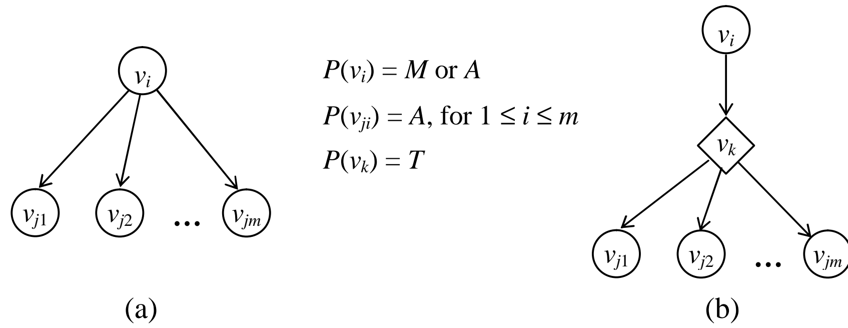



Figure 4.6. (a) A TDAG fragment, (b) after inserting the register transfer $v_k$.

1. Input: $G = (V, E, X, P)$, $n$;
2. Output: $G_t = (V_t, E_t, X_t, P_t)$;
3. $G_t = G$;
4. Suppose that $v_i \in V_t$ and $P_t(v_i) = M$ or $A$;
5. If ($v_i$ has more than $n$ immediate successors $v_1, \ldots, v_m$ with type $A$)
   Delete edges $e_{i1}, \ldots, e_{im}$ from $E_t$;
   Insert nodes $v_x$ into $V_t$ (set $P_t(v_x) = T$);
   Insert edges $e_{x1}, \ldots, e_{xm}$ into $E_t$;
6. Return $G_t$;

Figure 4.7. The register transfer inserting algorithm.

### 4.3.3 TDAG Modification

One of the main goals of RSSP is to avoid accumulator and register spills by predicting their occurrence in advance. In the third part of RSSP we analyze and modify the TDAG to resolve accumulator spills. Register spills will be dealt with in the fifth part later.

Three main steps are required for this TDAG modification: insertion of register transfers, analysis of TDAG, and insertion of memory accesses. Recall that we assume unlimited number of accumulators when constructing the TDAG. Hence, an ALU instructions with types *M*/*A* may have many immediate successors with type *A* in the TDAG. As shown in Figure 4.6(a), the ALU result of $v_j$ is a source operand of all additions $v_{j1}$ to $v_{jm}$. In this case we add a register transfer $v_k$ if $m > n$, if the architecture only consists of one data ALU and $n$ accumulators. Figure 4.6(b) contains the TDAG after inserting $v_k$ and the corresponding algorithm is listed in Figure 4.7.

1. Input: $G = (V, E, X, P)$;
2. Output: $G_{op} = (V_{op}, E_{op}, S)$, $G_{pr} = (V_{op}, E_{pr}, S)$;
3. $V_{op} = \{v \mid v \in V, P(v) = M \text{ or } A\}$;
4. $E_{op} = \{e_{ij} \mid e_{ij} \in E, v_i, v_j \in V_{op}\}$;
5. $E_{pr} = \{e_{ij} \mid e_{ji} \in E_{op}\}$;
6. $S(e) = \{F \mid e \in E_{op} \text{ and } E_{pr}\}$;
7. Return $(G_{op}, G_{pr})$;

Figure 4.8. The $G_{op}$ and $G_{pr}$ constructing algorithm.

1. Input: $G = (V, E, S)$, $E_i$;
2. Output: $G_r = (V_r, E_r, S_r)$;
3. $G_r = G$;
4. $label(v) = N$, $\forall\, v \in V$;
5. $label(v) = S$, $\forall\, v$ doesn't have any immediate predecessor;
6. While ($\exists\, label(v) == N$)
   - 6.1. $\exists\, e_{ij} \in E_i$, such that $v_i$ is the only immediate predecessor of $v_j$
     $$label(v_j) = \begin{cases} V & \text{if } label(v_i) == S \\ label(v_i) & \text{otherwise} \end{cases}$$
   - 6.2. $\exists\, e_{ik}, e_{jk} \in E_i$
     If ($label(v_i) == N$ or $label(v_j) == N$) $label(v_k) = N$;
     else if ($label(v_i) == S$ or $label(v_j) == S$) $label(v_k) = H$;
     else if ($label(v_i) == V$ and $label(v_j) == V$) $label(v_k) = H$;
     else $label(v_k) = G$; $S_r(e_{jk}) = T$;
7. Return $G_r$;

Figure 4.9. The *Mark_Edge* algorithm.

Then, we analyze TDAG topologies too predict the occurrence of accumulator spill. Two intermediate DAGs $G_{op}$ and $G_{pr}$, defined as follows, are constructed using algorithm listed in Figure 4.8. Initially we set $S(e) = F$ for all edges in $G_{op}$ and $G_{pr}$ to indicate no accumulator spill will occur. After applying algorithms listed in Figures 4.9 and 4.10, some edges in $G_{op}$ will be set $S(e) = T$ to represent the occurrence of accumulator spill. Figure 4.11 shows two $G_{op}$ fragments with accumulator spills that will be checked by algorithms *Mark_Edge* and *Check_Cycle*, respectively. Note that *Mark_Edge* and *Check_Cycle* algorithms are designed based on our analyses of TDAG topologies. That is, they only suit the architecture consisting of data ALU and two accumulators, such as the DSP56000.

1. Input: $G = (V, E, S)$, $G_t = (V_t, E_t, X_t, P_t)$;
2. Output: $G_r = (V_r, E_r, S_r)$;
3. $G_r = G$;
4. Delete edge $e$ from $E$, such that $S(e) = T$;
5. $\forall\, e_{ij} \in E_t$ such that $P_t(v_j) = T$
   $\forall\, e_{jk} \in E_t$, insert edge $e_{ik}$ into $E$ (set $S(e_{ik}) = X$);
6. Remove edge direction in $G$;
7. Level each node $v \in V$ ($level(v)$ indicates the longest path length from $v$ to any root node; $level(v) = 1$ if $v$ is a root node)
8. If ($\exists$ a cycle $v_i \rightarrow v_{i+1} \rightarrow \ldots \rightarrow v_k \rightarrow v_{k+1} \rightarrow \ldots \rightarrow v_j \rightarrow v_i$ in $G$)
   8.1. Suppose $v_i$ has the smallest $level(v)$ value in this path;
   8.2. If (($level(v_i) < level(v_{i+1})$ in path $v_i \rightarrow \ldots \rightarrow v_k$) and ($level(v_k) < level(v_{k+1})$ in path $v_k \rightarrow \ldots \rightarrow v_i$)) $S_r(e_{ji}) = T$;
   else $S_r(e_{ij}) = T$, $\forall\, level(v) = level(v_i)$ in this path;
9. Return $G_r$;

Figure 4.10. The *Check_Cycle* algorithm.

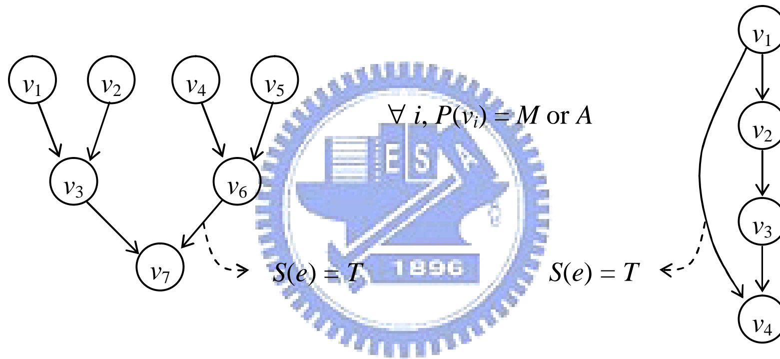



Figure 4.11. Two $G_{op}$ fragments with accumulator spill.

**Definition 4.2** A DAG $G_{op} = (V, E, S)$ is a direct graph, where $V$ is the node set representing ALU instructions; $E \subseteq V \times V$ is the edge set that defines the precedence relations over the nodes in $V$; $S(e)$ is an edge mark that represents two nodes that must be scheduled at separate time steps or not.

**Definition 4.3** A DAG $G_{op}$, corresponds to an undirected DAG $G_{pr} = (V, E, S)$ with the same topology and characteristics.

Finally, for an edge in $G_{op}$ with $S(e) = T$, two memory accesses with types $S$ and $L$ are inserted into the TDAG using algorithm listed in Figure 4.12. After completing steps 3.1~3.6 listed in Figure 4.3, we will get a modified TDAG which can be scheduled without any accumulator spill.

1. Input: $G = (V, E, S)$, $G_1 = (V_1, E_1, X_1, P_1)$;
2. Output: $G_t = (V_t, E_t, X_t, P_t)$;
3. $G_t = G_1$;
4. $\forall\ e_{ij} \in E$ such that $S(e_{ij}) == T$
   Delete edge $e_{ij}$ from $E_t$;
   Insert nodes $v_s$, $v_l$ into $V_t$ (set $P_t(v_s) = S$, $P_t(v_l) = L$);
   Insert edges $e_{is}$, $e_{sl}$, $e_{lj}$ into $E_t$ (set $X_t(e_{is}) = t$, $X_t(e_{lj}) = t$, where $t$ is a temporary variable);
5. Return $G_t$;

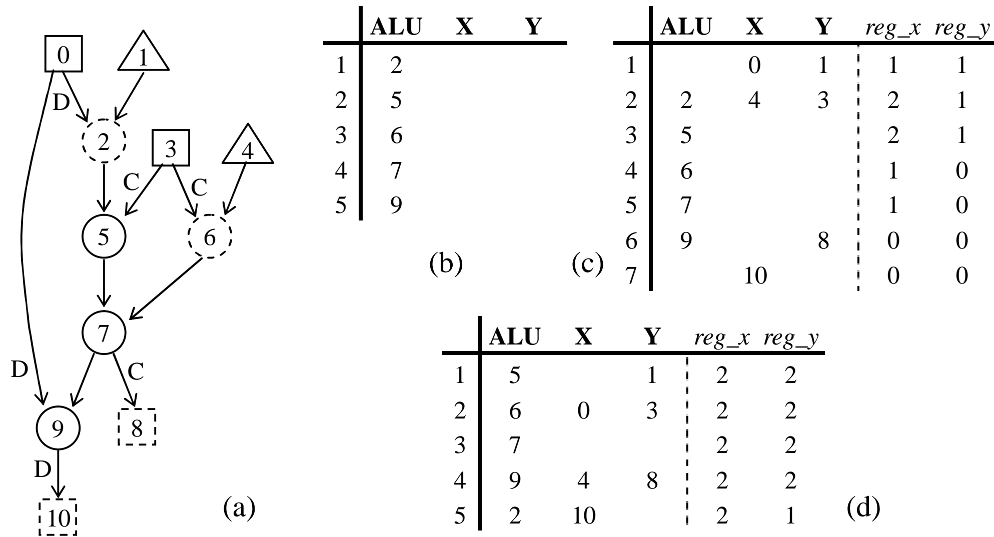
Figure 4.12. The memory access inserting algorithm.

### 4.3.4 ALU Instruction Scheduling

In the fourth part of RSSP, ALU instructions are scheduled considering the nature of Motorola DSP56000. We first list principles that a correct schedule must satisfy as follows, and propose scheduling rules based on these principles. For convenience, we only permit a variable or constant loaded from memory to be stored in a register.

1. For an edge $e_{ij}$ of a TDAG, if $P(v_i) = L/C/T$ and $P(v_j) = M/A$, $v_j$ must be executed no later than the next two instruction (in the same memory bank as $v_i$) with type $L/C/T$.
2. For an edge $e_{ij}$ of a TDAG, if $P(v_i) = M/A$ and $P(v_j) = S$, $v_j$ must be executed no later than the next two instruction with type $M/A$.
3. For an edge $e_{ij}$ of a TDAG, if $P(v_i) = M/A$ and $P(v_j) = M/A$, at most one ALU instruction can be executed between $v_i$ and $v_j$.

Basically, ALU instructions are scheduled using list scheduling based on $G_{op}$ $(V, E, S)$. Recall that the Motorola DSP56000 consists of one data ALU and two accumulators, and all instructions are completed in one time step. For an edge $e_{ij} \in E$, its edge mark $S(e_{ij})$ may be $F$, $T$, or $X$, which indicates different rules for scheduling $v_i$ and $v_j$. Assume that $v_i \in V$ is scheduled at time step $i$, and the ALU result $rt_i$ of $v_i$ is stored in accumulator $acc_i$. If $S(e_{ij}) = F/X$, $v_j$ must be scheduled at time step $i$+1 or $i$+2




(b)

| | ALU | X | Y |
|---|---|---|---|
| 1 | 2 | | |
| 2 | 5 | | |
| 3 | 6 | | |
| 4 | 7 | | |
| 5 | 9 | | |

(c)

| | ALU | X | Y | *reg_x* | *reg_y* |
|---|---|---|---|---|---|
| 1 | | 0 | 1 | 1 | 1 |
| 2 | 2 | 4 | 3 | 2 | 1 |
| 3 | 5 | | | 2 | 1 |
| 4 | 6 | | | 1 | 0 |
| 5 | 7 | | | 1 | 0 |
| 6 | 9 | | 8 | 0 | 0 |
| 7 | | 10 | | 0 | 0 |

(d)

| | ALU | X | Y | *reg_x* | *reg_y* |
|---|---|---|---|---|---|
| 1 | 5 | | 1 | 2 | 2 |
| 2 | 6 | 0 | 3 | 2 | 2 |
| 3 | 7 | | | 2 | 2 |
| 4 | 9 | 4 | 8 | 2 | 2 |
| 5 | 2 | 10 | | 2 | 1 |

Figure 4.13. Scheduling steps of RSSA. (a) An TDAG example, (b) ALU instructions only, (c) initial scheduling result, (d) retimed scheduling result.

to prevent $rt_i$ being recovered before being used. Conversely, if $S(e_{ij}) = T$, $v_j$ can be scheduled at time step later than $i$+2, because $rt_i$ will be transferred to register $reg_i$. In addition, if $S(e_{ij}) = X$ and $v_j$ is scheduled at time step $i$+1, an idle time step is inserted between $v_i$ and $v_j$ for scheduling register transfer instruction further. Because we have already considered the occurrences of accumulator spill, all ALU instructions can be scheduled exactly according to the above three rules. These rules for scheduling ALU instructions are essentially equivalent to the third principle listed above. Figure 4.13 (a) shows a TDAG example, and its scheduling result of the ALU instructions only is listed in Figure 4.13(b).

### 4.3.5 Other Instruction Scheduling

After scheduling ALU instructions, other instructions including memory accesses and register transfers are scheduled based on the modified TDAG. Meanwhile, we consider the limited number of registers during instruction scheduling, therefore no extra action is required to determine and deal with the occurrences of register spill. In RSSP, we use two variables $reg_x(t)$ and $reg_y(t)$ to record the number of registers

been occupied at time step $t$ for X and Y memory banks, respectively. When scheduling each instruction, these two variables are dynamically updated. Apparently, if we can generate a schedule where $reg_x(t)$ and $reg_y(t)$ do not exceed the limited number of registers for all time steps, register spills will not occur.

For a correct schedule, an operand residing in an accumulator/register obviously cannot be overwritten before being used. Recall that all instructions are completed in one time step in Motorola DSP56000. That is, if a variable (or constant) is loaded from memory at time step $i$ and used at time step $j$, it will occupy a register from time step $i$ to $j$-1. Similarly, an ALU result will occupy a register from time step $i$ to $j$–1 if it is transferred from an accumulator at time step $i$ and used at time step $j$. We conclude scheduling rules for memory accesses and register transfers as follows.

1. According to the execution sequence of ALU instructions, schedule their predecessors as soon as possible.
2. Principles 1~2 listed in subsection 4.3.4 must be satisfied, and $reg_x(t)$ and $reg_y(t)$ cannot exceed the number of registers for any time step.
3. If a variable is stored and loaded at consecutive time steps, these two memory accesses can be replaced by a single register transfer.
4. If a memory access or register transfer cannot be scheduled successfully due to insufficient registers, a variable currently occupied a register must be overwritten and reloaded again when required.
5. If an overwritten variable is not used after being transferred from the accumulator, the corresponding register transfer is replaced by a store variable instruction.

Figure 4.13(c) shows the scheduling result of the TDAG in Figure 4.13(a). Finally, because we have already considered accumulator and register spills, an appropriate assignment of physical accumulators and registers will exist.

### 4.3.6 Initial Schedule Retiming

After generating the initial scheduling result, we apply the multi-dimensional rotation scheduling to explore the instruction-level parallelism between different iterations. Retimed instructions in each rotation phase are originally rescheduled as soon as possible to reduce the schedule length. But in RSSP we redefine the rescheduling rules for retimed instructions, in order to guarantee that the number of registers used at all time steps will not exceed the limitation. Assume that the length of the initial schedule is *len*. In the following we present conditions so that a retimed instruction can be rescheduled at time step *i*. We will reschedule a retimed instruction at the earliest time step that satisfies all conditions listed below. Moreover, because constants are stored in both X and Y data memory banks in advance, a retimed load constant instruction can be rescheduled at any data memory bank to achieve higher performance. The retimed scheduling result of Figure 4.13(c) is shown in Figure 4.13(d).

1. A retimed instruction with type *L*/*C* must occupy a register from time step *i* to *len*, because this value or constant will be used for a later iteration.
2. A retimed instruction with type *T* must occupy a register from time step *i* to *len*, because this ALU result will be used for a later iteration. In addition, the second principle listed in subsection 4.3.4 has to be satisfied.
3. Rescheduling a retimed instruction with type *S* must satisfy the second principle listed in subsection 4.3.4.
4. Rescheduling a retimed instruction with type *M*/*A* must satisfy the first or third principle listed in subsection 4.3.4. In addition, $reg_x(t)$ and $reg_y(t)$ are updated after rescheduling this ALU instruction.

Table 4.1. Experimental results for a single iteration in the repetitive pattern.

| | Cho | Malik | Shiue | RSVR | RSF | RST | RSP | RSSP | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | RSVR | RSF | RST | RSP |
| Wave Digital Filter | 7 | 9 | 9 | 8 | 6 | 8.5 | 6 | 6 | 5 | 5.5 | 5.5 |
| Filter | 8 | 13 | 13 | 9 | 11.5 | 9 | 6 | 6 | 5.5 | 5 | 4.5 |
| IIR Filter 2D | 20 | 29 | 33 | 25 | 27.5 | 28 | 24.5 | 16 | 16 | 16 | 16 |
| forward-substitution | 7 | 12 | 12 | 9 | 10 | 11.5 | 7.5 | 5 | 5.5 | 5 | 5 |
| THCS | 6 | 8 | 8 | 6 | 6.5 | 5.5 | 5 | 4 | 4 | 4 | 4 |
| DFT | 14 | 21 | 21 | 18 | 21 | 18.5 | 18 | 13 | 12.5 | 13 | 12.5 |
| Floyd-Steinberg | 20 | 36 | 37 | 29 | 32.5 | 30.5 | 23.5 | 18 | 17.5 | 17 | 17 |
| Transmission Line | 15 | 20 | 21 | 19 | 18 | 25 | 18 | 12 | 12 | 12 | 12 |

### 4.4 Performance Evaluations [37]

In the following, we select several MDFGs represented DSP applications to evaluate methods including Cho [9], Malik [17], Shiue [20], and RSSP. Because four variable partition mechanisms proposed in RSVR, RSF, RST, and RSP can be applied in RSSP, three scheduling results are derived from RSSP. Meanwhile, scheduling results obtained by RSVR [30], RSF [14], RST [14], and RSP [36] are used for comparison, after inserting necessary spill codes. Among these methods, Cho [9] and RSSP are scheduled based on TDAG, and others are scheduled based on MDFG. Suppose that the target architecture is the Motorola DSP56000, which consists of one data ALU, two data memory banks, two accumulators, and four registers. All ALU instructions and memory accesses can be completed in one time step.

Similar as in chapter 3, we evaluate performances focus on both a single iteration in the repetitive pattern and the entire retimed loop. Table 4.1 lists schedule lengths obtained from different methods for a single iteration in the repetitive pattern. From these results, it is obvious that methods scheduled based on TDAG outperform methods scheduled based on MDFG. The direct reason is that we remove additional memory accesses during constructing the TDAG in advance, which decrease the

number of instructions actually been scheduled. Furthermore, both RSSP and Cho [9] are scheduled based on TDAG, and our RSSP can achieve shorter schedule lengths. This is because the retiming technique is applied in RSSP, in order to explore the potential instruction-level parallelism between different iterations. The effectiveness among four scheduling results derived from RSSP is very similar for most applications. This indicates that RSSP is sufficiently flexible and can achieve reasonable results using various variable partition mechanisms.

For the entire retimed loop, formulas listed in appendix A can be directly used to calculate the overall schedule length for RSSP. Figures 4.14 and 4.15 show the overall schedule lengths of every application. For each application, we only sketch the best result among methods RSVR, RSF, RST, and RSP. Four scheduling results are derived from RSSP with different variable partition mechanisms, and we also only sketch the best one. As shown in these figures, basically these results are the same as the evaluations focused on a single iteration in the repetitive pattern. Meanwhile, as the size of nested loop increases, the difference in overall schedule lengths between all methods increases. That is, the proposed RSSP can save more execution time in larger problem sizes.

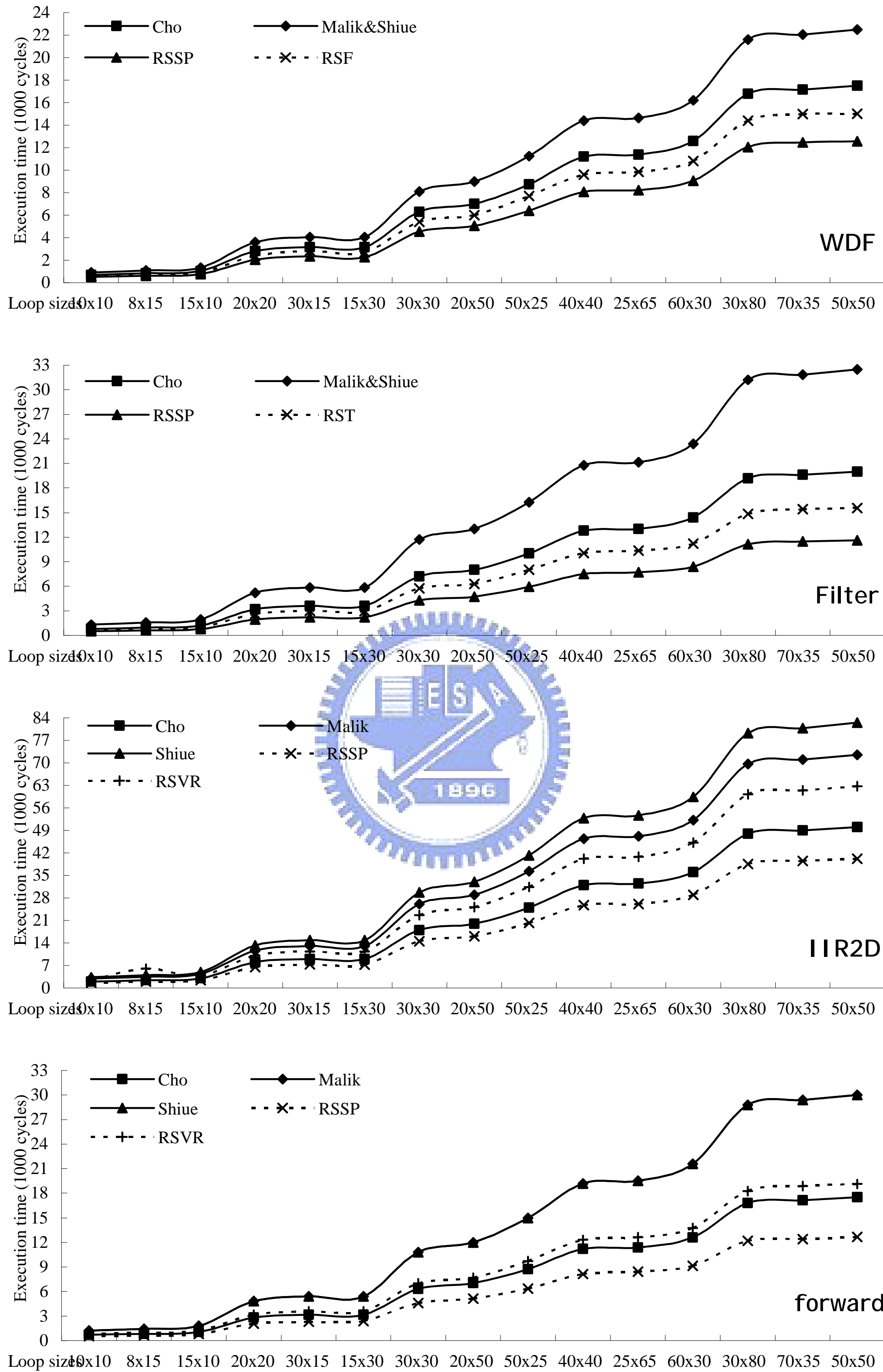
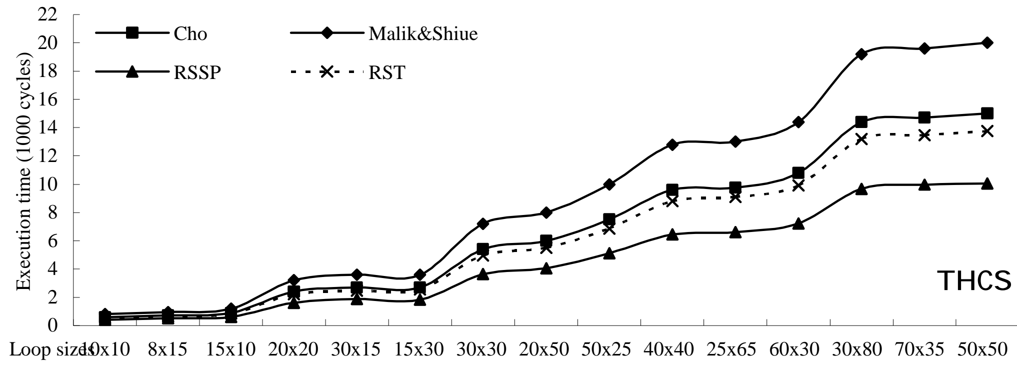
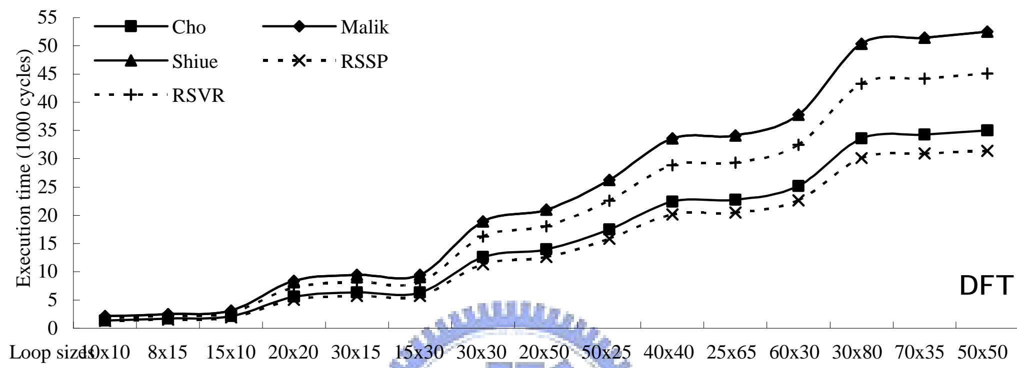
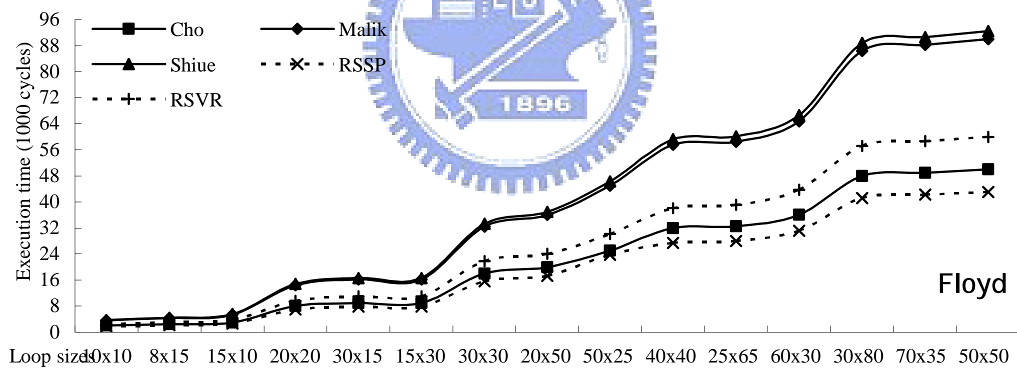



Figure 4.14. Overall schedule lengths of DSP applications.



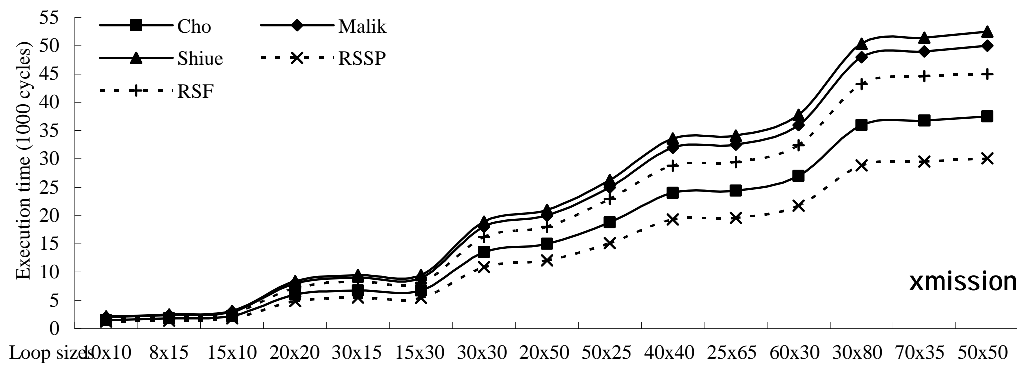




Figure 4.15. Overall schedule lengths of DSP applications.

# Chapter 5. Effective Generalized Code Generation Method

According to descriptions in chapter 4, RSSP looks really effective and efficient, but it is not scalable and specifically designed for Motorola DSP56000. In this chapter, we want to extend it to a more general version, which is suitable for various DSPs with similar architectural features. For the generalized code generation method, in addition to achieving shorter schedule length, we take fewer spill codes as the second scheduling goal due to its importance for DSP with heterogeneous register sets. Furthermore, in order to deep study the influence of differing number of resources on the scheduling result, we also define a parameterized machine model to simulate architectures with different number of resources. In section 5.1 we present the hypothetical machine model, and lists design motivations in section 5.2. Section 5.3 contains detailed steps of the generalized method. In section 5.4, we describe how to apply the proposed hypothetical machine model and code generation method to real DSP families. Finally, some performance evaluations are shown in section 5.5.

## 5.1 Hypothetical Machine Model [38]

As mentioned before, we need a parameterized architecture to model a scalable DSP with multiple data memory banks and heterogeneous register sets. Many parameterized architecture models have been developed to explore and investigate advanced compiler and architecture research [53-57]. Most of them are oriented towards *EPIC* (*explicitly parallel instruction computing*) or superscalar architectures, and support novel features such as prediction, control and data speculation, and memory hierarchy. However, none of them supports both architectural features we require, so here we define a hypothetical machine model in which more resources will be included.

Table 5.1 lists architectural features of some popular DSP families, especially for their data paths. Our parameterized machine model, defined as follows, is design to cover all these architectural features as far as possible. Assume that it contains $N$ data memory banks ($M_1 \ldots M_N$), $k$ function units ($FU_1 \ldots FU_k$), $k \times m$ accumulators ($acc_{11} \ldots acc_{1m}$, …, $acc_{k1} \ldots acc_{km}$), and $N \times n$ registers ($reg_{11} \ldots reg_{1n}$, …, $reg_{N1} \ldots reg_{Nn}$). All function units are identical and parallel executed, which can execute all ALU instructions including the multiplication. Source operands for all ALU instructions, except multiplication, must be registers or accumulators and destination operands (or ALU results) must always be accumulators. For the multiplication source operands must always be registers. This restriction is inherited from Motorola families. We assume that an instruction may require several time steps to complete in our hypothetical machine model.

Due to the nature of the target architecture, many irregular DSPs have parallel conditions to define which instructions can be performed in parallel. In the following, we list parallel conditions in our hypothetical machine model: (1) up to $N$ independent move operations and $k$ ALU instructions can be executed simultaneously in one cycle; (2) the $N$ move operations reference data in different data memory banks; (3) the $M_i$ memory access load into restricted locations $reg_{i1} \ldots reg_{in}$, or all accumulators; (4) the ALU result generated by $FU_i$ must be stored in accumulators $acc_{i1} \ldots acc_{im}$; (5) $2k$ source operands must be read from different registers or accumulators.

According to above descriptions, our hypothetical machine model is basically extended from Motorola DSP56000. However, with minor modifications, this model and the general code generation method can satisfy all architectural features listed in Table 5.1. We will further describe how to apply the machine model and general method to real DSP families in section 5.4. On the other hand, because our main purpose is to design code generation method, our hypothetical machine model is

Table 5.1. Architectural features of some popular DSPs.

| | | data memory bank | function unit | register set |
|---|---|---|---|---|
| Motorola | DSP56000<br>DSP56001 | * X/Y memory banks with independent data buses<br>* 2 memory reads/ writes can be performed simultaneously | * 1 MAC | * 4 24-bit reg. (X0, X1, Y0, Y1) for source operands<br>* 2 56-bit acc. (A, B) for source/destination operands |
| | DSP56300 family | | | |
| | DSP56800 family | * 1 memory bank<br>* 2 memory reads can be performed simultaneously using independent buses | | * 3 16-bit reg. (X0, Y0, Y1) for source operands<br>* 2 36-bit acc. (A, B) for source/destination operands |
| Analog Device | ADSP2100 family | | * 1 ALU, 1 MAC, 1 shifter<br>* Parallel execute | * ALU:<br>- 4 16-bit reg. (AX0, AX1, AY0, AY1) for source operands<br>- 1 16-bit reg. (AF/ AR) for source/ destination operand<br>* MAC:<br>- 4 16-bit reg. (MX0, MX1, MY0, MY1) for source operands<br>- 1 40-bit reg. (MR) for source/destination operand |
| NEC | uPD7701x family | * X/Y memory banks with independent data buses<br>* 2 memory reads/ writes can be performed simultaneously | * 1 ALU, 1 MAC, 1 shifter<br>* One unit executes at a time | * 8 40-bit reg. (R0~R7)<br>* Homogeneous |
| TI | TMS320C62x<br>TMS320C67x<br>family | * 1 memory bank<br>* 2 memory reads/ writes can be performed simultaneously using independent buses | * 2 multipliers, 6 ALUs<br>* Parallel execute | * 32 32-bit reg. (A0~ A15, B0~B15)<br>* 2 register files<br>* Homogeneous |
| | TMS320C64x family | | | * 64 32-bit reg. (A0~ A31, B0~B31)<br>* 2 register files<br>* Homogeneous |

defined focus on the data path of the DSP architecture. If we want to study other issues on the DSP architecture later, it is not difficult to further extend our model for the target DSP to include necessary resources.

### 5.2 Design Motivations [37]

In RSSP we list two design motivations: performing variable partition before code compaction and predicting accumulator spills in advance. The former is retained in the general method because it certainly can avoid the occurrence of spill codes. On the other hand, when the target architecture is no longer specific, to predict accumulator spills by topological analysis of the TDAG becomes much more difficult and inaccurate. Therefore, in the general method we design another mechanism to resolve accumulator spills and not predict their occurrences.

Next, we focus on the resolution of accumulator/register spills. When a register spill occurs, the overwritten variable must be stored back to memory and reloaded when required. As for an accumulator spill, in addition to memory, the overwritten ALU results also can be temporarily transferred to an available register before being used. Consider the example shown in Figure 5.1. Assume that the target architecture is the Motorola DSP56000, and Figure 5.1(c) lists the schedule obtained using method described in [17]. In this schedule we find that when an accumulator spill occurs at $I_3$, the overwritten ALU results, $m$, must be stored back to memory because all variables residing in four registers will be accessed later. Later another memory access is added to reload $m$ before $I_8$. However, if instruction $i_{13}$ is moved to $I_5$ as shown in Figure 5.1(d), $m$ can be transferred to register Y0 instead of memory to eliminate one extra spill cost. From this example, we find that it is preferable to transfer an overwritten ALU results to a register when an accumulator spill occurs. In the general method we will follow this principle to determine and resolve accumulator spills.

| MOVE | a, reg0 | ($i_0$) |
|---|---|---|
| MOVE | b, reg1 | ($i_1$) |
| MPY | acc0, reg0, reg1 | ($i_2$) |
| MOVE | c, reg2 | ($i_3$) |
| ADD | acc1, acc0, reg2 | ($i_4$) |
| MOVE | d, reg3 | ($i_5$) |
| ADD | acc2, acc1, reg3 | ($i_6$) |
| MOVE | acc2, p | ($i_7$) |
| MOVE | e, reg4 | ($i_8$) |
| ADD | acc3, acc1, reg4 | ($i_9$) |
| ADD | acc4, acc3, reg3 | ($i_{10}$) |
| MOVE | acc4, r | ($i_{11}$) |
| ADD | acc5, acc3, reg0 | ($i_{12}$) |
| MOVE | f, reg5 | ($i_{13}$) |
| ADD | acc6, acc5, reg5 | ($i_{14}$) |
| ADD | acc7, acc0, acc6 | ($i_{15}$) |
| MOVE | acc7, t | ($i_{16}$) |

(a)

| MOVE | | X: a, reg0 | Y: b, reg1 | ($I_0$) |
|---|---|---|---|---|
| MPY | acc0, reg0, reg1 | X: c, reg2 | Y: d, reg3 | ($I_1$) |
| ADD | acc1, acc0, reg2 | X: e, reg4 | Y: f, reg5 | ($I_2$) |
| ADD | acc2, acc1, reg3 | | | ($I_3$) |
| ADD | acc3, acc1, reg4 | acc2, X: p | | ($I_4$) |
| ADD | acc4, acc3, reg3 | | | ($I_5$) |
| ADD | acc5, acc3, reg0 | | acc4, Y: r | ($I_6$) |
| ADD | acc6, acc5, reg4 | | | ($I_7$) |
| ADD | acc7, acc0, acc6 | | | ($I_8$) |
| MOVE | | acc7, X: t | | ($I_9$) |

(b)

| MOVE | | X: a, X0 | Y: b, Y0 |
|---|---|---|---|
| MPY | A, X0, Y0 | X: c, X1 | Y: d, Y1 |
| ADD | B, A, X1 | X: e, X1 | Y: f, Y0 |
| MOVE | | A, X: m | |
| ADD | A, B, Y1 | | |
| ADD | A, B, X1 | A, X: p | |
| ADD | B, A, Y1 | | |
| ADD | B, A, X0 | | B, Y: r |
| ADD | A, B, Y0 | | |
| MOVE | | X: m, X0 | |
| ADD | B, A, X0 | | |
| MOVE | | B, X: t | |

(c)

| MOVE | | X: a, X0 | Y: b, Y0 |
|---|---|---|---|
| MPY | A, X0, Y0 | X: c, X1 | Y: d, Y1 |
| ADD | B, A, X1 | X: e, X1 | |
| MOVE | | | A, Y0 |
| ADD | A, B, Y1 | | |
| ADD | A, B, X1 | A, X: p | |
| ADD | B, A, Y1 | | Y: f, Y1 |
| ADD | B, A, X0 | | B, Y: r |
| ADD | A, B, Y1 | | |
| ADD | B, A, Y0 | | |
| MOVE | | B, X: t | |

(d)

Figure 5.1. An example of code compaction. (a) Uncompacted code, (b) compacted code, (c)(d) two scheduling results after resource assignment.

In order to give an overwritten ALU results higher priority to be transferred to a register, registers must be unfilled as far as possible while dealing with accumulator spills. But in methods [9, 17, 20] registers will be occupied by resource operands when accumulator spills are resolved, which is unfavorable for inserting additional register transfers. Thus, in the general method we divide the instruction scheduling phase into two steps, and let ALU instructions be scheduled before memory accesses. This mechanism makes registers remain unfilled during resolving accumulator spills, which is able to store overwritten ALU results and reduce additional spill costs.

Finally, we consider the time that accumulator/register spills been resolved in the entire code generation process. In methods [9, 17] they both perform this action at last, which may lengthen the schedule length like the example shown in Figure 5.1(c). The reason is that added spill codes cannot be scheduled in parallel with other instructions. Besides, although in methods [9, 17] they do not present detailed mechanisms to determine accumulator/register spills and insert spill codes, they definitely require an independent step to do this action. If the target architecture contains strict resource constraints, this step may cost considerable time. Thus, in the general method we want to design efficient mechanisms to determine and resolve accumulator/register spills, and integrate them into the instruction scheduling and code compaction phases. That is, we consider resource constraints during instruction scheduling, and then no extra action is required to determine accumulator/register spills and insert spill codes.

## 5.3 Rotation Scheduling with Spill Codes Avoiding (RSSA) [38]

In this section we introduce code generation method *rotation scheduling with spill codes avoiding* (*RSSA*), which is generalized from RSSP to suit various DSPs with similar architectural features. For RSSA it can handle target architectures with various numbers of function units, accumulators, registers, and data memory banks. As listed in Figure 5.2, RSSA contains five parts including MDFG construction, TDAG construction, instruction scheduling (I), instruction scheduling (II), and initial schedule retiming. First two parts are directly inherited from RSSP, and we present last three parts in some detail as follows.

### 5.3.1 Instruction Scheduling (I)

For a given loop written in high-level language, a TDAG is constructed after completing first two parts of RSSA. In the first instruction scheduling part, our goal is

1. $G_c$ = Construct MDFG;
   1.1. Partition variables to memory banks;
   1.2. Unfold or tile $G_c$ if necessary;
2. $G_t$ = Construct TDAG ($G_c$);
3. $S$ = Schedule all instructions except memory loads ($G_t$);
   3.1. $G_{op}$ = Construct DAG $G_{op}$ ($G_t$);
   3.2. $S$ = Schedule nodes in $G_{op}$ ($G_{op}$);
   3.3. $S$ = Determine and solve accumulator spills ($S$, $G_{op}$);
4. $S$ = Schedule memory load instructions ($S$, $G_t$);
5. $S$ = Retime the initial scheduling result ($S$, $G_t$);

Figure 5.2. The entire scheduling steps of RSSA.

to schedule all instructions except memory loads and resolve corresponding register/ accumulator spills. Steps marked 3.1~3.3 in Figure 5.2 belong to the first instruction scheduling part. In these steps we define an intermediate DAG $G_{op}$ which contains all nodes of the TDAG will be scheduled in this part.

**Definition 5.1** A DAG $G_{op} = (V, E, X, P)$ is a direct graph, where $V$ is the node set representing ALU instructions, register transfers, and store variables; $E \subseteq V \times V$ is the edge set that defines the precedence relations over nodes in $V$; $X(e)$ represents the variable accessed by an edge $e$; $P(v)$ represents the type of node $v$.

Figure 5.3 shows another TDAG example and its corresponding $G_{op}$. Next, we basically schedule nodes in $G_{op}$ using list scheduling method, assuming the number of accumulators/registers are unlimited. If an instruction $v_i$ requires $c$ time steps to complete, the destination operand of $v_i$ will be ready at the next $c$ time step. That is, for an edge $e_{ij} \in G_{op}$, if $v_i$ is scheduled at time step $t$, $v_j$ can be scheduled at time step $t+c$ or later. Assume that all instructions are completed in one time step. Figure 5.4(a) shows the scheduling result of $G_{op}$ in Figure 5.3(b) with only one function unit.

In the following we describe how to determine and resolve accumulator/register spills in our RSSA. Our idea is to calculate the number of accumulators/registers used at every time step, and variables listed in Table 5.2 are defined for our mechanism.

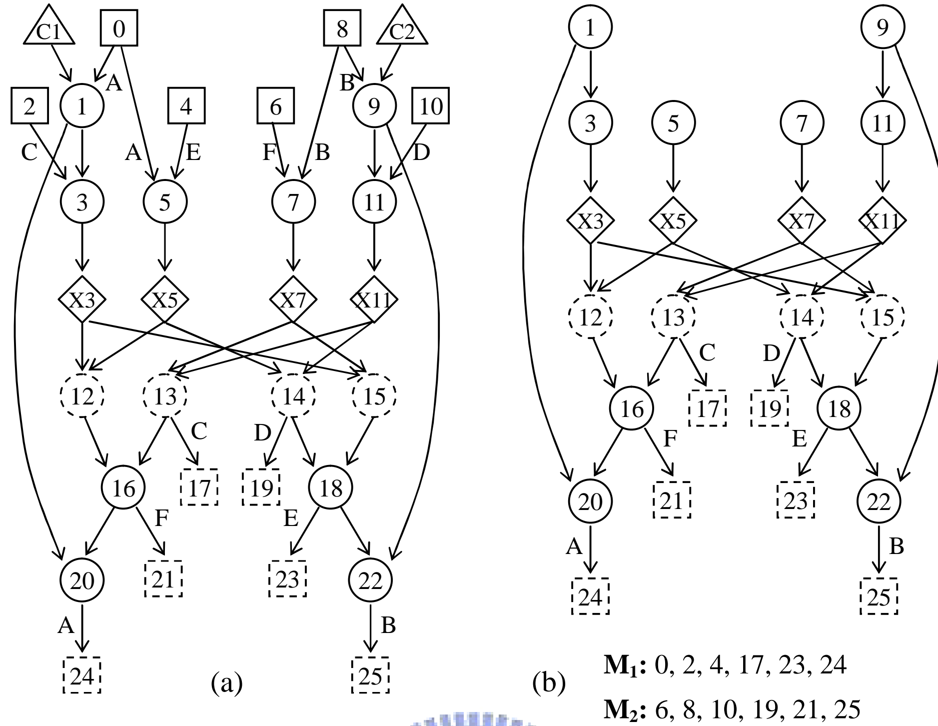



Figure 5.3. The $G_{op}$ example. (a) TDAG, (b) corresponding $G_{op}$.

| *t* | $FU_1$ | $M_1$ | $M_2$ | acc_1 | reg_1 | reg_2 | acclist_1 | reglist_1 | reglist_2 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | 1 | 0 | 0 | | | |
| 2 | 3 | | | 2 | 0 | 0 | 1 | | |
| 3 | 5 | X3 | | 2 | 1 | 0 | 1 | X3 | |
| 4 | 7 | | X5 | 2 | 1 | 1 | 1 | X3 | X5 |
| 5 | 9 | X7 | | 2 | 2 | 1 | 1 | X3, X7 | X5 |
| 6 | 11 | | | 3 | 2 | 1 | 1, 9 | X3, X7 | X5 |
| 7 | 12 | | X11 | 3 | 2 | 2 | 1, 9 | X3, X7 | X5, X11 |
| 8 | 13 | | | 4 | 2 | 2 | 1, 9, 12 | X3, X7 | X5, X11 |
| 9 | 16 | 17 | | 3 | 2 | 2 | 1, 9 | X3, X7 | X5, X11 |
| 10 | 20 | | 21 | 2 | 2 | 2 | 9 | X3, X7 | X5, X11 |
| 11 | 14 | 24 | | 2 | 2 | 0 | 9 | X3, X7 | |
| 12 | 15 | | 19 | 3 | 0 | 0 | 9, 14 | | |
| 13 | 18 | | | 2 | 0 | 0 | 9 | | |
| 14 | 22 | 23 | | 1 | 0 | 0 | | | |
| 15 | | | 25 | 0 | 0 | 0 | | | |

Figure 5.4(a). $G_{op}$ nodes only scheduling result of Figure 5.3(a), unlimited resource.

That is, if an ALU result is defined and used by instructions scheduled at time step *i* and *j*, respectively, it will occupy an accumulator from time steps *i* to *j*–1. Similarly, if an ALU result is transferred from an accumulator at time step *i* and used by the instruction scheduled at time step *j*, it will occupy a register from time steps *i* to *j*–1. Figure 5.4(a) also shows variables defined in Table 5.2 for that $G_{op}$ scheduling result.

Table 5.2. Variables defined for solving accumulator/register spills.

| Variable | Type | Definition |
|---|---|---|
| $sch(v)$ | integer | the time step that $G_{op}$ node $v$ is scheduled |
| $uselist(v)$ | integer list | time steps that $G_{op}$ nodes, which use node $v$, are scheduled |
| $acc_i(t)$ | integer | the number of accumulators $acc_{ij}$, for $j$ = 1…$m$, been occupied at time step $t$ |
| $acclist_i(t)$ | node list | $G_{op}$ nodes with types $M$ or $A$ whose generated ALU results reside in accumulators $acc_{ij}$, for $j$ = 1…$m$, at time step $t$, except the one that are scheduled in $FU_i$ at time step $t$ |
| $reg_i(t)$ | integer | the number of registers $reg_{ij}$, for $j$ = 1…$n$, been occupied at time step $t$ |
| $reglist_i(t)$ | node list | $G_{op}$ nodes with type $T$ whose transferred ALU results reside in registers $reg_{ij}$, for $j$ = 1…$n$, at time step $t$ |

#### 5.3.1.1 Mechanisms for Resolving Accumulator Spills

As described in section 5.2, we first transfer all overwritten ALU results to registers, and temporarily store them to memory only when the number of registers is insufficient. Suppose that an accumulator spill occurs at time step $t$ for accumulators $acc_{ij}$. In this situation we must select a node $v$ from $acclist_i(t)$ and transfer the ALU result generated by $v$ to a register. From all nodes in $acclist_i(t)$, we want to transfer an ALU result $rt$ that will release an accumulator with the longest time free of use. Next, an additional register transfer is scheduled at time step $t$ and all variables defined in Table 5.2 are updated accordingly. Note that the transferred value $rt$ will be ready at time step $t+c$ if a register transfer instruction requires $c$ time steps to complete. Therefore, if $rt$ is used by an instruction $u$ scheduled before time step $t+c$, additional time steps must be inserted to delay node $u$. These steps will be applied repeatedly until all accumulator spills are resolved.

Suppose the target architecture is the Motorola DSP56000, we use the schedule in Figure 5.4(a) to illustrate above steps. In this schedule accumulator spills occur at time steps 5~8 and 11. At time step 5, after checking the content of $uselist(u)$ for all nodes $u$ in $acclist_1(5)$, node 9 is selected to transfer its value. An additional register

| *t* | **FU$_1$** | **M$_1$** | **M$_2$** | acc_1 | reg_1 | reg_2 | acclist_1 | reglist_1 | reglist_2 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | 1 | 0 | 0 | | | |
| 2 | 3 | | | 2 | 0 | 0 | 1 | | |
| 3 | 5 | X3 | | 2 | 1 | 0 | 1 | X3 | |
| 4 | 7 | | X5 | 2 | 1 | 1 | 1 | X3 | X5 |
| 5 | 9 | X7 | | 2 | 2 | 1 | 1 | X3, X7 | X5 |
| 6 | 11 | | X9 | 2 | 2 | 2 | 1 | X3, X7 | X5, X9 |
| 7 | 12 | | X11 | 2 | 2 | 3 | 1 | X3, X7 | X5, X9, X11 |
| 8 | 13 | X1 | | 2 | 3 | 3 | 12 | X1, X3, X7 | X5, X9, X11 |
| 9 | 16 | 17 | | 1 | 3 | 3 | | X1, X3, X7 | X5, X9, X11 |
| 10 | 20 | | 21 | 1 | 2 | 3 | | X3, X7 | X5, X9, X11 |
| 11 | 14 | 24 | | 1 | 2 | 1 | | X3, X7 | X9 |
| 12 | 15 | | 19 | 2 | 0 | 1 | 14 | | X9 |
| 13 | 18 | | | 1 | 0 | 1 | | | X9 |
| 14 | 22 | 23 | | 1 | 0 | 0 | | | |
| 15 | | | 25 | 0 | 0 | 0 | | | |

Figure 5.4(b). $G_{op}$ nodes only scheduling result of Figure 5.3(a), with unlimited number of input registers.

transfer, **X9**, is scheduled at memory bank $M_2$ at time step 5, because *reg*_1(5) is smaller than *reg*_2(5). Another register transfer, **X1**, is also scheduled at $M_1$ at time step 7. Figure 5.4(b) shows the modified schedule without any accumulator spill.

#### 5.3.1.2 Mechanisms for Resolving Register Spills

After resolving accumulator spills, we still use variables defined in Table 5.2 to deal with register spills. Note that registers $reg_{ij}$ are dedicated for use for referencing data from $M_i$, so register spills occurring at each memory bank have to be resolved separately. Suppose that a register spill occurs at time step $t$ for register $reg_{ij}$. In this case we must select a node $v$ from *reglist_i*($t$) to temporarily store. From all nodes in *reglist_i*($t$), we store a value that will be used at latest to release a register with the longest time interval. Assume that the selected value $rt$ will be used at time steps $p < t$ and $q > t$. Then, $rt$ is stored later than time step $p$ and reloaded earlier than time step $q–c$, if a load variable instruction requires $c$ time steps to complete. If $rt$ is not yet used, the added store variable instruction can replace the corresponded register transfer. Moreover, an inserted memory access may not be successfully scheduled due

| $t$ | $FU_1$ | $M_1$ | $M_2$ | acc_1 | reg_1 | reg_2 | acclist_1 | reglist_1 | reglist_2 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | 1 | 0 | 0 | | | |
| 2 | 3 | | | 2 | 0 | 0 | 1 | | |
| 3 | 5 | X3 | | 2 | 1 | 0 | 1 | X3 | |
| 4 | 7 | | X5 | 2 | 1 | 1 | 1 | X3 | X5 |
| 5 | 9 | X7 | | 2 | 2 | 1 | 1 | X3, X7 | X5 |
| 6 | 11 | | S9 | 2 | 2 | 1 | 1 | X3, X7 | X5 |
| 7 | 12 | S3 | X11 | 2 | 1 | 2 | 1 | X7 | X5, X11 |
| 8 | 13 | X1 | | 2 | 2 | 2 | 12 | X1, X7 | X5, X11 |
| 9 | 16 | 17 | | 1 | 2 | 2 | | X1, X7 | X5, X11 |
| 10 | 20 | L3 | 21 | 1 | 2 | 2 | | L3, X7 | X5, X11 |
| 11 | 14 | 24 | | 1 | 2 | 0 | | L3, X7 | |
| 12 | 15 | | 19 | 2 | 0 | 0 | 14 | | |
| 13 | 18 | | L9 | 1 | 0 | 1 | | | L9 |
| 14 | 22 | 23 | | 1 | 0 | 0 | | | |
| 15 | | | 25 | 0 | 0 | 0 | | | |

Figure 5.4(c). $G_{op}$ nodes only scheduling result of Figure 5.3(a), without accumulator spills.

to insufficient time steps. In this case we insert an extra time step to schedule this instruction individually as late as possible. Similarly, we update variables defined in Table 5.2 accordingly, and repeat these steps until all register spills are resolved.

We use the schedule shown in Figure 5.4(b) to illustrate above steps. Suppose the target architecture is still the Motorola DSP56000, register spills occur at time steps 7~8 at $M_1$ and time steps 6~9 at $M_2$. At time step 7, after checking the content of *uselist*($u$) for all nodes $u$ in *reglist*_1(7), node **X3** is selected. Additional **S3** and **L3** are scheduled at time steps 6 and 9 respectively, because this value is used by nodes 12 and 15. **S9** and **L9** are also inserted at time steps 5 and 12, respectively. In this case **S9** can directly replace instruction **X9** because **X9** is not yet used at time step 5. Figure 5.4(c) shows the scheduling result without any accumulator/register spill.

### 5.3.2 Instruction Scheduling (II)

So far, we have obtained a schedule that contains all nodes in the TDAG except those with types *L*/*C*. In the second instruction scheduling part of RSSA, remaining memory loads will be inserted to complete the initial scheduling result. We list rules basically inherited from RSSP to schedule memory loads and avoid generating

| $t$ | $FU_1$ | $M_1$ | $M_2$ | acc_1 | reg_1 | reg_2 | acclist_1 | reglist_1 | reglist_2 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | 0 | C1 | 0 | 1 | 1 | | 0 | C1 |
| 2 | 1 | 2 | 6 | 1 | 2 | 1 | | 0, 2 | 6 |
| 3 | 3 | 4 | 8 | 2 | 2 | 2 | 1 | 0, 4 | 6, 8 |
| 4 | 5 | X3 | | 2 | 1 | 2 | 1 | X3 | 6, 8 |
| 5 | 7 | C2 | X5 | 2 | 2 | 2 | 1 | C2, X3 | 8, X5 |
| 6 | 9 | X7 | 10 | 2 | 2 | 2 | 1 | X3, X7 | 10, X5 |
| 7 | 11 | | S9 | 2 | 2 | 1 | 1 | X3, X7 | X5 |
| 8 | 12 | S3 | X11 | 2 | 1 | 2 | 1 | X7 | X5, X11 |
| 9 | 13 | X1 | | 2 | 2 | 2 | 12 | X1, X7 | X5, X11 |
| 10 | 16 | 17 | | 1 | 2 | 2 | | X1, X7 | X5, X11 |
| 11 | 20 | L3 | 21 | 1 | 2 | 2 | | L3, X7 | X5, X11 |
| 12 | 14 | 24 | | 1 | 2 | 0 | | L3, X7 | |
| 13 | 15 | | 19 | 2 | 0 | 0 | 14 | | |
| 14 | 18 | | L9 | 1 | 0 | 1 | | | L9 |
| 15 | 22 | 23 | | 1 | 0 | 0 | | | |
| 16 | | | 25 | 0 | 0 | 0 | | | |

Figure 5.4(d). The initial scheduling result of $G_t$ of Figure 5.3(a).

register spills below. Their main feature is to consider the limited number of registers during scheduling, so that no extra action is required to deal with register spills.

1. According to the execution sequence of ALU instructions, schedule their predecessors as soon as possible.
2. A variable or constant loaded into a register cannot be replaced before being used, and *reg_i*($t$) cannot exceed the number of registers at any time step.
3. If a memory load instruction cannot be scheduled successfully due to insufficient registers, a variable currently residing in a register is selected for storing and reloading using the mechanism described in subsection 5.3.1.2.
4. For previous rule, if the selected variable is not transferred from an accumulator, the additional store variable instruction is unnecessary.
5. If a memory load instruction cannot be scheduled successfully due to insufficient time steps, an additional time step is inserted to schedule this instruction individually as late as possible as in subsection 5.3.1.2.

Figure 5.4(d) shows the scheduling result of TDAG in Figure 5.3(a). Finally, because we have considered accumulator/register spills already, an appropriate physical accumulators/registers assignment certainly exists.

### 5.3.3 Initial Schedule Retiming

In the last part of RSSA, we still apply the multi-dimensional rotation scheduling to explore potential instruction-level parallelism between different iterations. Variables defined in Table 5.2 are dynamically updated during rotation phases to determine at which time step a retimed instruction can be rescheduled. We first describe after rescheduling an instruction at time step *t*, the time interval that its corresponding value must reside in an accumulator or register. Assume that the length of the initial schedule is *len*. If the retimed instruction is of type *L*/*C*/*T*, its referenced value *rt* must occupy a register from time step *t* to *len* because this value will be used for the later iteration. Similarly, the ALU result *rt* of the retimed ALU instruction must occupy an accumulator from time step *t* to *len*. Note that the value *rt* is ready at time step *t*+*c* if the rescheduled instruction requires *c* time steps to complete. Then, for a retimed instruction, we reschedule it at the earliest time step that satisfies precedence relations, gets all ready source operands, and will not cause any accumulator/register spill. Meanwhile, variables defined in Table 5.2 are also updated after rescheduling an ALU instruction, because some resources will be released by its predecessors. In addition, a load constant can be rescheduled for any memory bank to achieve higher performance, because we store constants in all memory banks in advance. The final retimed scheduling result of Figure 5.4(d) is shown in Figure 5.4(e).

## 5.4 Applying to Real DSP Families

In this section, we present how to apply the hypothetical machine model and RSSA, with some modifications, to various real DSP families with architectural features listed in Table 5.1. We divide the description into three parts including data memory bank, function unit, and register set.

| $t$ | **FU$_1$** | **M$_1$** | **M$_2$** | acc_1 | reg_1 | reg_2 | acclist_1 | reglist_1 | reglist_2 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 4 | 8 | 2 | 2 | 2 | 1 | 0, 4 | 6, 8 |
| 2 | 5 | X3 | | 2 | 1 | 2 | 1 | X3 | 6, 8 |
| 3 | 7 | C2 | X5 | 2 | 2 | 2 | 1 | C2, X3 | 8, X5 |
| 4 | 9 | X7 | 10 | 2 | 2 | 2 | 1 | X3, X7 | 10, X5 |
| 5 | 11 | | S9 | 2 | 2 | 1 | 1 | X3, X7 | X5 |
| 6 | 12 | S3 | X11 | 2 | 1 | 2 | 1 | X7 | X5, X11 |
| 7 | 13 | X1 | | 2 | 2 | 2 | 12 | X1, X7 | X5, X11 |
| 8 | 16 | 17 | | 1 | 2 | 2 | | X1, X7 | X5, X11 |
| 9 | 20 | L3 | 21 | 1 | 2 | 2 | | L3, X7 | X5, X11 |
| 10 | 14 | 24 | C1 | 1 | 2 | 1 | | L3, X7 | C1 |
| 11 | 15 | 0 | 19 | 2 | 1 | 1 | 14 | 0 | C1 |
| 12 | 18 | 2 | L9 | 1 | 2 | 2 | | 0, 2 | C1, L9 |
| 13 | 22 | 23 | 6 | 1 | 2 | 2 | | 0, 2 | 6, C1 |
| 14 | 1 | | 25 | 1 | 2 | 1 | 1 | 0, 2 | 6 |

Figure 5.4(e). The retimed scheduling result of $G_t$ of Figure 5.3(a).

### 5.4.1 Data Memory Bank

For the data memory bank, the original definition of the hypothetical machine model and RSSA can satisfy Motorola DSP56000/DSP56001, DSP56300, and NEC uPD7701x families. Other architectures contain of only one data memory bank, so the variable partition step is unnecessary if we apply RSSA to them. However, these architectures still permit parallel memory accesses, with special bus exchange unit or independent buses. In this case we have to modify the first parallel condition of our machine model to allow more than *N* move operations to be executed in one cycle. For members of TI TMS320C6x families, because two memory reads/writes can be performed simultaneously, we simply treat these architectures with two virtual data memory banks and apply the original RSSA. But for Motorola DSP56800 and NEC ADSP2100 families only two memory reads can be executed in parallel, an additional condition will be required in instruction scheduling phases to satisfy this restriction.

### 5.4.2 Function Unit

For the function unit, the original definition of the hypothetical machine model and RSSA directly satisfies all Motorola families. NEC uPD7701x family members

although contain three function units with different types, these function units must be executed exclusively. In this case we can treat these architectures contain a single function unit. Then, one instruction is scheduled at a time using the original RSSA, and allocated to the function unit that is suitable to execute. For members of TI and Analog Device families, they all consist of multiple function units which can be executed in parallel, but each function unit only can execute a restricted instruction set. That is, we must add some additional conditions in instruction scheduling phases in RSSA, to check only instructions that can be allocated to different function units are scheduled in parallel.

### 5.4.3 Register Set

As for the register, the original definition of the hypothetical machine model and RSSA directly satisfies all Motorola families. Because the functionality of registers AF/AR and MR in Analog Device ADSP2100 family is the same as accumulators in Motorola families, our machine model and RSSA basically suit these Analog Device members. From architectural features of ADSP2100 family, in RSSA we will use variables $acc_ALU(t)$, $acc_MAC(t)$, and $reg_1(t)$ to record the number of registers used at time step $t$. However, because in these architectures source operands of ALU and MAC must be stored in different register sets, a single $reg_1(t)$ variable cannot represent the usage of these registers. In this case we actually need two variables $reg_ALU(t)$ and $reg_MAC(t)$ to separately record the number of occupied registers for two register sets. Then, original scheduling rules and variable modifying mechanisms defined in RSSA can be directly applied. On the other hand, members in NEC and TI families contain a homogeneous register set, which indicates all registers are identical and used to store both source and destination operands. In our hypothetical machine model we originally define the heterogeneous register sets. Nevertheless, if we merge

all accumulators and registers into a single register file, it can simulate the architecture with a homogeneous register set. Meanwhile, since all registers are identical, in RSSA we only need variables $reg_i(t)$ to record the total number of registers occupied by all source and destination operands at time step $t$, and variables $acc_i(t)$, as well as variables $acclist_i(t)$, are no longer required. Mechanisms designed for updating variables $reg_i(t)$ and $reglist_i(t)$ can be inherited from the original RSSA. For NEC uPD7701x family members only a single $reg_1(t)$ is necessary. For members of TI families, because they further divide homogeneous registers into two independent register files, we have to use two variables $reg_A(t)$ and $reg_B(t)$ for each register file.

From above descriptions, with minor modifications, our hypothetical machine model and RSSA is capable for simulating all architectural features listed in Table 5.1. Therefore, we conclude that the proposed machine model and RSSA have enough flexibility, which can apply to real DSP families with various architectural features.

## 5.5 Performance Evaluations [38]

### 5.5.1 Comparison with Previous Work

At first we set the target architecture equivalent to the Motorola DSP56000, and select several MDFGs represented DSP applications to evaluate methods including Cho [9], Malik [17], Shiue [20], and RSSA. Similar as in chapter 4, four scheduling results are derived from RSSA with different variable partition mechanisms, and RSVR [30], RSF [14], RST [14], and RSP [36] are used for comparison after inserting necessary spill codes. For a single iteration in the repetitive pattern, we use two metrics including schedule length and instruction count to evaluate performance at the same time. Shorter schedule length basically indicates shorter execution time for both a single iteration and the entire retimed loop. On the other hand, less instruction count

Table 5.3. Schedule lengths obtained by different code generation algorithms.

| | Cho | Malik | Shiue | RSVR | RSF | RST | RSP | RSSA | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | RSVR | RSF | RST | RSP |
| Wave Digital Filter | 7 | 9 | 9 | 8 | 6 | 8.5 | 6 | 6 | 5 | 5.5 | 5.5 |
| Filter | 8 | 13 | 13 | 9 | 11.5 | 9 | 6 | 6 | 5.5 | 5 | 4.5 |
| IIR Filter 2D | 20 | 29 | 33 | 25 | 27.5 | 28 | 24.5 | 16 | 16 | 16 | 16 |
| forward-substitution | 7 | 12 | 12 | 9 | 10 | 11.5 | 7.5 | 5 | 5.5 | 5 | 5 |
| THCS | 6 | 8 | 8 | 6 | 6.5 | 5.5 | 5 | 4 | 4 | 4 | 4 |
| DFT | 14 | 21 | 21 | 18 | 21 | 18.5 | 18 | 13 | 12.5 | 13 | 12.5 |
| Floyd-Steinberg | 20 | 36 | 37 | 29 | 32.5 | 30.5 | 23.5 | 18 | 17.5 | 17 | 17 |
| Transmission Line | 15 | 20 | 21 | 19 | 18 | 25 | 18 | 12 | 12 | 12 | 12 |
| IIR Filter 1D | 11 | 15 | 15 | 11 | 14 | -- | -- | 8 | 8 | -- | -- |
| Differential Equation Solver | 16 | 20 | 21 | 18 | 21.5 | -- | -- | 13 | 11.5 | -- | -- |
| All-pole Lattice Filter | 21 | 37 | 37 | 35 | 28 | -- | -- | 17 | 16 | -- | -- |
| Elliptic Filter | 42 | 62 | 66 | 56 | 69 | -- | -- | 36 | 34 | -- | -- |

indicates not only less power consumption, but also less memory space required to store (smaller code size). Table 5.3 lists schedule lengths for a single iteration in the repetitive pattern for selected MDFGs. In this table we can see that RSSA usually achieves the shortest schedule lengths compared to other methods. The main reason is the usage of retiming technique, which reassigns instructions in consecutive iterations to explore potential instruction-level parallelism. As more compact codes are obtained, system resources are fully utilized and schedule lengths are shortened.

Table 5.4 lists the instruction counts for a single iteration in the repetitive pattern. From this table we find that RSSA and Cho [9] generate much less instruction counts, because they use accumulators and registers to store temporary variables. Other methods directly let temporary variables write back to memory and reload when required, so many memory accesses are really unnecessary. Therefore, instruction counts generated by RSSA and Cho [9] can be kept relatively low. In subsection 5.5.3 we will further describe the effectiveness of RSSA on both evaluation metrics.

Table 5.4. Number of operations really executed in an iteration obtained by different code generation algorithms.

| | Cho | Malik | Shiue | RSVR | RSF | RST | RSP | RSSA | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | RSVR | RSF | RST | RSP |
| Wave Digital Filter | 13 | 16 | 16 | 16 | 15.5 | 16 | 16 | 14 | 13 | 13 | 14 |
| Filter | 11 | 16 | 16 | 16 | 16 | 16 | 16 | 11 | 10.5 | 10.5 | 11 |
| IIR Filter 2D | 37 | 64 | 68 | 64 | 64 | 64 | 64 | 37 | 37 | 37 | 37 |
| forward-substitution | 11 | 20 | 20 | 18 | 17.5 | 18 | 18 | 11 | 10.5 | 10.5 | 11 |
| THCS | 10 | 16 | 16 | 14 | 14.5 | 14 | 14 | 10 | 9.5 | 10 | 10 |
| DFT | 29 | 48 | 49 | 44 | 44 | 44 | 45.5 | 32 | 30 | 31.5 | 32 |
| Floyd-Steinberg | 39 | 68 | 70 | 59 | 59 | 59.5 | 59 | 39 | 39.5 | 40 | 41 |
| Transmission Line | 28 | 48 | 48 | 42 | 42 | 42 | 42 | 29 | 29 | 28 | 29 |
| IIR Filter 1D | 19 | 32 | 32 | 30 | 29.5 | -- | -- | 18 | 17.5 | -- | -- |
| Differential Equation Solver | 23 | 44 | 44 | 37 | 37 | -- | -- | 26 | 25.5 | -- | -- |
| All-pole Lattice Filter | 37 | 60 | 60 | 51 | 51.5 | -- | -- | 35 | 34.5 | -- | -- |
| Elliptic Filter | 75 | 136 | 136 | 125 | 116.5 | -- | -- | 75 | 72 | -- | -- |

### 5.5.2 The Influence of Resources

To harvest the benefits provided by the irregular DSP architecture, using an effective code generation method to fully utilize system resources is obviously essential. However, in order to explore the instruction-level parallelism and reduce accumulator/register spills, increasing the number of resources is a more direct way. Hence, in the following we set our parameterized machine model to simulate target architectures with different number of resources, and use the general method RSSA to evaluate selected MDFGs. Scheduling results affected by different kinds of resources will be studied on both evaluation metrics.

We first list some preliminaries. After transferring MDFGs to TDAGs using RSSA, Table 5.5 lists the number of ALU instructions, the critical path length, and the number of nodes in every TDAG. This information can be treated as lower bounds of scheduling results. If the obtained schedule length is equal to or less than the critical

Table 5.5. Characteristics of selected TDAGs.

| Benchmarks | Number of ALU nodes in $G_t$ | Critical path of $G_t$ | Number of nodes in $G_t$ |
|---|---|---|---|
| Wave Digital Filter | 4 | 6 | 14 |
| Filter | 4 | 7 | 11 |
| IIR Filter 2D | 16 | 7 | 34 |
| Forward-substitution | 5 | 6 | 11 |
| THCS | 4 | 4 | 10 |
| DFT | 12 | 7 | 32 |
| Floyd-Steinberg | 17 | 12 | 38 |
| Transmission Line | 12 | 10 | 26 |
| IIR Filter 1D | 8 | 6 | 17 |
| Differential Equation Solver | 11 | 11 | 25 |
| All-pole Lattice Filter | 15 | 18 | 33 |
| Elliptic Filter | 34 | 19 | 65 |

path of the corresponding TDAG, it indicates that the shortest schedule is achieved. Besides, when architecture has only one function unit, a schedule with length equal to the number of ALU nodes is also shortest, because all ALU instructions must be executed in serial. On the other hand, if an iteration consists of exactly the same number of nodes as the TDAG, it means that no spill codes are inserted. In Tables 5.6~5.10 we use shaded values to represent a schedule with shortest length or without any spill code. Moreover, when either the schedule length or the instruction count is improved by additional resources, the improved result is shown as a bold value.

Table 5.6 shows results of different number of accumulators. From this table, the instruction count decreases obviously when the target architecture contains more accumulators, which represents that accumulator spills occur very often. If more ALU results can reside in additional accumulators, spill codes will be reduced due to less occurrences of accumulator spill. Furthermore, since fewer overwritten ALU results are temporarily transferred to registers, occurrences of register spill also can be reduced. As for the schedule length, because it is only slightly shortened, increasing the number of accumulators cannot explore the instruction-level parallelism.

Table 5.6. Experimental results, with target architectures contains different number of accumulators.

| | 1 FU, 2 acc, 4 reg, 2 mem | | | | | | | | 1 FU, 3 acc, 4 reg, 2 mem | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RSVR | | RSF | | RST | | RSP | | RSVR | | RSF | | RST | | RSP | |
| | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # |
| [1] | 6 | 14 | 5 | 13 | 5.5 | 13 | 5.5 | 14 | 6 | 14 | 5 | 13 | 5.5 | 13 | 5.5 | 14 |
| [2] | 6 | 11 | 5.5 | 10.5 | 5 | 10.5 | 4.5 | 11 | 6 | 11 | 5.5 | 10.5 | **4.5** | 10.5 | 4.5 | 11 |
| [3] | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 | 16 | **34** | 16 | **34** | 16 | **34** | 16 | **34** |
| [4] | 5 | 11 | 5.5 | 10.5 | 5 | 10.5 | 5 | 11 | 5 | 11 | 5.5 | 10.5 | 5 | 10.5 | 5 | 11 |
| [5] | 4 | 10 | 4 | 9.5 | 4 | 10 | 4 | 10 | 4 | 10 | 4 | 9.5 | 4 | 10 | 4 | 10 |
| [6] | 13 | 32 | 12.5 | 30 | 13 | 31.5 | 12.5 | 32 | **12** | 32 | **12** | **28.5** | **12** | **31** | **12** | 32 |
| [7] | 18 | 39 | 17.5 | 39.5 | 17 | 40 | 17 | 41 | 18 | **38** | 17.5 | **38.5** | 17 | **38.5** | 17 | **39** |
| [8] | 12 | 29 | 12 | 29 | 12 | 28 | 12 | 29 | 12 | **27** | 12 | **27** | 12 | **26** | 12 | **27** |
| [9] | 8 | 18 | 8 | 17.5 | -- | -- | -- | -- | 8 | **17** | 8 | **16.5** | -- | -- | -- | -- |
| [10] | 13 | 26 | 11.5 | 25.5 | -- | -- | -- | -- | 13 | 26 | 11.5 | **25** | -- | -- | -- | -- |
| [11] | 17 | 35 | 16 | 34.5 | -- | -- | -- | -- | 17 | **33** | 16 | **31.5** | -- | -- | -- | -- |
| [12] | 36 | 75 | 34 | 72 | -- | -- | -- | -- | **35** | **70** | **30.5** | **66.5** | -- | -- | -- | -- |

[1] Wave Digital Filter
[2] Filter
[3] Infinite Impulse Response Filter 2D
[4] Forward-substitution
[5] Toeplitz Hyperbolic Cholesky Solver
[6] Discrete Fourier Transform
[7] Floyd-Steinberg
[8] Transmission Line
[9] Infinite Impulse Response Filter 1D
[10] Differential Equation Solver
[11] All-pole Lattice Filter
[12] Elliptic Filter

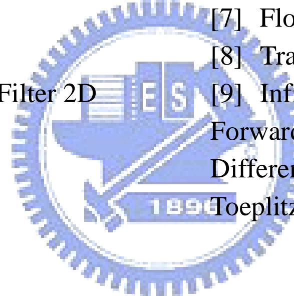
Table 5.7. Experimental results, with target architectures contains different number of input registers.

| | 1 FU, 2 acc, 4 reg, 2 mem | | | | | | | | 1 FU, 2 acc, 6 reg, 2 mem | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RSVR | | RSF | | RST | | RSP | | RSVR | | RSF | | RST | | RSP | |
| | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # |
| [1] | 6 | 14 | 5 | 13 | 5.5 | 13 | 5.5 | 14 | 6 | 14 | 5 | 13 | **5** | 13 | **5** | 14 |
| [2] | 6 | 11 | 5.5 | 10.5 | 5 | 10.5 | 4.5 | 11 | **5** | 11 | **5** | 10.5 | **4** | 10.5 | **4** | 11 |
| [3] | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 |
| [4] | 5 | 11 | 5.5 | 10.5 | 5 | 10.5 | 5 | 11 | 5 | 11 | **5** | 10.5 | 5 | 10.5 | 5 | 11 |
| [5] | 4 | 10 | 4 | 9.5 | 4 | 10 | 4 | 10 | 4 | 10 | 4 | 9.5 | 4 | 10 | 4 | 10 |
| [6] | 13 | 32 | 12.5 | 30 | 13 | 31.5 | 12.5 | 32 | 13 | 32 | 12.5 | **28.5** | **12** | **30.5** | 12.5 | 32 |
| [7] | 18 | 39 | 17.5 | 39.5 | 17 | 40 | 17 | 41 | 18 | 39 | 17.5 | 39.5 | 17 | **39.5** | 17 | **40** |
| [8] | 12 | 29 | 12 | 29 | 12 | 28 | 12 | 29 | 12 | 29 | 12 | 29 | 12 | 28 | 12 | 29 |
| [9] | 8 | 18 | 8 | 17.5 | -- | -- | -- | -- | 8 | 18 | 8 | 17.5 | -- | -- | -- | -- |
| [10] | 13 | 26 | 11.5 | 25.5 | -- | -- | -- | -- | 13 | **25** | 12 | **23** | -- | -- | -- | -- |
| [11] | 17 | 35 | 16 | 34.5 | -- | -- | -- | -- | 17 | 35 | 16 | **33** | -- | -- | -- | -- |
| [12] | 36 | 75 | 34 | 72 | -- | -- | -- | -- | **35** | **73** | **30.5** | **69.5** | -- | -- | -- | -- |

Table 5.8. Experimental results, with target architectures contains different number of function units.

| | 1 FU, 2 acc, 4 reg, 2 mem | | | | | | | | 2 FU, 2 acc, 4 reg, 2 mem | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RSVR | | RSF | | RST | | RSP | | RSVR | | RSF | | RST | | RSP | |
| | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # |
| [1] | 6 | 14 | 5 | 13 | 5.5 | 13 | 5.5 | 14 | 6 | 14 | 5 | 13 | 5.5 | 13 | **5** | 14 |
| [2] | 6 | 11 | 5.5 | 10.5 | 5 | 10.5 | 4.5 | 11 | 6 | 11 | 5.5 | 10.5 | **4.5** | 10.5 | **3.5** | 11 |
| [3] | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 | **12** | 39* | **13** | 40* | **12.5** | 40* | **12.5** | 40* |
| [4] | 5 | 11 | 5.5 | 10.5 | 5 | 10.5 | 5 | 11 | **4** | 11 | **4** | 10.5 | **4.5** | 10.5 | **4** | 11 |
| [5] | 4 | 10 | 4 | 9.5 | 4 | 10 | 4 | 10 | **3** | 10 | **3** | 9.5 | **3** | 10 | **3.5** | 10 |
| [6] | 13 | 32 | 12.5 | 30 | 13 | 31.5 | 12.5 | 32 | **12** | 34* | **11.5** | 32* | 13 | 35* | **12** | 34* |
| [7] | 18 | 39 | 17.5 | 39.5 | 17 | 40 | 17 | 41 | **15** | 41* | **16.5** | 45.5* | **16** | 45.5* | **14** | 42.5* |
| [8] | 12 | 29 | 12 | 29 | 12 | 28 | 12 | 29 | **8** | **28** | **11.5** | 30.5* | **9.5** | 29* | **11.5** | 30.5* |
| [9] | 8 | 18 | 8 | 17.5 | -- | -- | -- | -- | **6** | 18 | **7** | 18* | -- | -- | -- | -- |
| [10] | 13 | 26 | 11.5 | 25.5 | -- | -- | -- | -- | **10** | **25** | 11.5 | 26.5* | -- | -- | -- | -- |
| [11] | 17 | 35 | 16 | 34.5 | -- | -- | -- | -- | **16** | **33** | **14** | 35.5* | -- | -- | -- | -- |
| [12] | 36 | 75 | 34 | 72 | -- | -- | -- | -- | **27** | 80* | **29** | 80* | -- | -- | -- | -- |

[7] Wave Digital Filter [7] Floyd-Steinberg
[8] Filter [8] Transmission Line
[9] Infinite Impulse Response Filter 2D [9] Infinite Impulse Response Filter 1D
[10] Forward-substitution [10] Differential Equation Solver
[11] Toeplitz Hyperbolic Cholesky Solver



Then, Table 5.7 shows results of different number of registers. These results indicate that the instruction-level parallelism still cannot be explored by using more registers. In addition, the instruction count also be improved slightly, which means register spills rarely occur in fact. Thus, if we only increase the number of registers, scheduling results will be almost unchanged for both evaluation metrics.

In Table 5.8 we show results of different number of function units but the number of accumulators remains two. That is, when the target architecture has two function units, only one dedicated accumulator is capable to store destination operands calculated from each function unit. From this table schedule lengths are obviously shortened, because the second function unit is beneficial to explore instruction-level parallelism. However, instruction counts increase in some MDFGs as

Table 5.9. Experimental results, with target architectures contains different number of function units.

| | 1 FU, 2 acc, 4 reg, 2 mem | | | | | | | | 2 FU, 4 acc, 4 reg, 2 mem | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RSVR | | RSF | | RST | | RSP | | RSVR | | RSF | | RST | | RSP | |
| | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # |
| [1] | 6 | 14 | 5 | 13 | 5.5 | 13 | 5.5 | 14 | 6 | 14 | **4.5** | 13 | 5.5 | 13 | **5** | 14 |
| [2] | 6 | 11 | 5.5 | 10.5 | 5 | 10.5 | 4.5 | 11 | 6 | 11 | 5.5 | 10.5 | **4.5** | 10.5 | **3.5** | 11 |
| [3] | 16 | 37 | 16 | 37 | 16 | 37 | 16 | 37 | **10** | **34** | **10** | **34** | **9** | **34** | **9** | **34** |
| [4] | 5 | 11 | 5.5 | 10.5 | 5 | 10.5 | 5 | 11 | **4** | 11 | **4** | 10.5 | **4** | 10.5 | **4** | 11 |
| [5] | 4 | 10 | 4 | 9.5 | 4 | 10 | 4 | 10 | **3** | 10 | **3** | 9.5 | **3** | 10 | **3** | 10 |
| [6] | 13 | 32 | 12.5 | 30 | 13 | 31.5 | 12.5 | 32 | **10** | 32 | **9** | **28** | **9.5** | **30** | **10** | 32 |
| [7] | 18 | 39 | 17.5 | 39.5 | 17 | 40 | 17 | 41 | **13** | **38** | **14** | 39.5 | **13** | **39.5** | **12.5** | **38** |
| [8] | 12 | 29 | 12 | 29 | 12 | 28 | 12 | 29 | **8** | **26** | **8.5** | **26** | **8.5** | **25** | **8.5** | **26** |
| [9] | 8 | 18 | 8 | 17.5 | -- | -- | -- | -- | **6** | **17** | **6** | **16.5** | -- | -- | -- | -- |
| [10] | 13 | 26 | 11.5 | 25.5 | -- | -- | -- | -- | **10** | **25** | **10** | 25.5 | -- | -- | -- | -- |
| [11] | 17 | 35 | 16 | 34.5 | -- | -- | -- | -- | **16** | **33** | **13** | **30.5** | -- | -- | -- | -- |
| [12] | 36 | 75 | 34 | 72 | -- | -- | -- | -- | **23** | **70** | **24** | **68** | -- | -- | -- | -- |

[1] Wave Digital Filter
[2] Filter
[3] Infinite Impulse Response Filter 2D
[4] Forward-substitution
[5] Toeplitz Hyperbolic Cholesky Solver
[6] Discrete Fourier Transform
[7] Floyd-Steinberg
[8] Transmission Line
[9] Infinite Impulse Response Filter 1D
[10] Differential Equation Solver
[11] All-pole Lattice Filter
[12] Elliptic Filter

asterisked, which represents more spill codes are inserted. Apparently these additional spill codes are mainly incurred from frequently occurred accumulator spills. If an ALU result will be used later than next ALU instruction been executed, it must be temporarily stored to avoid being overwritten. Thuse, we conclude that using more function units only is not appropriate to explore instruction-level parallelism.

Similarly, Table 5.9 still shows results of different number of function units. This time we increase the number of accumulators to four and evenly allocate them to each function unit. Compared to Table 5.8, clearly that not only schedule lengths are further shortened, but also spill codes are inserted infrequently. These results are essentially the combination of results shown in Tables 5.6 and 5.8. Using more function units is beneficial to shorten schedule lengths, and adding additional

Table 5.10. Experimental results, with target architectures contains different number of data memory banks.

| | 1 FU, 3 acc, 6 reg, 2 mem | | | | | | | | 1 FU, 3 acc, 6 reg, 3 mem | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RSVR | | RSF | | RST | | RSP | | RSVR | | RSF | | RST | | RSP | |
| | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # | *len* | # |
| [1] | 6 | 14 | 5 | 13 | 5 | 13 | 5 | 14 | **4** | 14 | **3** | **12.7** | 5 | 13.3 | **4** | 14 |
| [2] | 6 | 11 | 5.5 | 10.5 | 4.5 | 10.5 | 4 | 11 | 6 | 11 | **5.3** | **10.3** | **4** | **10.3** | 4 | 11 |
| [3] | 16 | 34 | 16 | 34 | 16 | 34 | 16 | 34 | 16 | 34 | 16 | 34 | 16 | 34 | 16 | 34 |
| [4] | 5 | 11 | 5 | 10.5 | 5 | 10.5 | 5 | 11 | 5 | 11 | 5 | **10.3** | 5 | **10.3** | 5 | 11 |
| [5] | 4 | 10 | 4 | 9.5 | 4 | 10 | 4 | 10 | 4 | 10 | 4 | **9.3** | 4 | 10 | 4 | 10 |
| [6] | 12 | 32 | 12 | 28 | 12 | 30 | 12 | 32 | 12 | 32 | 12 | **26.7** | 12 | 30.3 | 12 | 32 |
| [7] | 18 | 38 | 17 | 38.5 | 17 | 38.5 | 17 | 39 | 18 | 38 | 17 | **38.3** | 17 | **38.3** | 17 | 39 |
| [8] | 12 | 27 | 12 | 27 | 12 | 26 | 12 | 27 | 12 | 27 | 12 | 27 | 12 | **25.7** | 12 | 27 |
| [9] | 8 | 17 | 8 | 16.5 | -- | -- | -- | -- | 8 | 17 | 8 | **16** | -- | -- | -- | -- |
| [10] | 13 | 25 | 11.5 | 22.5 | -- | -- | -- | -- | 14 | 26 | 11.7 | **21.7** | -- | -- | -- | -- |
| [11] | 17 | 33 | 16 | 31.5 | -- | -- | -- | -- | 17 | 33 | **15.7** | **30** | -- | -- | -- | -- |
| [12] | 35 | 68 | 30.5 | 65 | -- | -- | -- | -- | 35 | 68 | 35.7 | **64.3** | -- | -- | -- | -- |

[1] Wave Digital Filter
[2] Filter
[3] Infinite Impulse Response Filter 2D
[4] Forward-substitution
[5] Toeplitz Hyperbolic Cholesky Solver
[6] Discrete Fourier Transform
[7] Floyd-Steinberg
[8] Transmission Line
[9] Infinite Impulse Response Filter 1D
[10] Differential Equation Solver
[11] All-pole Lattice Filter
[12] Elliptic Filter

accumulators can reduce occurrences of spill codes efficiently. Hence, if we want to explore instruction-level parallelism, both numbers of function units and accumulators must be increased.

Finally, in Table 5.10 we show results of increasing the number of data memory banks. Both architectures consist of six input registers evenly allocated to each data memory bank. From these results schedule lengths are hardly improved without additional function units, and using more data memory banks seems helpful to reduce instruction counts. The reason is that with the number of data memory banks increasing, more independent memory accesses, as well as register transfers inserted to resolve accumulator spills, can be executed simultaneously. This situation lets the instruction-level parallelism between move operations be explored, which is

beneficial to reduce occurrences of register spills. However, implementing additional data memory banks, associated with dedicated data buses, definitely requires heavy hardware costs. Besides, recall that the TDAG is enlarged factor equal to the number of data memory banks before using variable partition mechanisms proposed in RSF, RST, and RSP. A larger TDAG also costs longer time doing code generation. Thus, we do not recommend using more data memory banks to reduce the instruction count, because the cost-performance is not worth.

### 5.5.3 Brief Summaries

After showing RSSA is effective compared to previous work under the Motorola DSP56000 architecture, we present its effectiveness in some detail on both evaluation metrics. As shown in Tables 5.8 and 5.9, when the target architectures consists of two function units, RSSA can achieve schedule lengths to their lower bounds in most selected MDFGs. If there is only one function unit, RSSA still can obtain schedule lengths almost equal to the number of ALU instructions, which indicates these schedules cannot be shortened further. On the other hand, according to Tables 5.5~5.10, RSSA really generates quite few spill codes especially when the target architecture has more than four accumulators. This is because we prefer to transfer an overwritten ALU result to a register, and insert spill codes only when required. In addition, we compact spill codes with regular codes as far as possible, in order to prevent lengthening the final schedule length. Whereas RSSA usually achieves optimal results on both evaluation metrics, we conclude that it is quite effective.

Then, we summarize the influence of differing number of resources on the scheduling result. From descriptions in subsection 5.5.2, adding more accumulators to keep more ALU results for further using is the most efficient way to reduce spill codes. According to our evaluation results, almost all spill codes can be eliminated if

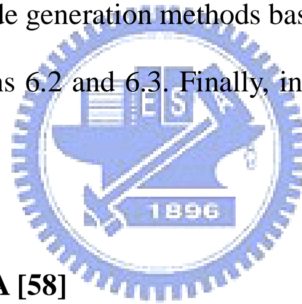
the target architecture contains more than four accumulators. Increasing the number of registers or data memory banks is also useful, but its improvement is not as obvious as using more accumulators. Besides, implementing additional data memory banks and associated data buses requires heavy hardware costs. Therefore, we think a target architecture that contains two data memory banks and four registers is appropriate. As for exploring the instruction-level parallelism, adding additional function units and accumulators concurrently is certainly necessary. Based on evaluation results shown in Table 5.9, RSSA generates the shortest schedules without too many spill codes in most MDFGs, so two function units with four accumulators are actually sufficient. Using more than two function units no doubt can further shorten schedule lengths, but the improvement will be clearly slight. Furthermore, we also find that the variable partition mechanism proposed in RSF is unsuitable for one-dimensional MDFGs. This is because loop-carried data dependences in one-dimensional MDFGs are usually with distance one, and most memory accesses will reference variables from the same data memory bank after applying loop unfolding. Thus, a memory access may easily fail to be scheduled successfully in time, which will lengthen the schedule.

In the following, we describe the efficiency of RSSA and compare to RSSP. Recall that both methods mainly contain following phases: the construction of graphs MDFG, TDAG, and $G_{op}$, variable partition, two separate instruction scheduling phases, the resolution of accumulator/register spills, and the initial schedule retiming. Among these phases, resolving accumulator spills is the most time-consuming and the main different phase between RSSP and RSSA. This phase is completed by analyzing the TDAG topology with a relatively complex mechanism in RSSP, and by constantly updating variables in RSSA. The mechanism used in RSSA is apparently more efficient, general, and accurate. Other phases between two methods are very similar in essence. Hence, we conclude that RSSA is efficient than RSSP.

Compared to related studies [9, 17], our RSSA still has advantages. Note that in methods [9, 17] they do not contain procedures to generate uncompacted codes, and schedule instructions using list scheduling similar as in RSSA. Thus, we omit steps of constructing graphs MDFG and TDAG, and focus on discussing their complexities in partitioning variables, allocating accumulators/registers, and resolving accumulator/ register spills. In method [17], it uses *graph labeling* to assign accumulators/registers and variables simultaneously, and applies *simulated annealing* to solve the *graph labeling* problem. The mechanism used to insert spill codes is not presented in detail, although it is definitely required. However, because the *simulated annealing* is time-consuming, it makes method [17] more complicated. Next, in method [9], it uses *graph coloring* to partition variables and allocate accumulators/registers separately, and gives a heuristic to solve the *graph coloring* problem. The mechanism used to insert spill codes is still lacked. Thus, clearly that the method proposed in [9] is efficient than the method proposed in [17]. Finally, in our RSSA, variable partitioning is very simple. Spill codes insertion and physical accumulators/registers assignment are also trivial, which means RSSA is efficient compared to both methods [9, 17].

# Chapter 6. Energy-efficient Code Generation Methods

In addition to shorter schedule length and less instruction count, low power consumption becomes another important constraint in the DSP design specification [24-25]. In section 1.2 we mentioned that to increase the potential for a function unit to reuse an operand is an appropriate way to reduce the power consumed by a function unit. An instruction-level power analysis and some ideas which can be exploited by software development tools have been also listed in subsection 2.4.4. Therefore, the third study issue of this thesis is to design energy-efficient code generation methods based on the operand sharing technique. At first we briefly analyze RSSA from the viewpoint of low power consumption in section 6.1. Two proposed energy-efficient code generation methods based on the hypothetical machine model is presented in sections 6.2 and 6.3. Finally, in section 6.4, some performance evaluations are shown.



## 6.1 Brief Analyses of RSSA [58]

RSSA is an effective code generation method suited for DSPs with various architectural features, and its design goal is to achieve shorter schedule length and less spill codes. From the viewpoint of low power consumption, RSSA has satisfied three positive features. First, to achieve the instruction-level parallelism, RSSA schedules unpacked instructions as soon as possible without violating data dependencies and resource constraints. This strategy leads to pack instructions as much as possible, which can reduce the energy consumption in DSP especially with multiple data memory banks. Second, with appropriate variable partition mechanisms, memory accesses are separately scheduled at all data memory banks to explore potential higher memory bandwidth. Third, during the TDAG construction, RSSA assumes unlimited

numbers of accumulators/registers and removes all possible unnecessary memory accesses. Then, it prefers to use register transfers to resolve accumulator spills, and inserts additional store/load variable instructions only when required. By using these scheduling rules RSSA can apparently produce schedules with retrenched memory accesses. This feature is beneficial in reducing power consumption as well as code size, because a memory access requires considerably more power to execute than an ALU instruction. However, the potential for operand sharing are not successfully explored because RSSA simply uses list scheduling to schedule ALU instruction. Thus, in the following we will propose two energy-efficient methods extended from RSSA, which will retain all above positive features and further consider the operand sharing technique.

## 6.2 Rotation Scheduling with Operand Reutilization (RSOR) [58]

In this section we introduce the first proposed method named *rotation scheduling with operand reutilization* (*RSOR*). After presenting its scheduling steps in subsection 6.2.1, some comparisons of RSSA and RSOR are described in subsection 6.2.2.

### 6.2.1 Detailed Algorithms of RSOR

Based on the scheduling steps of RSSA described in section 5.2, we only have to modify the mechanism used to schedule ALU instructions to consider the operand sharing technique. RSSA simply uses the list scheduling to individually schedule ALU instructions. In RSOR we define a *sharing set* to group ALU instructions with a common operand. Then, the list scheduling is still applied, and ALU instructions in the same sharing set will be restrictively scheduled to the same function unit at consecutive time steps to achieve the operand sharing. For a given TDAG, we use the following definition to describe the node grouping conditions for the sharing set.

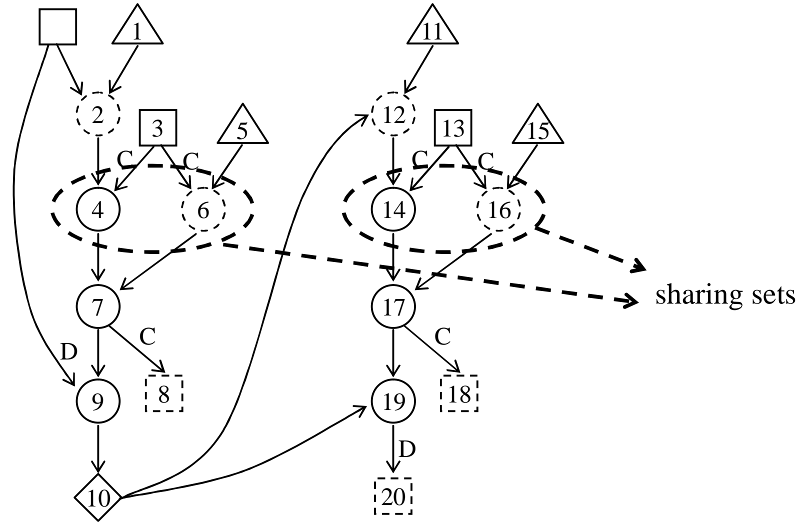
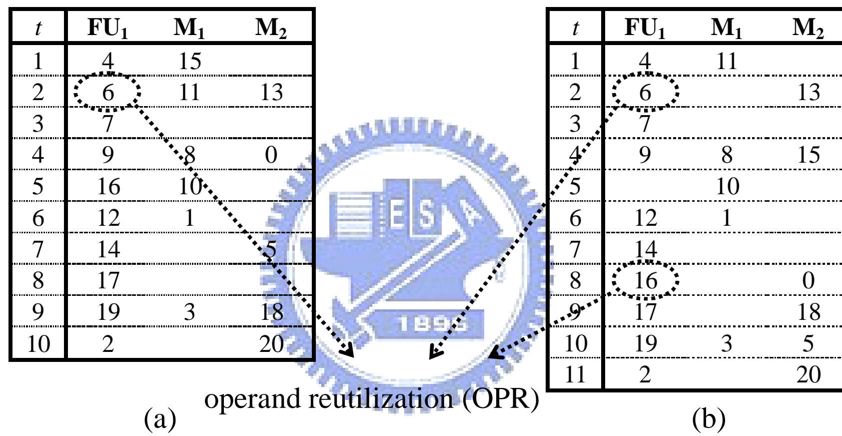



Figure 6.1. An example of TDAG and sharing sets.



| $t$ | $FU_1$ | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | 4 | 15 | |
| 2 | 6 | 11 | 13 |
| 3 | 7 | | |
| 4 | 9 | 8 | 0 |
| 5 | 16 | 10 | |
| 6 | 12 | 1 | |
| 7 | 14 | | 5 |
| 8 | 17 | | |
| 9 | 19 | 3 | 18 |
| 10 | 2 | | 20 |

(a)

| $t$ | $FU_1$ | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | 4 | 11 | |
| 2 | 6 | | 13 |
| 3 | 7 | | |
| 4 | 9 | 8 | 15 |
| 5 | | 10 | |
| 6 | 12 | 1 | |
| 7 | 14 | | |
| 8 | 16 | | 0 |
| 9 | 17 | | 18 |
| 10 | 19 | 3 | 5 |
| 11 | 2 | | 20 |

(b)



Figure 6.2. Scheduling results of Figure 6.1. (a) RSSA, (b) RSOR.

**Definition 6.1.** For a given TDAG $G = (V, E, X, P)$, nodes $v_1 \ldots v_m \in V$ with type $M/A$ are grouped into a *sharing set* if they satisfy two conditions: (1) All nodes $v_1 \ldots v_m$ have the same predecessor $v_u \in V$ ($P(v_u) = L/M/A$); (2) There is no path between any two nodes $v_i$ and $v_j$ in $G$ for $1 \le i, j \le m$.

For example, in the TDAG shown in Figure 6.1, nodes 4 and 6 are grouped into a sharing set as they share the operand loaded by node 3. Nodes 14 and 16 are grouped into another sharing set, as they share the operand loaded by node 13. Suppose that the target architecture is the Motorola DSP56000; Figure 6.2 shows scheduling results of Figure 6.1 using RSSA and RSOR, while nodes with common operand are grouped

1. $G_c$ = Construct MDFG;
    1.1. Partition variables to memory banks;
    1.2. Unfold or tile $G_c$ if necessary;
2. $G_t$ = Construct TDAG ($G_c$);
3. $S$ = Schedule all instructions except memory loads ($G_t$);
    3.1. **Group ALU instructions into sharing sets;**
    ***// additional step used in RSOR***
    3.2. $G_{op}$ = Construct DAG $G_{op}$ ($G_t$);
    3.3. **$S$ = Schedule nodes in $G_{op}$ ($G_{op}$);**
    ***// nodes in the same sharing set are restrictively scheduled***
    3.4. $S$ = Determine and solve accumulator spills ($S$, $G_{op}$);
4. $S$ = Schedule memory load instructions ($S$, $G_t$);
5. $S$ = Retime the initial scheduling result ($S$, $G_t$);

Figure 6.3. The overall scheduling algorithm of RSOR.

and consecutively scheduled or not. From these two schedules we find that one more *operand reutilization* is achieved in an iteration using RSOR. We call *operand reutilization* (*OPR*) the fact that an operand is reused by two instructions executed consecutively in the same function unit [4]. Recall that the average power consumed by the function unit is dramatically lower when one operand remains unchanged. That is, if a schedule has more OPRs, to execute this schedule will cost less power consumption at function units. However, because instructions resided in the same sharing set are restrictively scheduled to the same function unit at consecutive time steps, using RSOR may obtain longer schedules. This feature indicates that OPR may be a trade off for schedule length, which is similar to the LPLS method described in subsection 2.4.4. Moreover, if there are more than one function units in the target architecture, ALU instructions in the same sharing set are evenly distributed to all function units to explore the instruction-level parallelism. Note that an ALU instruction may reside in more than one sharing set, so after instruction scheduling not all potential operand sharing can be achieved. The overall scheduling steps for RSOR are listed in Figure 6.3. The main difference between RSSA and RSOR are the sharing set grouping and the mechanism used to schedule ALU instructions.

Table 6.1. Average current required for each instruction [20].

| Instruction | Current (mA) | Instruction | Current (mA) |
|---|---|---|---|
| Move | 90 | Mpy | 160 |
| Move Move | 120 | Mpy Move | 170 |
| Add | 100 | Mpy Move Move | 180 |
| Add Move | 140 | | |
| Add Move Move | 150 | | |

### 6.2.2 Comparisons between RSSA and RSOR

In this subsection, we evaluate RSOR using several selected MDFGs and our hypothetical machine model defined in section 5.1. Three scheduling results are derived from RSOR with variable partition mechanisms proposed in RSVR, RSF, and RST, and in the following we only list the best one of them. For a single iteration in the repetitive pattern, we use evaluation metrics including schedule length, instruction count, and the number of OPRs to compare RSOR and RSSA at the same time. According to the instruction-level power model presented in [45], a schedule with shorter schedule length and less instruction count obviously indicates lower power consumption. More OPRs represent more operands reused by two instructions consecutively executed in the same function unit, which leads to less power consumption at function units. Furthermore, in [20], authors list the average current required to execute each instruction in Motorola DSP56000. Table 6.1 shows their provided information, and we also borrow it to approximately estimate the required current for each schedule.

Table 6.2 lists the number of OPRs of a single iteration in the repetitive pattern for selected MDFGs. From this table clearly that using RSOR achieves more OPRs than using RSSA, and performs better when the target architecture has more function units. The reason is that some ALU instructions may essentially share a common operand but not be grouped into a sharing set, because these instructions violate the

Table 6.2. The comparison between RSOR and RSSA (the number of OPRs).

| | 1 FU, 2 acc, 4 reg, 2 mem | | 2 FU, 4 acc, 4 reg, 2 mem | |
|---|---|---|---|---|
| | RSSA | RSOR | RSSA | RSOR |
| Wave Digital Filter | 0.5 | 0.5 | 0 | 0.5 |
| Filter | 0 | 0 | 0 | 0 |
| IIR 2D | 0 | 0 | 0 | 0 |
| Forward-substitution | 1.5 | 1.5 | 0.5 | 2 |
| THCS | 1 | 1 | 0 | 2 |
| Discrete Fourier Transform | 1 | 4 | 0 | 4 |
| Floyd-Steinberg | 9 | 9 | 6 | 9 |
| Transmission Line | 4 | 4 | 0 | 5 |
| IIR 1D | 0.5 | 3 | 0.5 | 3 |
| Differential Equation Solver | 4 | 5 | 3 | 6 |
| All-pole Lattice Filter | 1.5 | 3 | 1.5 | 6 |
| Elliptic Filter | 4 | 9 | 2 | 11 |

second condition listed in Definition 6.1. Nodes 12 and 19 in Figure 6.1 are such an example. When the target architecture has only one function unit, due to data dependencies, nodes 12 and 19 will not be schedule at consecutive time steps. However, if there exists a second function unit, we can use it to separately execute nodes 12 and 19 without interfering with other instructions. Hence, the operand sharing between these two nodes is also achieved.

Table 6.3 lists the schedule length, instruction count, and approximate current of a single iteration in Motorola DSP56000 architecture. It shows that compared to using RSSA, using RSOR may generate schedules with, at most, 6% longer schedule length and 7% greater instruction count. This is because ALU instructions in a sharing set are restrictively scheduled to the same function unit at consecutive time steps, and some instruction-level parallelism cannot be successfully explored. Meanwhile, when more instructions are executed by the same function unit, accumulator spills may occur more easily, due to the frequent use of some dedicated accumulators. As for the approximate current, not in all cases using RSOR can be improved, especially when the schedule length is increased. The main reason is that the total current of a schedule

Table 6.3. The comparison between RSOR and RSSA (under Motorola DSP56000 architecture).

| | RSSA | | | RSOR | | |
|---|---|---|---|---|---|---|
| | length | ins. count | current | length | ins. count | current |
| Wave Digital Filter | 5 | 13 | 750 | 5 | 13 | 770 |
| Filter | 5 | 10.5 | 655 | 5 | 10.5 | 655 |
| IIR 2D | 16 | 37 | 2435 | 16 | 37 | 2435 |
| Forward-substitution | 5 | 10.5 | 711 | 5 | 10.5 | 717 |
| THCS | 4 | 9.5 | 611 | 4 | 9.5 | 587 |
| Discrete Fourier Transform | 12.5 | 30 | 1870 | 12.5 | 32 | 1756 |
| Floyd-Steinberg | 17.5 | 39 | 2440 | 17.5 | 39 | 2440 |
| Transmission Line | 12 | 28 | 1770 | 12 | 28 | 1770 |
| IIR 1D | 8 | 17.5 | 1201 | 8 | 19 | 1111 |
| Differential Equation Solver | 11.5 | 25.5 | 1580 | 12 | 24.5 | 1651 |
| All-pole Lattice Filter | 16 | 34.5 | 2280 | 17 | 34.5 | 2300 |
| Elliptic Filter | 34 | 72 | 5020 | 35 | 75 | 4827 |

is the sum of current required by all instructions. Although the power cost by function units is reduced, the total current may still increase due to longer schedule length. In summary, RSOR achieves more OPRs than RSSA and requires slightly longer schedule length and more instruction count in some cases. As long as the schedule length is not increased, using RSOR usually can obtain a schedule with lower approximate current. Further evaluations of RSOR and comparisons to other energy-efficient instruction scheduling methods will be given in section 6.4.

### 6.3 Rotation Scheduling with Exploiting Operand Reutilization (RSER) [58]

Although increasing the potential for a function unit to reuse an operand can obtain low-power schedules, common operands are not encountered very frequently in real designs [29]. That is, if just operand sharing within an iteration are explored, the power consumption will be reduced only slightly due to less opportunities of operand sharing. As mentioned in subsection 2.4.4, the retiming technique can be used to transform the given loop to generate instructions with common operands hidden

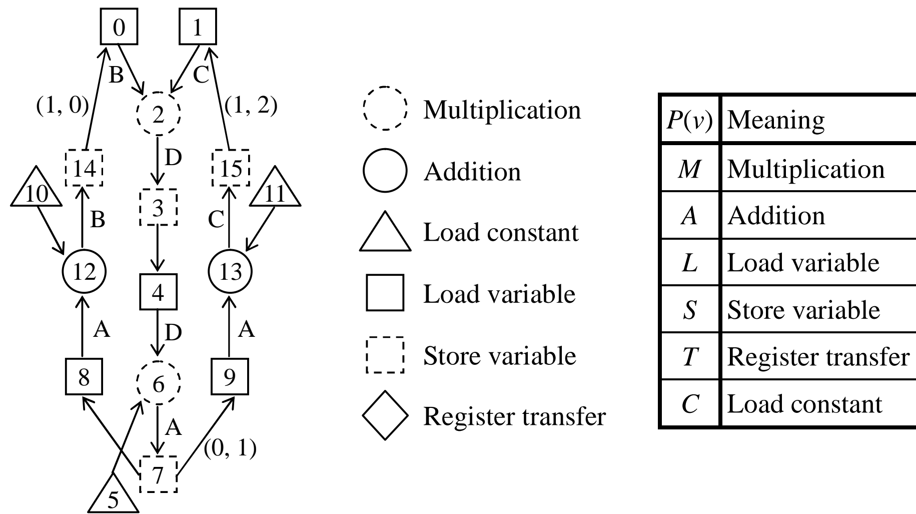
inside the original MDFG. Hence, we propose the second method *rotation scheduling with exploiting operand realization* (*RSER*), which is extended from RSOR and aimed to further exploit potential operand sharing between different iterations. Subsection 6.3.1 contains the mechanism for reconstructing the original MDFG. Detailed scheduling steps of RSER are described in subsection 6.3.2. In subsection 6.3.3, we list the difference among RSOR, RSER, and other related methods.

### 6.3.1 MDFG Reconstruction Mechanism

#### 6.3.1.1 Finding Potential Operand Reutilization in Different Iterations

To generate instructions with common operands hidden inside the given MDFG, first we have to find instructions sharing an operand in different iterations. Recall that a variable in a loop indicates an array. For a given loop, assume that two different elements of the same array are used as source operands of two ALU instructions $x_i$ and $y_i$ in iteration $i$. Apparently that $x_i$ and $y_i$ do not have common operand. However, there must exist another ALU instruction $y_j$ in iteration $j$, which references the same element as $x_i$. If we can move $x_i$ and $y_j$ to the same iteration, an additional OPR can be achieved. In the MDFG, if different elements of the same array are referenced in an iteration, we will find a node $v_u$ with type $S$ that has multiple successors, $v_i$, of type $L$ where all $d(e_{ui})$ are different. In RSER we group load variable instructions $v_i$ into an *exploitable sharing set*, which means these instructions may reference the same element of the same array after retiming. Node grouping conditions of the exploitable sharing set are described in the following definition.

**Definition 6.2.** For a given MDFG $G = (V, E, X, d, P)$, nodes $v_1 \ldots v_m \in V$ with type $L$ are grouped into an *exploitable sharing set* if they satisfy two conditions: (1) All nodes $v_1 \ldots v_m$ have the same predecessor $v_u \in V$ ($P(v_u) = S$); (2) For any two edges $e_{ui}$ and $e_{uj} \in E$, $d(e_{ui}) \neq d(e_{uj})$ for $1 \leq i, j \leq m$.




| $P(v)$ | Meaning |
|---|---|
| $M$ | Multiplication |
| $A$ | Addition |
| $L$ | Load variable |
| $S$ | Store variable |
| $T$ | Register transfer |
| $C$ | Load constant |

Figure 6.4. The MDFG example.

For example, for the MDFG shown in Figure 6.4, nodes 8 and 9 are grouped into an exploitable sharing set because they both connect to node 7 with different edge delays. This case indicates that load variable instructions 8 and 9 reference to the same array but different elements. If we can apply the retiming technique to make $d(e_{79})$ equal to $d(e_{89})$, ALU instructions 12 and 13 will share a common operand within an iteration. Therefore, for every exploitable sharing set, we require an MDFG reconstruction algorithm to make these instructions reference a common element as far as possible. This algorithm will be introduced in detail in the next subsection.

#### 6.3.1.2 MDFG Reconstruction Algorithm

Before describing the MDFG reconstruction algorithm, we list some features of an MDFG. According to the MDFG construction steps, data memory is the only place to store operands. That is, an instruction written in high-level language is directly transferred to four nodes, which are used to load two source operands, execute, and store result. Therefore, in an MDFG, a node with type $M/A$ will have two predecessors with type $L/C$ and one successor with type $S$, and at least one predecessor must be load variable instruction. Besides, a non-zero delay edge, $e_{ij}$, can only exist between

(1) **Input:** MDFG $G = (V', E, X, d, P)$, exploitable sharing set $S = \{v_1 \ldots v_m\}$;
(2) **Output:** $rf(v)$, retimed MDFG $G_r$;
(3) $rf(v) = (0,\ldots, 0)$, $\forall\, v \in V$; *// inilize retiming function of all nodes*
(4) Assume that $v_1 \ldots v_m$ have the same predecessor $v_u \in V$; // $P(v_u) = S$
(5) $v_{ij} \in V$ are successors of $v_i$ respectively; // $P(v_{ij}) = M/A$
(6) $s_{ij} \in V$ are successors of $v_{ij}$ respectively; // $P(s_{ij}) = S$
(7) $u_{ij} \in V$ are predessors of $v_{ij}$ respectively, $u_{ij} \neq v_i$; // $P(u_{ij}) = L/C$
(8) /* $v_i$ *and* $u_{ij}$ *are two input operands of* $v_{ij}$ */
(9) $check(e_{ui})$ = '$F$', $\forall\, d(e_{ui}) \neq (0,\ldots, 0)$; // $e_{ui} \in E$ *connects* $v_u$ *and* $v_i$
(10) $merge(v_i)$ = '$F$', $\forall\, d(e_{ui}) \neq (0,\ldots, 0)$;
(11) **While** (not all $d(e_{ui})$ are equal, $\forall\, check(e_{ui})$ = = '$F$')
(12) $r = d(e_{ui})$ such that $check(e_{ui})$ = = '$F$' and $d(e_{ui}) \neq (0,\ldots, 0)$;
(13) *// select a non-zero delay vector as the retiming base r*
(14) $rf(v_i) = rf(v_i) + r$; $rf(v_{ij}) = rf(v_{ij}) + r$; *// set retiming function*
(15) $rf(s_{ij}) = rf(s_{ij}) + r$; $rf(u_{ij}) = rf(u_{ij}) + r$; *// set retiming function*
(16) $G_r$ = retime $G$ using above retiming functions; // $G_r = (V, E, X, d_r, P)$
(17) **While** ($\exists\, e \in E$ such that $d_r(e) = = -r$) *// remove delay vector* $d_r(e) = -r$
(18) Assume that $e$ connects nodes $v_s$ and $v_l$; // $P(v_s) = S$, $P(v_l) = L$
(19) $v_a \in V$ is the predecessor of $v_s$; // $P(v_a) = M/A$
(20) $u_{l1}, u_{l2} \in V$ are predecessors of $v_a$; // $P(u_{li}) = L/C$
(21) /* $u_{l1}$ *and* $u_{l2}$ *are two input operands of* $v_a$ */
(22) $rf(v_s) = rf(v_s) + r$; $rf(v_a) = rf(v_a) + r$; *// set retiming function*
(23) **If** ($u_{li}$ has successors $v_{ai}$ other than $v_a$) // $P(v_{ai}) = M/A$, $P(u_{li}) = L$
(24) Insert node $v_{xi}$ into $V$, set $P(v_{xi}) = L$; // *split* $u_{li}$ *to* $u_{li}$ *and* $v_{xi}$
(25) Delete $e_{lia}$ from $E$; // $e_{lia}$ *connects* $u_{li}$ *and* $v_a$
(26) Insert edge $e_{xia}$ into $E$; // $e_{xia}$ *connects* $v_{xi}$ *and* $v_a$
(27) $rf(v_{xi}) = rf(v_{xi}) + r$; *// set retiming function*
(28) **Else** $rf(u_{li}) = rf(u_{li}) + r$; *// set retiming function*
(29) $G_r$ = retime $G$ using above retiming functions;
(30) **End while**
(31) **If** ($\exists$ a vector $s$ such that $s \cdot d_r(e) \geq 0$) // $G_r$ *is realizable*
(32) $G = G_r$; $merge(v_i)$ = '$T$'; **End if**
(33) $check(e_{ui})$ = '$T$';
(34) **End while** *// all* $d(e_{ui})$ *are checked*
(35) Insert $v_x$ into $V$; Insert $e_{ux}$ into $E$;
(36) **For** ($k = 1$; $k \leq m$, $k$++) *// merge* $v_i$ *with the same* $e_{ui}$
(37) **If** ($merge(v_i)$ = = '$T$')
(38) Delete $v_k$ from $V$; Delete $e_{kkj}$ from $E$; // $e_{kkj}$ *connects* $v_k$ *and* $v_{kj}$
(39) Delete $e_{uk}$ from $E$; Insert $e_{xkj}$ into $E$; // $e_{xkj}$ connects $v_x$ and $v_{kj}$
(40) **End if**
(41) **End for**
(42) **Return** $rf(v)$, $G_r$;

Figure 6.5. The MDFG reconstructing algorithm.

nodes $v_i$ and $v_j$ with type $S$ and $L$, respectively, which represents the loop-carried data dependence of the given loop. After reconstructing the MDFG, above features also must be satisfied in addition to guarantee the retimed MDFG is realizable.

The proposed MDFG reconstruction algorithm, listed in Figure 6.5, contains three main phases: node retiming (Lines 12~16), graph realization (Lines 17~32), and graph modification (Lines 35~41). For a given MDFG $G$ and an exploitable sharing set $S = \{v_1 \dots v_m\}$, assume that $v_1 \dots v_m$ have the same predecessor $v_u$; our goal is to make as many as possible $d(e_{ui})$ equal using the retiming technique. In our design, we simply select a non-zero delay vector $d(e_{ui})$ as the retiming base, which can transfer $d(e_{ui})$ to a zero delay edge after retiming node $v_i$. Then, to satisfy features of MDFG described above, all nodes listed in Lines 14~15 must be concurrently retimed. For example, in Figure 6.6(a), there exists a sharing set $S = \{6, 9\}$ and an exploitable sharing set $S' = \{5, 15\}$. After retiming nodes 4~10 with $r = (0, 1)$ equal to $d(e_{514})$, the sharing set $S$ is extended to $\{6, 9, 16\}$ as shown in Figure 6.6(b), which indicates the number of potential operand reutilizations is increased. This node retiming phase will be applied iteratively until all $d(e_{ui})$ have been selected as the retiming base.

In the second phase, we check and guarantee the retimed MDFG $G_r$ is realizable. As described in section 2.1, a realizable MDFG $G$ must have a schedule vector, $s$, such that $s \cdot d \geq 0$ for all loop-carried data dependencies $d$. However, because we directly select a non-zero delay vector, $r$, as the retiming base in the previous phase, two edges with opposite delay vectors $r$ and $-r$ may exist in $G_r$ simultaneously. In this case above realizable condition can be satisfied, but $G_r$ still will not be successfully executed. $d_r(e_{711})$ and $d_r(e_{34})$ in Figure 6.6(b) are such an example. Although a vector $s = (1, 0)$ makes $s \cdot d \geq 0$ for all $d \in G_r$, $G_r$ is actually illegal because iterations $(i, j)$ and $(i, j + 1)$ will depend on each other. In order to resolve this case, we design a mechanism to retime additional nodes backtracked from edge $e$ with $d_r(e) = -r$.

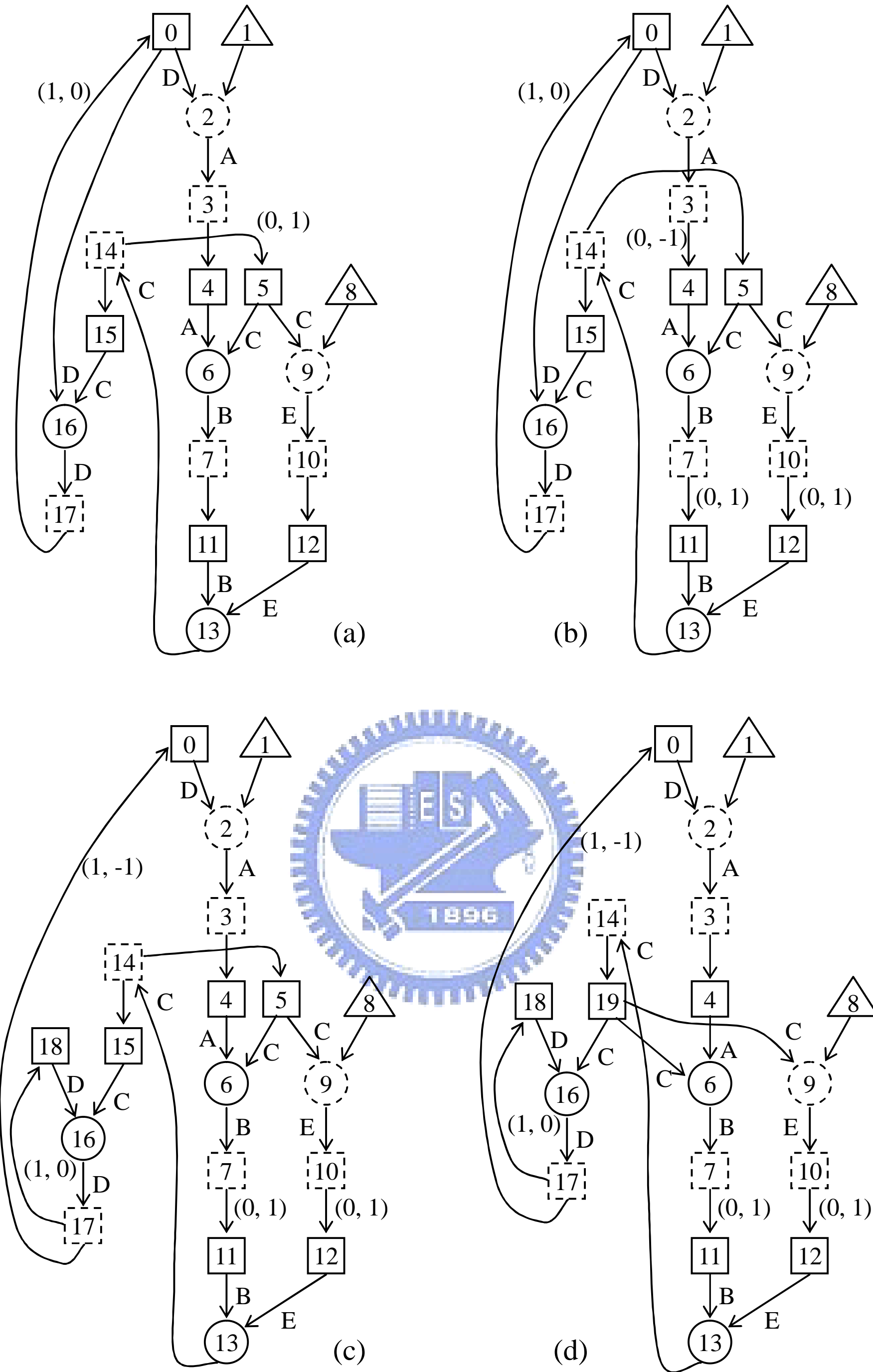


Figure 6.6. An example used to illustrate steps of RSER.

Meanwhile, if the backtracking steps reach a node with type *L* with multiple successors, we split that node (Lines 23~28). Figure 6.6(c) shows the modified realizable MDFG $G_r$ after splitting node 0 and retiming nodes 0~3. This phase will be also iteratively applied until the retimed graph is realizable.

1. $G_c$ = Construct MDFG;
    1.1. Partition variables to memory banks;
    1.2. Unfold or tile $G_c$ if necessary;
2. **$G_c$ = Reconstruct $G_c$;** ***// apply the algorithm listed in Figure* 6.5**
3. $G_t$ = Construct TDAG ($G_c$);
4. $S$ = Schedule all instructions except memory loads ($G_t$);
    4.1. Group ALU instructions into sharing sets;
    *// additional step used in RSOR and RSER*
    4.2. $G_{op}$ = Construct DAG $G_{op}$ ($G_t$);
    4.3. $S$ = Schedule nodes in $G_{op}$ ($G_{op}$);
    *// nodes in the same sharing set are restrictively scheduled*
    4.4. $S$ = Determine and solve accumulator spills ($S$, $G_{op}$);
5. $S$ = Schedule memory load instructions ($S$, $G_t$);
6. $S$ = Retime the initial scheduling result ($S$, $G_t$);

Figure 6.7. The overall scheduling algorithm of RSER.

Finally, the third phase is used to merge nodes in an exploitable sharing set that reference the same array element. In our design we will merge them to an additional node. The final graph $G_r$ after applying our MDFG reconstruction algorithm is shown in Figure 6.6(d).

### 6.3.2 Detailed Algorithms of RSER

In the previous subsection we describe the proposed algorithm to increase the number of potential OPRs. As shown in Figure 6.7, when we insert this algorithm into RSOR, we will obtain our second method RSER. Figure 6.8(a)(b) shows the corresponding TDAGs for Figure 6.6(a)(d). Suppose the target architecture is the Motorola DSP56000, Figure 6.9(a)(b) shows the scheduling results of Figure 6.8(a)(b), which are actually scheduling results of Figure 6.6(a) using RSOR and RSER, respectively. From these schedules, we find that for a single iteration in the repetitive pattern, using RSER achieves one more OPR with one-time step longer schedule length. More instruction counts are obviously required using RSER, because the original MDFG is reconstructed and some nodes are split. This feature indicates that

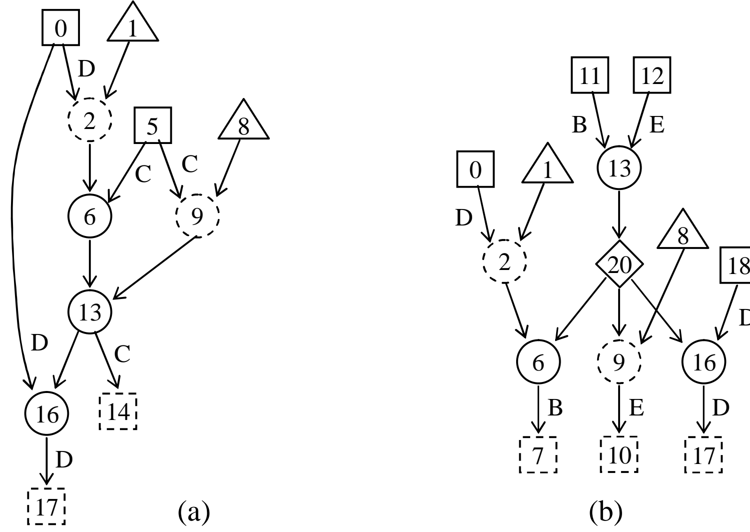



Figure 6.8. The corresponding TDAG of (a) Figure 6.6(a), (b) Figure 6.6(d).

| $t$ | $FU_1$ | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | 6 | | 1 |
| 2 | 9 | 0 | 5 |
| 3 | 13 | | |
| 4 | 16 | 8 | 14 |
| 5 | 2 | 17 | |

(a)

| $t$ | $FU_1$ | $M_1$ | $M_2$ |
|---|---|---|---|
| 1 | 13 | 0 | 1 |
| 2 | 2 | 20 | 8 |
| 3 | 6 | 18 | 12 |
| 4 | 9 | 7 | |
| 5 | 16 | 11 | 10 |
| 6 | | 17 | |

(b)

operand reutilization

Figure 6.9. Scheduling results of Figure 6.6(a). (a) RSOR, (b) RSOE.

in RSER, OPR is a trade off for schedule length as well as instruction count. In section 6.4 we will describe further evaluations of RSER.

### 6.3.3 The Difference between Proposed Methods and Other Methods

In the following, we describe the difference among methods RSOR, RSER, LPLS [4], power-conscious loop folding [24], and method proposed in [28], all are designed based on the operand sharing technique. Among these methods, the retiming technique is never applied in LPLS, which only uses a modified list scheduling to consider the operand sharing. RSOR focus on achieving potential OPRs within an iteration, and the retiming technique is used once to compact the schedule. Other three methods RSER, power-conscious loop folding, and the method [28] all use the

retiming technique to generate instructions with common operands hidden inside the MDFG. Power-conscious loop folding is a basic method. After finding instructions sharing an operand in different iterations, the retiming technique is used to move them to the same iteration. The method [28] contains a *force-directed retiming* mechanism to determine which instruction must be retimed, and aim to make as many instructions as possible take common operands as their inputs. Apparently, these two methods only apply the retiming technique to achieve more OPRs. On the other hand, in our RSER, the retiming technique is applied more than once for different purposes. First, after determining exploitable sharing sets, it is used to gather instructions sharing common operands. Note that before retiming a specific retiming base must be chosen. That is, to remove more non-zero delay edges during MDFG reconstruction we may retime the MDFG several times with different retiming bases. Then, to compact the initial scheduling result, the retiming technique is used once more to partial overlap the execution time of successive iterations. From above description, we expect that using RSER can produce schedules with shorter lengths than using methods in [24, 28].

However, applying the retiming technique will generate corresponding prologue and epilogue codes that must be executed separately before and after the iterative patterns. If code sizes of the prologue and epilogue are too large, they will cost greater overall execution time and more power consumption of the given loop. We have proven that the overall schedule length is strongly dependent on which schedule vector, as well as retiming base, been selected [41]. Therefore, to avoid generating too many prologue and epilogue codes, we restrict that only two retiming bases, (0, 1) and (1, 0), can be selected in the MDFG reconstruction algorithm. This restriction means that in RSER the retiming technique is applied at most three times. Detailed evaluations of RSOR, RSER, and other energy-efficient instruction scheduling methods will be given in section 6.4.

Table 6.4. The number of OPRs obtained by different scheduling methods.

| | 1 FU, 2 acc, 4 reg, 2 mem | | | | 2 FU, 4 acc, 4 reg, 2 mem | | | |
|---|---|---|---|---|---|---|---|---|
| | LPLS | RSOR | Kim | RSER | LPLS | RSOR | Kim | RSER |
| Wave Digital Filter | 0 | 0.5 | 1 | 1 | 0 | 0.5 | 0 | 1 |
| Filter | 0 | 0 | 0 | | 0 | 0 | 0 | |
| IIR2D | 0 | 0 | 4 | 4 | 0 | 0 | 4 | 4 |
| forward-substitution | 1 | 1.5 | 2 | 2 | 2 | 2 | 1 | 2 |
| THCS | 1 | 1 | 1 | | 2 | 2 | 0 | |
| DFT | 3 | 4 | 3 | 7 | 3 | 4 | 3 | 7 |
| Floyd-Steinberg | 9 | 9 | 9 | | 9 | 9 | 6 | |
| Transmission | 4 | 4 | 4 | | 5 | 5 | 4 | |
| IIR1D | 2 | 3 | 4 | 4 | 2 | 3 | 3 | 4 |
| Equation Solver | 5 | 5 | 5 | 4 | 5 | 6 | 4 | 5 |
| All-pole Lattice | 3 | 3 | 2 | | 6 | 6 | 2 | |
| Elliptic Filter | 9 | 9 | 9 | | 11 | 11 | 11 | |

Finally, in section 5.4 we have presented that with minor modifications, our hypothetical machine model and RSSA can be apply to real DSP families with various architectural features. Since in RSOR and RSER we apply the same mechanisms as in RSSA to schedule instructions and insert spill codes, both RSOR and RSER also can suit real DSP families.

## 6.4 Performance Evaluations [58]

In this section, we evaluate RSOR and RSER using selected MDFGs and the hypothetical machine model. LPLS [4] and Kim et al. [28] are also evaluated using the variable partition mechanism presented in RSVR [30] for comparison, after inserting necessary spill codes. Similar as in subsection 6.2.2, we still use evaluation metrics including schedule length, instruction count, the number of OPRs, and approximate current, and only show the best result derived by RSOR and RSER.

Table 6.4 lists the number of OPRs for a single iteration in the repetitive pattern. Note that not all selected MDFGs contain exploitable sharing sets, so we only apply RSER to MDFGs that have potential operand sharing in different iterations. From this

table, if the given MDFG has exploitable sharing sets, using RSER and Kim et al. [28] can clearly produce schedules with more OPRs compared to using LPLS and RSOR. That is, for a single iteration in the repetitive pattern, schedules generated by RSER and Kim et al. [28] will cost lower power consumption at function units. In addition, for an MDFG without exploitable sharing set, using RSOR still generates a similar number of OPRs to LPLS and Kim et al. [28]. This result shows that all three methods can successfully exploit potential operand sharing within an iteration. For comparison between two different architectures, Table 6.4 shows that all methods, except Kim et al. [28], perform better when the target architecture has more function units. This situation indicates whether an MDFG is reconstructed or not, using more function units is beneficial in achieving more OPRs. Thus, we conclude that when the number of OPRs is taken as the evaluation metric, RSOR and RSER are at least as effective as the previous methods. Furthermore, if the given loop contains potential operand sharing in different iterations, applying the retiming technique to exploit it is positive for energy-efficient instruction scheduling.

Table 6.5 lists the schedule length, instruction count, and approximate current of a single iteration in Motorola DSP56000 architecture. From these results, we find that RSOR and RSER achieve shorter schedules than LPLS and Kim et al. [28] in most cases, because both our methods apply the retiming technique to effectively explore the instruction-level parallelism between successive iterations. But the effectiveness between RSOR and RSER is uncertain, and will depend on the topological difference between the MDFGs before and after reconstruction. Hence, we conclude that RSOR and RSER are more effective than previous methods when the schedule length is the evaluation metric. On the other hand, in most cases using LPLS and Kim et al. [28] will generate schedules with the least and most instructions, respectively. If a MDFG contains exploitable sharing sets, applying RSER will require greater instruction

Table 6.5. The comparison among four methods (under Motorola DSP56000).

| | LPLS | | | RSOR | | | Kim | | | RSER | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | length | instr. count | current | length | instr. count | current | length | instr. count | current | length | instr. count | current |
| [1] | 6 | 13 | 820 | 5 | 13 | 770 | 6 | 13 | 830 | 4 | 11.5 | 640 |
| [2] | 8 | 11 | 920 | 5 | 10.5 | 655 | 6 | 10 | 760 | | | |
| [3] | 20 | 37 | 2690 | 16 | 37 | 2435 | 18 | 42 | 2604 | 17.5 | 39 | 2428 |
| [4] | 7 | 10 | 802 | 5 | 10.5 | 717 | 7 | 15 | 922 | 5 | 12.5 | 717 |
| [5] | 6 | 10 | 712 | 4 | 9.5 | 587 | 6 | 10 | 712 | | | |
| [6] | 14 | 30 | 1866 | 12.5 | 32 | 1756 | 15 | 34 | 1986 | 12.5 | 31.5 | 1691 |
| [7] | 20 | 39 | 1630 | 17.5 | 39 | 2440 | 19 | 39 | 1520 | | | |
| [8] | 14 | 29 | 1940 | 12 | 28 | 1770 | 14 | 29 | 1930 | | | |
| [9] | 10 | 18 | 1222 | 8 | 19 | 1111 | 10 | 23 | 1298 | 8.5 | 21 | 1118 |
| [10] | 14 | 24 | 1776 | 12 | 24.5 | 1651 | 13 | 30 | 1816 | 11.5 | 27.5 | 1646 |
| [11] | 21 | 35 | 2640 | 17 | 34.5 | 2300 | 17 | 39 | 2450 | | | |
| [12] | 40 | 77 | 5162 | 35 | 75 | 4827 | 36 | 73 | 4782 | | | |

[1] Wave Digital Filter
[2] Filter
[3] Infinite Impulse Response Filter 2D
[4] forward-substitution
[5] Toeplitz Hyperbolic Cholesky Solver
[6] Discrete Fourier Transform
[7] Floyd-Steinberg
[8] Transmission Line
[9] Infinite Impulse Response Filter 1D
[10] Differential Equation Solver
[11] All-pole Lattice Filter
[12] Elliptic Filter

count than RSOR but still less than Kim et al. [28]. Note that the number of ALU instructions for a MDFG is fixed whichever scheduling method is applied. That is, a schedule with more instruction counts represents more inserted spill codes, which are usually extra memory accesses. Based on the instruction-level power model presented in [45], to execute every instruction will cost the base cost, so a schedule with less instruction count will benefit code size as well as power consumption. As for the approximate current, in most cases RSOR and RSER outperform LPLS and Kim et al. [28]. Obviously the main reason is using our methods can obtain shorter schedules. For comparison between RSOR and RSER, RSER is usually better, even if the number of memory accesses may increase after MDFG reconstruction. This is because using RSER can further lower the power consumed at function units, and the schedule length is only slightly increased.

Table 6.6. Definitions of variables used in the analytic model.

| Variable | Definition |
|---|---|
| $N$ | Number of memory modules |
| $m$ | Loop bound of the outer loop for a two-dimensional nested loop<br>Loop bound for an one-dimensional loop |
| $n$ | Loop bound of the inner loop for a two-dimensional nested loop |
| $(s_1, s_2)$ | Schedule vector selected for retiming during instruction scheduling |
| *list* | Schedule length of an iteration in the repetitive pattern produced by *list scheduling* method |
| *length* | Schedule length of an iteration in the repetitive pattern |
| *prologue* | Schedule length of the prologue generated during instruction scheduling |
| *eplogue* | Schedule length of the prologue generated during instruction scheduling |
| $d$ | Retiming depth obtained during instruction scheduling |
| *half* ($k$, $N$) | Schedule length of $k$ original iterations under $N$ memory modules |
| *exp*1 | Schedule length of the prologue generated during MDFG reconstructing after first retiming |
| *exe*1 | Schedule length of the epilogue generated during MDFG reconstructing after first retiming |
| *exd*1 | Retiming depth obtained during MDFG reconstructing after first retiming |
| *exp*2 | Schedule length of the prologue generated during MDFG reconstructing after second retiming |
| *exe*2 | Schedule length of the epilogue generated during MDFG reconstructing after second retiming |
| *exd*2 | Retiming depth obtained during MDFG reconstructing after second retiming |

In the following, we focus on the entire retimed loop to compare the overall schedule length. In chapter 3 we have introduced an analytic model to calculate the overall schedule length of a retimed MDFG. Formulas (A.1)~(A.5) can be directly used to test methods RSOR and Kim et al. [28], and we extend it further to treat RSER. Table 6.6 lists variables used in the extended analytic model. Note that we restrict that only two retiming bases, (0, 1) and (1, 0), can be selected in the MDFG reconstruction algorithm. That is, the original MDFG is retimed at most twice during reconstructing, with two retiming bases been used in different sequences. In the extended analytic model, we directly assume that every MDFG is retimed twice, and design corresponded formulas to calculate the overall schedule length. If the given

MDFG is only retimed once, variables *exp*2, *exe*2, and *exd*2 can be simply set to zero. Detailed derivations of new formulas are listed in appendix B.

Figures 6.10 and 6.11 show the overall schedule lengths of the entire retimed loop when the target architecture has one or two function units, respectively. From these figures, for most applications RSOR obtain shorter overall schedule lengths than LPLS and Kim et al. [28]. If the given MDFG contains exploitable sharing sets, using RSER may not produce shorter overall schedule lengths compared to RSOR, but still outperforms LPLS and Kim et al. [28]. These results are the same as the evaluations based on a single iteration in the repetitive pattern. That is, although the two proposed methods, especially RSER, require longer time to run the prologue and epilogue, the overall performance is still better because they can effectively explore the instruction-level parallelism between successive iterations.

Finally, we summarize above evaluations. The overall schedule lengths obtained by RSOR and RSER are obviously shorter than those of previous methods, although RSER may require more time to run corresponding prologue and epilogue codes. If the number of OPRs is the evaluation metric, RSOR and RSER are at least as effective as LPLS and Kim et al. [28]. Recall that the average power consumption of the function unit is clearly less when an operand remains unchanged, and the total power consumption of a schedule equals to the sum of power consumed by all instructions. Since proposed RSOR and RSER perform better on both evaluation metrics schedule length and the number of OPRs, we conclude that they are energy-efficient code generation methods. As for the instruction count, our proposed methods are still very effective for the repetitive pattern due to fewer inserted spill codes. But their corresponding prologue and epilogue codes have to be stored in addition to the repetitive pattern, so our RSER will require much more memory space to store the scheduling results compared to other related methods.

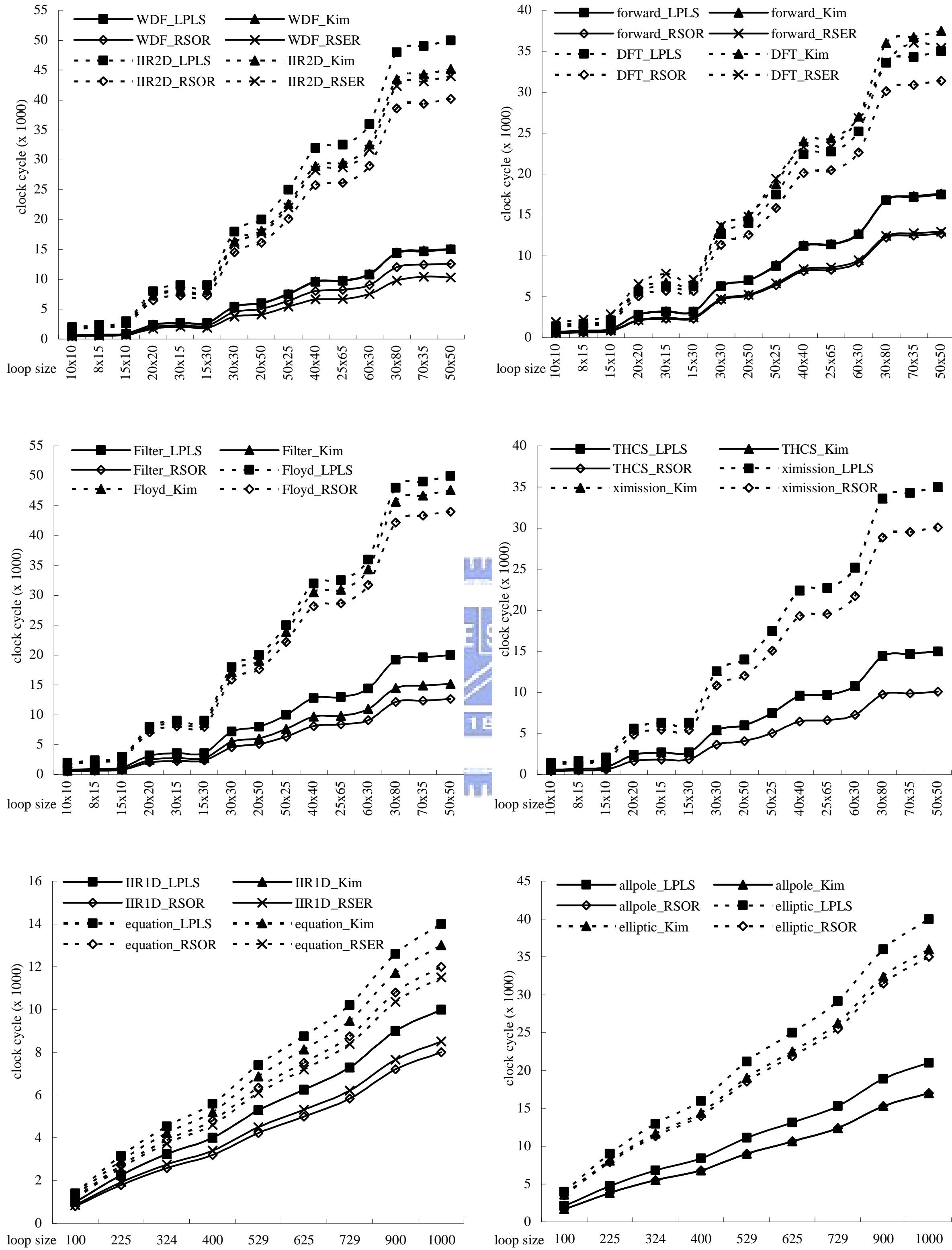



Figure 6.10. Experimental results of DSP applications (1 function unit, overall schedule length).

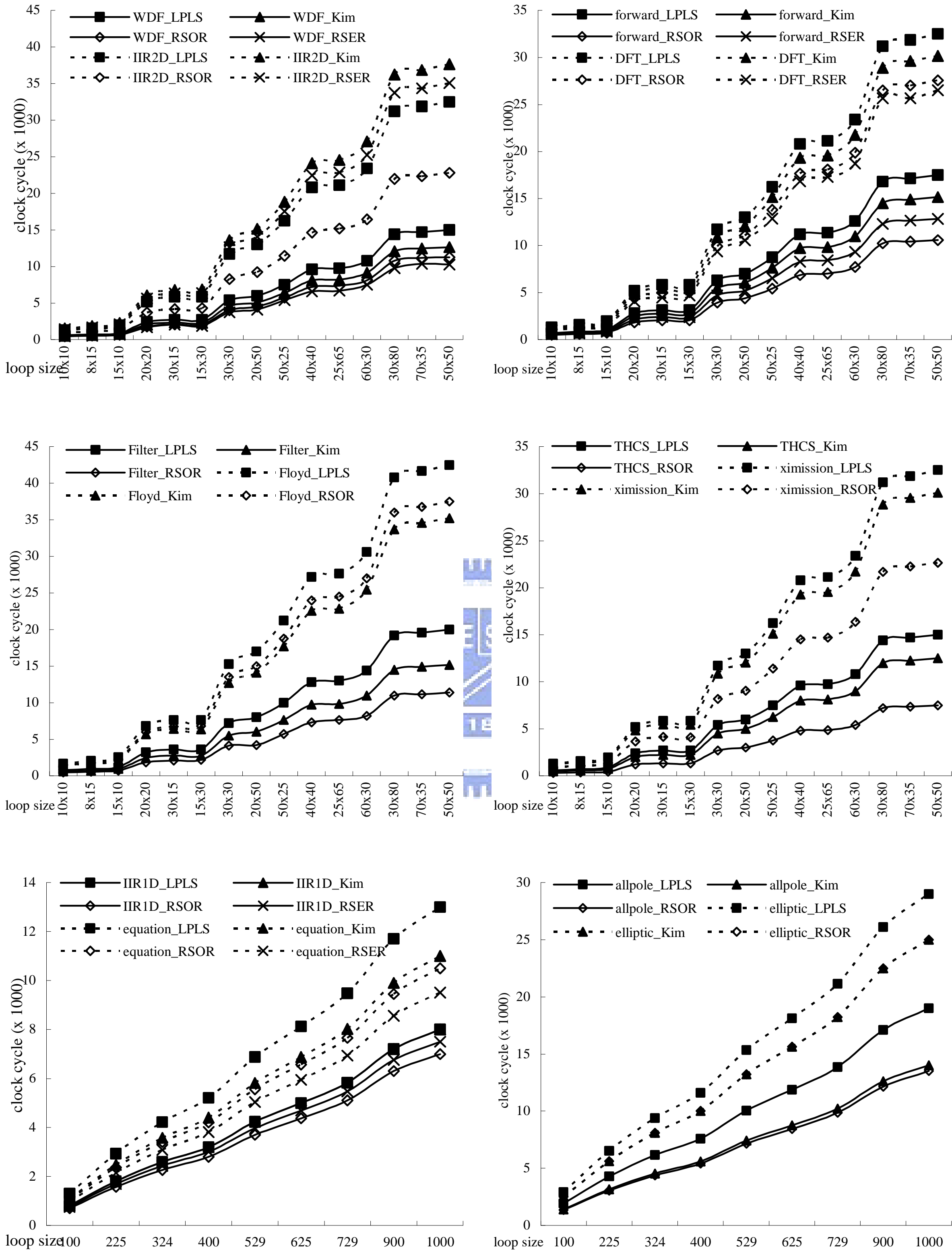



Figure 6.11. Experimental results of DSP applications (2 function units, overall schedule length).

# Chapter 7. Conclusions and Future Work

## 7.1 Conclusions

In this thesis we focus on proposing effective code generation method to schedule uniform loops on DSP with multiple data memory banks. A hypothetical machine model is also defined to simulate a scalable DSP architecture, in order to deep study the influence of differing number of resources on the scheduling result. Our research contains three main issues: variable partition mechanisms, effective methods containing all code generation phases, and energy-efficient methods based on the operand sharing technique. In each issue we proposed some effective methods, and evaluate those using selected MDFGs and an analytic model. In the following we give brief conclusions for our research.

(1) Variable partition mechanisms

In the first issue, we define three simple mechanisms to partition variables based on their array indices. After enlarging the given MDFG using different techniques to suit each variable partition mechanism, the multi-dimensional rotation scheduling is applied to schedule instructions and three corresponded code generation methods RSF, RST, and RSP are proposed. Because variables are never repartitioned during instruction scheduling, these three methods are apparently simpler and more efficient compared to a similar study RSVR. In addition, the enlarged iteration used in our methods gives a more global view of data dependencies, which is useful to explore the instruction-level parallelism between successive iterations using the retiming technique. We also define an analytic model and some formulas to calculate the overall schedule length of a retimed loop. From evaluation results, our methods achieve schedules with equal even shorter lengths compared to those of RSVR, not

only for a single iteration in the repetitive pattern but also for the entire retimed loop. Three variable partition mechanisms defined in RSVR, RSF, and RST are used in our subsequent several studies.

(2) Effective methods containing all code generation phases

For DSP with multiple data memory banks, the complete code generation process must include five phases. Because these phases are extremely data dependent, to consider more phases at a time will lead more effective results. In our second study issue, we first focus on Motorola DSP56000 and propose method RSSP to cover all code generation phases. In RSSP a TDAG is defined and transferred from the given MDFG to remove possible unnecessary memory accesses. Then, the main feature of RSSP is to predict the occurrence of accumulator spills and generate corresponding spill codes in advance. These spill codes will be scheduled in parallel with other instructions, which is beneficial to generate a more compact and shorter schedule. After generating an initial schedule, the retiming technique is also applied to fully utilize resources. From evaluation results, RSSP obviously outperforms methods RSF, RST, and RSP, because it schedules instructions based on the TDAG which contains less instructions than the MDFG. Comparing to other methods designed for Motorola DSP56000 our RSSP still generates schedules with shorter lengths, in both a single iteration in the repetitive pattern and the entire retimed loop.

However, although RSSP seems quire effective, it is designed dedicated to Motorola DSP56000 and not scalable. Therefore, we further propose a general method RSSA, which can suit various DSPs with different architectural features. In RSSA, in addition to shorter schedule length, we take fewer spill codes as the second scheduling goal due to its importance in DSP. Instructions are still scheduled based on the TDAG to remove possible unnecessary memory accesses. But we no longer predict the

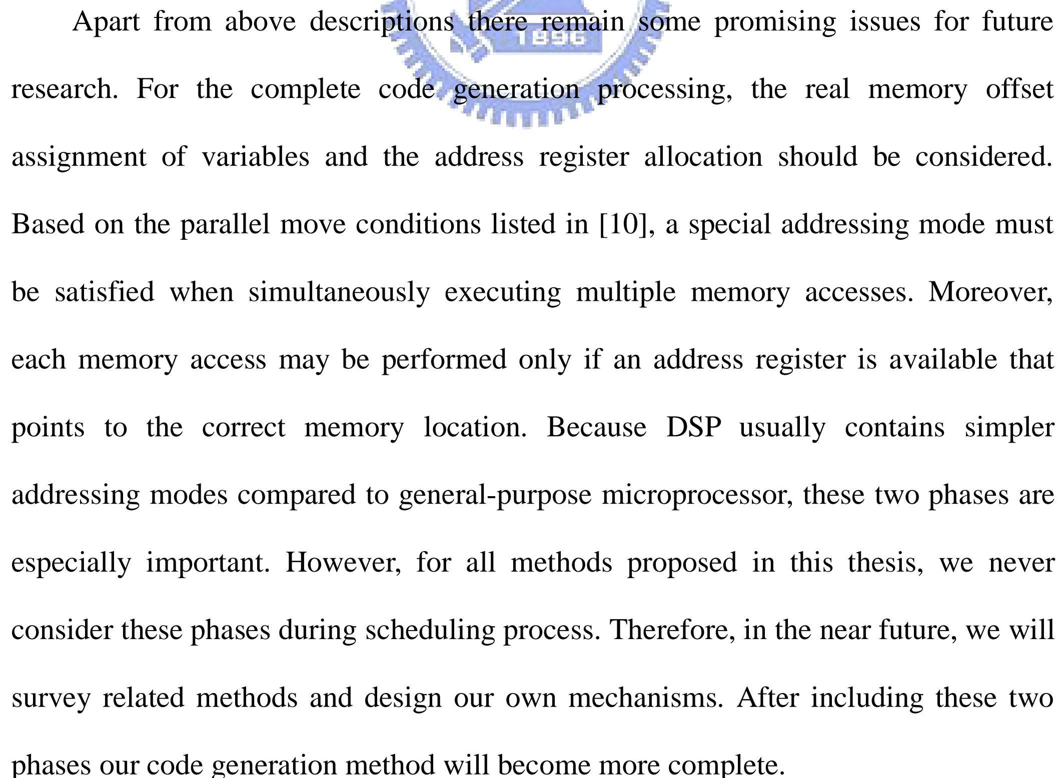
occurrences of accumulator spills in RSSA, because the predicting result becomes inaccurate easily when the target architecture is not specific. During scheduling instructions, several variables are dynamically updated to record the number of resources been occupied at every time step. When an accumulator spill is detected by checking those variables, we prefer to transfer the overwritten ALU result to an available register and temporarily store it to data memory only when required. After generating an initial schedule, the retiming technique is still applied in RSSA to explore the instruction-level parallelism between successive iterations. Suppose the target architecture equals to the Motorola DSP56000, RSSA usually achieves the shortest schedule length and considerably fewer spill codes compared to other related studies. We also define a hypothetical machine model to simulate architectures with different number of resources. This parameterized model is basically extended from the Motorola DSP56000, but can be apply to other real DSP families with minor modifications. After evaluating MDFGs using RSSA on this hypothetical machine model, the influence of differing number of resources on the scheduling results is deep studied. From evaluation results, we conclude that adding more accumulators to keep more ALU results for further using is the most efficient way to reduce spill codes. Increasing the number of registers or data memory banks is also useful, but its improvement is not as obvious as using more accumulators. As for achieving shorter schedule length, adding additional function units and accumulators concurrently is certainly necessary. Furthermore, we also find that the variable partition mechanism proposed in RSF is unsuitable for one-dimensional MDFGs. This is because most memory accesses will reference variables from the same data memory bank after applying loop unfolding, which may easily fail to be scheduled successfully in time and lengthen the scheduling results.

(3) Energy-efficient code generation methods

Because a function unit will cost less power to execute an instruction when one of its operand remains unchanged, the operand sharing is a useful technique in DSP for low power design. In our third study issue in this thesis, we propose two energy-efficient code generation methods RSOR and RSER, both based on the operand sharing technique. RSOR is directly extended from RSSA and aimed to explore potential operand sharing within an iteration. In RSOR we define a sharing set to group ALU instructions with common operand, and restrictedly schedule instructions in a sharing set to the same function unit at consecutive time steps to reuse operands. However, in real designs common operands are not encountered very frequently, so using RSOR only can reduce insignificant power consumption. But potential operand sharing may be hidden inside the original MDFG, which can be generated after loop transformations. Therefore, we propose our second energy-efficient method RSER, which is extended from RSOR and aimed to further explore operand sharing between different iterations. In RSER an exploitable sharing set is defined to group load variable instructions that reference the same array element in different iterations. Then, we design a MDFG reconstruction algorithm based on the retiming technique, and apply the method RSOR to schedule the reconstructed MDFG. Hence, operand sharing within an iteration and resided in different iterations can be both explored using RSER. We also extend the analytic model defined in methods RSF and RST to calculate the overall schedule length and number of OPRs for the entire retimed loop after applying RSOR and RSER. From evaluation results, we find that both RSOR and RSER can successfully explore operand sharing within an iteration. When the given MDFG contains exploitable sharing sets, using RSER achieves schedules with further more OPRs, which represents that exploiting the operand sharing in different iterations is beneficial for energy-efficient instruction scheduling. On the other hand,

schedules generated by RSOR and RSER may have slightly longer schedule lengths for a single repetitive iteration compared to those of RSSA. The main reason is that some ALU instructions are restrictedly scheduled to the same function unit, so the instruction-level parallelism between them cannot be successfully explored. But for the entire retimed loop, both RSOR and RSER still achieve shorter overall schedule lengths compared to related studies, because they apply the retiming technique to fully utilize resources as far as possible. Finally, as for the instruction count, our two proposed methods insert quite fewer spill codes for a repetitive iteration, but RSER will generate considerable prologue and epilogue codes. That is, if the instruction count is taken as the evaluation metric, RSER will require much more memory space to store scheduling results compared to related methods.

### 7.2 Future Work



Apart from above descriptions there remain some promising issues for future research. For the complete code generation processing, the real memory offset assignment of variables and the address register allocation should be considered. Based on the parallel move conditions listed in [10], a special addressing mode must be satisfied when simultaneously executing multiple memory accesses. Moreover, each memory access may be performed only if an address register is available that points to the correct memory location. Because DSP usually contains simpler addressing modes compared to general-purpose microprocessor, these two phases are especially important. However, for all methods proposed in this thesis, we never consider these phases during scheduling process. Therefore, in the near future, we will survey related methods and design our own mechanisms. After including these two phases our code generation method will become more complete.

The second promising issue is about the code size reduction. In our methods, we

frequently use the retiming technique to increase performances. Applying retiming to schedule uniform loops is actually effective to reduce the schedule length, but the main problem is the generation of prologue and epilogue. We have shown that in our proposed methods the prologue and epilogue will not cost too much execution time to degrade the performance. However, extra codes for prologue and epilogue require considerable space to be stored in memory, especially in RSER because we apply the retiming technique more than once. Authors of [59] propose a mechanism to avoid storing prologue and epilogue codes. Their main idea is to claim that the execution of prologue and epilogue can be simulated by conditionally executing the repetitive iteration. Hence, only a repetitive iteration has to be stored, and additional instructions are required to control the execution of the entire retimed loop. This idea can be used in our methods RSSP, RSSA, and RSOR. But in RSER it is unsuitable, because using RSER will generate several pairs of prologue and epilogue, and not all of them can be simulated by a single repetitive iteration. Therefore, in the near future, we will survey related methods and try to design effective mechanisms to reduce the prologue and epilogue codes. After reducing the required code size our code generation methods will be more practical.

Finally, we can try to realize proposed methods and do some precise evaluations. In this thesis we use analytic model to calculate the schedule length and instruction count. As for the power consumption, information provided in [20] also only can approximately estimate the required current. If our methods can be realized and tested by more accurate tools, their effectiveness and efficiency will be evaluated more precisely. Our code generation methods are all systematic and represented by definite algorithms. Therefore, we believe that they can be integrated into real DSP compiler and successfully executed.

**References**

[1] C. Hsu and Y.L. Jeang, "Pipeline Scheduling Techniques in High-Level Synthesis", *Proc. of 6th Annual IEEE International ASIC Conference and Exhibition, Rochester*, pp. 396-403, 1993.

[2] S.Y. Kung, **VLSI Array Processors**, Prentice Hall, Englewood, NJ, 1988.

[3] V.K. Madisetti, **VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis**, Butterworth-Heinemann, Boston, 1995.

[4] E. Musoll and J. Cortadella, "Scheduling and Resource Binding for Low Power", *Proc. of International Symposium on System Synthesis*, pp. 104-109, April 1995.

[5] K.S. Khouri, G. Lakshminarayana, and N.K. Jha, "High-level Synthesis of Low-power Control-flow Intensive Circuits", *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, Vol. 18, No. 12, pp. 1715-1729, Dec. 1999.

[6] Z. Wang and X.S. Hu, "Power Aware Variable Partitioning and Instruction Scheduling for Multiple Memory Banks", *Proc. of Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1, pp. 312-317, Feb. 2004.

[7] J. Eyre and J. Bier, "The Evolution of DSP Processors", *IEEE Signal Processing Magazine*, Vol. 17, Issue 2, pp. 43-51, March 2000.

[8] P. Lapsley, J. Bier, A. Shoham, and E.A. Lee, **DSP Processor Fundamentals: Architectures and Features**, Berkeley Design Technology, Inc. 1996.

[9] J. Cho, Y. Paek, and D. Whalley, "Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms", *Proc. of ACM Joint Conference LCTES-SCOPES*, pp. 130-138, June 2002.

[10] DSP56000/DSP56001 Digital Signal Processor User's Manual, Motorola Inc.; DSP56300 Family Manual, Motorola Inc.

[11] http://www.physics.otago.ac.nz/internal/ELEC401/ADSP2100/adsp2101.html

[12] http://www.chipcatalog.com/NEC/UPD77016.htm

[13] TMS320C6000 Technical Brief, Texas Instruments.

[14] Y.H. Lee and C. Chen, "Efficient Variable Partitioning and Scheduling Methods of Multiple Memory Modules for DSP", *Proc. of 10th Workshop on Compiler Techniques for High-Performance Computing*, pp. 80-89, March 2004.

[15] M.A.R. Saghir, P. Chow, and C.G. Lee, "Towards Better DSP Architectures and Compilers", *Proc. of International Conference on Signal Processing Applications and Technology*, pp. 658-664, Oct. 1994.

[16] R. Leupers and D. Kotte, "Variable Partitioning for Dual Memory Bank DSPs", *Proc. of International Conference on Acoustics, Speech, and Signal Processing*, Vol. 2, pp. 1121-1124, 2001.

[17] A. Sudarsanam and S. Malik, "Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, No. 2, pp. 242-264, April 2000.

[18] M.A.R. Saghir, P. Chow, and C.G. Lee, "Exploiting Dual-memory Banks in Digital Signal Processors", *Proc. of 7th International Conference on Architecture Support for Programming Language and Operating Systems*, pp. 234-243, 1996.

[19] Q. Zhuge, B. Xiao, and E.H.M. Sha, "Exploring Variable Partitioning for Dual Data-memory Bank Processors", *Proc. of 34th International Symposium on Microarchitecture*, pp. 45-52, Dec. 2001.

[20] W.T. Shiue, "Energy-efficient Backend Compiler Design for Embedded Systems", *Proc. of 10th International Conference on Electrical and Electronic Technology*, Vol. 1, pp. 103-109, Aug. 2001.

[21] J.M. Daveau, T. Thery, T. Lepley, and M. Santana, "A Retargetable Register Allocation Framework for Embedded Processors", *Proc. of ACM SIGPLAN/ SIGBED*, pp. 202-210, June 2004.

[22] B. Scholz and E. Eckstein, "Register Allocation for Irregular Architectures", *Proc. of ACM Joint Conference LCTES-SCOPES*, pp. 139-148, June 2002.

[23] X. Zhuang, T. Zhang, and S. Pande, "Hardware-managed Register Allocation for Embedded Processors", *Proc. of ACM SIGPLAN/SIGBED*, pp. 192-201, June 2004.

[24] D. Kim and K. Choi, "Power-conscious High Level Synthesis using Loop Folding", *Proc. of Design Automation Conference*, pp. 441-445, June 1997.

[25] V. Tiwari, S. Malik. A. Wolfe, and M.T.C. Lee, "Instruction Level Power Analysis and Optimization of Software", *Journal of VLSI Signal Processing*, Vol 13, Issue 2-3, pp. 223-238, Aug./Sep. 1996.

[26] R. Mehra and J. Rabaey, "Behavioral Level Power Estimation and Exploration", *Proc. of International Workshop Low Power Design*, pp. 255-270, Jan. 1994.

[27] E. Musoll and J. Cortadella, "High-level Synthesis Techniques for Reducing the Activity of Functional Units", *Proc. of International Symposium on Low Power Design*, pp. 99-104, April 1995.

[28] D. Kim, D. Shin, and K. Choi, "Pipelining with Common Operands for Power-efficient Linear Systems", *IEEE Transactions on VLSI Systems*, Vol. 13, No. 9, pp. 1023-1034, Sep. 2005.

[29] E. Macii, M. Pedram, and F. Somenzi, "High-level Power Modeling, Estimation, and Optimization", IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, Vol. 17, No. 11, pp. 1061-1079, Nov. 1998.

[30] Q. Zhuge, E.H.M. Sha, B. Xiao, and C. Chantrapornchai, "Efficient Variable Partitioning and Scheduling for DSP Processors with Multiple Memory Modules", *IEEE Transactions on Signal Processing*, Vol. 52, No. 4, pp. 1090-1099, April 2004.

[31] L.F. Chao and E.H.M. Sha, "Scheduling Data-flow Graphs via Retiming and

Unfolding", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 12, pp. 1259-1267, Dec. 1997.

[32] M. Wolfe, **High Performance Compilers for Parallel Computing**, Addison-Wesley, Redwood City, CA, USA, 1996.

[33] M.E. Wolf and M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 452-471, Oct. 1991.

[34] N.L. Passos and E.H.M. Sha, "Achieving Full Parallelism using Multi-dimensional Retiming", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 11, pp. 1150-1163, Nov. 1996.

[35] N.L. Passos and E.H.M. Sha, "Scheduling of Uniform Multi-dimensional Systems under Resource Constraints", *IEEE Transactions on VLSI Systems*, Vol. 6, No. 4, pp. 719-730, Dec. 1998.

[36] Y.H. Lee and C. Chen, "An Effective Variable Partitioning and Scheduling Algorithm for DSP with Multiple Memory Modules", *Proc. of International Computer Symposium*, Dec. 2004.

[37] Y.H. Lee and C. Chen, "An Efficient Code Generation Algorithm for Non-orthogonal DSP Architecture", accepted and to appear to *Journal of VLSI Signal Processing Systems*.

[38] Y.H. Lee and C. Chen, "An Effective and Efficient Code Generation Algorithm for Uniform Loops on Non-orthogonal DSP Architecture", *Journal of Systems and Software*, Vol. 80, Issue 3, pp. 410-428, March 2007.

[39] C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry", *Algorithmica*, Vol. 6, No. 1, pp. 5-35, June 1991.

[40] L. Lamport, "The Parallel Execution of DO Loops", *Comm. ACM SIGPLAN*, Vol. 17, No. 2, pp. 82-93, Feb. 1974.

[41] Y.H. Lee, M.L. Tsai, and C. Chen, "RPUSM: An Effective Instruction Scheduling Method for Nested Loops", *Proc. of National Computer Symposium*, pp. C025-C036, Dec. 2001.

[42] Y.H. Lee and C. Chen, "A Two-level Scheduling Method: An Effective Parallelizing Technique for Uniform Nested Loops on a DSP Multiprocessor", *Journal of Systems and Software*, Vol. 75, Issue 1-2, pp 155-170, Feb. 2005.

[43] L.F. Chao, A. LaPaugh, and E.H.M. Sha, "Rotation Scheduling: A Loop Pipelining Algorithm", *IEEE Transactions on Computer Aided Design*, Vol. 16, No. 3, pp. 229-239, March 1997.

[44] C. Kessler and A. Bednarski, "Optimal Integrated Code Generation for Clustered VLIW Architectures", *Proc. of ACM Joint conference LCTES-SCOPES*, pp. 102-111, June 2002.

[45] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step towards Software Power Minimization", *IEEE Transactions on VLSI Systems*, Vol. 2, No 4, pp. 437-445, Dec. 1994.

[46] M.T.C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software", *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 5, No. 1, pp. 123-135, March 1997.

[47] W. Xu, A. Parikh, M. Kandemir, and M.J. Irwin, "Fine-grain Instruction Scheduling for Low Energy", *Proc. of IEEE Workshop on Signal Processing Systems*, pp. 258-263, Oct. 2002.

[48] K.W. Choi and A. Chatterjee, "Efficient Instruction-level Optimization Methodology for Low-power Embedded Systems", *Proc. of the 14th International Symposium on Systems Synthesis*, pp. 147-152, Oct. 2001.

[49] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin, "Energy-aware Instruction Scheduling", *Proc. of the 7th International Conference on High*

*Performance Computing*, pp. 335-344, 2000.

[50] A. Raghunathan and N.K. Jha, "SCALP: An Iterative Improvement based low-power Data Path Synthesis System", *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, Vol. 16, No. 11, pp. 1260-1277, Nov. 1997.

[51] C.G. Lyuh, T. Kim, and C.L. Liu, "An Integrated Data Path Optimization for Low Power based on Network Flow Method", *Proc. of ACM/IEEE Design Automation Conference*, pp. 553-559, 2001.

[52] J.M. Chang and M. Pedram, "Module Assignment for Low Power", *Proc. of European Design Automation Conference*, pp. 376-381, 1996.

[53] http://www.princeton.edu/~mescal/; http://embedded.eecs.berkeley.edu/mescal/

[54] http://www.arm.com/products/CPUs/families/OptimoDE.html

[55] http://ipf-orc.sourceforge.net/

[56] http://www.tensilica.com/

[57] http://www.trimaran.org/

[58] Y.H. Lee and C. Chen, "Energy-efficient Code Generation Algorithms with Operand Sharing for DSP Architecture with Multiple Data Memory Banks", submitted (revised) to *Journal of Systems and Software*.

[59] Q. Zhuge, B. Xiao, and E.H.M. Sha, "Code Size Reduction Technique and Implementation for Software-pipelined DSP Applications", *ACM Transactions on Embedded Computing Systems*, Vol. 2, No. 4, pp. 590-613, Nov. 2003.

# Appendix A. The Analytic Model for RSVR, RSF, RST, and RSP

In this appendix we introduce the analytic model defined to calculate the overall schedule length of a retimed one or two-dimensional MDFG [14, 36]. Nevertheless, it can be easily extended to cover nested loop with depths greater than two. Table A.1, the same as Table 3.3, lists variables used in our analytic model. The following formulas are used to calculate overall schedule lengths after applying different methods. Note that formula (A.5) for RST is only suited for the nested loop can be tiled directly. If the nested loop needs to be skewed before tiling, its scheduling results using RST is unacceptable and the calculation of overall schedule length becomes very complicated. We suggest not using RST if the nested loop cannot be tiled directly, and corresponding formulas are omitted.

« **For RSVR, 1-dim MDFG, the overall schedule length =**

$$length \times (m - d) + prologue + epilogue \qquad \text{(A.1)}$$

Figure A.1(a) shows the original iteration space of a loop with depth one which contains *m* iterations. When we apply the retiming technique, the retiming base $r = 1$ is always feasible for a one-dimensional MDFG. Figure A.1(b) shows the modified iteration space of Figure A.1(a) after applying RSVR with retiming depth *d*. From this figure, clearly that *d* iterations are moved to prologue and epilogue, and the repetitive pattern contains *m*–*d* iterations. Thus, formula (A.1) can be directly approved.

« **For RSF, 1-dim MDFG, the overall schedule length =**

$$length \times (\lfloor m/N \rfloor - d) + prologue + epilogue + half((m \textbf{ mod } N), N) \qquad \text{(A.2)}$$

When we apply RSF to schedule a given loop, we first need to unfold the original MDFG with factor *N*. Therefore, for a loop with depth with depth one which

Table A.1. Variables defined in the analytic model.

| Variable | Definition |
|---|---|
| $N$ | Number of memory banks |
| $m$ | Loop bound of the outer loop for a two-dimensional nested loop<br>Loop bound for an one-dimensional loop |
| $n$ | Loop bound of the inner loop for a two-dimensional nested loop |
| *prologue* | Schedule length of the prologue part of a retimed loop |
| *epilogue* | Schedule length of the epilogue part of a retimed loop |
| *length* | Schedule length of a single iteration in the repetitive pattern of a retimed loop |
| *list* | Schedule length of a single iteration produced by list scheduling |
| $d$ | Retiming depth, the number of iterations that must be moved into the prologue and epilogue |
| $w$ | Skew factor used to parallelize the inner loop |
| *half* ($k$, $N$) | Schedule length of $k$ original iterations under $N$ memory banks |

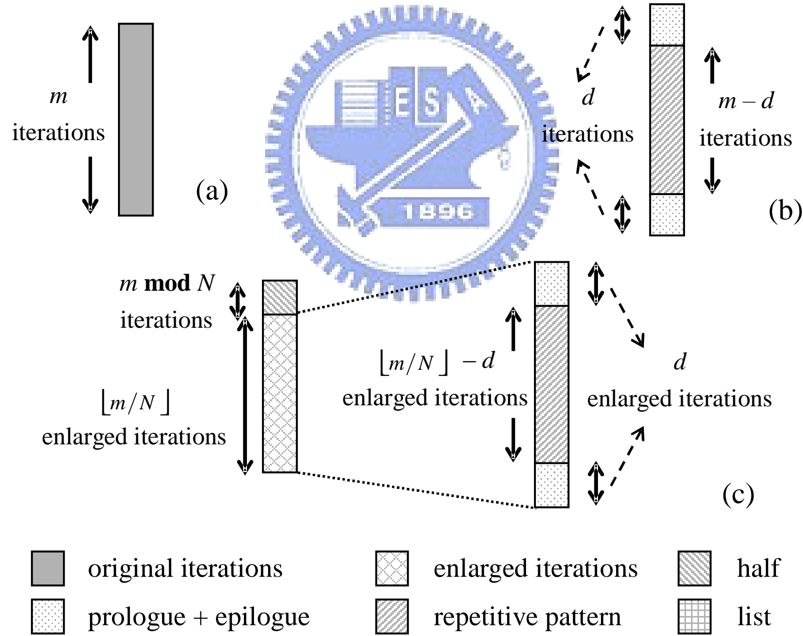



Figure A.1. Iteration spaces of a loop with depth one. (a) Original, (b) applying RSVR, (c) applying RSF.

contains $m$ iterations, we will obtain $\lfloor m/N \rfloor$ enlarged iterations and a *half* portion with ($m$ **mod** $N$) original iterations. Then, we apply the retiming technique for these enlarged iterations with retiming depth $d$, and the modified iteration space is shown in Figure A.1(c). Similar as formula (A.1), formula (A.2) is also directly approved.

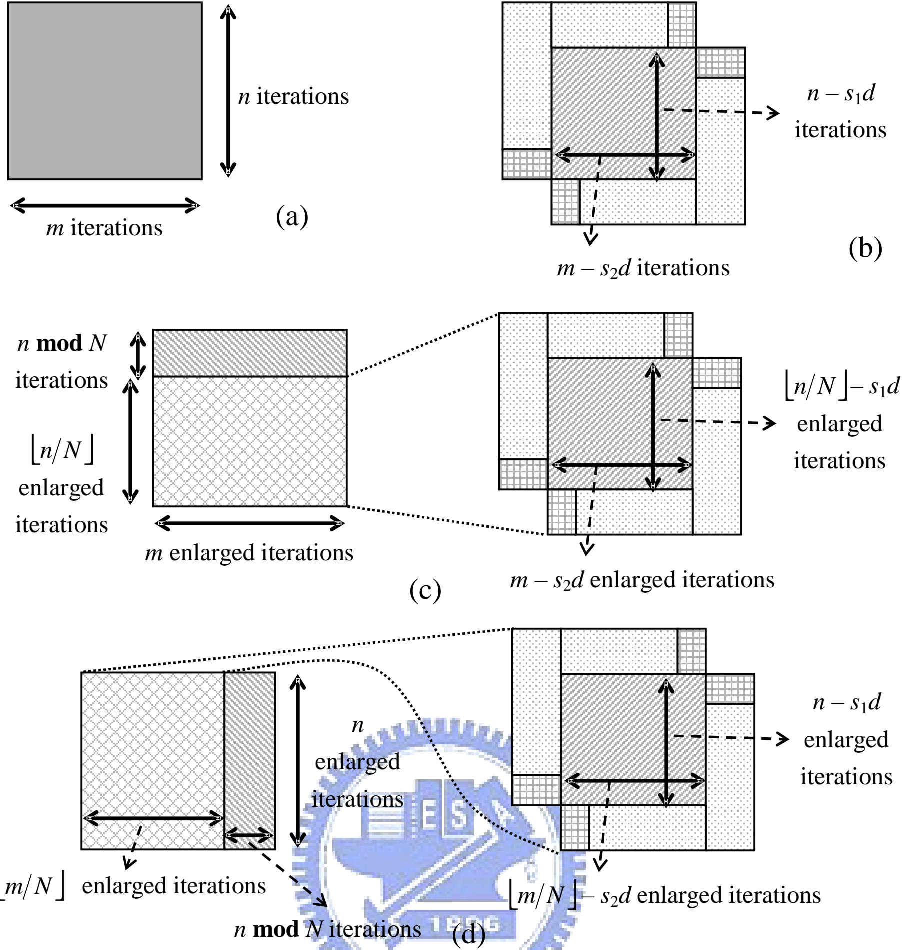



Figure A.2. Iteration spaces of a loop with depth two. (a) Original, (b) applying RSVR, (c) applying RSF, (d) applying RST.

« **For RSVR, 2-dim MDFG, the overall schedule length =**

$$length \times (m - s_2 d)(n - s_1 d) + (prologue + epilogue) \times d(s_1 m + s_2 n - s_1 s_2 - 2ds_1 s_2) + list \times s_1 s_2 d(d + 1) \quad \text{(A.3)}$$

As shown in Figure 2.3, two cases of modified iteration space will be generated after applying the retiming technique using different schedule vectors for a nested loop with depth two. In fact, Figure 2.3(b) is a special case of Figure 2.3(c), so in the following we directly select schedule vector $s = (s_1, s_2)$ to retime a two-dimensional MDFG. Figure A.2(a) shows the original iteration space of a loop with depth two which contains $m \times n$ iterations. After applying RSVR with retiming base $(s_2, -s_1)$, which is orthogonal to $s$, and retiming depth $d$, the modified iteration space is shown

Figure A.2(b). From this figure, we can see the repetitive pattern contains $(m - s_2d)(n - s_1d)$ iterations, and $d(s_1m + s_2n - s_1s_2 - 2ds_1s_2)$ iterations are moved to prologue and epilogue. The remaining $s_1s_2d(d + 1)$ iterations will be required to be executed using the list scheduling, because they are moved out of the nested loop. Formula (A.3) still can be simply approved.

« **For RSF, 2-dim MDFG, the overall schedule length =**

$$length \times (m - s_2d)(\lfloor n/N \rfloor - s_1d) + (prologue + epilogue) \times d(s_1m + s_2\lfloor n/N \rfloor - s_1s_2 - 2ds_1s_2) + list \times s_1s_2d(d + 1) + half((n \textbf{ mod } N), N) \times m \quad \text{(A.4)}$$

For a nested loop with depth two contains $m \times n$ iterations, the inner loop will be unfolded with factor $N$ when we apply RSF to schedule it. After loop unfolding, we will get $m \times \lfloor n/N \rfloor$ enlarged iterations and $m \times (n \textbf{ mod } N)$ iterations in the half portion, as shown in Figure A.2(c). Then, these enlarged iterations are applied the retiming technique with schedule vector $(s_1, s_2)$ and retiming depth $d$. The overall schedule length of the retimed iterations can be calculated using formula (A.3). Therefore, formula (A.4) is directly approved.

« **For RST, the overall schedule length =**

$$length \times (\lfloor m/N \rfloor - s_2d)(n - s_1d) + (prologue + epilogue) \times d(s_1\lfloor m/N \rfloor + s_2n - s_1s_2 - 2ds_1s_2) + list \times s_1s_2d(d + 1) + half((m \textbf{ mod } N), N) \times n \quad \text{(A.5)}$$

When we apply RST to schedule a nested loop with depth two contains $m \times n$ iterations, the modified iteration space is shown in Figure A.2(d). Similar as above, formula (A.5) still can be simply approved.

« **For RSP, if $wm + 1$ £ $w + n$, the overall schedule length =**

$$length \times w(\lfloor (m-1)/N \rfloor - d)(m - Nd - N + 1 + ((m - 1) \textbf{ mod } N)) + length \times (w + n - mw)(\lfloor m/N \rfloor - d) + (prologue + epilogue) \times (wm + w + n - 2wNd) + list \times dwN(d - 1) + 2w\lfloor (m-1)/N \rfloor \times \sum_{i=1}^{N-1} half(i, N) + 2w \times \sum_{i=1}^{(m-1) \bmod N} half(i, N) + (w + n - mw) \times half((m \textbf{ mod } N), N) \quad \text{(A.6)}$$

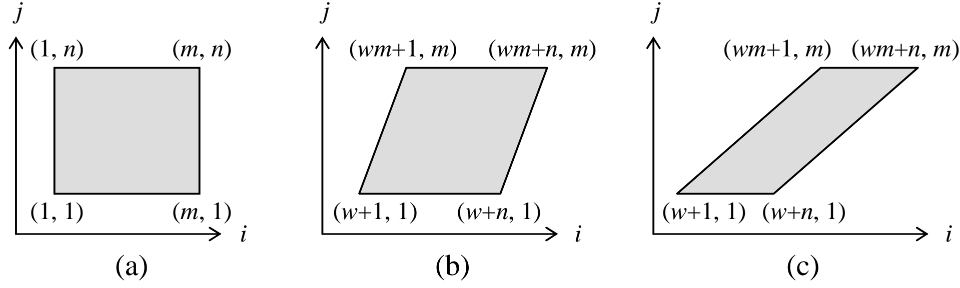



Figure A.3 Iteration spaces of a loop with depth two. (a) Original, (b)(c) applying RSP.

« **For RSP, if $wm + 1 > w + n$, the overall schedule length =**

$$length \times w(\lfloor (n-w)/wN \rfloor - d)(\lfloor n/w \rfloor - Nd - N + 1 + ((\lfloor n/w \rfloor - 1) \textbf{ mod } N)) + length \times (2w + ((mw - w - n) \textbf{ mod } w)\lceil (mw-w-n)/w \rceil)(\lfloor n/wN \rfloor - d) + length \times (2(n \textbf{ mod } w) + (w - ((mw - w - n) \textbf{ mod } w))\lfloor (mw-w-n)/w \rfloor)(\lfloor \lceil n/w \rceil / N \rfloor - d) + (prologue + epilogue) \times (2w\lfloor n/w \rfloor - 2wNd + 2w + ((mw - w - n) \textbf{ mod } w)\lceil (mw-w-n)/w \rceil + 2(n \textbf{ mod } w) + (w - ((mw - w - n) \textbf{ mod } w))\lfloor (mw-w-n)/w \rfloor) + list \times dwN(d-1) + 2w\lfloor (n-w)/wN \rfloor \times \sum_{i=1}^{N-1} half(i, N) + 2w \times \sum_{i=1}^{(\lfloor n/w \rfloor - 1) \textbf{ mod } N} half(i, N) + (2w + ((mw - w - n) \textbf{ mod } w)\lceil (mw-w-n)/w \rceil) \times half((\lfloor n/w \rfloor \textbf{ mod } N), N) + (2C + (w - ((mw - w - n) \textbf{ mod } w))\lceil (mw-w-n)/w \rceil) \times half((\lceil n/w \rceil \textbf{ mod } N), N) \quad (A.7)$$

If we use RSP to schedule a nested loop, to calculate its overall schedule length becomes much complex. As shown in Figure A.3, two modified iteration spaces will be generated after parallelizing the inner loop, based on variables $w$, $m$, and $n$. After parallelizing, we unfold the inner loop, and retime enlarged iterations using schedule vector (1, 0). Note that if the inner loop doesn't contain enough enlarged iterations for retiming, we will simply use list scheduling to schedule it. Based on two modified iteration spaces shown in Figure A.3, we conclude formulas (A.6) and (A.7) to calculate the overall schedule length for a given two-dimensional MDFG after applying RSP.

# Appendix B. The Analytic Model for RSOR and RSER

In this appendix we introduce the extended analytic model to calculate the overall schedule length and the number of operand reutilizations of a retimed one or two-dimensional MDFG for RSOR and RSER. Nevertheless, it also can be easily extended to cover nested loop with depths greater than two. Table B.1, the same as Table 6.6, lists variables used in the analytic model. After applying different methods to schedule the given MDFG, the following formulas are defined to calculate their overall schedule length. Note that three scheduling results can be derived by RSOR and RSER, which applies different variable partition mechanisms proposed in RSVR, RSF, and RST. In the following, we use RSOR(RSVR) to represent using RSOR with variable partition mechanism proposed in RSVR to schedule the given loop. The variable partition mechanism proposed in RST is only available for multi-dimensional MDFGs. Furthermore, formulas defined for RSOR(RSVR) are also suitable for method proposed in Kim et al. [30].

- **For RSOR(RSVR), 1-dim MDFG, the overall schedule length =**

$$prologue + epilogue + length \times (m - d) \quad \text{(B.1)}$$

- **For RSOR(RSF), 1-dim MDFG, the overall schedule length =**

$$prologue + epilogue + length \times (\lfloor m/N \rfloor - d) + half((m \ \mathbf{mod}\ N), N) \quad \text{(B.2)}$$

For a loop with depth one containing $m$ iterations, Figure B.1 shows iteration spaces before and after applying methods RSOR(RSVR) and RSOR(RSF). Figure B.1 is actually the same as Figure A.1. Thus, formulas (B.1) and (B.2) are equivalent to formulas (A.1) and (A.2) and can be directly approved as described in appendix A.

Table B.1. Definitions of variables used in the analytic model.

| Variable | Definition |
|---|---|
| $N$ | Number of memory modules |
| $m$ | Loop bound of the outer loop for a two-dimensional nested loop<br>Loop bound for an one-dimensional loop |
| $n$ | Loop bound of the inner loop for a two-dimensional nested loop |
| $(s_1, s_2)$ | Schedule vector selected for retiming during instruction scheduling |
| *list* | Schedule length of an iteration in the repetitive pattern produced by *list scheduling* method |
| *length* | Schedule length of an iteration in the repetitive pattern |
| *prologue* | Schedule length of the prologue generated during instruction scheduling |
| *eplogue* | Schedule length of the prologue generated during instruction scheduling |
| $d$ | Retiming depth obtained during instruction scheduling |
| *half* $(k, N)$ | Schedule length of $k$ original iterations under $N$ memory modules |
| *exp*1 | Schedule length of the prologue generated during MDFG reconstructing after first retiming |
| *exe*1 | Schedule length of the epilogue generated during MDFG reconstructing after first retiming |
| *exd*1 | Retiming depth obtained during MDFG reconstructing after first retiming |
| *exp*2 | Schedule length of the prologue generated during MDFG reconstructing after second retiming |
| *exe*2 | Schedule length of the epilogue generated during MDFG reconstructing after second retiming |
| *exd*2 | Retiming depth obtained during MDFG reconstructing after second retiming |

« **For RSER(RSVR), 1-dim MDFG, the overall schedule length =**

$$exp1 + exe1 + prologue + epilogue + length \times (m - exd1 - d) \quad \text{(B.3)}$$

« **For RSER(RSF), 1-dim MDFG, the overall schedule length =**

$$exp1 + exe1 + prologue + epilogue + length \times (\lfloor m/N \rfloor - exd1 - d) + half((m \textbf{ mod } N), N) \quad \text{(B.4)}$$

When we use RSER(RSVR) to schedule a loop with depth one containing $m$ iterations, the retiming technique will be applied twice to reconstruct the given MDFG and compact the initial scheduling result sequentially. As shown in Figure B.2(a), after reconstructing the MDFG, *exd*1 iterations are moved into the *exp*1 and

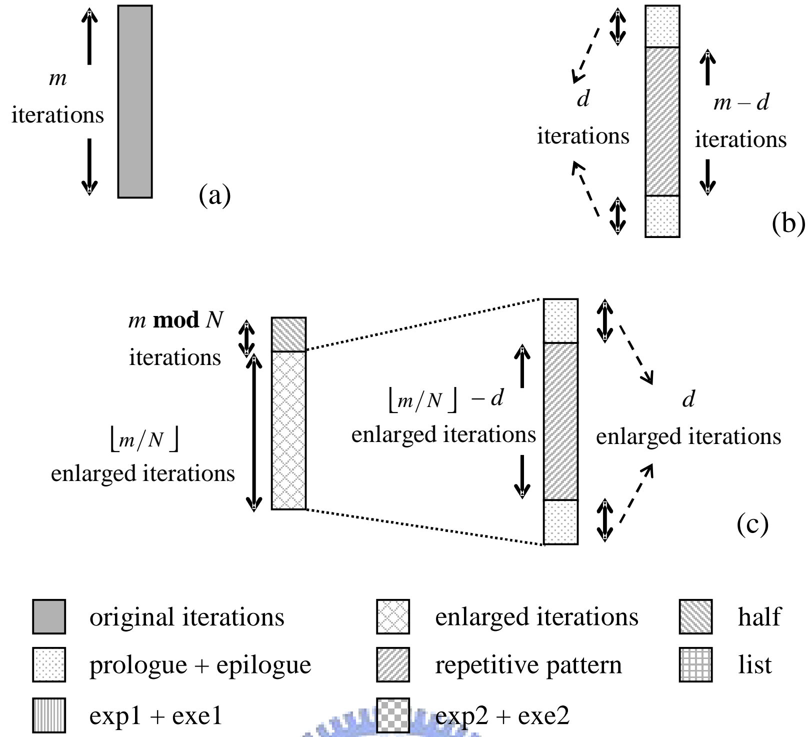



Figure B.1. Iteration spaces of a loop with depth one. (a) Original, (b) applying RSOR(RSVR), (c) applying RSOR(RSF).

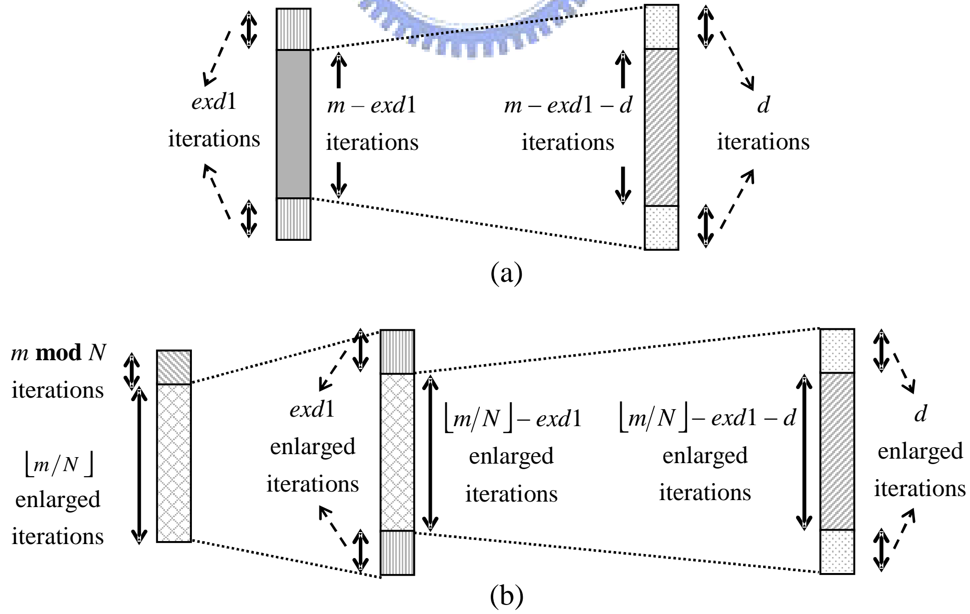



Figure B.2. Iteration spaces of a loop with depth one. (a) Applying RSER(RSVR), (b) applying RSER(RSF).

*exe*1 portions. The remaining *m–exd*1 iterations will be further retimed with retiming depth *d*, and their overall schedule length can be calculated using above formula (B.1). On the other hand, to schedule a loop with depth one using RSER(RSF), the first step is to move ($m$ **mod** $N$) iterations to the *half* portion and unfold the given MDFG. Other steps are actually the same as using RSER(RSVR), and the modified iteration space after RSER(RSF) is shown in Figure B.2(b). Therefore, formulas (B.3) and (B.4) can be simply approved.

- **For RSOR(RSVR), 2-dim MDFG, the overall schedule length =**

  $$(prologue + epilogue) \times d(s_1 m + s_2 n - s_1 s_2 - 2ds_1 s_2) + length \times (m - s_2 d)(n - s_1 d) + list \times s_1 s_2 d(d + 1) \quad \text{(B.5)}$$

- **For RSOR(RSF), 2-dim MDFG, the overall schedule length =**

  $$(prologue + epilogue) \times d(s_1 m + s_2 \lfloor n/N \rfloor - s_1 s_2 - 2ds_1 s_2) + length \times (m - s_2 d)(\lfloor n/N \rfloor - s_1 d) + list \times s_1 s_2 d(d + 1) + half((n \textbf{ mod } N), N) \times m \quad \text{(B.6)}$$

- **For RSOR(RST), 2-dim MDFG, the overall schedule length =**

  $$(prologue + epilogue) \times d(s_1 \lfloor m/N \rfloor + s_2 n - s_1 s_2 - 2ds_1 s_2) + length \times (\lfloor m/N \rfloor - s_2 d)(n - s_1 d) + list \times s_1 s_2 d(d + 1) + half((m \textbf{ mod } N), N) \times n \quad \text{(B.7)}$$

For a loop with depth one which contains $m \times n$ iterations, Figure B.3 shows iteration spaces before and after applying methods RSOR(RSVR), RSOR(RSF), and RSOR(RST). Figure B.3 is actually the same as Figure A.2. Thus, formulas (B.5)~(B.7) are equivalent to formulas (A.3)~(A.5) and can be directly approved as described in appendix A.

- **For RSER (RSVR), 2-dim MDFG, retiming base (0, 1) → (1, 0), the overall schedule length =**

  $$(exp1 + exe1) \times m + (exp2 + exe2) \times (n - exd1) + (prologue + epilogue) \times d(s_1(m - exd2) + s_2(n - exd1) - s_1 s_2 - 2ds_1 s_2) + length \times (m - exd2 - s_2 d)(n - exd1 - s_1 d) + list \times s_1 s_2 d(d + 1) \quad \text{(B.8)}$$

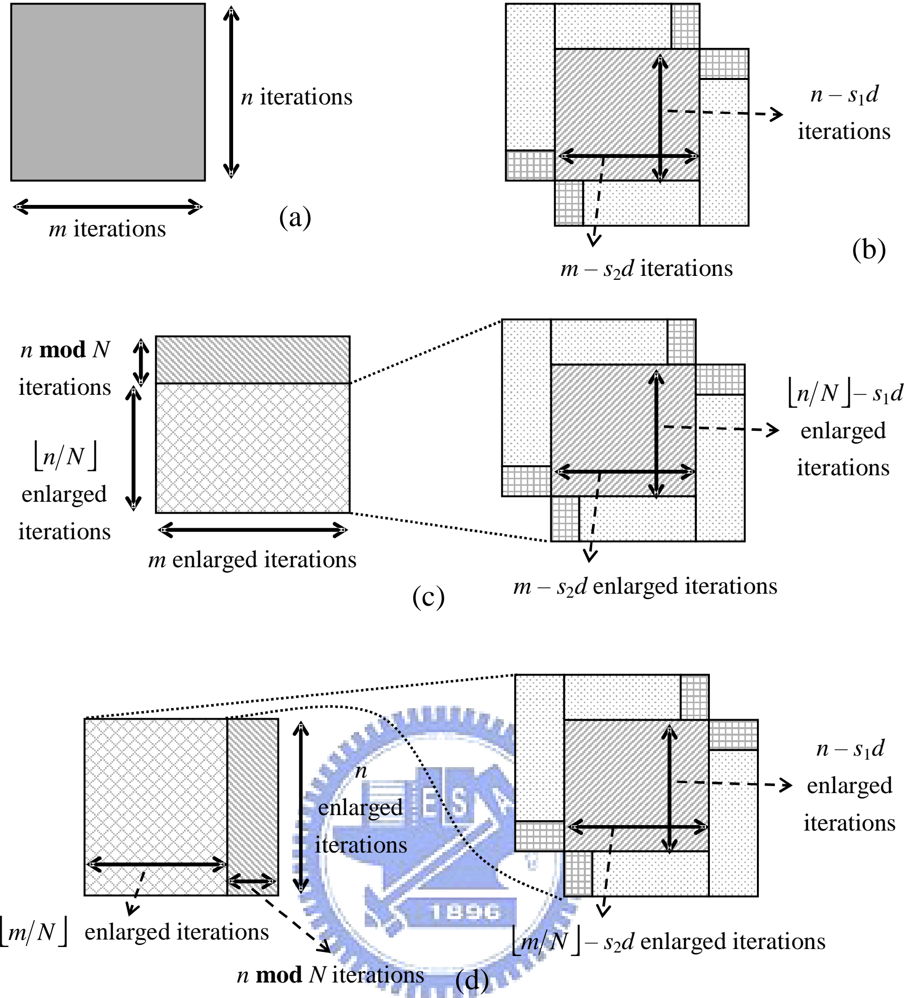



Figure B.3. Iteration spaces of a loop with depth two. (a) Original, (b) applying RSOR(RSVR), (c) applying RSOR(RSF), (d) applying RSOR(RST).

For a nested loop with depth two which contains $m \times n$ iterations, when we use RSER(RSVR) to schedule it, the retiming technique is applied at most twice during MDFG reconstructing with specific retiming bases. At first, we assume the MDFG is retimed twice when it is reconstructed with retiming bases (0, 1) and (1, 0) sequentially. As shown in Figure B.4(a), $m \times exd1$ iterations are moved into the *exp*1 and *exe*1 portion, and the *exp*2 and *exe*2 portion contains $(n–exd1) \times exd2$ iterations. Then, $(m–exd2) \times (m–exd1)$ iterations, represented by the reconstructed MDFG, will be further retimed with schedule vector $(s_1, s_2)$, and their overall schedule length can be calculated using formula (B.5). Hence, formula (B.8) is directly approved. When the

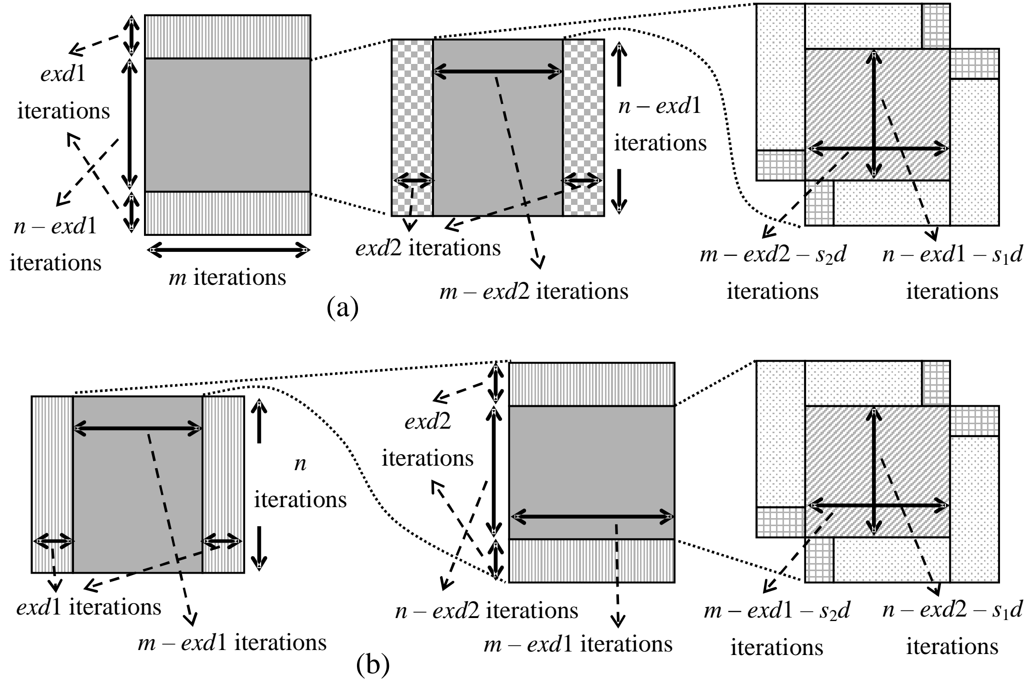



Figure B.4. Iteration spaces of a loop with depth two. (a) Applying RSER(RSVR), formula (B.8), (b) applying RSER(RSVR), formula (B.9).

given MDFG is retimed only once using retiming base (0, 1), formula (B.8) is also available if variables *exp*2, *exe*2, and *exd*2 are set to zero.

« **For RSER (RSVR), 2-dim MDFG, retiming base (1, 0) → (0, 1), the overall schedule length =**

$$(exp1 + exe1) \times n + (exp2 + exe2) \times (m - exd1) + (prologue + epilogue) \times d(s_1(m - exd1) + s_2(n - exd2) - s_1 s_2 - 2ds_1 s_2) + length \times (m - exd1 - s_2 d)(n - exd2 - s_1 d) + list \times s_1 s_2 d(d + 1) \qquad \text{(B.9)}$$

When we apply RSER(RSVR) to schedule a nested loop with depth two contains $m \times n$ iterations, Figure B.4(b) shows the modified iteration space if the MDFG is retimed twice with retiming bases (1, 0) and (0, 1) sequentially during MDFG reconstruction. In fact, these steps are very similar as those of Figure B.4(a), so formula (B.9) can be easily approved according to descriptions above. When the given MDFG is retimed only once using retiming base (1, 0), formula (B.8) is also available if variables *exp*2, *exe*2, and *exd*2 are set to zero.

« **For RSER (RSF), 2-dim MDFG, retiming base (0, 1) → (1, 0), the overall schedule length =**

$(exp1 + exe1) \times m + (exp2 + exe2) \times (\lfloor n/N \rfloor - exd1) + (prologue + epilogue) \times d(s_1(m - exd2) + s_2(\lfloor n/N \rfloor - exd1) - s_1s_2 - 2ds_1s_2) + length \times (m - exd2 - s_2d)(\lfloor n/N \rfloor - exd1 - s_1d) + list \times s_1s_2d(d + 1) + half((n \textbf{ mod } N), N) \times m$ (B.10)

« **For RSER (RSF), 2-dim MDFG, retiming base (1, 0) → (0, 1), the overall schedule length =**

$(exp1 + exe1) \times \lfloor n/N \rfloor + (exp2 + exe2) \times (m - exd1) + (prologue + epilogue) \times d(s_1(m - exd1) + s_2(\lfloor n/N \rfloor - exd2) - s_1s_2 - 2ds_1s_2) + length \times (m - exd1 - s_2d)(\lfloor n/N \rfloor - exd2 - s_1d) + list \times s_1s_2d(d + 1) + half((n \textbf{ mod } N), N) \times m$ (B.11)

For a two-dimensional MDFG scheduled using RSER(RSF), Figure B.5 shows two modified iteration spaces which correspond to difference sequences of applied retiming bases during MDFG reconstruction. From this figure, after moving $m \times (n$ **mod** $N)$ iterations into half portion and unfolding the MDFG, other steps are similar as those of Figure B.4. Therefore, these two formulas can be easily approved according to formulas (B.8) and (B.9).

« **For RSER (RST), 2-dim MDFG, retiming base (0, 1) → (1, 0), the overall schedule length =**

$(exp1 + exe1) \times \lfloor m/N \rfloor + (exp2 + exe2) \times (n - exd1) + (prologue + epilogue) \times d(s_1(\lfloor m/N \rfloor - exd2) + s_2(n - exd1) - s_1s_2 - 2ds_1s_2) + length \times (\lfloor m/N \rfloor - exd2 - s_2d)(n - exd1 - s_1d) + list \times s_1s_2d(d + 1) + half((m \textbf{ mod } N), N) \times n$ (B.12)

« **For RSER (RST), 2-dim MDFG, retiming base (1, 0) → (0, 1), the overall schedule length =**

$(exp1 + exe1) \times n + (exp2 + exe2) \times (\lfloor m/N \rfloor - exd1) + (prologue + epilogue) \times d(s_1(\lfloor m/N \rfloor - exd1) + s_2(n - exd2) - s_1s_2 - 2ds_1s_2) + length \times (\lfloor m/N \rfloor - exd1 - s_2d)(n - exd2 - s_1d) + list \times s_1s_2d(d + 1) + half((m \textbf{ mod } N), N) \times n$ (B.13)

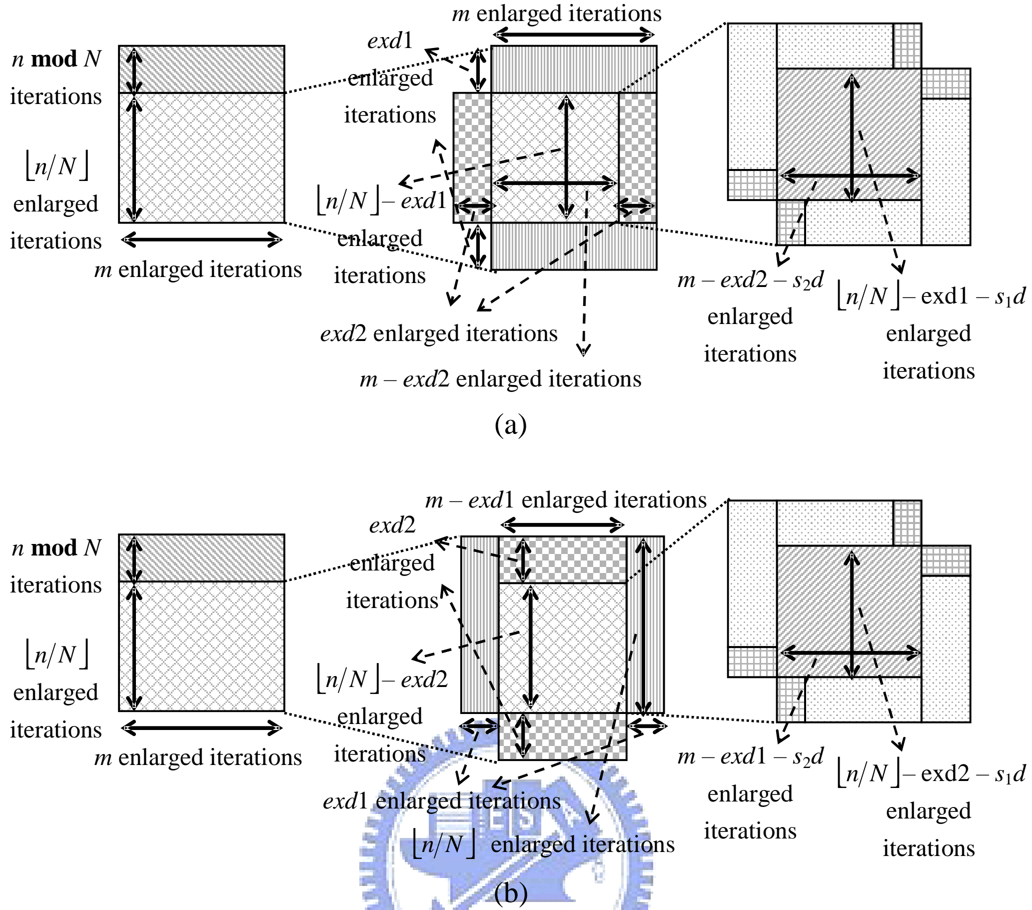



Figure B.5. Iteration spaces of a loop with depth two. (a) Applying RSER(RSF), formula (B.10), (b) applying RSER(RSF), formula (B.11).

For a two-dimensional MDFG scheduled using RSER(RSF), Figure B.6 shows two modified iteration spaces which correspond to difference sequences of applied retiming bases during MDFG reconstruction. From this figure, after moving $n \times (m$ **mod** $N)$ iterations into half portion and unfolding the MDFG, other steps are similar as those of Figure B.4. Therefore, these two formulas can be easily approved according to formulas (B.8) and (B.9).

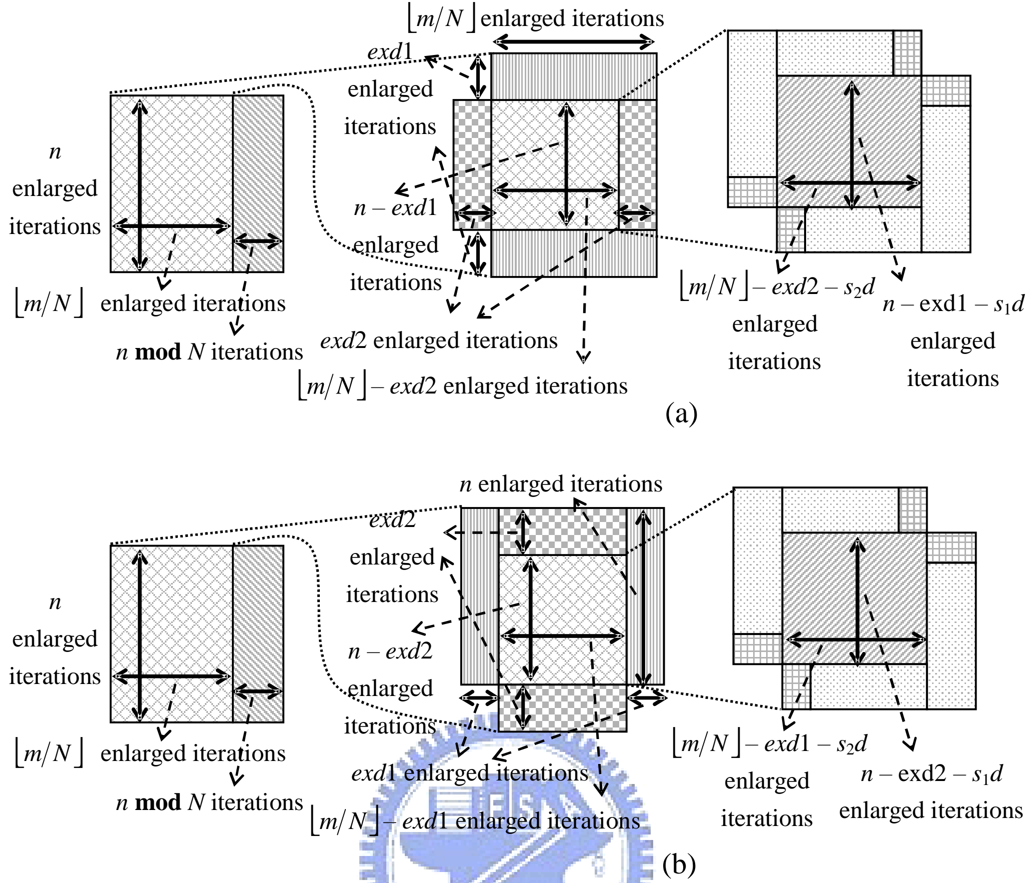



Figure B.6. Iteration spaces of a loop with depth two. (a) Applying RSER(RST), formula (B.12), (b) applying RSER(RST), formula (B.13).