國立交通大學

資 訊 工 程 學 系

碩 士 論 文

考慮分支指令行為的迴圈排班方法

Probabilistic Loop Scheduling Method for Nested Loop with

Conditional Branch on DSP Architecture

研 究 生： 李嘉淳

指導教授： 陳 正 教授

中 華 民 國 九 十 四 年 六 月

考慮分支指令行為的迴圈排班方法

Probabilistic Loop Scheduling Method for Nested Loop with Conditional Branch on DSP Architecture

研 究 生：李嘉淳　　Student：Jia-Chun Lee

指導教授：陳　正　　Advisor：Prof. Cheng Chen

國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

A Thesis

Submitted to

Institute of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Institute of Computer Science and Information Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

考慮分支指令行為的迴圈排班方法

研究生：李嘉淳　　　指導教授：陳 正 教授

國立交通大學資訊工程學系碩士班

摘要

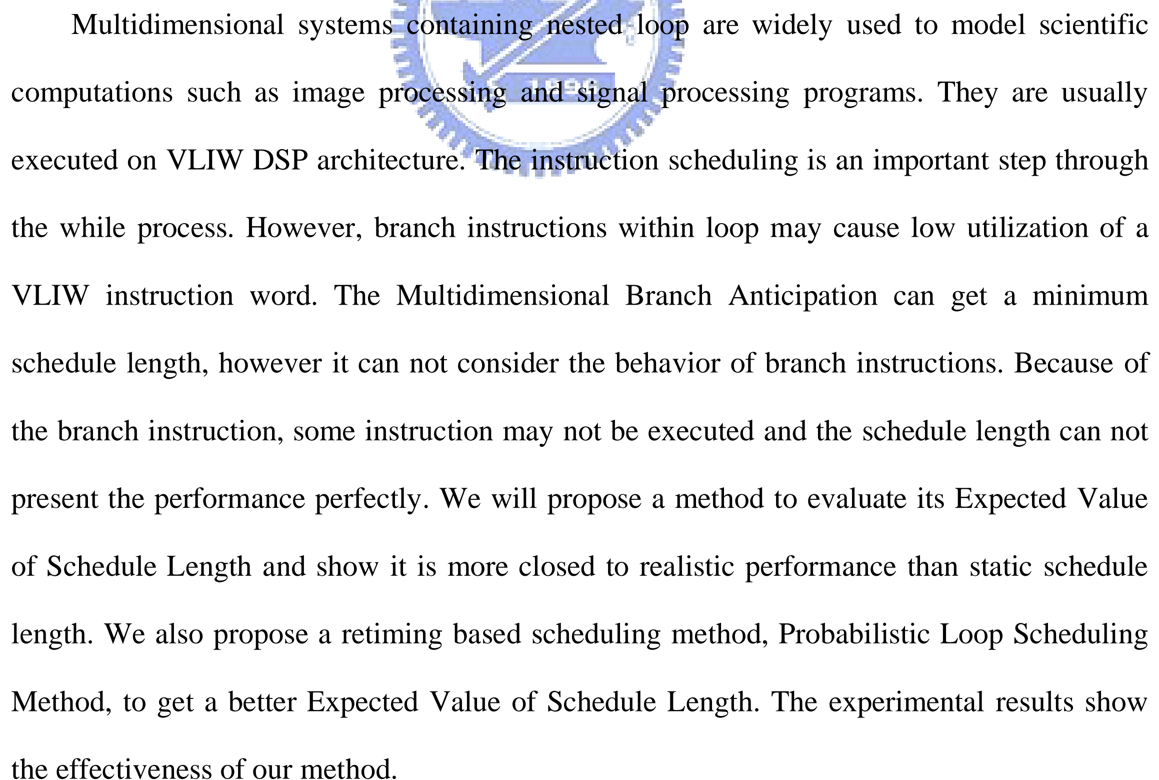
隨著個人攜帶式應用產品的普及，數位訊號處理機的應用也隨之廣泛，且性能迅速改進，使得攜帶式的應用程式日益複雜。在數位訊號處理機中，巢狀迴圈經常佔去大部分的計算時間，所以需要探討針對巢狀迴圈中包含分支指令的排班方法。在過去已經完成的研究中，Push-Up Scheduling Method 與 Bottom Up Scheduling Method 皆是基於 Retiming 且可以處理巢狀迴圈的指令排班的排班方法，目的在得到最短的 Schedule Length，但是不能處理分支指令。Multidimensional Branch Anticipation 可以處理含分支指令的巢狀迴圈問題但是不考慮分支指令的行為。當迴圈中的分支指令越多，分支指令的行為將會對應用程式的產能影響越大。由於分支指令的影響，迴圈中的某些指令不會被執行，所以傳統的 Schedule Length 無法完全反應排班結果的優劣。我們提出 Expected Value of Schedule Length 用來評估含有分支指令的巢狀迴圈排班結果的優劣。之後我們基於 Retiming 的觀念，並考慮分支指令的行為發展出 Probabilistic Loop Scheduling Method (PLSM)，可以達到相當短的 Expected Value of Schedule Length。最後利用實驗，可以看出 PLSM 的優勢與效果。

Probabilistic Loop Schedule Method for Nested Loop with Conditional Branch on DSP Architecture

Student: Jia-Chun Lee Advisor: Prof. Cheng Chen

Institute of Computer Science and Information Engineering National Chiao Tung University

Abstract



Multidimensional systems containing nested loop are widely used to model scientific computations such as image processing and signal processing programs. They are usually executed on VLIW DSP architecture. The instruction scheduling is an important step through the while process. However, branch instructions within loop may cause low utilization of a VLIW instruction word. The Multidimensional Branch Anticipation can get a minimum schedule length, however it can not consider the behavior of branch instructions. Because of the branch instruction, some instruction may not be executed and the schedule length can not present the performance perfectly. We will propose a method to evaluate its Expected Value of Schedule Length and show it is more closed to realistic performance than static schedule length. We also propose a retiming based scheduling method, Probabilistic Loop Scheduling Method, to get a better Expected Value of Schedule Length. The experimental results show the effectiveness of our method.

Acknowledgements

I would like to express my sincere thanks to my advisor, Prof. Cheng Chen, for his supervision and advice. Without his guidance and encouragement, I could not finish this thesis. I also thank Prof. Jyj-Jiun Shann and Prof. Guan-Joe Lai for their valuable suggestions.

There are many others whom I wish to thank. I thanks to Yi-Hsuan Lee for her kindly advice suggestion. Wen-Pin Liu, Wei-Fen Yang, Che-Yin Liao and Ming-Hsien Tsai are delightful follows, I felt happy and relaxed because of your presence. Finally, I am grateful to my dearest family.

Table of Contents

摘要……………………………………………………………………………………... i

Abstract…………………………………………………………………………………..ii

Acknowledgements……………………………………………………………………iii

Table of Contents………………………………………………………………………... iv

List of Figures…………………………………………………………………………… vi

Chapter 1. Instruction…………………………………………………………………….1

Chapter 2. Fundamental Background & Related Work……………………………………. 3

2.1 Modeling the Problem…………………………………………………………..3

2.2 Multidimensional Retiming………………….. 6

2.3 Resource Sharing……………………………………………………………………9

2.4 Related Work…………………………………………………………………………10

2.4.1 Push-Up Scheduling Method………………………………………………... 10

2.4.2 Bottom-Up Scheduling Method…………………………………………………11.

2.4.3 Multidimensional Branch Anticipation……………………………………….12

Chapter 3. Probabilistic Loop Scheduling Method……………………………………………14

3.1 Motivation……………………………………………………………………………14

3.2 Basic Concept………………………………………………………………………….. 16

3.2.1 Exclusive Nodes……………………………………………………………………18

3.2.2 Inclusive Nodes………………………………………………………………………19

3.2.3 Expected Value of Schedule Length………………………………………. 21

3.3 Scheduling Strategy………………………………………………………………. 22

3.4 Probabilistic Loop Scheduling Method…………………………………………... 25

Chapter 4. Preliminary Performance Evaluations……………………………………………..35

4.1 Evaluating the Execution Time…………………………………………………………35

4.2 Experiment……………………………………………………………………………...36

4.3 Summary of Experimental Result……………………………………………………42

Chapter 5. Conclusion and Future Work…………………………………………………44

5.1 Conclusion……………………………………………………………………... 44

5.2 Future Work……………………………………………………………………. 45

Bibliography…………………………………………………………………………….. 47

Appendix A……………………………………………………………………………… 49

List of Figures

Figure 2.1 (a) High-level language code for DSP program

(b) An equivalent MD-CdDFG………………………………………….. 4

Figure 2.2 (a) A DG respect to Fig 2.1

(b) A DG without intra iteration data dependence…………………………. 5

Figure 2.3 A retimed MD-CdDFG of Fig2.1(b)……………………………………..6

Figure 2.4 (a) A retimed DG respect to Fig 2.3

(b) A retimed DG without intra iteration data dependence………………… 7

Figure 2.5 A retimed DG………………………………………………………….9

Figure 2.6 (a) A part of MD-CdDFG

(b) A schedule table with sharing prevention cycle

(c) A schedule table without sharing prevention cycle…………………….. 13

Figure 3.1 (a) A source code fragment

(b) MD-CdDFG respect to (a)…………………………………………….15

Figure 3.2 (a) Schedule result of our motivation

(b) The consecutive two iterations…………………………………………. 15

Figure 3.3 (a) Schedule result of MDBA

(b) The consecutive two iterations…………………………………………. 16

Figure 3.4 A example of MD-CdDFG………………………………………………17

Figure 3.5 A example of MD-CdDFG………………………………………………22

Figure 3.6 The resource sharing policy of our example……………………………..23

Figure 3.7 Step 1 of PLSM : Calculating the executing probability for each node…… 26

Figure 3.8 After calculating executing probability of each node……………………… 26

Figure 3.9 Step 2 of PLSM : Algorithm for finding resource sharing policy…………. 27

Figure 3.10 (a) After finding sharing policy among addition

(b) After finding all sharing policy………………………………………… 28

Figure 3.11 Operations after finding resource sharing policy…………………………...28
Figure 3.12 Scheduling processes for allocating operations……………………………29
Figure 3.13 Step 3 of PLSM : allocating operations into schedule table……………….. 31
Figure 3.14 Step 4 of PLSM : Algorithm for setting retiming count…………………… 32
Figure 3.15 Process for retiming count setting……………………………………….....33
Figure 4.1 (a) Benchmark "VerySmall"
(b) Scheduling result by PLSM
(c) Scheduling Result of MDBA……………………………………………37
Figure 4.2 Maximum, average and minimum execution time (cycle) of the Fig 4.1(b)(c)
in a 5x5 multidimensional loop with depth 2……………………………......37
Figure 4.3 Execution times in different size of nested loop for "VerySmall"…………38
Figure 4.4 "VerySmall" in different functional units and branch behaviors…………... 39
Figure 4.5 Benchmark "Floyd-Stienberg"……………………………………………...40
Figure 4.6 Experiment Results of "Floyd-Stienberg"…………………………………. 40
Figure 4.7 Execution times in different size of nested loop for "Floyd-Stienber"……..41
Figure 4.8 Experiment Results of "SC"……………………………………………….. 42
Figure 4.9 Experiment Results of "Kim"……………………………………………… 42

Chapter 1. Introduction

In embedded system, high performance *Digital Signal Processing*（DSP）is usually used in image processing, multimedia and wireless security, etc. These applications usually contain time-critical sections consisting of nested loops of instructions [4]. However, branch instructions within these loops may waste many computing resource by failure of branch testing. The optimization of such loops, considering processing resource constraints, is required in order to improve their entire computational time [1-4].

Many retiming-based methods can deal with nested loop to get a lower entire execution time, including Push-Up Schedule Method (PUSM) [1], Relax Push-Up Schedule Method (RPUSM) [2] and Bottom Up Schedule Method (BUSM) [3]. In such scheduling methods, PUSM achieve a minimum scheduling length, RPUSM reduces the entire execution time by selecting a better retiming function, and BUSM reduces the entire execution time by reducing the retiming depth. However they can not deal with the conditional branch, we use such ideas to improve entire performance.

Early work for loop scheduling with conditional branches, such as Sumit [5], Radivojevic [6] and Xie [13] do not consider the executing probability of instructions. Lakshminarayana et al. [7] and Sha [8] only consider the executing probability in task level. However, an instruction level probability definition can help us to get a more precisely schedule result than task level. In this thesis, we focus on probability definition on instruction level problem.

Resource Sharing is a useful mechanism to enhance the computing resource utilization, and many researches have discussed it already [4,6,10-11]. In [6,10], scheduling on directed acyclic graphics is considered, and in [11], one dimensional data-flow graph is studied. However, they do not consider the two dimensional nested loops.

Multidimensional Branch Anticipation (MDBA) [4] is retiming-based scheduling method used to schedule nested loops with conditional branch under resource constraint environment. It can fully utilize functional units to achieve the minimum static schedule length by the concept of PUSM. Because of the branch instruction, some instructions may not be executed. It means that some long instructions may not be executed and schedule length is not fixed. The traditional static schedule length can not perfectly present the performance of scheduling result. We need to define a more realistic measurement for nested loop with conditional branch.

Firstly, we will propose a method to evaluate the expected value of schedule length by the behavior of each branch instruction and explain why it is closed to the realistic performance. Because MDBA do not consider the behavior of conditional branches, it usually gets a higher average schedule length. Secondly, we will propose a retiming-based scheduling method to reduce the average schedule length of nested loop with conditional branch. And we will use some benchmarks to show the efficiency of our scheduling method.

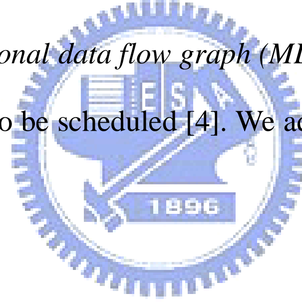
This thesis is organized as follows. In chapter 2, we introduce the fundamental background and the related work. In chapter 3, we define the expected value of schedule length and discuss our scheduling strategies. Probability Loop Scheduling Method is presented in this chapter. In chapter 4, we do some experiments and give some summary for our experimental results with MDBA to show its efficiency. Finally, we conclude our thesis in chapter 5, and list the future work of our research.

Chapter 2. Fundamental Background & Related Work

In this chapter, we define the *Multidimensional Conditional Data Flow Diagram (MD-CdDFG)* to model the nested loop with conditional branch to be scheduled and describe the concept of resource sharing. We survey several basic techniques in loop scheduling and resource sharing with conditional branch, including *Push-Up Schedule Method* [1], *Bottom Up Schedule Method* [3] and *Multidimensional Branch Anticipation* [4].

2.1 Modeling the Problem

Multidimensional conditional data flow graph (MD-CdDFG) is used to model the nested loop with conditional branch to be scheduled [4]. We add some attributes for our problem and redefine it in Definition 2.1 .



Definition 2.1 A *Multidimensional Conditional Data Flow Graph (MD-CdDFG)* $G = (V, E, d, t, k, f)$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, $E \in V \times V$ represents the set of dependence edges, d is a function from E to Z^n , representing the multi-dimensional delay between two adjacent nodes where n is the number of dimensions, t is a function from V to the positive integers, representing the computation time of each node, k is a function from V to the set of types, e.g. {*fork, join, alu, mult, div*}, f is a function from V to positive real number representing the truth probability when $v \in V$ is a fork node.

A *fork node* represents a conditional instruction in the loop body. A *join node* with zero computing time is a dummy node representing the ending of conditional statements.

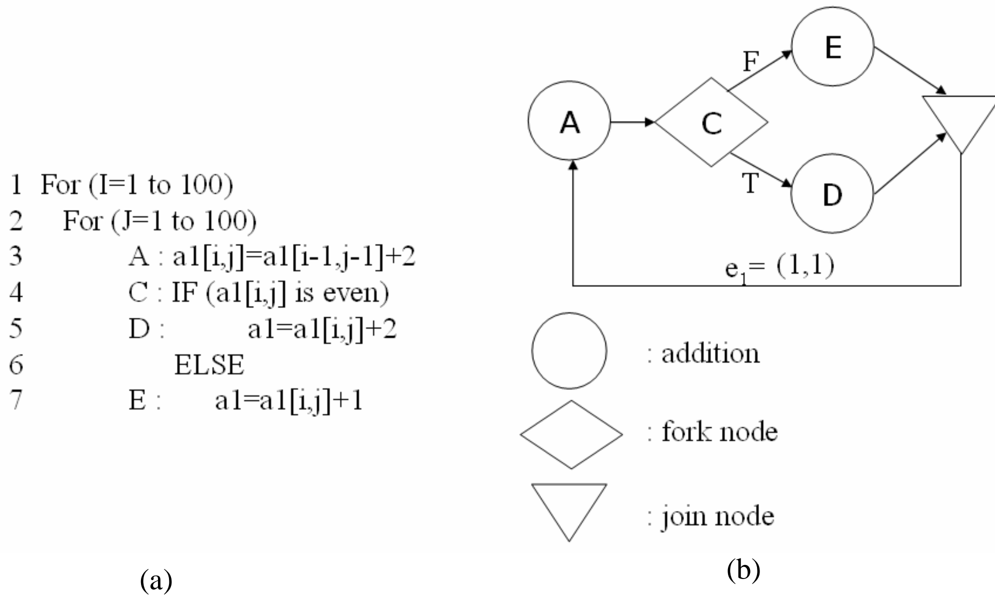


Fig 2.1 (a) High-level language code for DSP program; (b) A MD-CdDFG respect to (a) .

Fig 2.1(a) shows an example of high-level language codes of one DSP program and its equivalent two dimensional conditional data flow graph is shown in Fig 2.1(b). It's a nested loop with depth two and contains a branch instruction *C*. When *C* is true, *D* is enable and *E* is disable. When *C* is false, *D* is disable and *E* is enable. We use $d(e_1) = (d.x, d.y)$ to represent any delay edge e in a two-dimensional data flow graph.

A *conditional block* is a set of nodes including a fork node, its corresponding join node, and all nodes controlled by the fork node. Nested conditional block is allowed, which means one node may be controlled by several fork nodes. We use *conditional depth* to present the number of fork nodes which decide execution of one node. Nodes belong to the true (or false) path respect to *C* means that they are enable if the fork is true (or false). For example from Fig 2.1(b), nodes *C, D, E* and the join node consist of the conditional block. And node *D* and *E* belong to the true and false paths respect to *C* respectively.

An *iteration* is equivalent to the execution of each node in *V* exactly once, excluding disable nodes by branch instruction. For the example in Fig 2.1(b), if fork node *C* is true, an

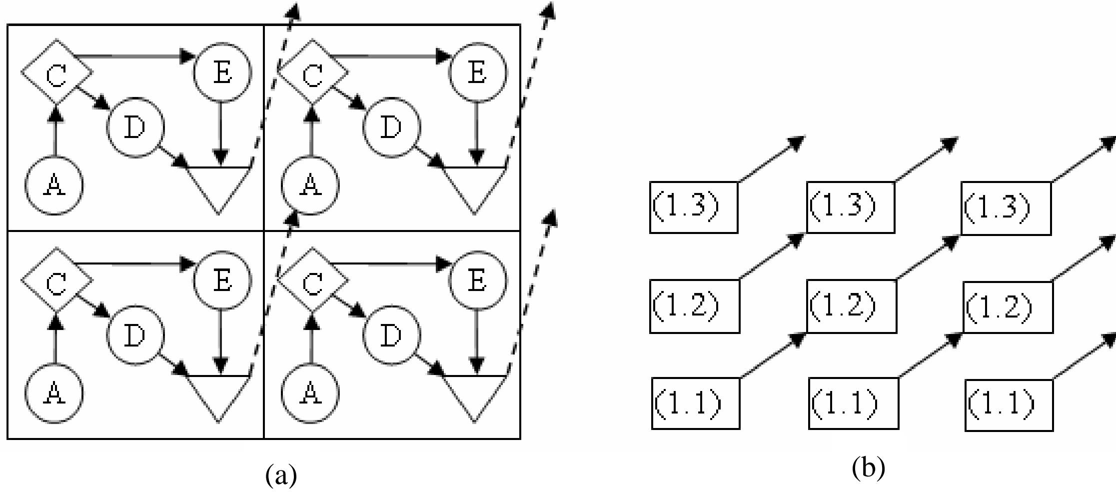


Fig 2.2 (a) A DG respect to Fig 2.1; (b) A DG without intra iteration data dependence.

iteration means the completion to node *A*, *C* and *D*. Iteration is identified by a vector *I*, equivalent to a multidimensional index, starting from *(1,1...,1)*. Inter-iteration dependencies are represented by vector-weighted edge in an MD-CdDFG such as $e_1 = (1,1)$ in the Fig 2.1(b). It means the result of fork node will be used by another iteration with indexing distance *(1,1)* in the iteration space.

For any iteration *j*, edge from node *u* to node *v* with delay vector *d(e)* means that the computation of node *v* at iteration *j* depends the execution of node *u* at iteration *j-d(e)*. An edge with delay *(0,0...,0)* in an MD-CdDFG represents a data dependence within the same iteration. A *legal* MD-CdDFG must have no zero-delay cycle, i.e., the summation of the delay vectors along any cycle can't be (0,0…,0) [ ].

A *cell dependence graph (DG)* of the MD-CdDFG *G* is the directed acyclic graph, showing the dependence between copies of nodes representing an MD-CdDFG *G*. Fig 2.2(a) shows the DG based on the replication of the MD-CdDFG in Fig 2.1, and Fig 2.2(b) shows its data dependence of DG represented by computational cells. A computation cell is the DG node that represents a copy of the MD-CdDFG, excluded the edges with delay vectors different from (0,0…,0). The computation cell is considered as an atomic execution unit.

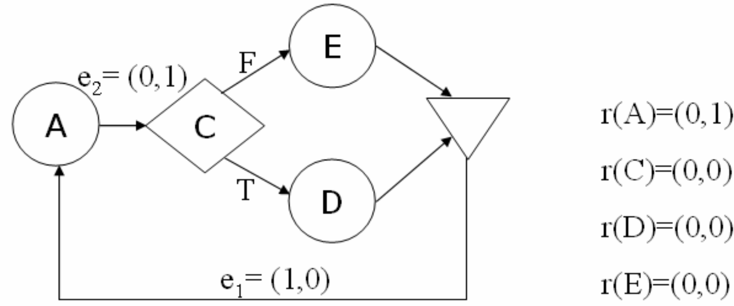
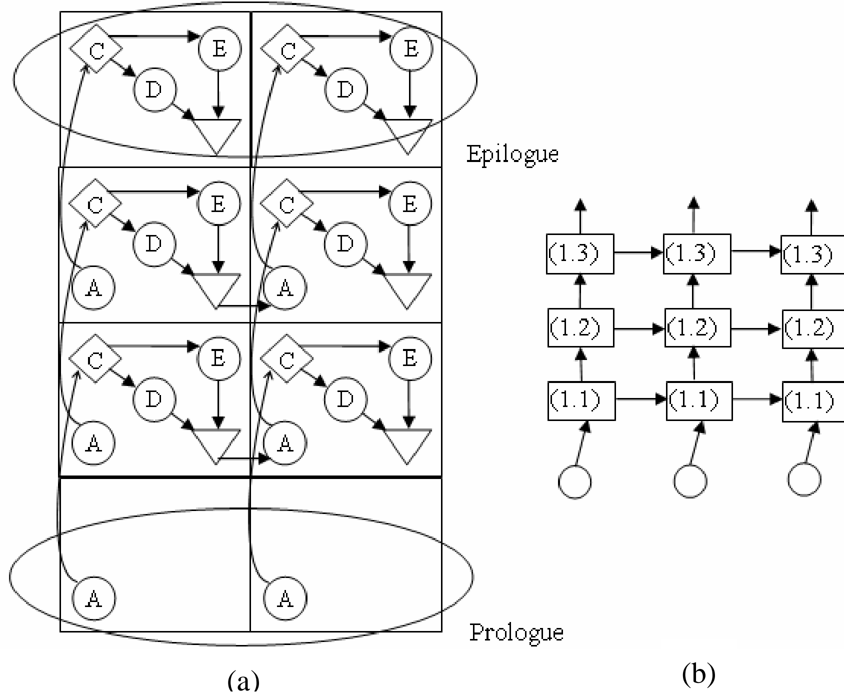


Fig 2.3 A retimed MD-CdDFG of Fig2.1(b).

2.2 Multidimensional Retiming [1,9]

For as MD-CdDFG *G*, the *multidimensional retiming r* is a function from *V* to Z^n that redistributes nodes in consecutive iteration. Each iteration, represented by loop index, still contains all nodes in V with one instance. A new MD-CdDFG $G_r = (V, E, d_r, t, k, f)$ is created, such that the summation of delay vectors of any cycle is unchanged. The retiming vector *r(u)* of a node $u \in G$ represents the offset between the original iteration containing *u* and the one after retiming. The delay vectors change accordingly to preserve data dependencies, i.e., *r(u)* represents delay components pushed into the edge $u \rightarrow v$, and subtracted from the edge $w \rightarrow u$, where $u,v,w \in V$. After retiming, the execution of node *u* in iteration *i* is moved to the iteration *i-r(u)*. For example, Fig. 2.3 shows the retimed MDFG G_r after applying retiming function *r(A)=(0,1)* on *G*. We can use the Definition 2.2 to obtain the retimed delay vector for every edge *e* in *E*.

Definition 2.2 For any MD-CdDFG *G=(V,E,d,t,k,f)*, retiming function *r*, and retimed MD-CdDFG $G_r=(V,E,d_r,t,k,f)$, we define the retimed delay vector for every edge *e* in *E*, the retimed delay vector for every path in *G*, and the retiming delay vector for every cycle in *G*, denoted as $d_r(e)$, $d_r(p)$, $d_r(l)$ respectively by the following formulas:



(a) (b)

Fig 2.4 (a) A retimed DG respect to Fig 2.3; (b) A retimed DG without intra iteration data dependence .

1896

(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 any path $u \xrightarrow{e} v$, $u,v \in V$ and $p \in G$.

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

In Fig. 2.4(a), we show the retimed DG based on the replication of the MD-CdDFG in Fig. 2.3 and the retimed DG represented by computational cells is shown in Fig. 2.4(b). The retiming function applied to an MD-CdDFG may create *prologue* and *epilogue*. Prologue is the set of instructions that must be executed to provide the necessary data for the beginning of the iterative process. Epilogue is the set of instructions that must be executed to complete the process [2]. These two sets of instructions are complementary. For example in Fig. 2.4(a), the instruction *A* becomes the prologue, and the instruction *C*, *D* and *E* become epilogue for this problem.

A *schedule vector s* is the normal vector for a set of parallel equitemporal hyper planes that define the sequence of execution of the cell dependence graph. To get a schedule vector *s*, we can solve the inequalities $d(e) \cdot s \geq 0$ for every $e \in E$ [2]. For example, (1,0) is a schedule vector of the MD-CdDFG in Fig. 2.1(b).

Definition 2.3 A legal MD-CdDFG $G = (V, E, d, t, k, f)$ that have no zero-delay cycle is *realizable* if there exits a schedule vector *s* for the cell dependence graph with respect to *G*, i.e., $s \cdot d \geq 0$ for any $d \in G$ [1-3].

Property 2.4 Given a realizable MD-CdDFG *G*, a *legal multidimensional retiming* for *G* is the multidimensional retiming function *r* that transforms *G* into G_r, such that G_r is still realizable [1-3].

A legal multidimensional retiming on an MD-CdDFG $G = (V, E, d, t, k, f)$ requires that the execution sequence of the corresponding retimed DG does not contain any cycle. This constraint is enforced through the use of a *schedule vector* that supports the realization of the retimed graph.

Property 2.5 If *r* is a multidimensional retiming function orthogonal to a schedule vector *s* that realizes an MD-CdDFG $G = (V, E, d, t, k, f)$, and $u \in V$, then $(k \times r)(u)$ is also a legal multidimensional retiming on that MDFG [1-3].

From the Property 2.5, which is called as *chained multidimensional retiming*, we know that the retiming function of every node in the retimed MDFG can be in the form $(k \times r)$. Here, *r* is called *retiming base*, and *k* is called *retiming depth* [2].

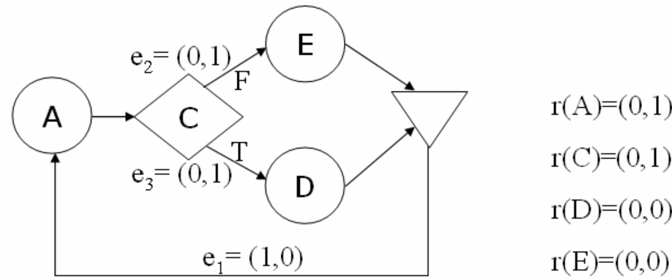


Fig 2.5 (a) A retimed DG respect to Fig 2.1 .

2.3 Resource Sharing [4,6,10-12]

Our method works on a uni-processor VLIW system with multiple various functional units which can be executed concurrently [10]. We assume that each type of functional unit has the same execution time, called a "*Control Step*", or a "*C-Step*". That is in one C-Step, a long instruction consist of several individual operations. To enhance the utilization of computing resource, we have some difference in execution stage. *Branch Anticipation logic* and *Address Control logic* are the first and second steps in the execution stage [4,10].

Branch Anticipation logic allow multiple conditional signal to cross iteration. From Fig 2.5, branch conditional testing of fork node *C* decides either *D* or *E* should be executed. However, *C* and *D, E* are in different iteration with offset (0,1). Branch Anticipation logic can use some hardware to record the testing result to the later iteration.

Because of the branch instruction, we observe some nodes are mutual exclusive. It means that only one of them can be executed. For example in Fig 2.5, D and E are mutual exclusive. We use a mechanism to enhance the utilization of functional unit called *resource sharing*. If two nodes are resource sharing, we assign them into the same functional unit. However, it needs some hardware to support such mechanism.

Address Control logic can enhance resource utilization by assigning several nodes to the same functional unit at one step. From Fig 2.5, if node *D* and *E* use the same functional unit, they can share the same operation and only one of them can be executed in an iteration. We

denote "*D/E*" to present that node *D* and *E* share one operation. Branch Anticipation logic and Address Control logic can work together. When a long instruction is fetched, two logics decide which one instruction is assigned to which one functional unit. It is easy to observe in an acyclic conditional block, all nodes sharing the same functional unit must come from the same iteration.

2.4 Related Work

Many scheduling algorithms are designed based on multidimensional retiming technique. In this section we briefly describe some of them, including Push-Up Schedule Method (PUSM) [1], Bottom-Up Schedule Method (BUSM) [3] and Multidimensional Branch Anticipation (MDBA) [4].

2.4.1 Push-Up Schedule Method [1]

In order to make the schedule length shorter, PUSM uses retiming technique to change the dependence in the MDFGs. PUSM will first analyze that if a node could be scheduled, and then use retiming technique to make the node *schedulable* as early as possible. Now, we define what a *schedulable node* as follows.

Definition 2.7 *(Schedulable Condition)* Given an MDFG $G = (V, E, d, t)$ and a node $u \in V$, u is a *schedulable node* at a C-Step *cs*, if it satisfies one of the following conditions:

(a) u has no incoming edges

(b) all incoming edges of u have a nonzero multidimensional delay

(c) all predecessors of u, connected to u by a zero-delay edge, have been scheduled to earlier control steps

Before traversing the MDFG *G*, a schedule vector *s* realizing *G* and a legal retiming *r* on *G* will be found. Then scheduling an MDFG *G* by PUSM, it uses a queue to maintain the set of schedulable nodes. Scheduler fetches a schedulable node and places it into schedule table at an earliest C-Step to get a minimum schedule length. Schedulable nodes are filled into schedule table sequentially and get a minimum schedule length.

During traversing *G*, every traversed node will record the retiming count function $RC(u), u \in V$. $RC(u)$ represents the number of extra nonzero delays required by any path from roots of *G* to node *u*. When we schedule one node *u* into schedule table , we “push-up” *u* to the earliest available functional unit. If this scheduling violate the data dependence, we increase *RC(u)* to add the retiming depth. Retiming count will propagate to its successors. After traversing *G*, PUSM uses retiming count to calculate the retiming function of every node by the following formula:

$$\forall u \in V, r(u) = (Max\{RC(v), \forall v \in V\} - RC(u)) \times r$$

PUSM promises to get a minimum static schedule length but ignores the effect of the retiming depth which affects the entire execution time of a scheduled nested loop. In the next section, we survey the BUSM which provides a method to reduce the retiming depth.

2.4.2 Bottom-Up Schedule Method [3]

The PUSM gets a minimum schedule length but cause high retiming depth. Bottom-Up Schedule Method (BUSM) reduces retiming depth and holds the same schedule length. The main idea of BUSM calculate the *maximum schedule length* of MDFG *G* before allocating nodes into schedule table. The formula for computing *maximum schedule length (MSL)* is

$$MSL(G) = \max\{1, \lceil ADD / A \rceil, \lceil Mult / M \rceil\}$$

The *ADD* means the number of addition and *A* means the number of adder in the processor. It is the same in the multiplication, and easy to extend to the architecture with more types of

functional units. Scheduler prepares an empty schedule table with size as *MSL* which is equal to the schedule length of PUSM.

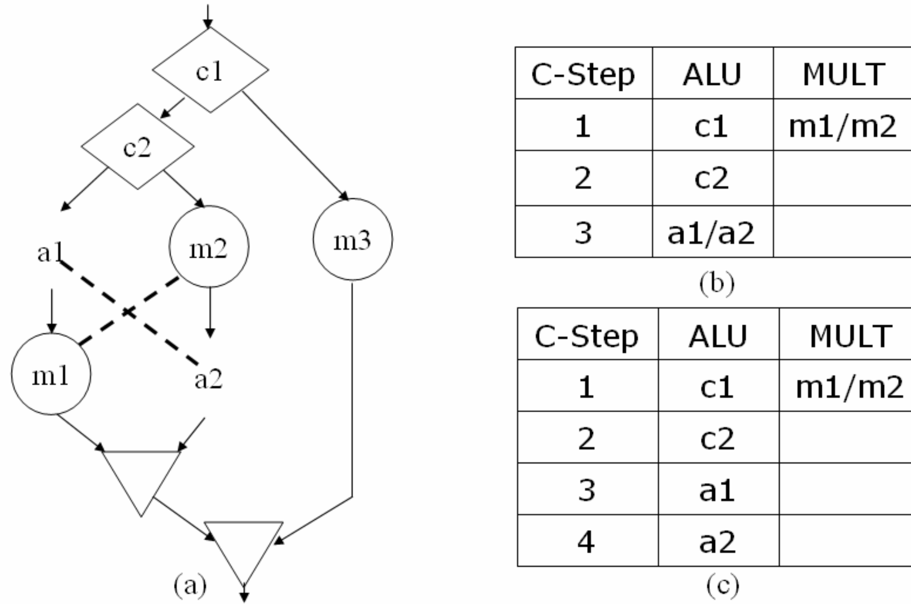
Then BUSM use the same schedulable condition in Definition 2.7 to allocate node into schedule table. The policy for node allocation is trying to allocate node into schedule table without increasing retiming depth. It is different from PUSM which always tries to allocate node in an earlier C-Step.

However, BUSM can not deal with conditional branch instruction and resource sharing problem. We will improve it in our method..

2.4.3 Multidimensional Branch Anticipation [4]

Multidimensional Branch Anticipation is based on PUSM with resource sharing to deal with nested loop with conditional branch problem in VLIW architecture [ ]. When we use PUSM to find an earlier available functional unit, we also find all scheduled operations for the possibility of resource sharing.

As we schedule one node *u* into schedule table and share it with other nodes, such operations should have the same retiming depth. If we always use the earliest functional unit to allocate node *u*, there may cause a contradiction in data dependence as a cycle. Fig 2.6(a) is a part of MD-CdDFG *G*, *m1*,*m2* and *a1* have scheduled into schedule table and *m1,m2* share the same multiplier. The dotted edges in Fig 2.7(a) are *sharing indication edges*. The graph consisting of *G* and sharing indication edges is denoted by G_s consisted by MD-CdDFG and resource sharing indication edges. The retiming depth of *m1* and *m2* is the same, so as *a1*and *a2*. By the propagation of RC in the PUSM, retiming depth will propagate and increase infinitely along the path a*1*→ *m1*→*m2*→*a2*→*a1*. We note such situation as a "*sharing-prevention cycle*".



C-Step	ALU	MULT
1	c1	m1/m2
2	c2	
3	a1/a2	

C-Step	ALU	MULT
1	c1	m1/m2
2	c2	
3	a1	
4	a2	

Fig 2.6 (a) A part of MD-CdDFG; (b) A schedule table with sharing prevention cycle; (c) A schedule table without sharing prevention cycle.

MDBA provide a mechanism, called "*AVAIL-PREVENT*" to solve such problem. In *AVAIL-PREVENT*, we pseudo-allocate a node into schedule and try to detect any sharing-prevention cycle in graph G_s. If a sharing-prevention cycle is detected, we recover the schedule table and allocate the node to a later C-Step and try to detect again until no sharing-prevention cycle in G_s. Similar to PUSM, we use the retiming base to calculate the retiming function of each node $u \in V$.

In this section, we have surveyed several retiming based scheduling algorithm. PUSM can achieve a minimum schedule length but cause a high retiming depth. BUSM reduces the retiming depth but can not deal with branch instruction. MDBA bases on PUSM to get a minimum schedule length but does not consider the behavior of conditional branch. In the next chapter, we will discuss the importance of branch behavior and present our motivation to propose a new scheduling method. Then our new method will be described clearly.

Chapter 3. Probabilistic Loop Scheduling Method

In this chapter, we will finely introduce Probability Loop Schedule Method (PLSM) to deals with the nested loop with conditional branch. This chapter is organized as follows. In section 3.1, we explain our motivations and discuss our scheduling goal. In section 3.2, we define the instruction level executing probability and a new measurement for average performance. For the definition of average schedule length, we propose some schedule strategies in section 3.3. Finally in the section 3.4, we completely propose an algorithm to achieve the scheduling goal.

3.1 Motivation

From the related work, PUSM and MDBA use the schedule length as a performance measurement [1,4]. Because of the conditional branch, some operations may not be executed and the schedule length is not fixed. Using schedule length as the measurement may overcount the performance. It is necessary to define a measurement to deal with the nested loop with conditional branch.

We use an example to explain the shortcoming of schedule length. Fig 3.1(a) is a source fragment code in high level language and Fig 3.1(b) is its respected MD-CdDFG. It is a one-level conditional block and we assume the true probability of the fork node *C* is 0.5. It means the executing probability of node *D* is 0.5 and so as *E*.

A simple architecture is used to explain our motivation in this section, and it can complete two operations in a C-Step. A retimed scheduling result of source code is shown in Fig 3.2(a). The integral superscript of each operation represents the multiple of retiming base by Property 2.5. Fig 3.2(b) shows two consecutive iterations which distance is one unit of

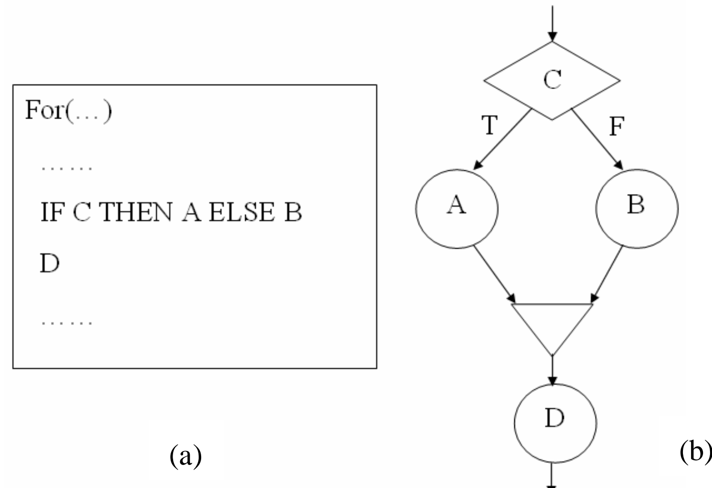
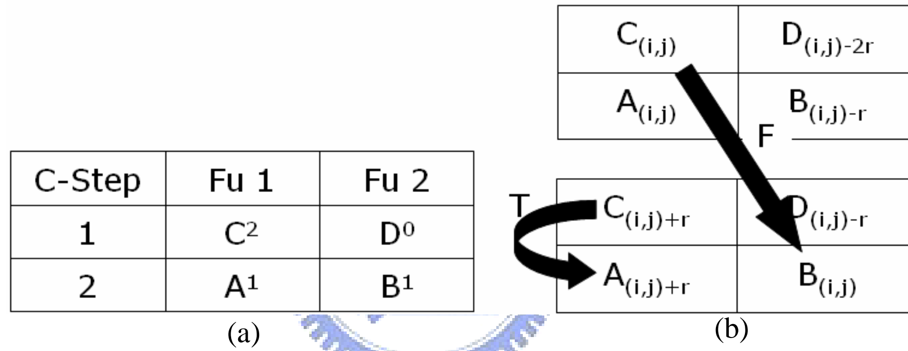


Fig 3.1 (a) A source code fragment (b) MD-CdDFG respect to (a).



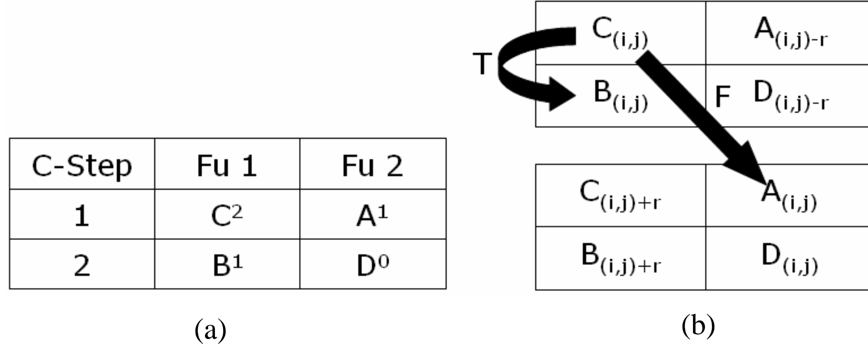
C-Step	Fu 1	Fu 2
1	C^2	D^0
2	A^1	B^1

Fig 3.2 (a) Schedule result of our motivation (b) The consecutive two iterations.

retiming base r in the iteration space. The subscript of each operation represents its loop index.

In the Fig 3.2(b), it is an example to show our motivation. Although its schedule length is two, we can observe that the second long instruction which contains nodes A and B may not be executed. When $C_{(i,j)}$ is true and $C_{(i,j)+r}$ is false, the second long instruction in the second iteration does not be executed. We assume that the true probability of $C_{(i,j)}$ and $C_{(i,j)+r}$ are independent and expected executing probability of the second long instruction is $1\text{-}0.5\times0.5=0.75$.

From the related work, a long instruction uses Branch Anticipation logic and Address Control logic firstly in the execution stage [10]. If all operations in a long instruction need not



C-Step	Fu 1	Fu 2
1	C^2	A^1
2	B^1	D^0

(a) (b)

Fig 3.3 (a) Schedule result of MDBA (b) The consecutive two iterations.

to be executed, execution stage stops after Address Control logic [ ]. Because time consumption of such two logics are small, we can regard that execution time is saved.

We can compute the average schedule length of the first scheduling result. The first long instruction is always executed and the second has probability *0.75* to be executed. The average schedule length is *1.75*. On the contrary in Fig 3.3(a), we use MDBA to get a scheduling result which schedule length is two. However the first and second long instructions are always executed and the average schedule length is 2 C-Step. We discover that the schedule length can not present the difference between such two scheduling results, and we need to define the average schedule length as our new measurement. It is more closed to the realistic executing performance of nested loop problem.

MDBA does not consider the behavior of branch instruction, and usually get a high schedule length in average. We need to design a method to deal with such problem and get a lower average schedule length. In the next section, we define the measurement to evaluate the average schedule length.

3.2 Basic Concept

In this section, we first define the executing probability of each operation in a given MD-CdDFG. Second, we infer the relationship from the executing probability of an operation to long instruction. Finally, we define the expected schedule length of a scheduling result and

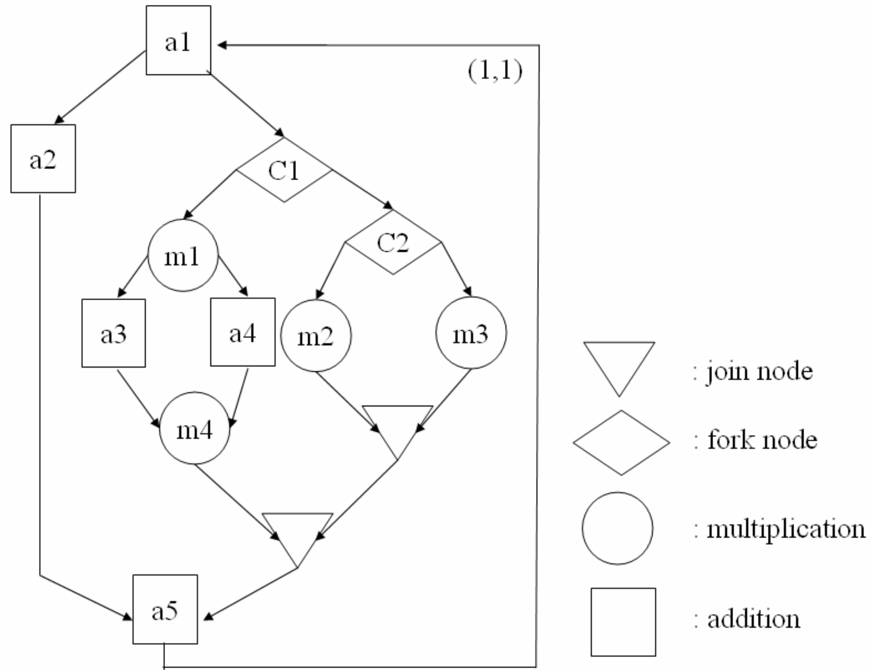


Fig 3.4 A example of MD-CdDFG.

explain its relationship to the average performance of nested loop.

Definition 3.1 Given an MD-CdDFG $G=(V,E,d,t,k,f)$, for each node $u \in V$, u belongs to conditional block $c_1, c_2, \ldots, c_n$. The executing probability of node u is defined as

$$p(u)=k(c_1)k(c_2)...k(c_n) \quad \text{where} \quad \begin{cases} k(c_i)=f(c_i) & \text{where } u \text{ belongs to the true path of } c_i \\ k(c_i)=1-f(c_i) & \text{where } u \text{ belongs to the false path of } c_i \end{cases}$$

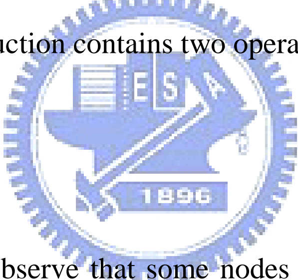
For example in Fig 3.4, if *f(c1)=0.2* and *f(c2)=0.5,* the executing probability of *a3* is *p(a3)=f(c1)* and *m3* is *p(m3)=(1-f(c1))(1-f(c2))=0.8×0.5=0.4.*

Because a long instruction is consisted by several operations, we use *PL* to present the execution probabilistic of long instruction and explain the relationship between $p(u)$ and PL. In a general case, we assume a long instruction can complete n operations in parallel and having executing probability as $p(1), p(2)...p(n)$. The PL is equal to the complement that all of n operations are not be executed in the same time. We use a formula to present it.

$$PL(i) = 1-(1-p(1))(1-p(2))...(1-p(n)) \quad \text{...} \quad (1)$$

It is correct in most cases where the event of $p(1), p(2)...p(n)$ are independent. However, we know the probability of each node execution is not always independent. For example in Fig 3.4, it is a dependent case on *a3* and *a4*, they are always executed in the same time when *C1* is true. We need some modifications for dependent cases.

In the later, when a operation and long instruction with probability less than one, we describe them as *conditional*. On the contrary, a operation and long instruction is described as *unconditional* when their executing probability are equal to one. We define a term "*source*" to present the latest common predecessor of two nodes in a given MD-CdDFG. One function *t=Source(u,v)* presents that *t* is the source of *u* and v, where $t,u,v \in V$. For example in Fig 3.4, it shows that *Source(m2,m3)=c2*, *Source(m2,a3)=c1* and *Source(a1,m3)=a1*. We use a simple architecture which a long instruction contains two operations in this section.



3.2.1 Exclusive Nodes

In the example, we can observe that some nodes are *mutual exclusive* and only one of such mutual exclusive nodes can be executed. We define the relationship to describe the relationship.

Definition 3.2 For a given *MD-CdDFG G*, $\forall$ *u*, $v \in V$ are *exclusive* if they satisfy the following property: There exists a fork node $c \in V$ such that *u* belongs to the true path and *v* belongs to the false path of *c*.

For example in the Fig 3.4, *m2* and *m3* belong to the different side of fork node *c2*. If *m2* and *m3* come form the same iteration, they are exclusive and *p(m2)*, *p(m3)* are dependent. We need to modify the formula (1) to deal with such exclusive nodes. We assume that there are

two independent nodes *u*, *v* in a long instruction, its executing probability is:

$$PL = 1-(1-p(u))(1-p(v)) = p(u)p(v) + p(u)'p(v) + p(u)p(v)' \quad (2)$$

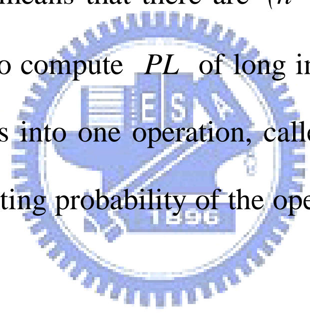
Now we assume that *u* and *v* are exclusive, *u* and *v* can not be executed in the same time and $p(u)p(v)=0$. We modify (2) as:

$$PL = p(u)'p(v) + p(u)p(v)' \quad (3)$$

When *u* has executed, *v* is always disabled. It means that $p(v)'=1$ and $p(u)p(v)'=p(u)$. We can modify (3) as:

$$PL = p(u) + p(v) \quad (4)$$

Finally, we apply (4) to modify (1) where there are *n* operations in one long instruction. If there are two exclusive operations, we sum there executing probability and use only one operation to replace them. It means that there are $(n-1)$ independent operations in a long instruction and we apply (1) to compute PL of long instruction. In our architecture, we can assign several exclusive nodes into one operation, called "resource sharing" and all sharing nodes are exclusive, the executing probability of the operation is the summation of all sharing nodes.



For example, if a long instruction consist two operations, one is *a4* and the other is shared by *m2/m3*. The probability of first operation is equal to the executing probability of *a4*. The other operation has probability $p(m2/m3)=p(m2)+p(m3)$ because they share one functional unit. Then we discover that *a4* and *m2/m3* are exclusive, the executing probability of long instruction equal to the summation of such two operations; $PL=p(a4)+p(m2/m3)=p(a4)+p(m2)+p(m3)$.

3.2.2 Inclusive Nodes

We discover that some nodes are always executed in the same time. For example in Fig 3.4, when *c1* is true, *a3* and *a4* are executed in the same time. Executing probability of *a3* and

a4 are not independent and we need a modification for (1). Now we define a relationship to describe this situation.

Definition 3.3 For a given *MD-CdDFG G*, $\forall$ *u*, $v \in V$ are *inclusive* if they satisfy the following property: For all fork nodes $c \in V$ such that *u* and *v* belong the conditional block of *c*, they are in the same side.

We assume two independent nodes *u*, *v* into a long instruction, its executing probability is:

$$PL = 1-(1-p(u))(1-p(v)) = p(u)p(v) + p(u)'p(v) + p(u)p(v)' \quad (2)$$

If u, v are inclusive, they should be executed in the same time. It means $p(u)'p(v) = p(u)p(v)' = 0$. We modify (2) as :

$$PL = 1-(1-p(u))(1-p(v)) = p(u)p(v) \quad (5)$$

When *u* has been executed, $p(v)=1$ and we modify formula (5) is as:

$$PL = p(v) \quad (6)$$

Finally, we apply (6) to modify (1) where are *n* operations in a long instruction. If there are two operations which are inclusive, we delete one of them. It means that there are $(n-1)$ independent operations in a long instruction and we apply (1) to compute the PL. For example in Fig 3.4, when *a3* and *a4* are scheduled in a long instruction, we delete the probability of *a4* and $PL=p(a3)$.

We have proposed some modification to correct the executing probability of a long instruction. Now we discuss how to compute the executing probability of a long instruction. Firstly, we find the inclusive relationship in a long instruction and delete the redundant operations. Second, we merge the executing probability of sharing nodes. Third, we find the exclusive operations in the long instruction and use only one operation to merge their executing probability. Finally, we use formula (1) to compute the executing probability from

the remaining independent operations and get the executing probability of each long instruction.

3.2.3 Expected Value of Schedule Length

We have defined the executing probability of each long instruction. Because we schedule the MD-CdDFG into several long instructions, we can sum PL to define the expected value of schedule length as follow.

Definition 3.4 *Expected value of schedule length (ESL)* is defined as $ESL = \sum_{i=1}^{MSL} PL(i)$, where MSL is the maximum schedule length of *MD-CdDFG G* and $PL(i)$ is the executing probability of long instruction *PL* at C-Step i.

The behavior of fork node decides the average number of instructions to be executed in an iteration. We schedule instructions into several long instructions and get the maximum schedule length. Schedule length can not present the performance perfectly because it is static. We propose the *ESL* which is computed by the behavior of fork nodes means that the average computation time of loop body. The *ESL* is a more fair measurement for the performance of scheduling result.

Our schedule goal is *ESL* reduction. The *ESL* reduction means that we collect some conditional operations into a conditional long instruction. Because the loop body usually use large part of computation time, *ESL* reduction can decrease the entire execution time. In the next section, we propose a new method for *ESL* reduction.

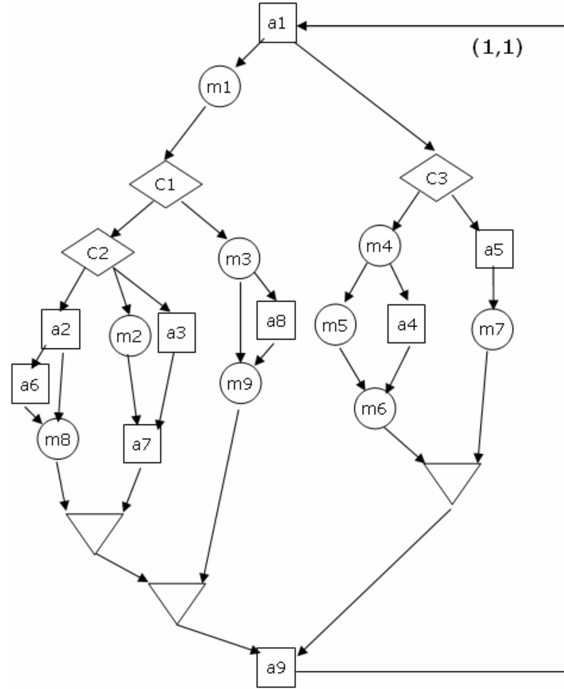


Fig 3.5 A example of MD-CdDFG.

3.3 Schedule Strategy

Our scheduling goal is to reduce the *ESL* and we propose several heuristic strategies to achieve it. From the related work, *MSL* is defined as :

$$MSL = max\{ \frac{\#\ of\ \ operations}{\#\ of\ \ functional\ unit} \}$$

MSL is the upper bound of schedule length. We need to reduce the number of operations to reduce the upper bound and to share several nodes into one operation. We define the resource sharing condition to indicate which nodes can share one operation.

Definition 3.6 *(Resource Sharing Condition)* In a given MD-CdDFG *G*, $u, v \in V$, *u* and *v* can share the same operation if they satisfy:

(1) *u* and *v* use the same type of functional unit.

(2) *u* and *v* are exclusive

(3) *u* and *v* do not exist *"sharing-prevention cycle"*

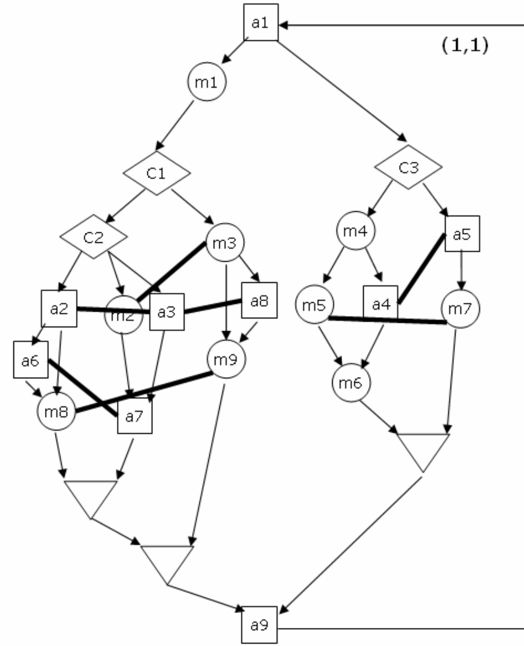


Fig 3.6 The resource sharing policy of our example

1896

We use this definition to check all pairs of nodes in the MD-CdDFG, and get a G_S to present resource sharing policy. The condition (3) can be detected by traversing the G_S in the MDBA.

We use Fig 3.6 as our example in this section and the result for finding resource sharing policy is shown in Fig 3.7. It shows that *a2/a3/a8*, *a6/a7* and *a4/a5* share one adder. By formula (4) in 3.3.2, we can compute that *a2/a3/a8* and *a4/a5* are unconditional. After finding all resource sharing policy, the number of operations is reduced. MSL can be also reduced by resource sharing. Thus, our scheduling strategy is given below.

Schedule strategy 1 To reduce the upper bound of schedule length, we find more and more nodes to share one operation.

We also need to reduce the lower bound of schedule length. From (1), if there exists an unconditional operation in a long instruction, the executing probability of this long instruction PL is equal to 1. It is an unconditional long instruction which always be executed in each

iteration. Because the resource sharing policy has been found, we define the number of unconditional long instruction as *Minimum Schedule Length (NSL).*

Definition 3.7 Given an *MD-CdDFG G* after finding resource sharing policy, the *Minimum Schedule Length (NSL)* is defined as $NSL = \max\{\frac{\#_of_unconditional_operations}{\#_of_functional_unit}\}$

For example in Fig 3.7(b), we have found the resource sharing policy and get unconditional operations as { *a1, m1, c1, c3, a4/a5, m5/m7, m2/m3, a2/a3/a8* }. If we have two adders and one multiplier in our architecture, the $NSL = max\{\frac{5}{2}, \frac{3}{1}\} = 3$. *NSL* is the lower bound of schedule length, it physical meaning is that three long instructions always be executed. However, there are three conditional long instructions with executing probability, the $ESL = 3 + PL(4) + PL(5) + PL(6)$ by Definition 3.5. Thus, given an *MD-CdDFG G* after finding resource sharing policy, we can compute the *MSL*, *NSL*, *ESL* and know that $MSL \geq ESL \geq NSL$.

We need a minimum number of unconditional long instructions to reduce *NSL*. Our second scheduling strategy can be given as below.

Schedule Strategy 2 To reduce the lower bound of schedule, we schedule unconditional operations into minimum number of long instructions.

After scheduling all unconditional operations into minimum number of long instruction, there are still some conditional operations to be scheduled. We need to schedule them into conditional long instructions and also need to reduce its executing probability. We consider the pervious simple architecture with only two operations, and *p*, *q* are scheduled into the long instruction.

If p and q are exclusive, its PL is $p(p)+p(q)$. However, if p and q are independent, its PL is $1-(1-p(p))(1-p(q))$ by (1). Because *p(p)* and *p(q)* are positive, $p(p)+p(q) > 1-(1-p(p))(1-p(q))$ is always right. It means that exclusive nodes will increase PL and we should propose a heuristic method to avoid such exclusive nodes. We have many methods to avoid exclusive and we choice the easiest one which select one operation to increase its retiming count. Then such two operations will become independent. This heuristic method will increase the retiming depth of our scheduling result, but reduce *ESL* which is our schedule goal. This heuristic method has a disadvantage which retiming depth will increase. Our third scheduling strategy is shown as following.

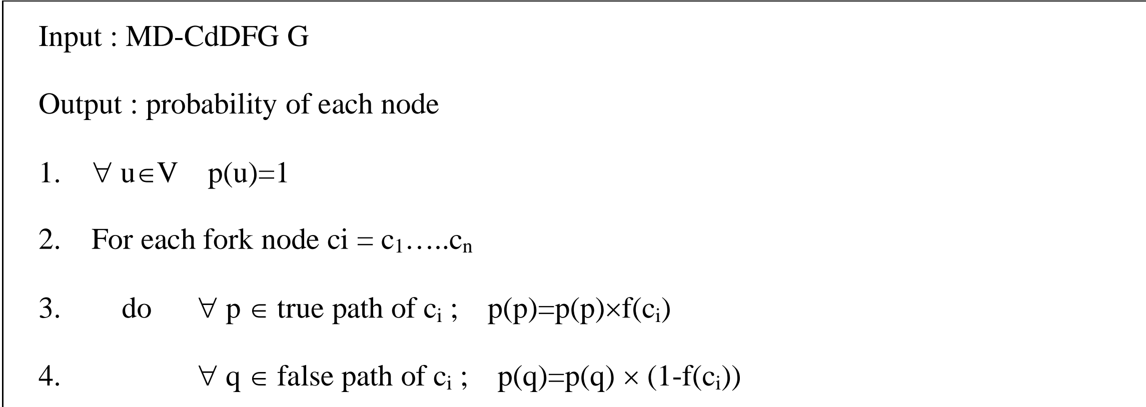
Schedule Strategy 3 When we schedule two exclusive conditional operations into a long instruction, they need to be set different retiming count.

We have proposed three scheduling method and we will propose our scheduling method in the next section. Such three scheduling method will be used to decrease the average schedule length *ESL*.

3.4 Probabilistic Loop Schedule Method

In this section, we propose our detail algorithm and use an example to explain how they work. Our scheduling goal is a static schedule result with shorter *ESL*. Input of our algorithm is the MD-CdDFG *G*, and outputs is the retiming of each node $u \in V$..

Our method which called Probabilistic Loop Scheduling Method (PLSM) contains four steps. The first step computes the executing probabilistic of each node. The second step finds the sharing policy of graph. The third step allocates operations into schedule table which is one output. The final step computes the retiming of each node $u \in V$.



Input : MD-CdDFG G

Output : probability of each node

1. $\forall$ u$\in$V p(u)=1
2. For each fork node ci = $c_1 \ldots c_n$
3.   do $\forall$ p $\in$ true path of c_i ; $p(p)=p(p)\times f(c_i)$
4. $\forall$ q $\in$ false path of c_i ; $p(q)=p(q) \times (1-f(c_i))$

Fig 3.7 Step 1 of PLSM : Calculating the executing probability for each node

Fig 3.7 shows the algorithm for calculating the probability of each $u \in V$. In the algorithm is an implement of Definition 3.1, Line 1 initially sets all nodes with probability equal to 1. From Line 2 to Line 4 is a loop indexed by the number of fork nodes. When loop is indexed by c_i, we multiply the true probability to every node which belongs to the true path of c_i and multiply the false probability to every node which belongs to the false path of c_i. After processing all fork nodes, each node gets an executing probability. For example in Fig 3.8, we assume the executing probability of each node and the italic real number means the result of probability calculation. We can know the executing probability of each node after the first step, the second step we need to find the resource sharing policy.

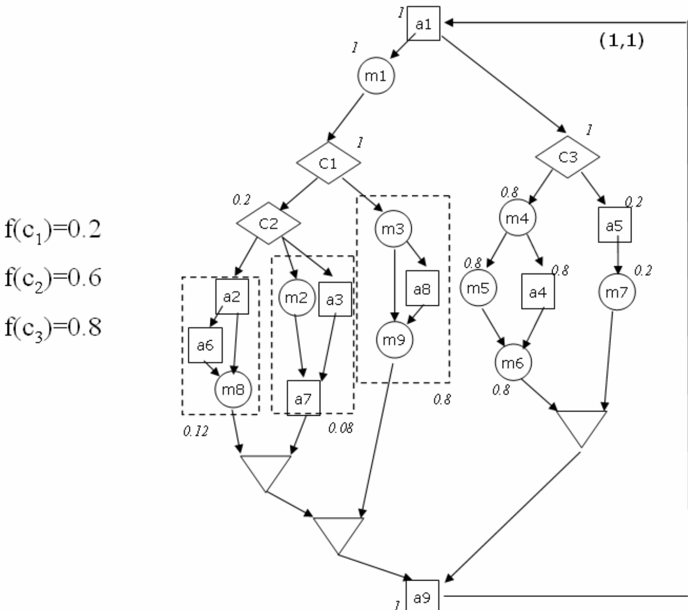


Fig 3.8 After calculating executing probability of each node.

```
Input : MD-CdDFG with p(u)    ∀u∈V
Output : G_S : MD-CdDFG with sharing indication edges
1   Sort fork node Ci=C1…Cm   By its Decreasing Order of Conditional Depth
2   Sort type of FU FUj=FU1…Fun By its (number of nodes / FU in system)
3   For (Fui=Fu1…Fun)
4     For (Cj=C1…Cm)
5         For each u s.t {(u ∈ true path respect to Ci ; shared u first) }
6               For each v s.t {(v ∈ false path respect to Ci ; shared v first )}
7                     IF (   (Fu(u)=Fu(v)=Fuj)
8                            AND ( u and v are exclusive)
9                            AND ( u and v are not sharing-prevention cycle ) )
10                  THEN   A sharing indication edge u→v
11                           p(u)=p(u)+p(v) ; p(v)=p(u);
```

Fig 3.9 Step 2 of PLSM : Algorithm for finding resource sharing policy

Fig 3.9 is the algorithm for finding resource sharing policy. Input of this algorithm is the MD-CdDFG with calculated the executing probability of each node in the first step. From the Line 5 and Line 9, we select a pair of nodes to check the resource sharing conditions in Definition 3.6. If we find two nodes which satisfy the sharing conditional, we use a sharing indication edge to connect them and sum their executing probability in Line 10 and 11. By the first schedule strategy, Line 1 and Line 2 try to merge more nodes into one operation. The Line 1 means that we find the sharing policy from the deeper conditional block first. The Line 2 means that we find the sharing nodes from the functional units which use longer schedule length. They are useful to enhance the number of resource sharing nodes. Fig 3.10(a) shows the resource sharing policy after finding addition, and Fig 3.10(b) shows the finding resource sharing policy after finding multiplication. After finding sharing policy, operations are classified into *unconditional* and *conditional* by its executing probability in Fig 3.11.

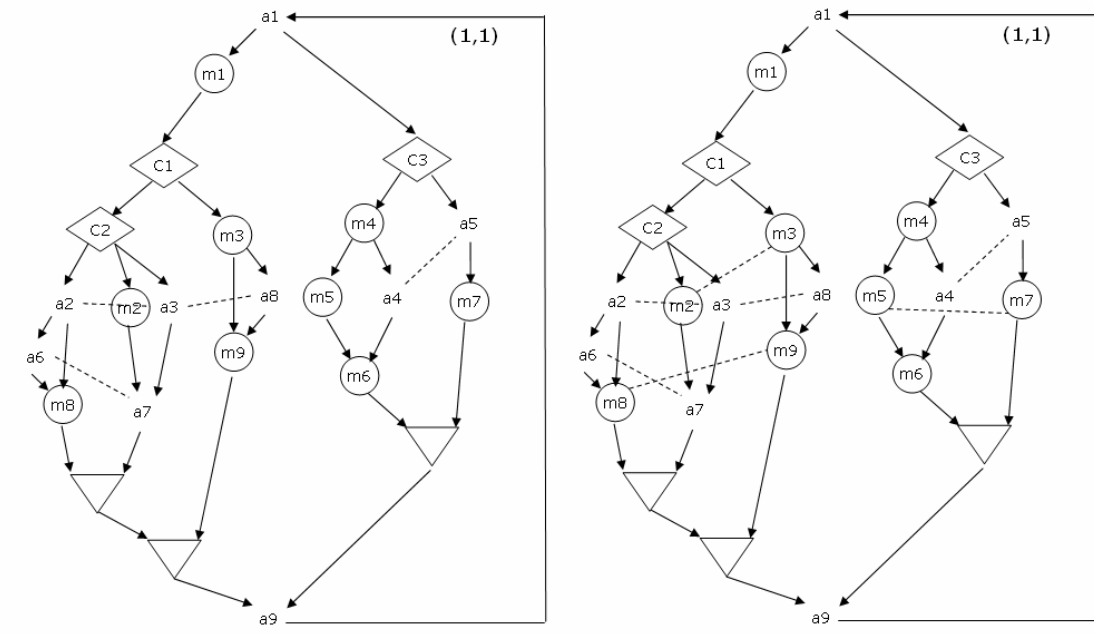


Fig 3.10 (a) After finding sharing policy among addition; (b) After finding all sharing policy.

Unconditional	Conditional
a1	c2 (0.2)
c1	a6/a7 (0.2)
c3	m2/m3 (0.88)
a2/a3/a8	m8/m9 (0.92)
a4/a5	m4 (0.8)
a9	m6 (0.8)
m1	
m5/m7	

Fig 3.11 Operations after finding resource sharing policy.

For example, after finding the resource sharing policy, we can find the number of unconditional and conditional operations as shown in Fig 3.11. The *MSL* and *NSL* can be computed by the related works and Definition 3.7. In this example, if we have two adders and one multiplier, then

C-Step	Adder	Adder	Mult
1	a1		
2	c3		m1
3	c1		
4			
5			
6			

(a)

C-Step	Adder	Adder	Mult
1	a1	a9	m2/m3
2	c3	a2/a3/a8	m1
3	c1	a4/a5	
4			
5			
6			

(b)

C-Step	Adder	Adder	Mult
1	a1	a9	m2/m3
2	c3	a2/a3/a8	m1
3	c1	a4/a5	m8/m9
4			
5			
6			

(c)

C-Step	Adder	Adder	Mult
1	a1	a9	m2/m3
2	c3	a2/a3/a8	m1
3	c1	a4/a5	m8/m9
4	c2	a6/a7	m2/m3
5			m4
6			A6

(d)

Fig 3.12 Scheduling processes for allocating operations

$$MSL = \max\{\frac{\#\,of\;\; operations}{\#\,of\;\; functional\; units}\} = max\{\;\frac{8}{2},\frac{6}{1}\;\} = 6$$ and

$$NSL = \max\{\frac{\#\,of\;\; unconditional\; operations}{\#\,of\;\; functional\; units}\} = max\{\;\frac{6}{2},\frac{2}{1}\;\} = 3\,.$$

The third step, we allocate all operations into schedule table which size is equal to the *MSL*. By second scheduling strategy, unconditional operations should be scheduled into minimum number of long instructions from C-step 0 to *NSL*. Fig 3.12 shows the schedule tables and the dotted line means the *NSL*. Algorithm for allocating operations is shown in Fig 3.13 which contains two phases. The first phase allocates all unconditional operations into schedule table from C-Step 1 to *NSL* by BUSM [3] to reduce the retiming depth. After first phase, there are some conditional operations to be scheduled. The second phase allocates conditional operations into schedule table by its executing probability. A conditional operation with higher probability is scheduled to the earlier C-Step to reduce the *PL* of later long instructions.

Now, we use the example to show how the algorithm works. Initially *a1* is in the *QueueV*

in the Line 4 and we can find an available FU in the C-Step 1 in the Line 9 to 11. We schedule *a1* into schedule table and decrease the indegree count of its successors to find schedulable operations. Then *m1* and *c3* are added into *QueueV* and we try to find an available functional unit for *m1* at C-Step from 2 to 4 and schedule it into C-Step 2.

After *a1, m1, c1, c3* are schedulable in Fig 3.12(a), we need to schedule *c2* but it's a conditional operation in the Line 19. We decrease its successors' indegree count but added *c2* into *QueueS*. Then the sharing operation *m2/m3* has zero indegree and is schedulable. There are not available functional unit from *ES(m2/m3)=3* to *NSL* and we find another one at C-Step 1 to *ES(m2/m3)-1* in the line 13 to 16. It's the idea of BUSM to reduce retiming depth. In the Fig 3.12(b), when *Queue* is empty, all unconditional operations are allocated into schedule table, there are some conditional operation to be scheduled in the *QueueS*.

We sort such conditional operations by its executing probability and allocate them into schedule table at C-Step from 1 to *MSL*. The highest probability operation is *m8/m9* and will be scheduled into table firstly in Fig 3.12(c). After allocating all operations, we get the schedule table as Fig 3.12(d). We can observe that the first three long instructions will always be executed, and the other three depend on the branch testing. By the definition of *ESL*, we can compute *ESL=3+PL(4)+PL(5)+PL(6)* as the average performance of our scheduling result. However, the retiming count of each operation is unknown, we finally use a algorithm to set the retiming count.

Input : MD-CdDFG G_S ,*MSL*,*NSL*

Output : An Allocated Schedule Table

1. $ES(\forall v \in V) \leftarrow 1$
2. $Queue \leftarrow v \in V$; $QueueV=QueueS=\varnothing$
3. $\forall e \in E$, $E \leftarrow E-\{e, \text{ s.t. } d(e) \neq (0...0)\}$
4. $QueueV \leftarrow QueueV \cup \{u \in V, \text{ s.t. } Indegree(u)=0\}$

```
5. While (Queue≠∅)
6.      DO      u←DeQueue(QueueV)
7.              IF (u is unconditional)
8.              THEN      for i=ES(u) to NSL
9.                          if (exist an empty FU for u)
10.                             ES(u)=i ;
11.                             Assign operation u to an available FU at C-Step i
12.                             Assign operation shared with u to an available FU at C-Step i
13.                         for i=1 to ES(u)-1
14.                           if (exist an empty resource for u)
15.                             ES(u)=i ; Assign node u to an available FU at C-Step i
16.                             Assign operation shared with u to an available FU at C-Step i
17.                         ∀v s.t. u→v
18.                           ES(v)←Max{ES(u),ES(u)+t(v)}
19.               ELSE      QueueS=QueueS∪{u , nodes shared with u}
20.               ∀v s.t. u→v
21.                   Indegree(v)=Indegree(v)-1
22.                   if Indegree(v and nodes shared with v)=0
23.                      QueueV=QueueV∪{v}
24. While (QueueS≠∅)
25.               u←DeQueue(QueueV)    s.t. u has the highest p(u) in QueueV
26.               for i =1 to MSL
27.                      if (exist an empty FU for u)
28.                            Assign operation u to an available FU at C-Step i
```

Fig 3.13   Step 3 of PLSM : allocating operations into schedule table

```
Input : MD-CdDFG and scheduled table
Output : retiming function of each node v∈V
1. ∀ u∈V RC(v)=null
2. RCmax=0
3. E ← E – { e, s.t. d(e)=(0...0) }
4. ∀ u∈V s.t. Indegree(u)=0    RC(u)=0
5. While (∀ u∈V, RC(u) ≠null)
6.      do   For CS=1 to MSL
7.                for each u s.t u is a operation in the control step
8.                     do   IF (RC(u)=null AND RC(all parents of u) ≠ null)
9.                          THEN    RC(u)=RCmax
10.                         IF (exist operation v, s.t. u, v are exclusuive)
11.                            AND RCmax>NSL
12.                            AND RC(v)=RCmax
13.                         Then     RC(u)=RC(u)+1
14.            RCmax=RCmax+1
15. Choose s=(s1,s2…sn) s.t. s•d(e ) > 0 for any e∈E whose d(e)≠(0,…,0)
16. Choose r s.t. r ⊥ s
17. for each u∈V
18.     do   r(u) = (RCmax – RC(u))*r
```

Fig 3.14 Step 4 of PLSM : Algorithm for setting retiming count

The final step is retiming count setting. We need to avoid the exclusive by Schedule Strategy 3. Nodes which do not violate the data dependence in the schedule table can use the same retiming count. Detail algorithm is shown in the Fig 3.14, and we use the allocated

C-Step	Adder	Adder	Mult
1	$a1^{0}$	a9	m5/m7
2	c3	a2/a3/a8	m1
3	c1	a4/a5	m8/m9
4	c2	a6/a7	m2/m3
5			m4
6			m6

(a)

C-Step	Adder	Adder	Mult
1	$a1^{0}$	a9	m5/m7
2	$c3^{0}$	a2/a3/a8	$m1^{0}$
3	$c1^{0}$	a4/a5	m8/m9
4	$c2^{0}$	a6/a7	m2/m3
5			$m4^{0}$
6			m6

(b)

C-Step	Adder	Adder	Mult
1	$a1^{0}$	a9	$m5/m7^{2}$
2	$c3^{0}$	$a2/a3/a8^{2}$	$m1^{0}$
3	$c1^{0}$	$a4/a5^{1}$	m8/m9
4	$c2^{0}$	$a6/a7^{2}$	$m2/m3^{1}$
5			$m4^{0}$
6			$m6^{2}$

(c)

C-Step	Adder	Adder	Mult
1	$a1^{0}$	$a9^{4}$	$m5/m7^{2}$
2	$c3^{0}$	$a2/a3/a8^{2}$	$m1^{0}$
3	$c1^{0}$	$a4/a5^{1}$	$m8/m9^{3}$
4	$c2^{0}$	$a6/a7^{2}$	$m2/m3^{1}$
5			$m4^{0}$
6			$m6^{2}$

(d)

Fig 3.15 Process for retiming count setting.

result in Fig 3.12 to show how retiming count setting works.

The process of retiming count setting is shown in Fig 3.15. The input of algorithm is the graph MD-CdDFG *G* and scheduled table. Initially we set the retiming count of each node as null in Line 1. We set the nodes without incoming edges as zero retiming count with retiming count 0 in the Line 4. In the Fig 3.15(a), only *a1* has zero retiming count initially. Then we find all nodes which satisfy zero retiming count in the schedule table from the Line 5 to 13. One operation can be set its retiming count if their parents are set the retiming count. For example, when we search the second C-Step and observe that all parents of *c2* have set the retiming count, we can set the retiming count of *c2* as zero. The first round from Line 6 to 9 set all operations which can apply zero retiming count and the result is shown in Fig 3.15(b).

After setting zero retiming count, we search the table twice to find nodes which can apply retiming count 1 and 2 in the Fig 3.15(c). From Line 10 to 13, we avoid schedule two exclusive operations into one long instruction by the Schedule Strategy 3. We increase the

retiming count of a later scheduled node. The while loop is finished until that all operations have its retiming count as Fig 3.15(d) . The maximum retiming count is recorded by *RCmax*. From the Line 15 to 18, we find the base retiming function by RPUSM [2] and compute the retiming function of each node $v \in V$.

So far, we have scheduled all operations into schedule table and set the retiming count. The example has shown how algorithm works. By property2.5, the scheduling result is a legal retiming to get a realize scheduling result. In the next chapter, we will use several benchmarks to evaluate the efficiency of our method.

Chapter 4. Preliminary Performance Evaluations

In this chapter, we will show some performance evaluations of the DSP program. First we will introduce the formula of evaluating the execution time for nested loops with conditional branch. Then, we will explain how the *ESL* affects the entire execution time. Secondly, we use some benchmarks to evaluate our method PLSM and MDBA in some detail. At the end, we will give some summary of time complexity and entire execution time between MDBA and our method PLSM.

4.1 Evaluating the Execution Time

In DSP applications, most nested loops are 2-dimensional loops. Thus, we use four benchmarks to evaluate the performance of PLSM and they are all 2-dimensional MD-CdDFG. In the following, we introduce the formula to calculate the execution time of 2-dimensional loops whose indexes are *m* and *n*. Before applying the retiming technique to a 2-dimensional loop, its execution time can be represented by $m \times n \times A$, where *A* is the static schedule length.

After applying the retiming technique, the execution time of a 2-dimensional loop can be divided into three parts, the loop body, the prologue and epilogue inside the first level loop, and the prologue and epilogue are out of the nested loop[2-3]. The formula of evaluating execution time of a 2-dimensional loop is shown as follows [2]:

$$A(m - s_2 \times d)(n - s_1 \times d) + (B + C)(s_1 \times m + s_2 \times n - s_1 \times s_2 - 2 \times d \times s_1 \times s_2) + D \times s_1 \times s_2 \times d(d + 1) \ldots(7)$$

, where (s_1,s_2) is the schedule vector, *d* is the maximum retiming depth, *A* is the average schedule length after applying some algorithm for optimization, *D* is the static schedule length of an iteration after applying "List Scheduling", *B* is the length of prologue inside the first

level loop, and *C* is the length of epilogue inside the first level loop. Following, we use the formula to compare the performance of PLSM and MDBA. We both use average schedule length in *A* to calculate their entire execution time.

4.2 Experiments Results

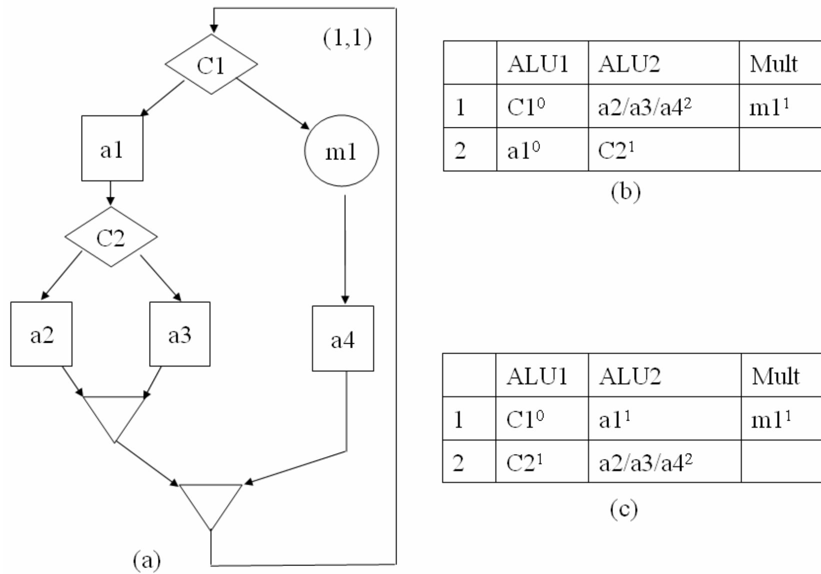
We use some benchmarks to evaluate the effects of MDBA and PLSM. They are all nested loops with conditional branches. We use (7) to compute their entire execution time, including loop bodies and overheads. The four benchmarks are the Floyd-Steinberg [4], VerySmall [6], SC [4,6], Kim [4,6.11,14] and VeryLarge[11,14] are shown in the Appendix A. These benchmarks are all 2-dimensional loops with conditional branches. We change the behavior of branch instructions and number of functional units to get different results which including *MSL*, *NSL*, *ESL* and retiming depth by our method. We also use MDBA to schedule such benchmarks and get *MSL* and retiming depth.

In the comparison of entire execution time, we use formula (7) to compute the execution time of scheduling result. Because the execution time is various, we show the maximum, minimal and average execution times. We change the size of nested loop to observe its influence of loop size on the entire execution time.

Fig 4.1(a) is the benchmark “VerySmall”, we assume that there are one adder and two multipliers in our architecture and the parameters are *f(c1)=0.2* and *f(c2)=0.8.* We select a best schedule vector (1,0) by RPUSM [2] and (0,1) as retiming base. The PLSM’s scheduling result is shown in Fig 4.1(b) and MDBA’s scheduling result is shown in Fig 4.1(c). Both the retiming depth of PLSM and MDBA are 2. After scheduling, we can compute the *ESL* are *1.36* in the PLSM and *2* in the MDBA by our average schedule length measurement.

We compute the maximum, average and minimum execution times by (7) in a 2-dimensional nested loop. Fig 4.2 shows that such two scheduling results are executed on a

5×5 nested loop. In this graph, the left three bars mean the maximum, average and minimum execution time produced by MDBA in the Fig 4.1(c). Each bar contains tow part, overhead and loop body. The overhead means instructions including prologues and epilogues beside loop body. The loop body means the instructions executed by nested loop in a retiming problem. Integral numbers in the central of each bar means the execution time of overhead or loop body. Because PLSM and MDBA cause the same retiming depth, the overhead both need 20 cycles to complete.



	ALU1	ALU2	Mult
1	$C1^0$	$a2/a3/a4^2$	$m1^1$
2	$a1^0$	$C2^1$	

(b)

	ALU1	ALU2	Mult
1	$C1^0$	$a1^1$	$m1^1$
2	$C2^1$	$a2/a3/a4^2$	

(c)

Fig 4.1 (a) Benchmark “VerySmall”; (b) Scheduling result by PLSM; (c) Scheduling Result of MDBA

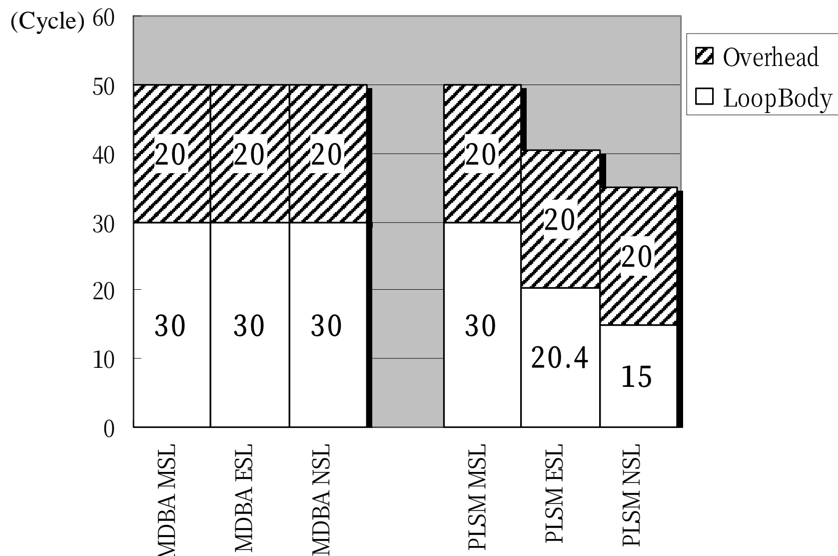


Fig 4.2 Maximum, average and minimum execution time (cycle) of the Fig 4.1(b)(c) in a 5x5 nested loop

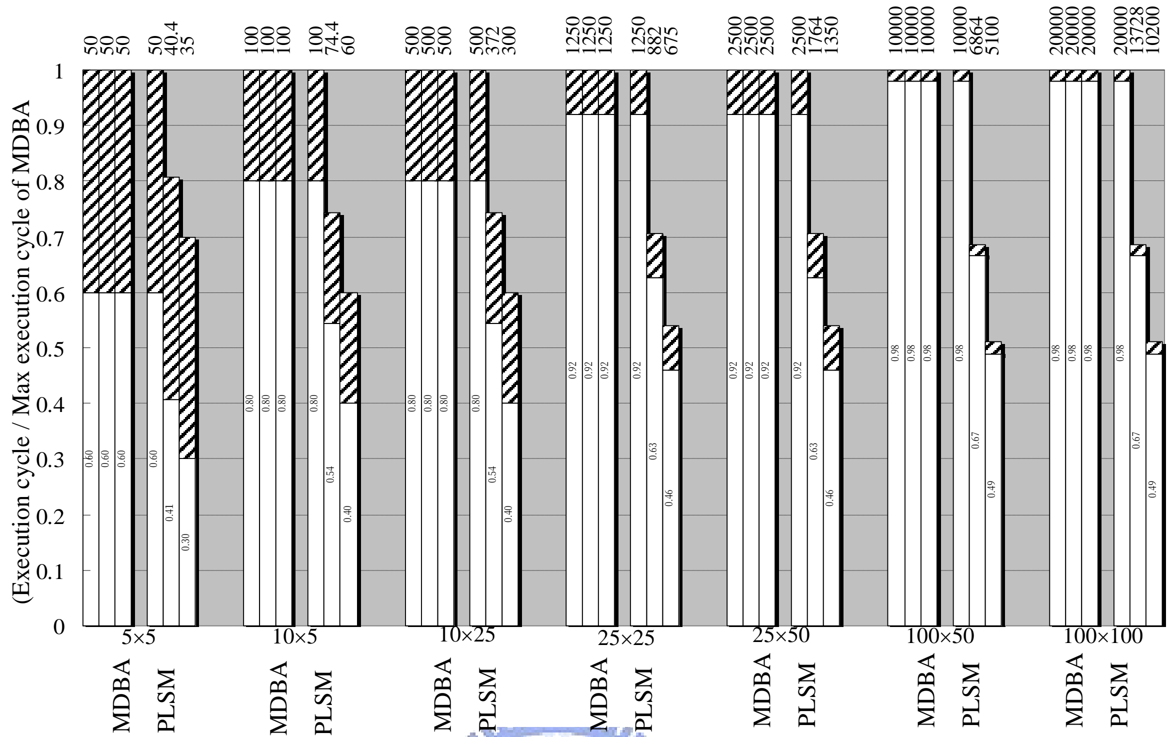


Fig 4.3 Execution times in different size of nested loop

In both scheduling results, we use the same retiming function and get the same retiming depth. We assume that prologues and epilogues use the same execution time. There are both 15 iterations in the loop body in such two scheduling results. In the MDBA, the schedule length is always 2, we always need 2×15=30 cycles to complete the loop body.

In our PLSM, the second long instruction is not always executed. In the worst case, if the second long instructions are executed in the every iteration, the execution time of loop body reaches to the maximum value 30, equal to the MDBA. However, if all second long instructions are not executed in the loop body, the best case need only 15 cycles to complete loop body. When the loop body is larger, maximum and minimum execution time is very small possibility to achieve. The entire execution time is usually close to the average value.

We use different size of nested loops to evaluate the effect from size and shape of loop body. Fig 4.3 shows the experiment results, the x-axis means the different size and shape of

	1ALU,1MULT						2ALU,1MULT						2ALU, 2MULT					
	MDBA			PLSM			MDBA			PLSM			MDBA			PLSM		
	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd
f(c1)=0.2 f(c2)=0.8	4	2.4	1	4	2.4	1	**2**	**2**	**2**	**2**	**1.36**	**2**	2	2	2	2	1.36	2
f(c1)=0.5 f(c2)=0.5	4	3	1	4	3	1	2	2	2	2	1.75	2	2	2	2	2	1.75	2
f(c1)=0.8 f(c2)=0.8	4	3.6	1	4	3.6	1	2	2	2	2	1.96	2	2	2	2	2	1.96	2
f(c1)=0.8 f(c2)=0.2	4	3.6	1	4	3.6	1	2	2	2	2	1.96	2	2	2	2	2	1.96	2

Fig 4.4 Experiments of "VerySmall" in different functional units and branch behaviors

loop body. Same as Fig 4.2, we also use six bars to present the maximum, average, minimum execution time of MDBA and PLSM for every type of loop body. The y-axis is a ratio over the maximum execution time of MDBA in each group. The floating point in each bar means the ratio of loop body over the entire execution time. In the top of graph are the entire execution times of each bar.

In the Fig 4.3, we observe the shape of loop body is less effect on the entire execution time. The improvement on the average execution time is more obvious when loop size is larger. We change the functional units and behavior of branch instructions to evaluate MDBA and our PLSM. Fig 4.4 is the scheduling result of "VerySmall" under variant type of functional units and branch behaviors. The high-lighting part is the scheduling result in Fig 4.1 which is a maximum improvement case. In other cases, if we can not get some improvement in *ESL,* we can assure that MSL of PLSM is equal MDBA and the entire execution is the same.

Fig 4.5 is another benchmark "Floyd-Stienberg"[4]. Its characteristic is a small conditional block and less conditional operations. It means that *MSL*, *NSL* and *ESL* are so closed, and we hardly to get a conditional long instruction to save entire execution time. In other words, such type of program contains numerous unconditional operations. Because we use BUSM [3] to allocate unconditional operations into unconditional long instruction, we may reduce some retiming depth here. We also use variant branch behaviors and functional

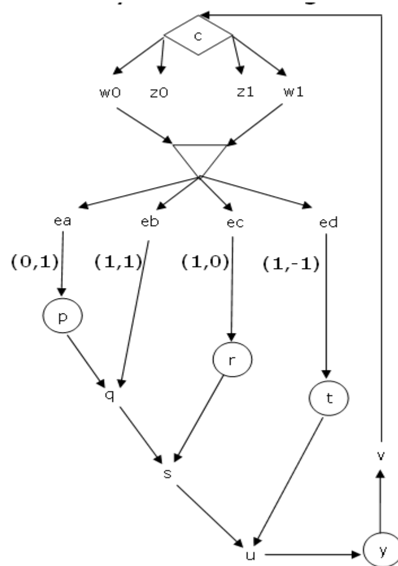


Fig 4.5 Benchmark "Floyd-Stienberg"

	1ALU,1MULT						2ALU,1MULT						2ALU, 2MULT					
	MDBA			PLSM			MDBA			PLSM			MDBA			PLSM		
	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd
f(c)=0.2	11	11	2	11	11	1	**6**	**6**	**3**	**6**	**6**	**2**	6	6	4	6	6	1
f(c)=0.5	11	11	2	11	11	1	6	6	3	6	6	2	6	6	4	6	6	1
f(c)=0.8	11	11	2	11	11	1	6	6	3	6	6	2	6	6	4	6	6	1

Fig 4.6 Experiment Results of "Floyd-Stienberg"

units to compute its execution. The experimental results are shown in Fig 4.6, and we use the high-lighting part to do experiments on different size of loop size in Fig 4.7. In the Fig 4.7, we observe some improvement by reducing retiming count. However the ratio of such improvement is dim with the increase of loop body. Retiming depth reduction decrease the ratio of overhead in the execution time, but the instructions of overhead is almost fixed. Improvements in the overhead will diluted with the loop body in any scheduling method.

We also use the benchmark "SC" and "Kim" to run experiments and the results are shown in Fig 4.8 and 4.9. In the benchmark "SC" and "Kim" are not extreme case of conditional block. We can get scheduling result with lower average schedule length.

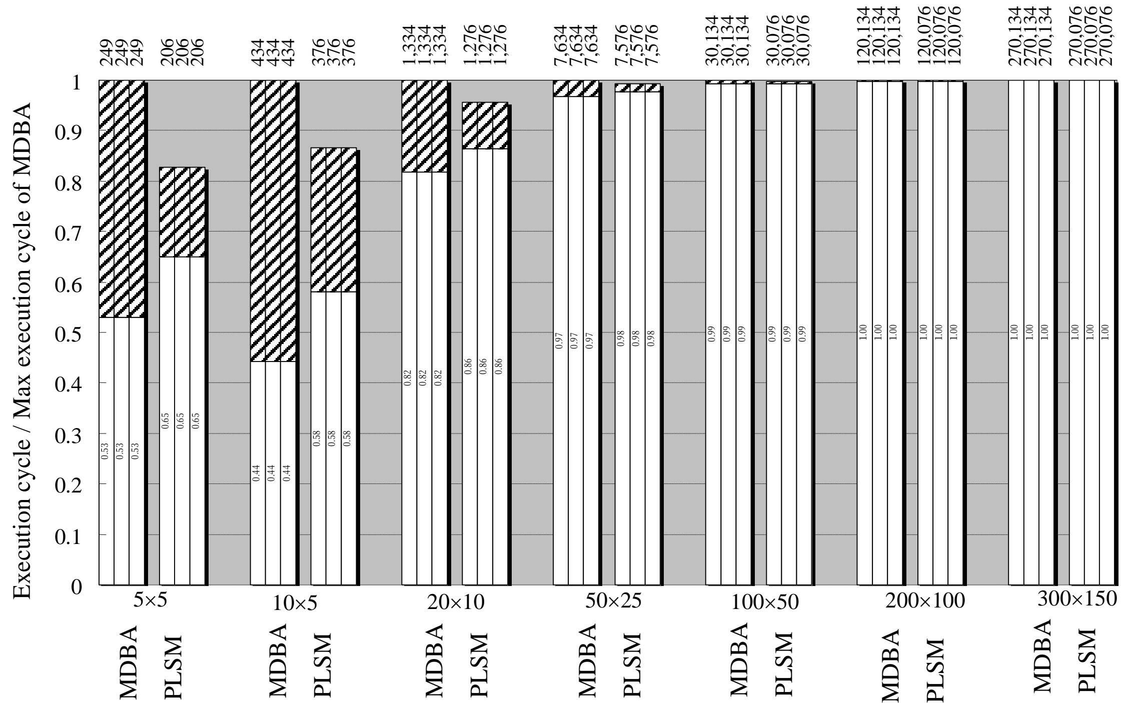


Fig 4.7 Experiments of "Floyd-Stienberg" in different functional units and

And in the final benchmark "VeryLarge" as shown in Appendix A, When there are one adder multiplier and divider, we can use PLSM to get a maximum schedule length 12, better than 15 by MDBA. Because the maximum is the upper bound of schedule length, we need not compare other measurements. The reason is that we find the resource sharing policy before allocate nodes. We can assign more operations to one functional unit and get a shorter maximum schedule length. In the next section, we will give summary of our method in time complexity and effectiveness.

	1ALU,1MULT						2ALU,1MULT						2ALU, 2MULT					
	MDBA			PLSM			MDBA			PLSM			MDBA			PLSM		
	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd
f(c1)=0.2 f(c2)=0.8 f(c3)=0.6	8	7.2	3	8	7.2	2	6	6	4	6	4.6	3	4	4	4	4	3.36	3
f(c1)=0.5 f(c2)=0.5 f(c3)=0.5	8	7.5	3	8	7.5	2	6	6	4	6	4.5	3	4	4	4	4	3.75	3
f(c1)=0.8 f(c2)=0.8 f(c3)=0.2	8	7.8	3	8	7.8	2	6	6	4	6	4.2	3	4	4	4	4	3.96	3
f(c1)=0.8 f(c2)=0.2 f(c3)=0.5	8	7.8	3	8	7.8	2	6	6	4	6	4.5	3	4	4	4	4	3.96	3

Fig 4.8 Experiment Results of “SC”

	1ALU,1MULT						2ALU,1MULT						2ALU, 2MULT					
	MDBA			PLSM			MDBA			PLSM			MDBA			PLSM		
	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd	MSL	ESL	rd
f(c1)=0.2 f(c2)=0.8	10	9.2	2	10 8	8.4	2	5	5	4	5 3	4.38	3	5	5	4	5 4	4.36	4
f(c1)=0.5 f(c2)=0.5	10	9.5	2	10 8	9	2	5	5	4	5 3	4.93	3	5	5	4	5 4	4.75	4
f(c1)=0.8 f(c2)=0.8	10	9.8	2	10 8	9.6	2	5	5	4	5 3	4.99	3	5	5	4	5 4	4.96	4
f(c1)=0.8 f(c2)=0.2	10	9.8	2	10 8	9.6	2	5	5	4	5 3	4.89	3	5	5	4	5 4	4.96	4

Fig 4.9 Experiment Results of “Kim”

4.3 Summary of Experiment Result

We give some summary the time complexity of our algorithm PLSM and MDBA. In MDBA, its time complexity is $O(|V|^2 + E)$. MDBA traverse the MD-CdDFG once in $O(|E|)$ and search the sharing policy in $O(|V|^2)$. In our algorithm PLSM, we have four steps to get a scheduling result. The first sets the executing probability in $O(|V|)$. The second step finds resource policy by checking the resource sharing conditionals for a pair of node in the MD-CdDFG. Its time complexity is $O(|V|^2)$. The third step allocates operations into schedule table, it traverse the graph once in the time complexity $O(|E|)$. And the final step sets the

retiming count of all operations in the schedule table. It need to check the parents' retiming count for each operation with time complexity $O(|E|)$. Thus, totally time complexity of PLSM is $O(|V|^2 + 2|E|)$, almost equal to MDBA.

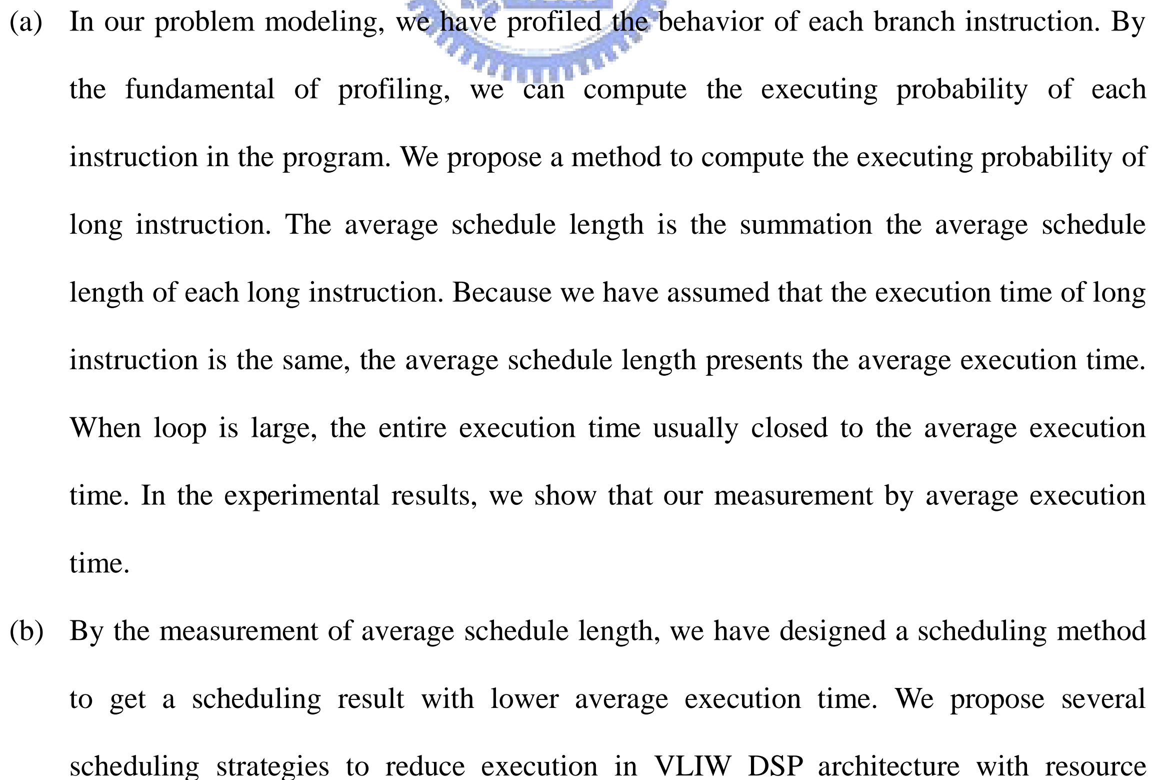
Secondly, we summary the efficiency on average schedule length of our method. It is different from the MDBA that we find the resource sharing policy before allocating operations into schedule table. It allows us to compute the maximum and minimum schedule length. Then we use three schedule strategies to achieve a minimum average schedule length. A low average schedule length can reduce the entire execution time and experimental results show the improvement of entire execution time. We can classify problems into tow cases. One is large conditional instructions and the other is small conditional instructions. When there are more conditional instructions, we can get a smaller average schedule length. On the contrary, when most instructions are unconditional, we are hard to get improvement from average schedule length. However, long instructions in this problem are almost unconditional. We can use the concept of BUSM to reduce the retiming depth. Because the average schedule length and retiming depth are trade-off, we improve one of them in the respected extreme cases.

Chapter 5. Conclusion and Future Work

In this thesis, we have designed an effective retiming based scheduling method PLSM to reduce the average performance. The experimental results have shown the effectiveness of PLSM and we will finally conclude our thesis and propose some future work for our research.

5.1 Conclusion

Because DSP processor becomes popular, high performance DSP used in such devices need to be processed with high data throughput. Applications in such systems become more complexity and cause numerous branch instructions. Static scheduling length can not present the inference which is caused by branch instructions. For this issue, we have proposed a method to evaluate the average performance. In summary, we give the following conclusions:



(a) In our problem modeling, we have profiled the behavior of each branch instruction. By the fundamental of profiling, we can compute the executing probability of each instruction in the program. We propose a method to compute the executing probability of long instruction. The average schedule length is the summation the average schedule length of each long instruction. Because we have assumed that the execution time of long instruction is the same, the average schedule length presents the average execution time. When loop is large, the entire execution time usually closed to the average execution time. In the experimental results, we show that our measurement by average execution time.

(b) By the measurement of average schedule length, we have designed a scheduling method to get a scheduling result with lower average execution time. We propose several scheduling strategies to reduce execution in VLIW DSP architecture with resource

sharing and resource constraining. After finding resource sharing policy, we can compute the maximum schedule length as the upper bound and the minimum schedule length as the lower bound. We propose a method to allocate nodes into schedule by a mixed method. Finally, we compare our algorithm with MDBA in the maximum, average, minimum schedule length and retiming depth. From the experimental results, we can conclude the retiming depth and average schedule length are trade-off. Experiments results have shown such extreme cases. In a problem with large conditional block, we can get a lower average execution time but cause high retiming depth. However, in a problem with small conditional block, average schedule length can not get large improvements in schedule length, but we can also reduce the entire execution time by lower retiming depth. In general cases, we can usually get a lower entire execution time.

5.2 Future Work

There are still some issues in the research that can be improved in the future.

(a) When we compute the average schedule length of loop body, we assume that happening of all long instructions are independent and compute the average schedule length by summing their executing probability. However, happening of long instruction may be dependent events. It means our measurement may overcount the entire execution time. In the future, we can analyze the relationship between long instructions' executing probability and get a more precisely.

(b) In our modeling of nested loop, all operations cost the same execution time. It may not respect the real case. In the future, we need to analyze the relationship between average schedule length and average execution time under considering different execution time of functional units. And, we use such measurement to develop a schedule method to get a shorter entire execution time.

(c) In modern microprocessors, branch prediction is usually used to enhance executing performance. However, we do not consider its efficiency to focus on branch behavior. In the future, we can also use branch prediction to evaluate a more precisely scheduling performance. It is possible to consider the efficiency of branch prediction into our scheduling method.

Bibliography

[1] N.L Passo and E.H.-M Sha, "Push-up scheduling: optimal polynomial-time resource constrained scheduling for multi-dimensional applications," *IEEE/ACM International Conference on Computer-Aided Design*, 1995. Digest of Technical Papers, pp. 588-591, Nov. 1995.

[2] M. L. Tsai, **A Study of Instruction Scheduling Techniques for VLIW-based DSP and Implementation of Its Simulation and Evaluation Environment,** Master Thesis, National Chiao-Tung University, Jun. 2001.

[3] M. C. Chen, **Instructions Scheduling with Less Power Consumption for nested Loop on DSP Architecture**, Master Thesis, National Chiao-Tung University, Jun. 2004.

[4] T. Z. Yu, E. H.-M. Sha, N. Passos, and R. D.C. Ju, "Algorithms and hardware support for branch anticipation," *In Great Lakes Symposium on VLSI*, pp. 163-168, 1997.

[5] Gupta, S.; Dutt, N.; Gupta, R. and Nicolau, A., "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," *Design, Automation and Test in Europe Conference and Exhibition, 2004*, Proceedings Volume 1, 16-20, pp. 114-119, Feb, 2004.

[6] Radivojevic, I.P and Brewer, F., **"**Analysis of conditional resource sharing using a guard-based control representation," *Computer Design: VLSI in Computers and Processors, 1995.*, IEEE International Conference on 2-4, pp. 434-439, Oct. 1995.

[7] Lakshminarayana, G.; Raghunathan, A. and Jha, N.K., **"**Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions," *Design Automation Conference, 1998.* Proceedings 15-19, pp. 108-113, Jun 1998.

[8] S. Tongsima, E.H.-M Sha, C. Chantrapornchai and D.R. Surma and N.L. Passos, "Probabilistic Loop Scheduling for Applications with Uncertain Execution Time," *IEEE Trans. Computers 49(1),* pp. 65-80, 2000.

[9] N.L. Passo and E.H.-M. Sha, "Achieving full parallelism using multidimensional retiming," *IEEE Transactions on Parallel and Distributed Systems*, Volume: 7 Issue:11, pp. 1150-1163, Nov. 1996.

[10] V. Kathail, M. Schlansker, and R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0" *Hewlett-Packard Laboratories Technical Report*, HPL-93-80, Feb 1993.

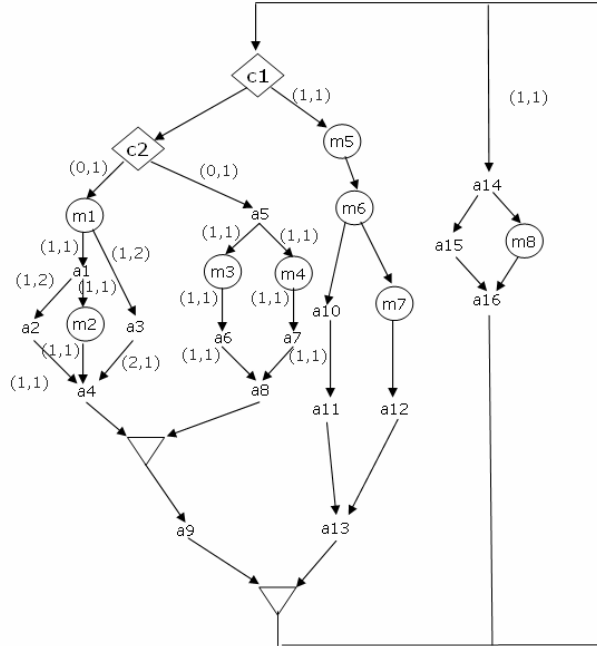
[11] T. Kim, N. Yonezawa, W.S. Liu, and C.L. Liu, "A Scheduling Algorithm for Conditional Resource Sharing ~ A Hierarchical Reduction Approach". *Computer-Aided Design of Integrated Circuits and Systems,* IEEE Transactions on Volume 13, Issue 4, 4. pp. 425-438, April 1994.

[12] J. Siddluwala and L. F. Cliao, "Scheduling Conditional Data-Flow Graphs with Resource Sharing", *Fifth Great Lakes Symposium 1995*, Proceedings on VLSI 16-18, pp 94-97, Mar. 1995.

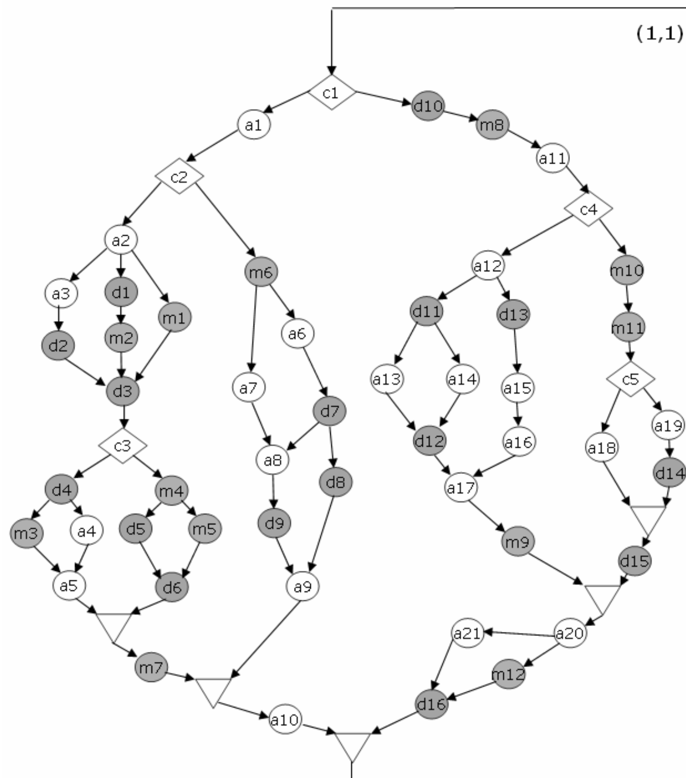
[13] Y. Xie and W. Wolf, "Allocation and scheduling of conditional task graph in hardware/software co-synthesis," *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001.* Proceedings 13-16, pp. 620-625, Mar. 2001.

[14] T. Kim, N. Yonezawa, J.W.S. Liu, and C.L. Liu, "A scheduling algorithm for conditional resource sharing," *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers. 1991*, IEEE International Conference on 11-14, pp. 84-87, Nov. 1991.

Appendix A



Performance Benchmark “Kim” [4,6,11,14]



Performance Benchmark “VeryLarge” [11,14]