國立交通大學

資訊工程學系

碩士論文

針對數位訊號處理器中的巢狀迴圈考慮功率消耗的指令排程方法

Instruction Scheduling with Less Power Consumption for Nested Loop on DSP Architecture

研 究 生：陳明志

指導教授：陳 正 教授

中華民國九十三年六月

針對數位訊號處理器中的巢狀迴圈考慮功率消耗的指令排程方法

Instruction Scheduling with Less Power Consumption for Nested Loop on DSP Architecture

研 究 生：陳明志　　Student: Ming-Chih Chen

指導教授：陳 正 教授　　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

Computer Science and Information Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年六月

針對數位訊號處理器中的巢狀迴圈考慮功率消耗的指令排程方法

研究生：陳明志　　　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

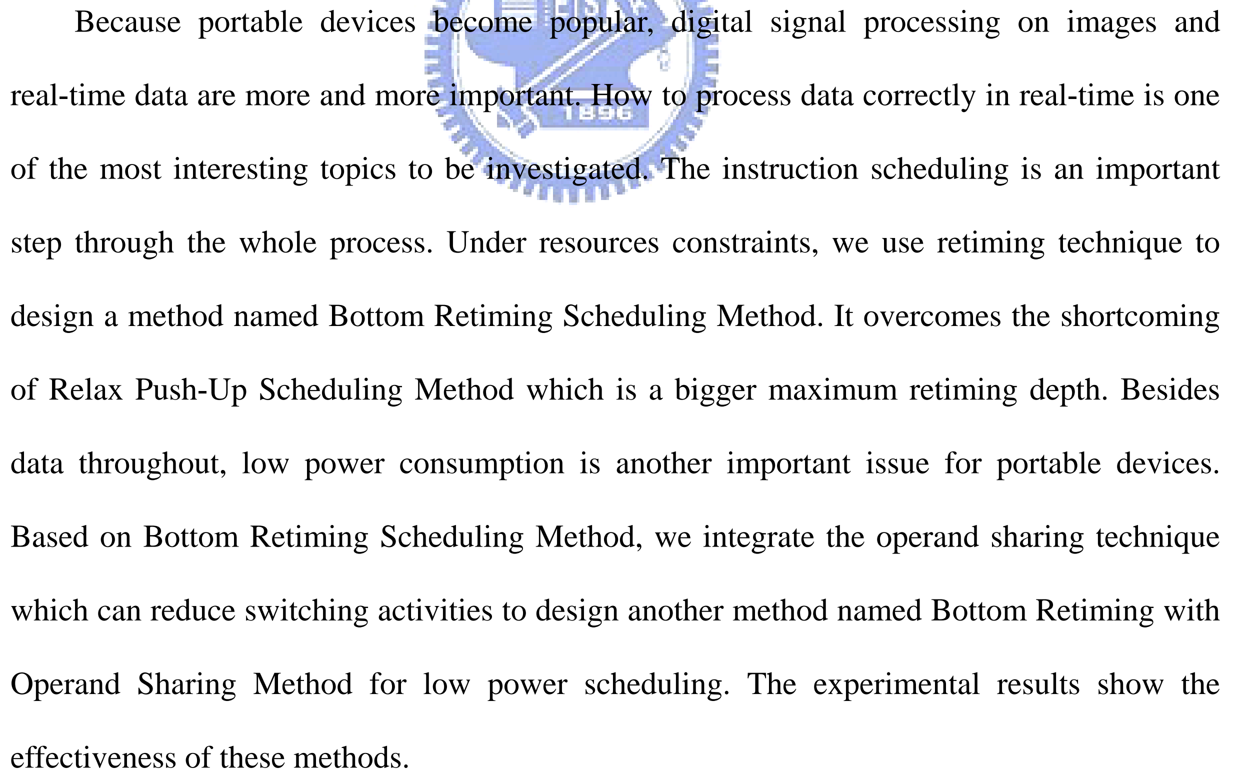
隨著個人攜帶式應用產品的普及，對於影音資料與即時性資料的需求日益增加，因此，數位訊號處理器扮演的角色也日趨重要。如何能使資料即時且正確的展現在使用者面前成了一個重要的課題，而指令的排程在整個過程中是一個很關鍵的步驟。我們利用 Retiming 的觀念，設計了一個在有限資源情況下的排程方法，名為 Bottom Retiming Scheduling Method，改善了 Relax Push-Up Scheduling Method 會造成較大 maximum retiming depth 的缺點。另外，由於可攜帶式的產品大都由電池供電，如何降低消耗功率以延長使用時間，亦是一個重要的課題；我們以 Bottom Retiming Scheduling Method 為基礎，加入了 operand sharing 可以減少 switching activities 的觀念，設計了一個降低功率消耗的指令排程方法 Bottom Retiming with Operand Sharing Method。由實驗結果可以看出，這兩個方法都可以達到不錯的效果。

Instruction Scheduling with Less Power Consumption for Nested Loop on DSP Architecture

Student: Ming-Chih Chen   Advisor: Prof. Cheng Chen

Institute of Computer Science and Information Engineering National Chiao Tung University

Abstract



Because portable devices become popular, digital signal processing on images and real-time data are more and more important. How to process data correctly in real-time is one of the most interesting topics to be investigated. The instruction scheduling is an important step through the whole process. Under resources constraints, we use retiming technique to design a method named Bottom Retiming Scheduling Method. It overcomes the shortcoming of Relax Push-Up Scheduling Method which is a bigger maximum retiming depth. Besides data throughout, low power consumption is another important issue for portable devices. Based on Bottom Retiming Scheduling Method, we integrate the operand sharing technique which can reduce switching activities to design another method named Bottom Retiming with Operand Sharing Method for low power scheduling. The experimental results show the effectiveness of these methods.

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 Dr. Guan-Joe Lai for their valuable suggestions.

There are many others whom I wish to thank. My thanks to Yi-Hsuan Lee for her kindly advice suggestion. Ming-Tien Chang, Chien-Wei Chen, Shun-Min Hsu, Wen-Pin Liu, Chia-Chun Lee and Wei-Fen Yang are delightful fellows, I felt happy and relaxed because of your presence.

Finally, I am grateful to my dearest family. I also send my sincere thanks to my best friends, Shuo-Zhan Ho and Jing-Yuan Lin. They accompany me all the time.

Table of Contents

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

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

Acknowledgements..iii

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

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

List of Tables..………………………………………………………….............viii

Chapter 1. Introduction……………………………………………………...........…..1

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

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

2.2 Retiming an MDFG…………………………………………………….................5

2.3 Related Work...9

2.3.1 Push-Up Scheduling Method...9

2.3.2 Relax Push-Up Scheduling Method...11

2.3.3 List Scheduling for Low Power...12

Chapter 3. Bottom Retiming Scheduling Method…………...13

3.1 Motivation……………………………………………………….............................13

3.2 Basic Concepts…………………………………………………….............................14

3.3 Bottom Retiming Scheduling Algorithm.....………………………….........…..17

3.4 Experimental Results...24

3.4.1 Evaluating the Execution Time...25

3.4.2 Performance Evaluation..25

Chapter 4. Bottom Retiming with Operand Sharing Method……………………..............30

4.1 Motivation…………………………………………………….................................30

4.2 Grouping Nodes……………………………………………….............................31

4.3 Bottom Retiming with Operand Sharing Algorithm...38

4.4 Experimental Results...40

4.4.1 Evaluating Operand Reutilizations...40

4.4.2 Preliminary Performance Evaluations..41

Chapter 5. Conclusion and Future Work……………………………………...............46

5.1 Conclusion……………………………………………………………………..............46

5.2 Future Work……………………………………………………………….............…47

Bibliography…………………………………………………………………..................48

Appendix A...52

List of Figures

Figure 2.1 High-level language code of a DSP program……………….............…………4

Figure 2.2 An equivalent two-dimensional data flow graph………………..……................4

Figure 2.3 Cell dependence graph……………………………………………............…..5

Figure 2.4 The retimed MDFG by retiming function *r*………………………………................6

Figure 2.5 Retimed cell dependence graph……………………………………............…7

Figure 2.6 An example of illegal retiming……………………………………….............9

Figure 3.1 Formula of ML…...………...……………..……………..………….................15

Figure 3.2 An partial schedule of an MDFG …..……………………………….................16

Figure 3.3 An MDFG modeling Floyd-Steimberg problem……………………............…..18

Figure 3.4 Scheduling information……………………………………………............…18

Figure 3.5 The algorithm of Assign function……………………………………..............19

Figure 3.6 An example of the multidimensional delay counting function……..................21

Figure 3.7 Bottom Retiming Scheduling Algorithm……………………...…..................23

Figure 3.8 The retiming function……………………………………………….................23

Figure 3.9 The retimed MDFG and scheduling sequences of BRSM………….................24

Figure 3.10 The execution time of benchmarks……………………………………...............28

Figure 4.1 An MDFG……………………..……………………………………….................32

Figure 4.2 The final schedule and an retimed MDFG…………………………….................32

Figure 4.3 The algorithm of Group……………………………………………….................34

Figure 4.4 Example for available continuous cycles……………………………..................34

Figure 4.5 The algorithm of Allocation…………..………………………………................35

Figure 4.6 The algorithm of BROS……………………………………………….................37

Figure 4.7 The information of scheduling nodes…………..……………………............…39

Figure 4.8 Scheduling sequences……………………..…………………………….............39

Figure 4.9 The retimed MDFG and the retiming function……………………….................39

Figure 4.10 Operand reutilizations and execution time of Floyd-Steimberg……................43

Figure 4.11 Operand reutilizations and execution time of 2-D Filter…………….............43

Figure 4.12 Operand reutilizations and execution time of IIR Section………….................44

Figure 4.13 Operand reutilizations and execution time of Transmission Line…................44
Figure 4.14 Operand reutilizations and execution time of DFT………………….............45

List of Tables

Table 3.1 The available number of functional units of every benchmark…………..…..............26

Table 3.2 The schedule vector and maximum retiming depth of every benchmark.................26

Table 4.1 The schedule vector and maximum retiming depth……………………….............41

Chapter 1. Introduction

In embedded system, high performance *Digital Signal Processing*（DSP）used in image processing, multimedia, wireless security, etc., needs to be processed not only with high data throughout but also with low power consumption [14]. These applications usually contain time-critical sections consisting of nested loops of instructions. The optimization of such loops, considering processing resource constraints, is required in order to improve their computational time [6].

Push-Up Scheduling Method (PUSM) [4] and *Relax Push-Up Scheduling Method* (RPUSM) [6] are *retiming*-based methods used to schedule instructions of nested loops under resources constraint. They can fully utilize functional units to achieve the minimum static schedule length, and RPUSM can further select a better schedule vector to reduce the entire execution time. However, they usually result in a bigger maximum retiming depth, which will longer prologue and epilogue and increase the entire execution time. Hence, in this thesis, we propose a method named *Bottom Retiming Scheduling Method* (BRSM) to overcome this shortcoming. In our BRSM, it can result in a smaller maximum retiming depth. From the experimental results, it shows that BRSM gives an improvement from 20.98% to 41.13% over the RPUSM in Floyd-Steimberg problem with various loop indexes [7].

As for low power scheduling, many techniques used for nested loops have been studied [2-6, 10-16, 19]. Based on *operand sharing approach*, a loop pipelining methodology to reduce both latency and power is first proposed in [13]. After that, a list-based loop pipelining technique is proposed to first minimize power and then maximize throughout [12]. Since list scheduling only considers the node with highest priority in ready list, it can't get an optimal solution when the number of functional units is more than one in every scheduling step. In order to fully utilize functional units and reduce power consumption, we integrate the operand sharing technique into BRSM to design another method, *Bottom Retiming with Operand*

Sharing method (BROS). It can bind operations with a common operand into the same functional unit and result in a smaller maximum retiming depth. BROS has advantages of BRSM and the operand sharing technique. From the experimental results, we can find that the performance of BROS is very close BRSM, and *operand reutilizations* are very high.

This thesis is organized as follows. In chapter 2, we will introduce the fundamental background and the related work. In chapter 3, BRSM is presented in detail, and the experimental results are shown. BROS is finely presented in chapter 4, and the corresponding experimental results are shown. 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 will introduce the *Multidimensional Data Flow Graph* (MDFG) to model the nested loop to be scheduled. Then, the retiming technique will be presented. Finally, we will go through some related work, including *Push-up Scheduling Method* [4], *Relax Push-up Scheduling Method* [6], and *List Scheduling for Low Power Method* [2].

2.1 Modeling the Problem [4-5]

Multidimensional data flow graph (MDFG) is used to model the nested loop to be scheduled [4, 6, 24-26]. Definition 2.1 defines what an MDFG is.

Definition 2.1. A *Multidimensional Data Flow Graph* (MDFG) $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, E is the set of dependence edges, d is a function from E to Z^n, representing the multidimensional delays between two nodes, where n is the number of dimensions, and t is the computation time of each node.

Fig. 2.1 shows the high-level language code of a DSP program. We use $d(e) = (d.x, d.y)$ to represent any delay edge e in a two-dimensional data flow graph. The equivalent two-dimensional data flow graph is shown in Fig. 2.2. In this thesis, we assume that execution time of any operation is one time unit.

An *iteration* is equivalent to the execution of each node in V of an MDFG G exactly once, i.e., the execution of one instance of the loop body [4]. An iteration is associated with a *static schedule*, that is repeatedly executed for the loop. Iterations are identified by a vector

```
for i = 1 to m
    for j = 1 to n
    {
        D:d(i,j)=b(i-1,j+1)*c(i-1,j-1);
        A: a(i,j)=d(i,j)*0.5;
        B: b(i,j)=a(i,j)+1;
        C: c(i,j)=a(i,j)+2;
    }
```

Fig. 2.1 High-level language code of a DSP program

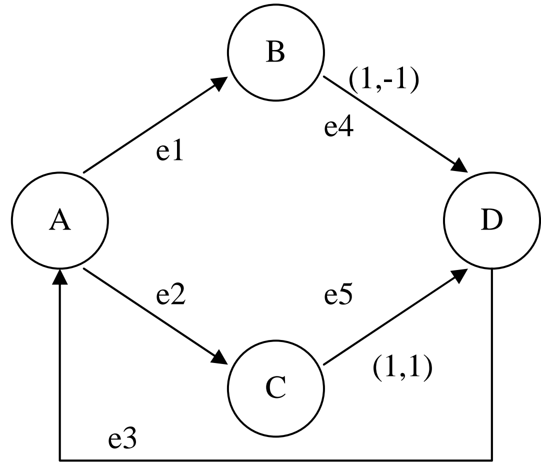


Fig. 2.2 An equivalent two-dimensional data flow graph

I, equivalent to a multidimensional index, starting from $(1,1,...,1)$. Inter-iteration dependencies are represented by vector-weighted edges in an MDFG. For any iteration *j*, in an MDFG an edge from node *u* to node *v* with delay vector $d(e)$ means that the computation of node *v* at iteration *j* depends on the execution of node *u* at iteration $j-d(e)$. An edge with delay $(0,0,...,0)$ in an MDFG represents a data dependence within the same iteration. A *legal MDFG* must have no zero-delay cycle, i.e., the summation of the delay vectors along any cycle can't be $(0,0,...,0)$.

Definition 2.2. A *cell dependence graph* (*DG*) of the MDFG *G* is the directed acyclic graph, showing the dependences between copies of nodes representing an MDFG *G*.

The cell dependence graph is bounded by the dimensions of the problem which it represents [5]. A *computational cell* is the DG node that represents a copy of the MDFG, excluded the edges with delay vectors different from $(0,0,...,0)$. The computational cell is considered as an atomic execution unit. Fig. 2.3(a) shows the DG based on the replication of

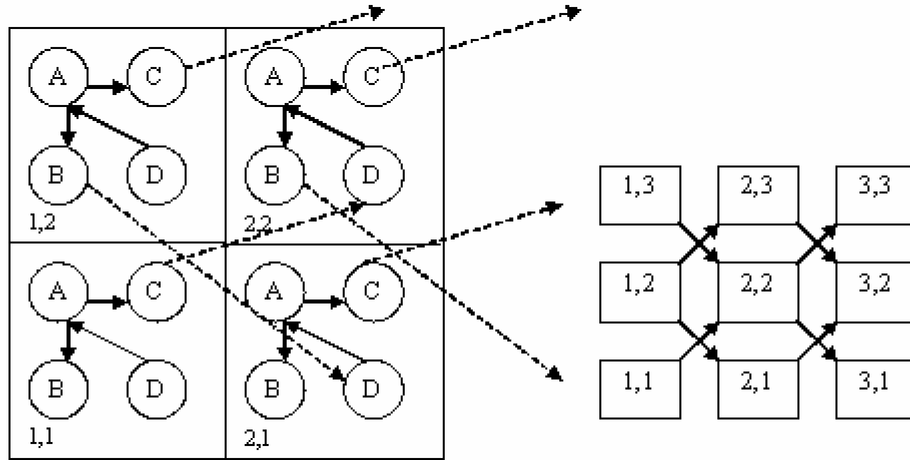


Fig. 2.3(a) DG based on the replication of the MDFG in Fig. 2.1

Fig. 2.3(b) DG represented by computational cells

the MDFG in Fig. 2.1, and Fig. 2.3(b) shows its DG represented by computational cells.

2.2 Retiming a Multidimensional Data Flow Graph [3-5]

A *multidimensional retiming* *r* is a function from *V* to Z^n that redistributes the nodes in the original dependence graph created by the replication of an MDFG $G=(V,E,d,t)$ [27]. A new MDFG $G_r=(V,E,d_r,t)$ is created after applying retiming function *r*, and each iteration still has one execution of each node in *G*. The purpose of using retiming technique is to construct a new MDFG with better instruction level parallelism (ILP). The retiming vector $r(u)$ of a node $u \in V$ represents the offset between the original iteration and the one after retiming. The delay vectors change accordingly to preserve dependencies. The retiming vector $r(u)$ of a node *u* represents delay components pushed into the edges $u \to v$, and subtracted from the edges $w \to u$, where $u, v, w \in V$. The execution of node *u* in iteration *i* which is represented by a multidimensional vector is moved to the iteration $i - r(u)$. Here we give some definitions and properties of the retiming technique as follows.

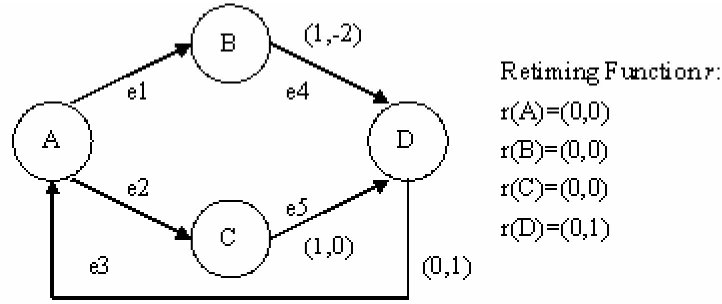


Fig. 2.4 The retimed MDFG by retiming function *r*

Definition 2.3. For any MDFG $G = (V, E, d, t)$, retiming function *r*, and retimed MDFG $G_r = (V, E, d_r, t)$, we define the retimed delay vector for every edge *e* in *E*, the retimed delay vector for every path in *G*, and the retimed delay vector for every cycle in *G*, denoted as $d_r(e)$, $d_r(p)$, $d_r(l)$ respectively by the following formulas:

(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 $u \xrightarrow{p} v$, $u, v \in V$ and $p \in G$;

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

For example, Fig. 2.4 shows the retimed MDFG G_r after applying retiming function *r* on G. We can use the definition to obtain the retimed delay vector for every edge *e* in *E*.

In Fig. 2.5(a), we show the retimed DG based on the replication of the MDFG in Fig. 2.4 and the retimed DG represented by computational cells is shown in Fig. 2.5(b). The retiming function applied to an MDFG 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. These two sets of instructions are complementary. For example, in Fig. 2.5(a) the instruction *D* becomes the prologue, and the instruction *A*, *B* and *C* become epilogue for this problem. If the retiming function of node D of the MDFG in Fig. 2.2 is equal to (0,2) and the

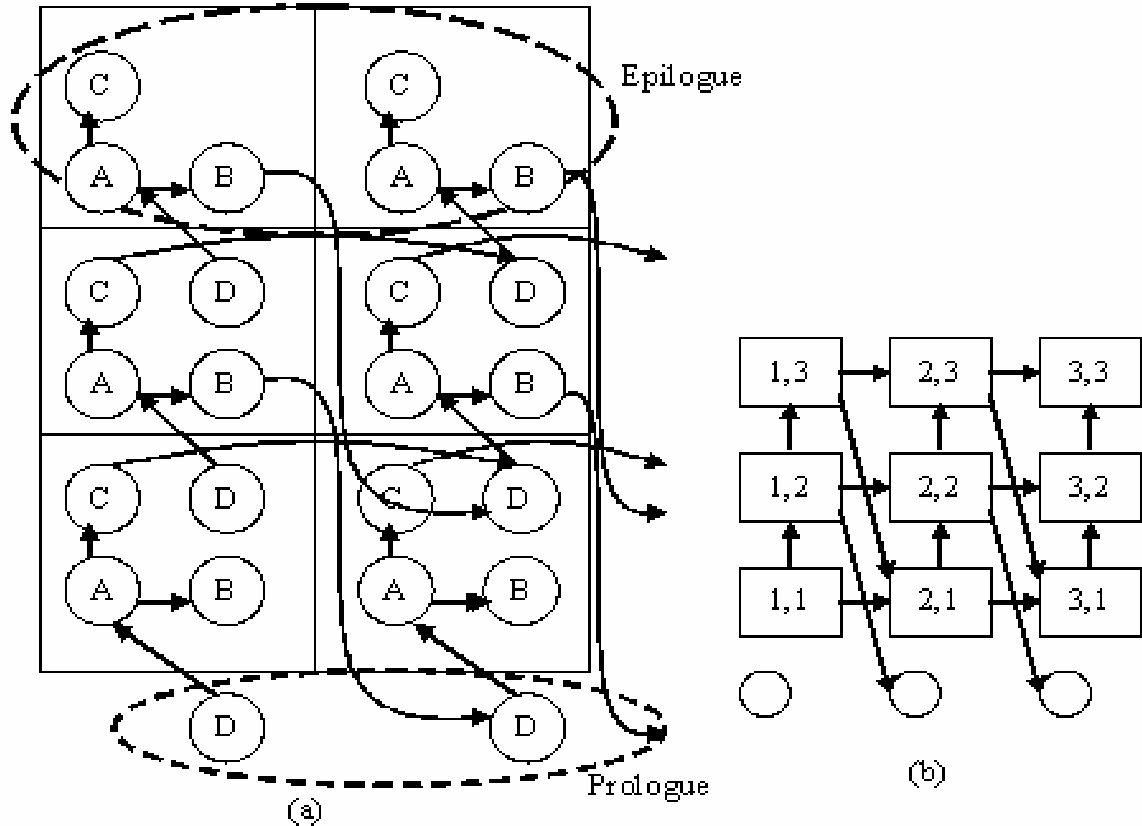


Fig. 2.5 (a) DG based on the replication of the MDFG, after retiming; (b) DG represented by computational cells

retiming function of other nodes is equal to (0,0), we can find that the instructions of the prologue and epilogue become more. So, the size of prologue and epilogue varies with the retiming function on the MDFGs.

A *schedule vector s* is the normal vector for a set of parallel equitemporal hyperplanes 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 $e \in E$ [5]. For example, (1,0) is a schedule vector of the MDFG in Fig. 2.2.

Definition 2.6. A legal MDFG $G = (V, E, d, t)$ that must 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 $d \in G$.

Definition 2.7. Given a realizable MDFG *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.

A legal multidimensional retiming on an MDFG $G=(V,E,d,t)$ 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.

The selection of retiming function may result in illegal retiming. Fig. 2.6(a) shows an illegal retiming function applied to the MDFG in Fig. 2.2. By simple inspection of the cell dependence graph in Fig. 2.6(b), we can find that there exists a cycle created by the dependencies (0,1) and (0,-1).

To get a legal multidimensional retiming *r*, we need to find a schedule vector *s*, such that $s \cdot d > 0$ for any $d \in G$. For a two-dimensional problem, we choose $s=(s.x, s.y)$ such that $s.x + s.y$ is minimum. Then, a legal multidimensional retiming *r* of node *u* is any vector orthogonal to *s* [5]. So, we can find that (0,1) is a legal retiming function on the MDFG in Fig. 2.2, and (1,0) is an illegal retiming function on the MDFG in Fig. 2.2. Further, we can have the corollary 2.1 [5].

Corollary 2.1. If *r* is a multidimensional retiming function orthogonal to a schedule vector *s* that realizes an MDFG $G=(V,E,d,t)$, and then $(k \times r)$ is also a legal multidimensional retiming on that MDFG.

From the corollary 2.1, 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*

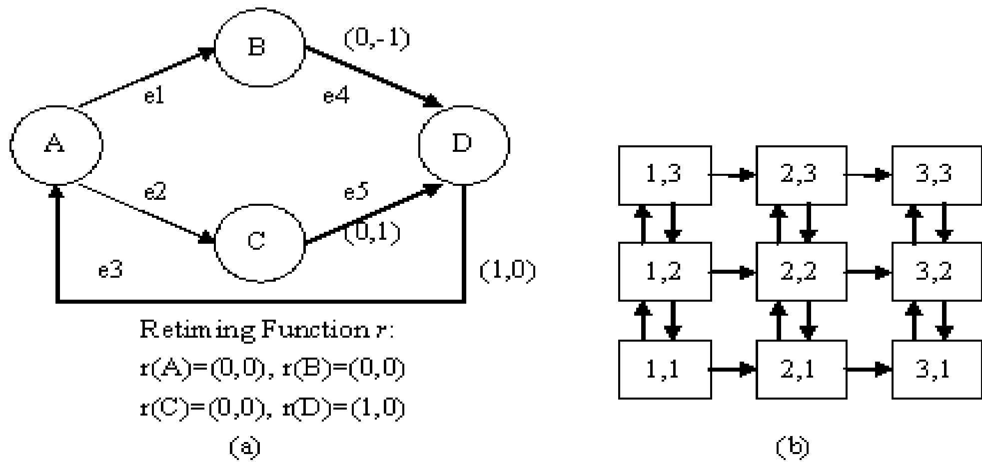


Fig 2.6 (a) Example of illegal retiming, (b) the corresponding DG

depth [6].

2.3 Related Work [1-2, 4, 6]

In this section, we'll show how *Push-Up Scheduling Method* (PUSM) works, and then *Relax Push-up Scheduling Method* (RPUSM) will also be introduced. Finally, we will introduce the *operand sharing technique* and a scheduling method, *List Scheduling for Low Power method* (LPLS), using list scheduling method combined with operand sharing technique [2].

2.3.1 Push-up Scheduling Method [4]

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 2.8. (Scheduling Conditions): Given an MDFG $G=(V,E,d,t)$ and a node $u \in V$,

u is a *schedulable node* at a control 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.

When scheduling an MDFG G by PUSM, it traverses G using BFS algorithm and checks that if the current traversing node satisfies the scheduling conditions or not. If the current traversing node satisfies the scheduling conditions, it will be scheduled in that control step. Otherwise, retiming technique will be used to make the node satisfy the scheduling conditions and be scheduled in that control step. During traversing G, every traversed node will record the multidimensional delay counting function $MC(u), u \in V$. $MC(u)$ represents the upper bound on the number of extra nonzero delays required by any path from roots of G to node u.

Before traversing the MDFG G, a schedule vector s realizing G and a legal retiming r on G will be found. After traversing G, PUSM uses multidimensional delay counting function MC to calculate the retiming function of every node by the following formula:

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

PUSM can promise to get schedule with a minimum static schedule length. But PUSM ignores the effect of the schedule vector and the retiming depth. Both of them affect the execution time of a scheduled nested loop. Following, RPUSM provides a method to select a better schedule vector to reduce the execution time.

2.3.2 Relax Push-up Scheduling Method [6]

One of the main shortcomings of PUSM is that it doesn't consider the effect of the schedule vector on the execution time. RPUSM finds that if the schedule vector could be kept as (1,0), the execution time is minimum as compared with other schedule vector different with

(1,0). The author also proposed a method to check if (1,0) could be a schedule vector, as shown in theorem 2.1.

Theorem 2.1. For an MDFG $G = (V, E, d, t)$, the retiming depth of any node u in V is *rd(u)*. We can use schedule vector $s = (1,0)$ and retiming base $r = (0,1)$ under one of the following two conditions which make the MDFG realizable:

(a) If there doesn't exist any delay vector $(0, a)$ for $a > 0$ in the original MDFG;

(b) If there exists the delay vectors $(0, a)$ for $a > 0$, and after finding out the retiming depth of every node in V we must make sure that $rd(u) + a > rd(v)$ for all $u \xrightarrow{(0,a)} v$ in the original MDFG;

Relax Push-up scheduling method (RPUSM) uses theorem 2.1 to modify PUSM to select a better schedule vector. The main difference is that RPUSM find the schedule vector and the retiming base after traversing an MDFG. Thus, RPUSM can use the multidimensional delay counting function MC obtained after traversing an MDFG to get the retiming depth of every node. RPUSM uses the retiming depth of every node to check if the conditions in theorem 2.1 are satisfied or not. If not, the same method of finding a schedule vector and a retiming base as PUSM will be performed.

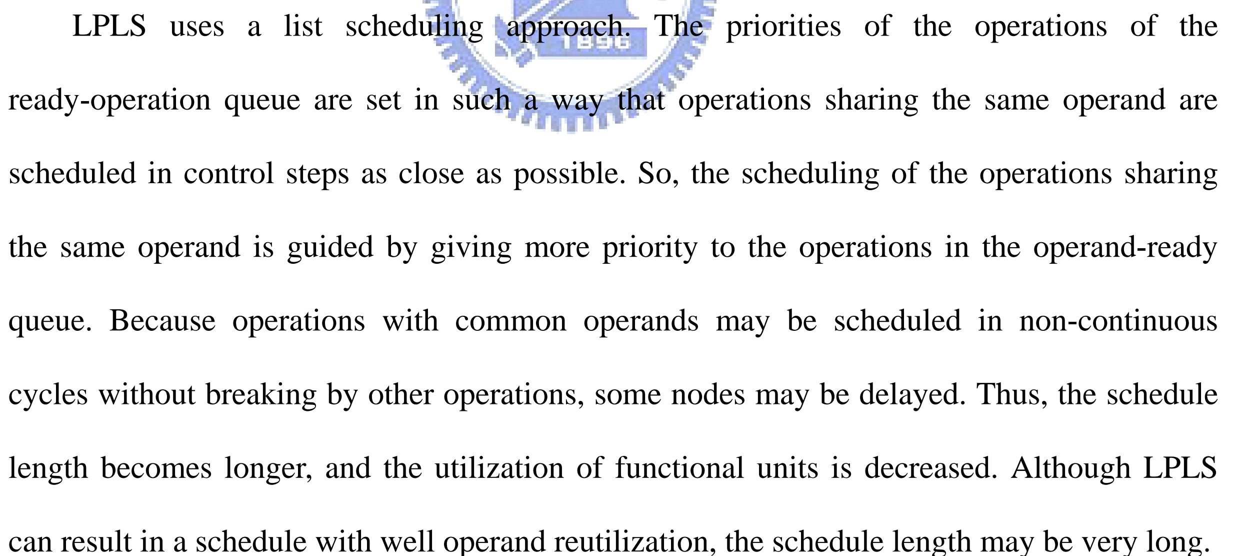
Although RPUSM provides a method to select (1,0) as a schedule vector, RPUSM, like PUSM, doesn't consider the effect of the retiming depth of nodes. We'll discuss this issue in chapter 3.

2.3.3 List Scheduling for Low Power Method [2]

There have been a few research results about power reduction using the operand sharing technique [1-2]. Here, we will briefly explain the operand sharing technique and then go

through the key feature of list scheduling for low power method (LPLS).

A functional unit in a data-path consumes both useful and useless power. It consumes useful power when it executes an operation and consumes useless power when there is an input operand transition while the functional unit is idle. The power consumption of a functional unit depends on the operand variability of its inputs. So, the operand sharing technique will try to bind operations with a common operand to the same functional unit such that the input activity of the shared functional unit decreases. In an MDFG, if a node has more than one outgoing edge, it reveals that the children of that node share the same data generated by that node. So, these children can be bind to the same functional unit in continuous cycles or non-continuous cycles without breaking by other operations. If two adjacent instructions executed on the same functional units have one common operand, it is called that one operand reutilization exists, or operand transition is reduced. One operand transition can reduce input activity of functional units.



LPLS uses a list scheduling approach. The priorities of the operations of the ready-operation queue are set in such a way that operations sharing the same operand are scheduled in control steps as close as possible. So, the scheduling of the operations sharing the same operand is guided by giving more priority to the operations in the operand-ready queue. Because operations with common operands may be scheduled in non-continuous cycles without breaking by other operations, some nodes may be delayed. Thus, the schedule length becomes longer, and the utilization of functional units is decreased. Although LPLS can result in a schedule with well operand reutilization, the schedule length may be very long.

In the chapter 3, we will present the bottom retiming scheduling method aiming at decreasing the retiming depth. In chapter 4, we will combine operand sharing technique with bottom retiming scheduling method to get a schedule with well operand reutilization and performance for reducing the power.

Chapter 3. Bottom Retiming Scheduling Method

In this chapter, we will finely introduce *Bottom Retiming Scheduling Method* (BRSM). First, we will explain our motivation to propose a method reducing the execution time of a nested loop. Then, we will describe the main concept and principle of BRSM. Finally, we will give some basic experimental results. From the results, we can find that BRSM produces the schedule of a nested loop with less execution time compared with RPUSM.

3.1 Motivation

From the related work, we know that in order to schedule nodes as early as possible, PUSM first analyzes the scheduling conditions (definition 2.8). If necessary, it uses the retiming technique to make instructions satisfying the scheduling conditions to be schedulable earlier. Although PUSM can achieve minimum static schedule length, effects of the schedule vector and retiming depth did not be considered. Both of them affect the execution time of applications. In [6], the author proposed a method, RPUSM, to get a better schedule vector to reduce the execution time. Different from PRUSM, we focus on effects of the retiming depth.

From Fig. 2.5, we find that some instructions become prologue and epilogue after retiming function is applied to an MDFG. Further, the prologue and epilogue will be longer while the retiming base is fixed and the *maximum retiming depth*, the maximum value of the retiming depth of all the nodes in a retimed MDFG, becomes bigger. That is to say, the optimized portion of the nested loop, the loop body, is decreased, and the un-optimized portion, prologue and epilogue, is increased. In order to reduce the execution time of nested loops, we have to increase the optimized portion of a nested loop. Thus, we need to decrease the maximum retiming depth.

By observing RPUSM, we find that in order to make nodes schedulable as early as

possible, many zero-delay edges will be changed to nonzero-delay edges. Thus, RPUSM will result in a static schedule with a bigger maximum retiming depth. In the following, we will introduce our BRSM to produce a static schedule with a smaller maximum retiming depth to reduce the execution time.

3.2 Basic Concept

In this section, basic concepts will be presented to explain how we decrease the maximum retiming depth under the minimum static schedule length with limited resources constraints. We describe how RPUSM works first. If *m* adders and *n* multipliers are available, RPUSM will schedule the first *m* add operations and the first *n* multiply operations in control step 1, the next *m* add operations and the next *n* multiply operations in control step 2,…until all operations are scheduled. If some node can't be scheduled in that control step, RPUSM uses the retiming technique to change the delay dependences to make it schedulable earlier. Although RPUSM can fully utilize functional units to achieve minimum static schedule length, it will produce a static schedule with a bigger maximum retiming depth. A bigger maximum retiming depth will result in a longer prologue and epilogue to increase the execution time. We will propose a new method which not only fully utilizes functional units to achieve minimum static schedule length but also has a smaller maximum retiming depth.

In order to fully utilize functional units, base on some information of architecture and applications, we can calculate the minimum static schedule length before scheduling. In a DSP application, it is usually composed of additions, multiplications, and assignments. Additions and multiplications are executed by adders and multipliers respectively. Assignments can be executed by adders or multipliers. To calculate the minimum static schedule length of some MDFG, denoted by $ML(G)$, some information is needed, the total number of adders(A), multipliers(M), additions(ADD), multiplications(MUL), and

1. $t = Max\{ADD/A, MUL/M\}$
2. $ML = Max\{t, t+[ADD+MUL+AS-t*(A+M)]/(A+M)\}$
3. $AA = ML \times A - AO$
4. $AM = ML \times M - MO$

t: a temporary data
ML: minimum static schedule length
A: the total number of adders
M: the total number of multipliers
ADD: the total number of additions
MUL: the total number of multiplications
AS: the total number of assignments
AA: the maximum number of assignments executed by adders
AM: the maximum number of assignments executed by multipliers

Fig. 3.1 Formula of $ML(G)$

assignments(AS). It costs ADD/A and MUL/M cycles to execute all additions and multiplications respectively. If no assignments exist, the bigger one of ADD/A and MUL/M is $ML(G)$. If assignments exist, we need to calculate that how many cycles it costs to execute them. The formula for calculating $ML(G)$ is shown in line 1 and line 2 of Fig. 3.1. Because assignments can be executed by adders or multipliers, we can calculate the maximum number of assignments executed by adders and multipliers, as shown in line 3 and line 4 of Fig. 3.1.

Generally speaking, in order to get a schedule with a smaller maximum retiming depth, we need to reduce the number of edges requiring additional multidimensional delays after scheduling. In other words, if there are fewer edges changing delay dependences of the MDFG after retiming, there are fewer edges requiring additional multidimensional delays after scheduling. Thus, we know we need to make fewer edges changing delay dependences. In order to reduce such edges, we retain the delay dependences to schedule as many nodes as possible. And in order to fully utilize functional units, nodes have to be scheduled before

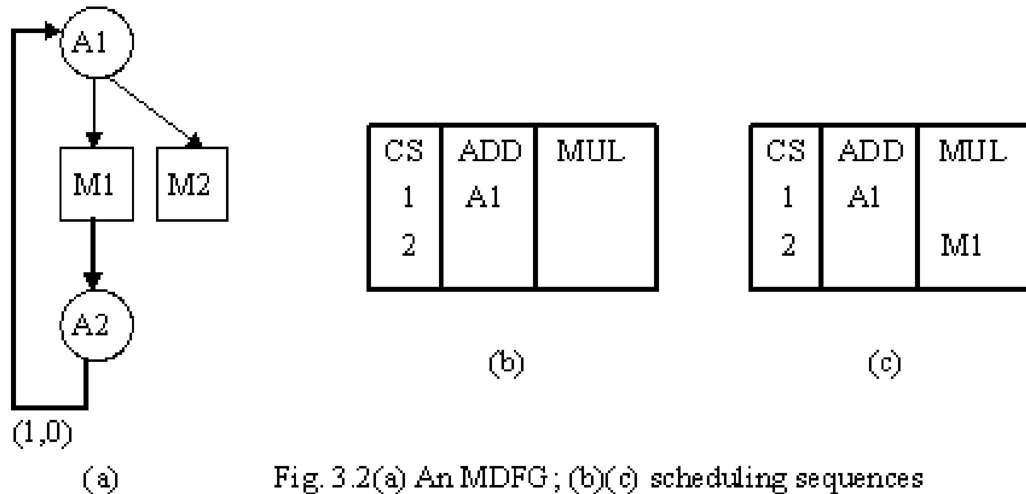


Fig. 3.2(a) An MDFG; (b)(c) scheduling sequences

minimum static schedule length. Under minimum static schedule length and resources constraints, when a node *u* is schedulable in control step *cs*, there are three conditions as follows:

(1) *cs* is smaller than minimum static schedule length, and there is an available functional units between control step *cs* and minimum static schedule length;

(2) *cs* is smaller than minimum static schedule length, and no functional units is available between control step *cs* and minimum static schedule length;

(3) *cs* is bigger than minimum static schedule length.

We give an example to explain these three conditions. In Fig. 3.2(a), we assume that one adder and multiplier are available. The minimum static schedule is two. In control step 1, node *A*1 is schedulable and one adder is available in that control step which is satisfying condition 1. We schedule *A*1 in control step 1, as shown in Fig. 3.2(b). Then, node *M*1 and *M*2 are schedulable in control step 2 and one multiplier is available. Assume that *M*1 are scheduled first, as shown in Fig. 3.2(c). We can find that *M*2 is schedulable in control step 2 but no available functional units in that control step which is satisfying condition 2. Node *A*2 is schedulable in control step 3 and the minimum static schedule length equals to 2 which is satisfying condition 3. From these three conditions and under minimum static schedule length constraints, we can know which conditions will result in additional multidimensional delays.

In condition 1, node u can be scheduled in some control step from cs to minimum static schedule length to retain the delay dependence. In other conditions, all incoming edges of node u require additional multidimensional delays to make u schedulable earlier. Based on above concepts, we will introduce Bottom Retiming Scheduling Algorithm in the next section.

3.3 Bottom Retiming Scheduling Algorithm

In this section, we will finely explain BRSM. Based on the above three conditions, we can use two functions, *ES*(u) and *Assign*(u), to decide when a multidimensional delay is needed. Given an MDFG $G = (V, E, d, t)$ and a node $u \in V$, the *earliest starting time* for the execution of node u, *ES*(u), is the first control step following the end of the execution of all predecessors of u by a zero-delay edge. It can be represented as:

$$ES(u) = Max\{1, ES(v_i\} + t(v_i)\}$$

for all v_i preceding u by an edge e_i such that $d(e_i) = (0,0,...,0)$.

From the definition of *ES*(u), we can know *ES*(u) is the earliest time that node u can start to be scheduled. For example, in Fig. 3.3, *ES*($M1$) = *ES*($M2$) = 1, and *ES*($A1$) = 2. Thus, *ES*($A2$) = 3.

In the following, we will introduce the function *Assign*(u) to record which control step node u assign to. Under minimum static schedule length and resources constraints, we would like to schedule node u in some control step to make incoming edges of u retaining their delay dependences. If that control step is smaller than minimum static schedule length and a functional unit is available in that control step, u will be scheduled in that control step. Thus, *Assign*(u) will equal to that control step. Or incoming edges of u will require additional multidimensional delays to make u schedulable earlier. Thus, *Assign*(u) will equal to an earlier control step in which a functional unit is available. *Assign*(u) can be determined as follows:

(a) When node u is schedulable and one functional unit is available between control step,

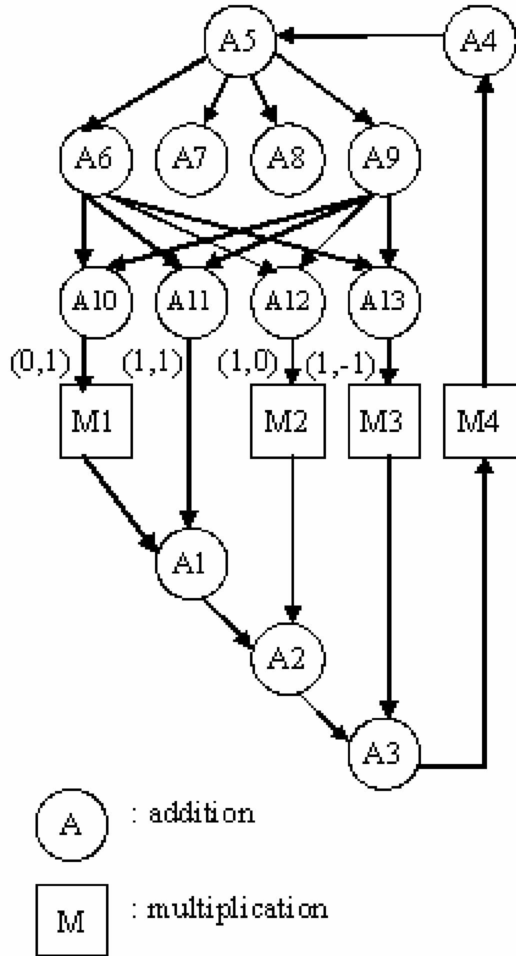


Fig. 3.3 An MDFG modeling Floyd-Steimberg problem

Node	Priority	MC(u)	Retiming Depth
A1	7	0	2
A2	6	0	2
A3	5	0	2
A4	3	1	1
A5	2	1	1
A6	1	1	1
A7	0	1	1
A8	0	1	1
A9	1	1	1
A10	0	1	1
A11	0	1	1
A12	0	1	1
A13	0	2	0
M1	8	0	2
M2	7	0	2
M3	6	0	2
M4	4	0	2

Fig. 3.4 Scheduling information

ES(*u*) and minimum static schedule length, *Assign*(*u*) equals to *Min* { *cs* , such that $ES(u) \le cs \le$ minimum static schedule length, and a functional unit is available in *cs*};

(b) When node *u* is schedulable and no function unit is available between control step *ES*(*u*) and minimum static schedule length, *Assign*(*u*) equals to *Min*{ *cs*, such that $1 \le cs \le ES(u) - 1$, and a functional unit is available in *cs*};

(c) When node *u* is schedulable and *ES*(*u*) is bigger than minimum static schedule length, *Assign*(*u*) equals to *Min*{ *cs*, such that $1 \le cs \le ES(u) - 1$, and a functional unit is available in *cs*}.

From those, we know that if possible, Assign(u) will record an control step which can retain

```
Algorithm Assign (node u)
1.   for  i = ES(u)  to ML(G)
2.        if u is an assignment
3.            if   AA > 0 and an adder is available in control step i
4.                AA--
5.                Return i
6.            elseif  AM > 0  and an adder is available in control step i
7.                AM--
8.                Return i
9.            endif
10.       elseif  the functional unit used by u is available in control step i
11.               Return i
12.       endif
13.  endfor
14.  for  i = 1  to ES(u)-1
15.       if u is an assignment
16.           if  AA > 0  and an adder is available in control step i
17.               AA--
18.               Return i
19.           elseif  AM > 0  and an adder is available in control step i
20.               AM--
21.               Return i
22.           endif
23.       elseif  the functional unit used by u is available in control step i
24.               Return i
25.       endif
26.  endfor
```

Fig. 3.5 The algorithm of Assign function

the delay dependences for node *u*, or Assign(u) will record an earlier control step to make *u* schedulable earlier. The algorithm of Assign is shown in Fig. 3.5. According to the definitions of *ES*(*u*) and *Assign*(*u*), we can determine when a multidimensional delay is needed by the following. If $ES(u) > Assign(u)$ for node *u*, all incoming edges of *u* require additional multidimensional delays, because *u* is assigned to the control step earlier than *ES*(*u*).

Inspecting from the function *Assign*, we find that the second and third condition result in

some edges requiring multidimensional delays. In order to avoid the second condition, more resources are required. In order to avoid the third condition, we have to avoid the earliest starting time of nodes being delayed. For example, in Fig. 3.3, node *M*1, *M*2, and *M*3 are schedulable in control step 1, and are scheduled in control step 3, 2, and 1 respectively. Thus, *ES*(*A*3) = 6. If *M*1, *M*2, and *M*3 are scheduled in control step 1, 2, and 3 respectively, *ES*(*A*3) = 4. In order to avoid *ES*(*u*) being delayed, we need to schedule nodes on critical path, the longest path, earlier. We will give every node a priority, and nodes on critical path are given higher priorities to be scheduled earlier. A BFS-like algorithm is used to assign priorities. The traversal direction is from leaves to roots. We can find that if the height of leaves is zero and the height of roots is highest, the nodes on longer path are at higher height. Thus, we prioritize nodes by their height. Priorities of all leaves equals to zero, and the priority of node *u*, *P*(*u*), is assigned by the following formula:

$$P(u) = Max\{P(u), P(v_i) + 1\}$$

for all v_i succeeding *u* by an edge e_i such that $d(e_i) = (0,0,...,0)$.

For example, after prioritizing nodes in Fig. 3.3, priorities are shown in Fig. 3.4.

We use two functions, *ES*(*u*) and *Assign*(*u*), to find which edges need for additional multidimensional delays. From the information, the retiming depth of every node can be found. The multidimensional delay counting function *MC*(*u*) is the upper bound on the number of extra nonzero delays required by any path from roots of *G* to node *u*. We can use it to compute the retiming depth of every node. If *w* precedes *u* by an edge requiring a multidimensional delay *MC*(*u*) = *Max*{*MC*(*u*),*MC*(*w*)+1}. Otherwise, *MC*(*u*) = *Max*{*MC*(*u*), *MC*(*w*)}. Fig. 3.6 is an example of computing the multidimensional delay counting function. Assume that edges $A \rightarrow B$, $D \rightarrow E$, $C \rightarrow F$, and $G \rightarrow H$ require multidimensional delays. After the function *MC*(*u*) is calculated, the retiming depth of every node *u* equals to $MC\max - MC(u)$, where *MCmax* is the maximum retiming depth of the retimed MDFG.

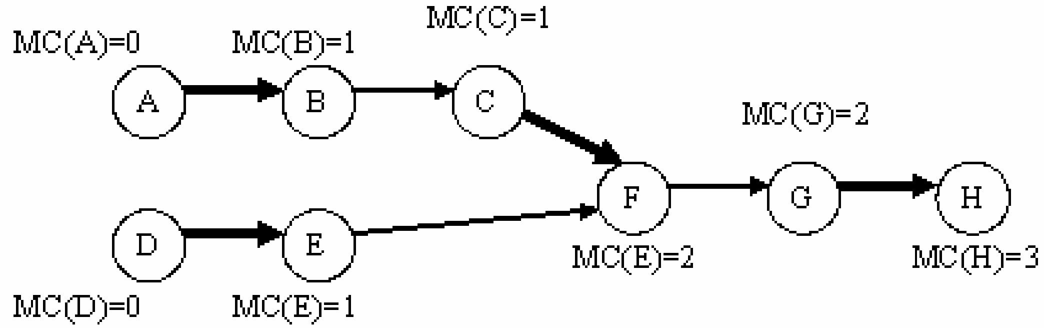


Fig. 3.6 An example of the multidimensional delay counting function

We use similar method proposed in [6] which uses the retiming depth of every nodes to check if (1,0) can be the schedule vector. It is shown in line 24 to line 33 of Fig. 3.6. From these lines, we know that if the condition $\forall u \xrightarrow{(0,a)} v, rd(u) + a > rd(v)$ is satisfied, (1,0) can be the schedule vector. For BRSM and RPUSM, $rd(u)$ equals to the maximum retiming depth. BRSM usually results in a smaller maximum retiming depth, so it has more chances to satisfy the condition and then select (1,0) as the schedule vector.

The algorithm of BRSM is shown in Fig. 3.7. Now, we give an example of the MDFG in Fig. 3.3 to show how BRSM works. We assume that three adders and a multiplier are available. The priority of every node is shown in Fig. 3.4, and the minimum static schedule length equals to five. We start to schedule nodes. In control step 1, *M1*, *M2* and *M3* are schedulable nodes and scheduled in control step 1, 2, and 3 respectively, as shown in Fig. 3.9(b). In control step 2, *A1* is schedulable. Although there is an available adder in control step 1, we don't schedule *A1* in that control step. Because we want to retain the delay dependences to schedule as many nodes as possible, *A1* is scheduled in control step 2, as shown in Fig. 3.9(c). Then, *A2*, *A3*, and *M4* are schedulable in control step 3, 4, and 5 respectively and scheduled in those control steps. *A4* is schedulable in control step 6, but $ML(G)$, equals to five. We need to make edge $M4 \rightarrow A4$ having a multidimensional delay to schedule *A4* earlier. Since edge $M4 \rightarrow A4$ has a multidimensional delay, *A4* can be

```
Algorithm Bottom Retiming Scheduling Method  (G = (V, E, d, t))
      /* traverse G by modified breadth first search algorithm */
1.    ES(∀u ∈ V) ← 0
2.    MC(∀u ∈ V) ← 0
3.    MCmax ← 0
4.    QueueV ← φ
      /* remove original edges with non-zero delays */
5.    ∀e ∈ E, E ← E − {e, s.t. d(e) ≠ (0,0,...,0)}
6.    QueueV ← QueueV ∪ {u ∈ V, s.t. INDEGREE(u) = 0}
7.    Prioritize_Nodes(G)
8.    Compute_Minimum_Static_Schedule_Length()
9.    while  QueueV ≠ φ
            /* fetch a node with highest priority in QueueV */
10.         GET(u, QueueV)
            /* check if u needs an extra non-zero delay in the incoming edges */
11.         if  Assign(u) < ES(u)
12.               MC(u) ← MC(u) + 1
13.               MCmax ← Max{MC(u), MCmax}
14.         endif
15.         ES(u) ← Assign(u)
16.         Assign node u to an available functional unit at control step ES(u)
            /* propagate the values to successor nodes of u */
17.         ∀v  such that  u → v
18.               INDEGREE(v) ← INDEGREE(v) − 1
19.               ES(v) ← Max{ES(v), ES(u) + t(u)}
20.               MC(v) ← Max{MC(v), MC(u)}
                  /* check for now schedulable nodes */
21.               if  INDEGREE(v) = 0
22.                     QueueV ← QueueV ∪ {v}
23.               endif
24.   endwhile
```

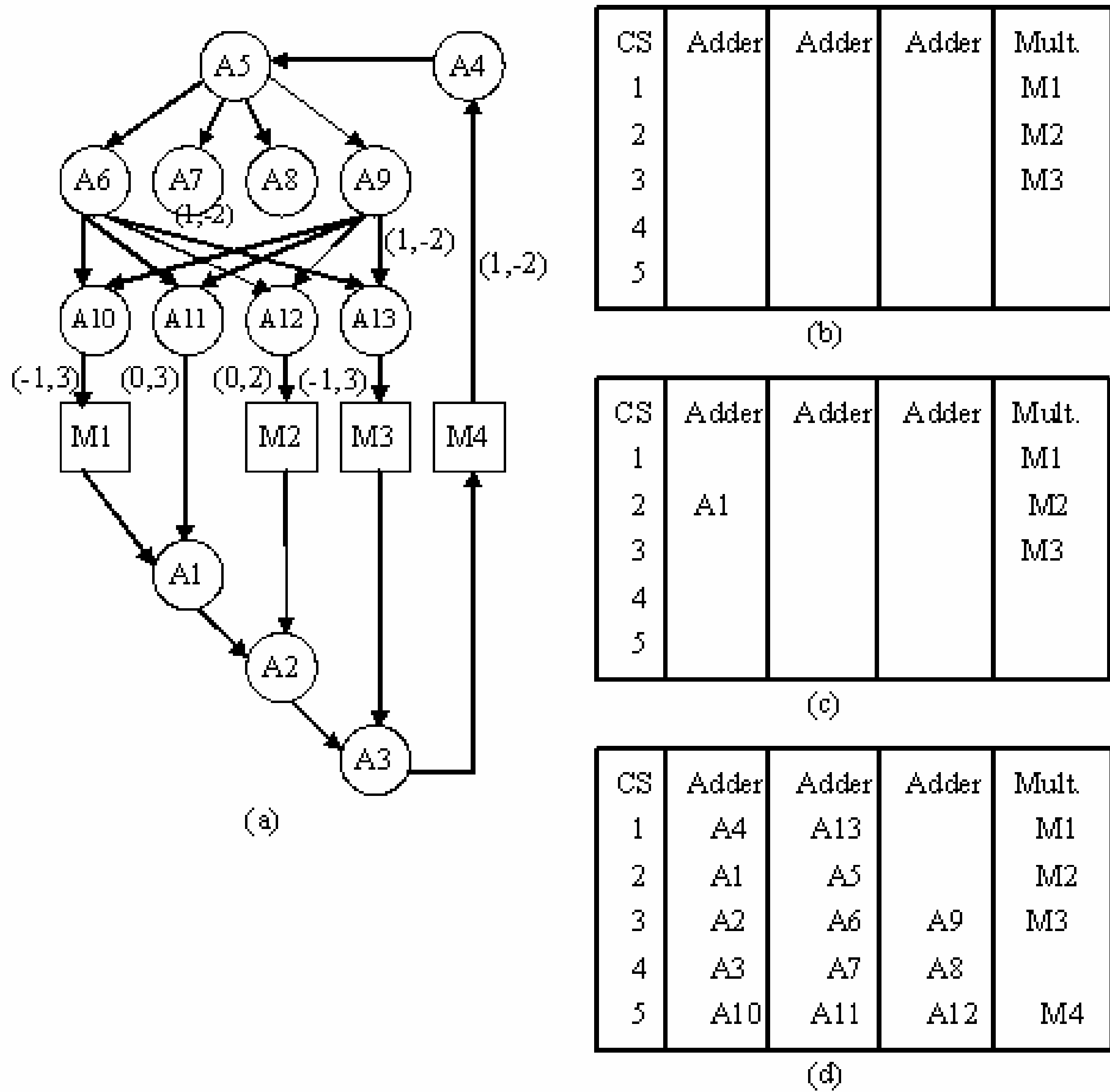
```
    /* get a schedule vector by the method of RPUSM */
1.  ∀u ∈ V, rd(u) ← MCmax − MC(u)
2.  if !∃d(e) = (0,a)  for any  e ∈ E
3.       Choose  s = (1,0)  such that  s·d > 0  for any  e ∈ E
4.  elseif  ∃d(e) = (0,a)  for any  e ∈ E
5.           if  ∀u —(0,a)→ v, rd(u) + a > rd(v)
6.                Choose  s = (1,0)  such that  s·d(e) > 0  for any  e ∈ E
7.           else
8.                Choose  s = (s1,s2)  such that  s·d(e) > 0  for any  e ∈ E
9.           endif
10. endif
11. Choose r  such that  r ⊥ s
    /* compute the multidimensional retiming */
12. ∀u ∈ V, r(u) ← rd(u) × r
```

Fig. 3.7 Bottom retiming scheduling algorithm

```
r(M1)=r(M2)=r(M3)=r(M4)=2 ×(1,-2)
r(A1)=r(A2)=r(A3)=2 ×(1,-2)
r(A4)=r(A5)=r(A6)=r(A7)=1 ×(1,-2)
r(A8)=r(A9)=r(A10)=1 ×(1,-2)
r(A11)=r(A12)=1 ×(1,-2)
r(A13)=0 ×(1,-2)
```

Fig. 3.8 The retiming function

scheduled in control step 1 and then *A5* is schedulable in control step 2. The final schedule is shown in Fig. 3.9(d). From the final schedule we know that besides edge $M4 \rightarrow A4$, edges $A6 \rightarrow A13$ and $A9 \rightarrow A13$ also need a multidimensional delay. The multidimensional delay counting function is shown in Fig. 3.4. Then, we find the schedule vector (2,1), and the retiming base (1,-2). Finally, the retiming depth and the retiming function are calculated, as shown in Fig. 3.4 and Fig. 3.8 respectively. The retimed MDFG is in Fig. 3.9(a). The maximum retiming depth equals to two. If this MDFG is scheduled by RPUSM or PUSM, the



CS	Adder	Adder	Adder	Mult.
1				M1
2				M2
3				M3
4				
5				

(b)

CS	Adder	Adder	Adder	Mult.
1				M1
2	A1			M2
3				M3
4				
5				

(c)

CS	Adder	Adder	Adder	Mult.
1	A4	A13		M1
2	A1	A5		M2
3	A2	A6	A9	M3
4	A3	A7	A8	
5	A10	A11	A12	M4

(d)

Fig. 3.9(a) The retimed MDFG; (b)(c)(d) scheduling sequences of BRSM

maximum retiming depth equals to six. In the next section, we use eight benchmarks to compare the performance of BROS and RPUSM.

3.4 Experimental Results

In this section, we will show some performance evaluations of the DSP benchmark. At first, we will introduce the formula of evaluating the execution time for nested loops and then compare the performance of RPUSH and BRSM. At the end, we will give an analysis of the comparison and conclude the advantage of BRSM.

3.4.1 Evaluating the Execution Time

In DSP applications, most nested loops are 2-dimensional loops. Thus, we use eight benchmarks to evaluate the performance of BRSM and they are all 2-dimensional DFG. In the following, we introduce the formula to execute 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 $m \times n \times D$, where D 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 out of the nested loop. The formula of evaluating execution time of a 2-dimensional loop is shown as follows [6]:

$$A(m - s_2 \times d)(n - s_1 \times d) + (B + C)(s_1 \times m + s_2 \times n - s_1 \times s_2 - 2 \times d \times s_1 \times s_2) + D \times s_1 \times s_2 \times d(d + 1) \quad \text{------ (1)}$$

, where (s_1,s_2) is the schedule vector, d is the maximum retiming depth, A is the static 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 BRSM and RPUSM.

3.4.2 Performance Evaluation

Here, we have used eight benchmarks, shown in Appendix A, to evaluate the effect between BRSM and RPUSM. Loop indexes vary from 10 to 50 with various combinations for each benchmark. The available number of functional units for each benchmark is shown in Table 3.1. The principle of determining the number of functional units is to balance the execution time of total additions and multiplications in an iteration. After scheduled by BRSM

Table 3.1 The available number of functional units of every benchmark

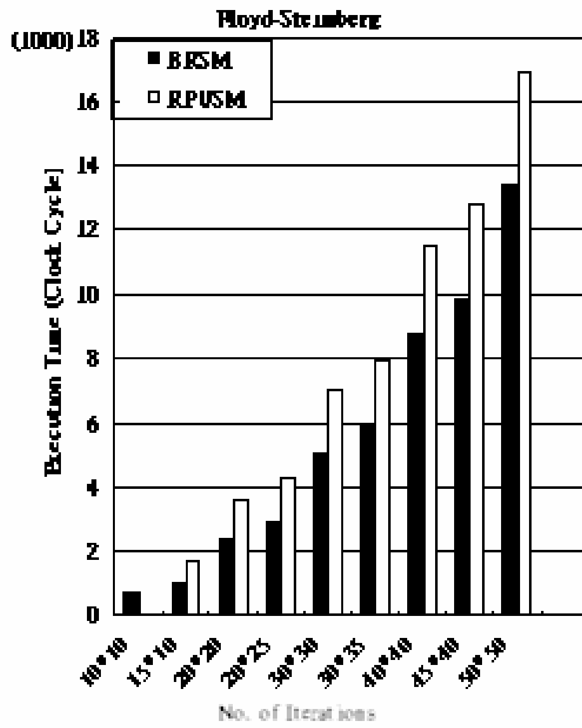
	Floyd-Steimberg	IIR Section	Transmission Line	IIR Filter	DFT	2-D Filter	Forward Substitution	THC Solver
Adder	3	2	2	2	2	2	1	1
Multiplier	1	2	1	2	1	2	1	1

Table 3.2 The schedule vector and maximum retiming depth of every benchmark

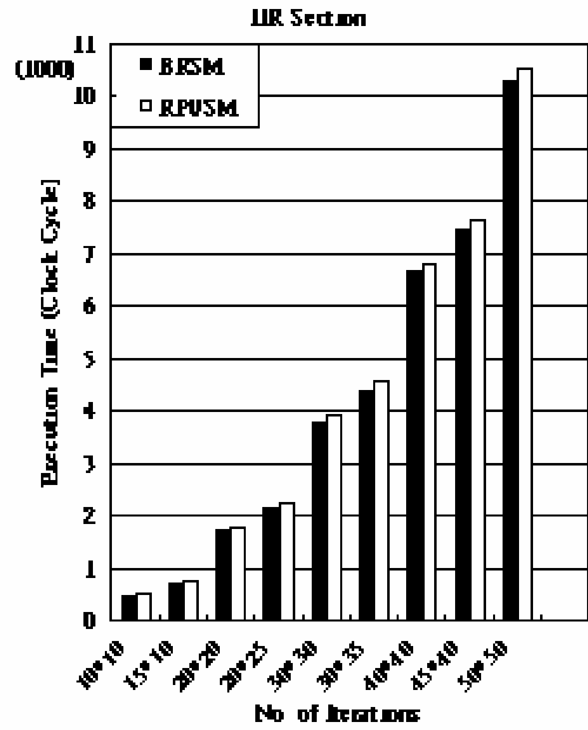
		Floyd Steimberg	IIR Section	Transmission Line	IIR Filter	DFT	2-D Filter	Forward Substitution	THC Solver
RPUSM	Schedule vector	(2,1)	(1,1)	(1,0)	(1,1)	(1,1)	(1,1)	(1,1)	(0,1)
	Maximum retiming depth	6	2	2	2	2	3	2	1
	Static schedule length	5	4	4	4	3	9	3	2
BRSM	Schedule vector	(2,1)	(1,1)	(1,0)	(1,1)	(1,1)	(1,0)	(1,0)	(0,1)
	Maximum retiming depth	2	1	2	1	1	0	1	1
	Static schedule length	5	4	4	4	3	9	3	2

and RPUSM, the schedule vector and maximum retiming depth are shown in Table 3.2. The execution time of each benchmark is shown in Fig. 3.10.

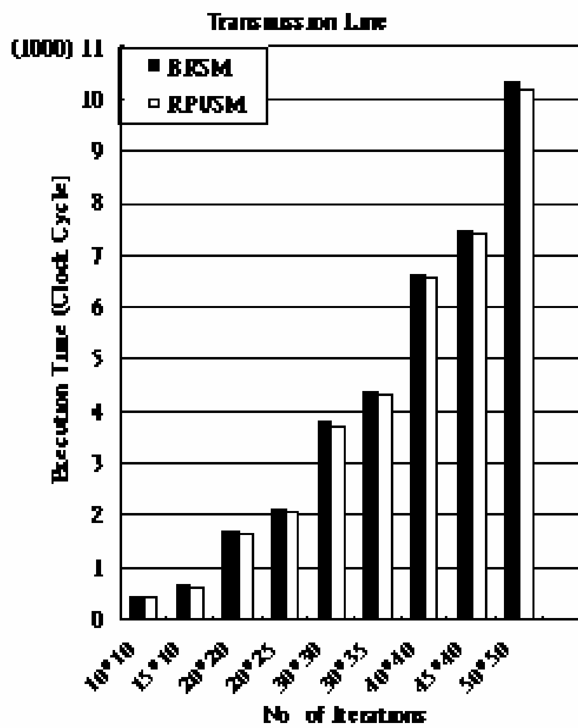
From the Table 3.2, we find that after Floyd-Steimberg is scheduled by BRSM and RPUSM, they select the same schedule vector, but BRSM results in a smaller maximum retiming depth. Thus, the execution time of Floyd-Steimberg scheduled by BRSM is smaller than that scheduled by RPUSM, as shown in Fig. 3.10(a), and some other benchmarks have the similar results. When the number of iterations of Floyd-Steimberg equals to 10×10, RPUSM can't apply to this benchmark, because RPUSM results in the maximum retiming depth equal to six and schedule vector equal to (2,1). Thus, the minimal index of the first level loop is 12. For 2-D Filter [8], after scheduled by BRSM and RPUSM, BRSM selects (1, 0) as the schedule vector but RPUSM can't. And BRSM results in a smaller maximum retiming depth. Thus, the execution time of 2-D Filter scheduled by BRSM is smaller than that scheduled by RPUSM , as shown in Fig. 3.10 (f) , and this circumstance also happens to the benchmark, Forward-Substitution [9], as shown in Fig. 3.10(g). After Transmission Line [4] is



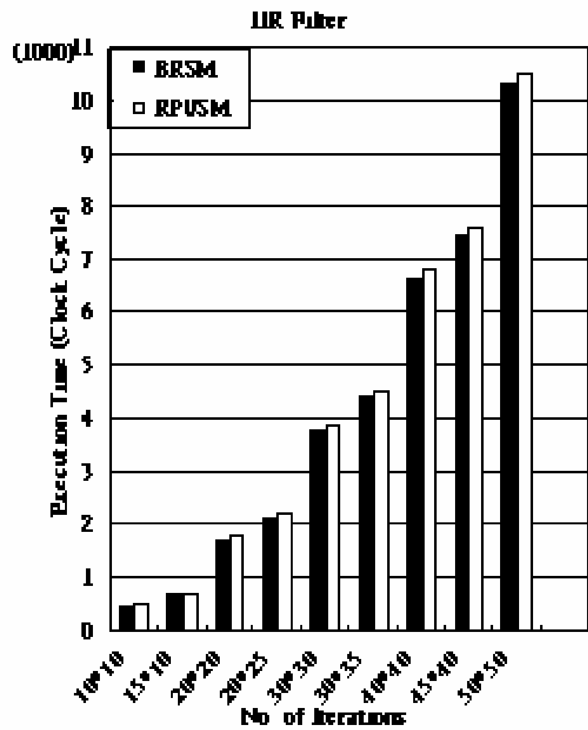
(a)



(b)

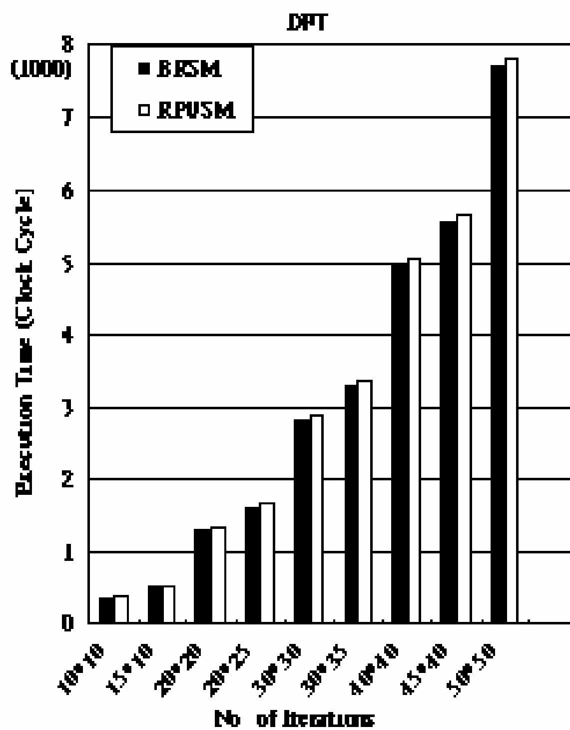


(c)

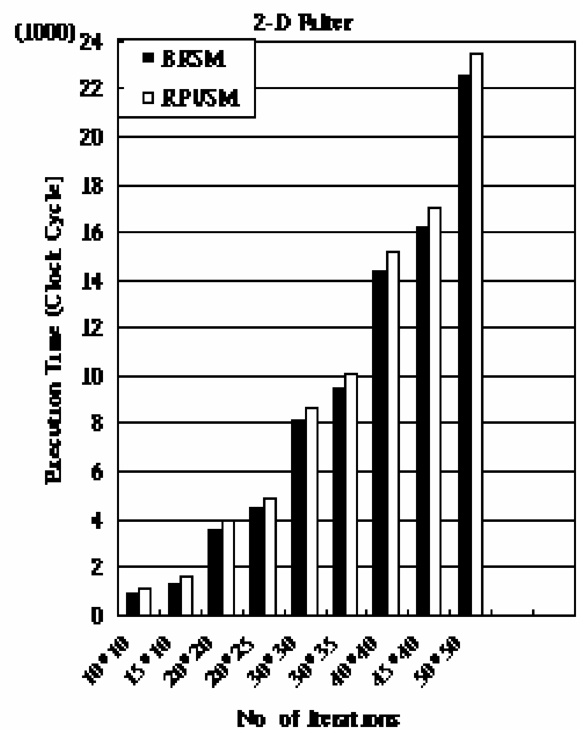


(d)

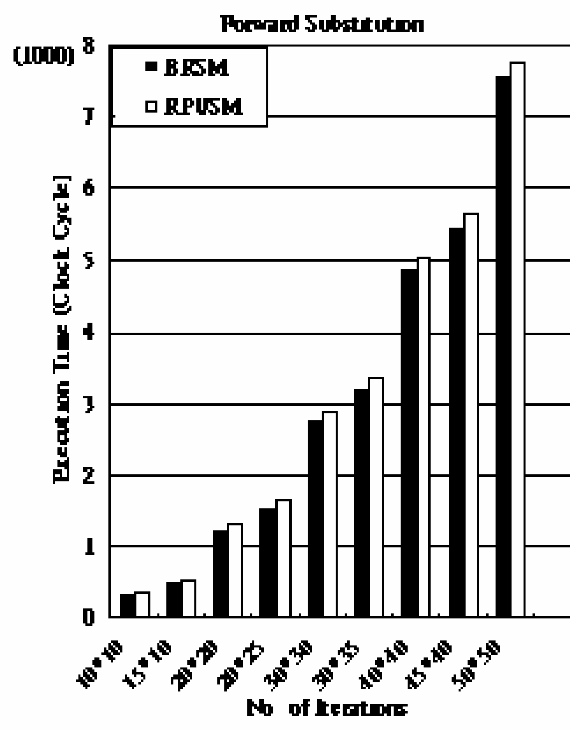
Fig. 3.10 Execution time of every benchmark



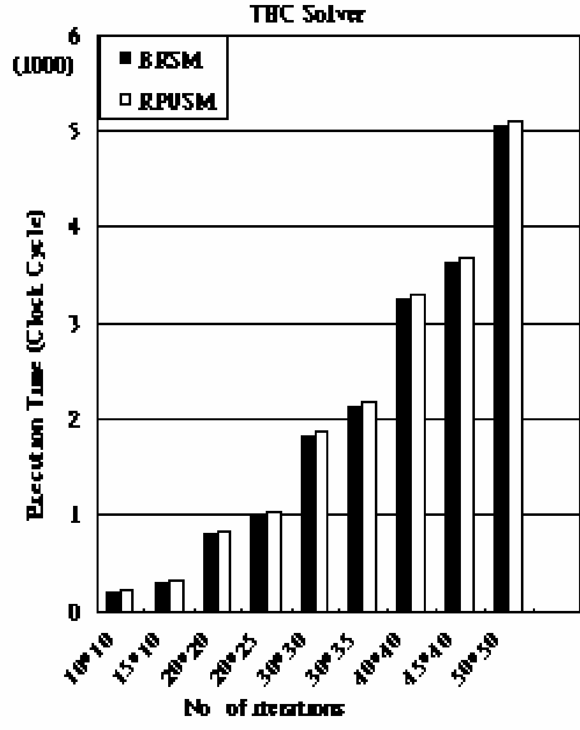
(e)



(f)



(g)



(h)

Fig. 3.10 Execution time of every benchmark

scheduled by BRSM and RPUSM, they result in the same schedule vector and maximum retiming depth. From Fig.3.10(c), we find that the execution time of Transmission Line scheduled by RPUSM is smaller than that scheduled by BRSM. This is because that the schedule length of prologue and epilogue inside the first level loop scheduled by RPUSM is smaller than that scheduled by BRSM. The benchmark, THC Solver [4], is just the opposite. From Fig. 3.10, if we inspect carefully, we find that the difference of the execution time of each benchmark is getting bigger while the number of iterations increases. This is because the prologue and epilogue inside the first level loop increase along with the number of iterations. This phenomenon can be observed by the formula (1).

In 2-Dimensional Filter, BRSM can achieve the minimum schedule length without using the retiming technique. This is because that BRSM gives nodes on the longest path higher priority. Thus, these nodes can be scheduled as early as possible. As mentioned earlier, BRSM has more chances to select (1,0) as a schedule vector. 2-D Filter and Forward Substitution are two instances.

From the evaluation results listed above, we find that BRSM is better than RPUSM. BRSM can result in a schedule with the smaller maximum retiming depth and have more chances to select (1,0) as the schedule vector. The smaller maximum retiming depth and (1,0) as the schedule vector can reduce the execution time of a nested loop. Thus, the execution time of a nested loop scheduled by BRSM is usually less than that scheduled by RPUSM. Next chapter, we'll integrate operand sharing technique into BRSM for low power scheduling.

Chapter 4. Bottom Retiming with Operand Sharing Method

In this chapter, we will introduce our second method named *Bottom Retiming with Operand Sharing method* (BROS). First, we'll show how to group nodes with common operands. Then, BROS will be finely explained. Finally, some basic experimental results will also be shown.

4.1 Motivation

Low power becomes the critical design issue due to wide use of the portable devices, especially those powered by batteries [14]. Reducing switching activities is one of most important power optimization methods when the hardware is built up. Based on the energy model proposed by [14], the energy E_S for a schedule S can be computed by

$$E_S = \sum_{k=1}^{L} P_{cycle}^{(k)} = L \times P_{base} + \sum_{k=1}^{L} \sum_{Inst_i^{(k)}} P_{Inst_i^{(k)}} + \sum_{k=1}^{L} \sum_{Inst_i^{(k)}} SP^{(k)}(i, j)$$

where P_{cycle} is the power consumption of one control step in which several instructions can be executed, P_{base} is the base power needed to execute one control step, P_{Inst_i} is the basic power to execute an instruction $Inst_i$ on a functional unit, $SP(i,j)$ is the switching power caused by switching activities between $Inst_i$ (current sub-instruction) and $Inst_j$ (last sub-instruction) executed on the same functional unit, and L is the schedule length of S. $\sum_{k=1}^{L} \sum_{Inst_i^{(k)}} P_{Inst_i^{(k)}}$ is the summation of basic power consumptions for all instructions of an application. It doesn't change with different schedules. L and $\sum_{k=1}^{L} \sum_{Inst_i^{(k)}} SP^{(k)}(i, j)$ will change with different schedules length. Therefore, in order to minimize the energy consumption of an application,

schedule length of applications and switching activities between instructions both need to be considered in scheduling. In the following, we will show how we consider both issues in scheduling.

4.2 Grouping Nodes

BRSM can produce a schedule with a smaller maximum retiming depth and smaller execution time of nested loops. And the operand sharing technique can reduce the input activities of functional units. Thus, in order to reduce the schedule length and switching activities, we intend to integrate the operand sharing technique into BRSM.

Operand sharing technique tries to bind operations with a common operand to the same functional unit such that the input activity of the shared functional unit decreases. In the MDFG of Fig. 3.4, *A6*, *A7*, *A8* and *A9* share the same operand produced by *A5*. When we bind these four nodes in continuous cycles, three operand transitions in an iteration can be reduced. That is to say, there are three operand reutilizations in an iteration.

In order to have as many as possible operand reutilizations in an iteration, we must make sure that operations with a common operand are scheduled in continuous cycles. Thus, we have to find out those operations first. We can use BFS-like method to traverse MDFGs. If the number of outgoing edges of the current traversing node is bigger than one, there is more than one operation sharing a common operand. Thus, by checking the number of outgoing edges of the current traversing node, we can determine if its children have to be grouped in an *operand sharing set* (SS).

In some cases, we group node *u* into some SS, but after scheduling, node *u* has different operands from other nodes in the same SS. For example, in Fig. 4.1(a), we assume that two adders are available, and the minimum static schedule length equals to two. From that MDFG we know *A2* and *A3* share the same operand produced by *A1*. In order to bind *A2* and *A3* in

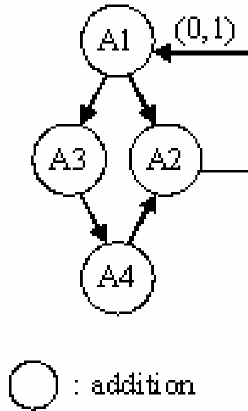
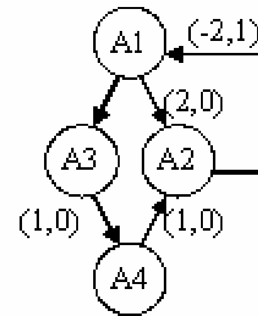


Fig. 4.1 An MDFG

CS	Adder	Adder
1	A1	A2
2	A4	A3

(a)



(b)

Fig. 4.2(a) the final schedule; (b) the retimed MDFG

the same functional unit, one of the possible schedules is shown in Fig. 4.2(a) and the retimed MDFG is shown in Fig. 4.2(b). Even though *A2* and *A3* are bound in the same functional unit and in continuous control steps, A2 and A3 share different operands, and no operand transition is reduced. We call such node, *A2*, as an *unnecessary node* of the SS. We can define an unnecessary node more formally as follows. If node u and v belong to the same SS sharing the data produced by w and a zero-delay path exists from some child of node v to u, u is an unnecessary node of the SS and will be deleted from the SS. Why a node is unnecessary is that in order to bind u and v in continuous cycles, we need to schedule u before schedule those nodes on the path from v to u. Some edges on that path usually require multidimensional delays to make u being scheduled earlier. The side effect is that edge $w \rightarrow u$ has a multidimensional delay but edge $w \rightarrow v$ doesn't. The algorithm in Fig. 4.3 is used to find all the SSs of an MDFG and delete unnecessary nodes from SSs. For example, in Fig. 3.4, after we traverse the MDFG, we group $A6$, $A7$, $A8$ and $A9$ in one SS, and group $A10$, $A11$, $A12$ and $A13$ in another SS.

After finding out all the SSs of an MDFG, we'll pre-schedule all the SSs in order to make sure that nodes in an SS are bound in the same functional unit in continuous cycles. Because one big SS will have more operand reutilizations than several small SSs, the SS with bigger

```
Algorithm Group (G = (V, E, d, t))
1.   SS = Queue = ∅
2.   ∀e ∈ E, E ← E − {e, s.t. d(e) ≠ (0,0,...,0)}
3.   Queue = {v | such that INDEFREE(v)=0}
4.   While Queue != ∅
          /* fetch the first node in Queue*/
5.        u = Get(Queue)
6.        Set_A = Set_M = ∅
7.        ∀v_i  i = 1,2,…,n such that v_i is the child of u and v_i is an addition
8.             if v_i hasn't been grouped and isn't an unnecessary node
9.                  Set_A = Set_A ∪ {v_i}
10.            endif
11.       ∀v_i  i = 1,2,…,n such that v_i is the child of u and v_i is an multiplication
12.            if v_i hasn't been grouped and isn't an unnecessary node
13.                 Set_M = Set_M ∪ {v_i} \
14.            endif
15.       ∀v_i  i = 1,2,…,n such that v_i is the child of u which is an assignment,
                                        v_i hasn't been grouped, and isn't an unnecessary node
               /* group assignment nodes to a bigger set */
16.            if |Set_A| + Min{n, AA} > |Set_M| + Min{n, AM}
17.                 if n < AA
18.                      Set_A = Set_A ∪ {v_i}  i=1,2,…,n
19.                      AA = AA - n
20.                 else
21.                      Set_A = Set_A ∪ {v_i}  i=1,2,…,AA
22.                      Set_M = Set_M ∪ {v_i}  i=AA+1,AA+2,…,n
23.                      AA = 0; AM = AM − n + AA
24.                 endif
25.            else
26.                 if n < AM
27.                      Set_M = Set_M ∪ {v_i}  i=1,2,…,n
28.                      AM = AM - n
29.                 else
30.                      Set_M = Set_M ∪ {v_i}  i=1,2,…,XA
31.                      Set_A = Set_A ∪ {v_i}  i=XA+1,XA+2,…,n
32.                      AM = 0; AA=AA − n + AM
33.                 endif
34.            endif
```

```
35.         delete every edge  u → v
36.         if INDEGREE(v) = 0
37.             Queue = Queue ∪ {v}
38.         endif
39.         if |Set_A| > 1
40.             SS = SS ∪ Set_A
41.         endif
42.         if |Set_M| > 1
43.             SS = SS ∪ Set_M
44.         endif
45.     endwhile
46.     Return SS
```

Fig. 4.3 The algorithm of Group()

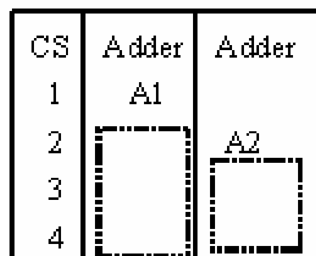


Fig. 4.4 Example for available continuous cycles

size will be scheduled first. For example, one SS with eight elements has seven possible operand reutilizations, but two SSs with four elements have six possible operand reutilizations. Before we schedule an SS in continuous cycles, we have to check if any one functional unit is available in any |SS| continuous cycles, where |SS| means the size of an SS. The *available continuous cycles* is the maximum number of continuous cycles in which a functional unit is available. For example, in Fig. 4.4, two adders are available and two nodes have been scheduled and the minimum static schedule length is four. The available continuous cycles of the two adders are two and three respectively. The *maximum continuous cycles* is the maximum number of the available continuous cycles of every functional unit of the same kind. For example, in Fig. 4.4 the maximum continuous cycles of adders is three. BRSM is under

```
Algorithm Allocation( SS)
1.    While |Group| > 0 and (maximum available continuous control steps of adders > 1
              or maximum available continuous control steps of multipliers > 1)
2.        LG = largest set in SS
3.        MCS = maximum available continuous control steps of FUs used by nodes in LG
4.        if |LG| <= MCS
5.            assign nodes in LG to MCS
6.            delete LG from SS
7.        elseif |LG| > MCS
8.            if MCS >1
9.                SLG = Subset of LG with its size equal to MCS
10.               assign nodes in SLG to MCS
11.               LG = LG - SLG
12.           else
13.               delete LG from SS
14.           endif
15.       endif
16.   endwhile
```

Fig. 4.5 The algorithm of Allocation ( )

the minimum static schedule length with limited resources constraint, thus the maximum value of the maximum continuous cycles equals to the minimum static schedule length. If |SS| is bigger than the maximum continuous cycles, we have to subtract a subset with its size equal to the maximum continuous cycles from the SS, and this subset is pre-scheduled immediately. The remainder becomes a new SS. The algorithm in Fig. 4.5 is shown how to pre-schedule the SSs. For example, in Fig. 3.4, after applying the function Group, two SSs are found, {*A*6, *A*7, *A*8, *A*9} and {*A*10, *A*11, *A*12, *A*13}. After we pre-schedule these two SSs, the result is shown in Fig. 4.8(a).

In some cases, after applying BRSM to MDFGs, the maximum operand reutilizations would be decreased. For example, there are at most nine operand reutilizations in an iteration in Fig. 3.4. Three operand reutilizations are due to *A6*, *A7*, *A8* and *A9* sharing the data

Algorithm Bottom Retiming with Operand Sharing Method $(G = (V, E, d, f))$

1. $ES(\forall u \in V) \leftarrow 0$
2. $MC(\forall u \in V) \leftarrow 0$
3. $MC\max \leftarrow 0$
4. $QueueV \leftarrow \phi$
5. $SS \leftarrow \phi$
6. $\forall e \in E, E \leftarrow E - \{e, s.t. d(e) \neq (0,0,\ldots,0)\}$
7. $QueueV \leftarrow QueueV \cup \{u \in V, s.t. INDEGREE(u) = 0\}$
8. $Prioritize_Nodes(G)$
9. $Compute_Minimum_Schedule_Length()$
10. $SS = Group(G)$
11. $Allocation(SS)$
12. while $QueueV \neq \phi$
    - /* fetch a node with highest priority in QueueV */
13. $GET(u, QueueV)$
14.   if $Assign(u) < ES(u)$
15. $MC(u) \leftarrow MC(u) + 1$
16. $MC\max \leftarrow Max\{MC(u), MC\max\}$
    - /* the adjusting behavior */
17.   if u belongs to set S in SS and nodes in S share the data produced by w
18. $\forall s \in S,\ MC(s) \leftarrow Max\{MC(s), MC(w) + 1\}$
19.   endif
20.   endif
21. $ES(u) \leftarrow Assign(u)$
22.   Assign node u to an available functional unit at control step $ES(u)$
23. $\forall v$ such that $u \rightarrow v$
24. $INDEGREE(v) \leftarrow INDEGREE(v) - 1$
25. $ES(v) \leftarrow Max\{ES(v), ES(u) + t(u)\}$
26. $MC(v) \leftarrow Max\{MC(v), MC(u)\}$
27.   if $INDEGREE(v) = 0$
28. $QueueV \leftarrow QueueV \cup \{v\}$
29.   endif
30. endwhile

```
    /* get a schedule vector by the method of RPUSM */
31. ∀u ∈ V, rd(u) ← MCmax − MC(u)
32. if !∃d(e) = (0, a) for any e ∈ E
33.     Choose s = (1,0) such that s·d > 0 for any e ∈ E
34. elseif ∃d(e) = (0, a) for any e ∈ E
35.     if ∀u --(0,a)--> v, rd(u) + a > rd(v)
36.         Choose s = (1,0) such that s·d(e) > 0 for any e ∈ E
37.     else
38.         Choose s = (s1, s2) such that s·d(e) > 0 for any e ∈ E
39.     endif
40. endif
41. Choose r such that r ⊥ s
42. ∀u ∈ V, r(u) ← (MCmax − MC(u)) × r
```

Fig. 4.6 The algorithm of BROS

produced by *A5*. And six operand reutilizations are due to *A10*, *A11*, *A12* and *A13* sharing the data produced by *A6* and *A9*. After applying BRSM to that MDFG, from the retimed MDFG in Fig. 3.9(a) we find that the maximum operand reutilizations in an iteration are seven. It is because that both operands of *A13* are different from operands of *A10*, *A11*, and *A12*.

In order to have as many as possible operand reutilizations in an iteration, we have to make sure that not only operations with a common operand are scheduled in continuous clock cycles, but also the maximum operand reutilizations in an iteration aren't decreased. By using the two functions *Group* and *Allocation*, we can achieve the former one. In order to achieve the later one, we only need to make sure that for every node v such that v belongs to some SS and shares the data produced by node w, edge $w \rightarrow v$ has the same multidimensional delay. Thus, if some such edge requires an additional multidimensional delay, we make all the other edges having the same multidimensional delays. We call this behavior as *adjusting*. By adjusting, we can make sure that the maximum operand reutilizations in an iteration aren't decreased. In the following, we'll introduce BROS.

4.3 Bottom Retiming with Operand Sharing Algorithm

In order to integrate the operand sharing technique into BRSM, we have to modify the function *Assign*, used in BRSM. *Assign*(*u*) records the control step in which node *u* is scheduled. If node *u* belongs to some SS, *u* must have been scheduled. Thus, *Assign*(*u*) equals to the control step which *u* is scheduled in. If node *u* doesn't belong to any SS, the data of *Assign*(*u*) is determined by the original definition as in chapter 3.

The algorithm shown in Fig. 4.6 is Bottom Retiming with Operand Sharing method (BROS). The adjusting behavior is from line 17 to line 19 in the algorithm. We give an example in Fig. 3.4 to show how BROS works. Assume that three adders and a multiplier are available. After prioritizing nodes, the priority of every node is shown in Fig. 4.7. The minimum static schedule length equals to five. Then, we find out two operand sharing sets,{*A6*, *A7*, *A8*, *A9*} and {*A10*, *A11*, *A12*, *A13*}, and then these sets are pre-scheduled, as shown in Fig. 4.8(a). The final schedule is shown in Fig. 4.8(b). Before applying the adjusting behavior, edges requiring multidimensional delays are $M4 \rightarrow A4$, $A5 \rightarrow A6$, $A5 \rightarrow A7$, $A5 \rightarrow A8$, $A5 \rightarrow A9$, $A6 \rightarrow A10$, $A6 \rightarrow A11$, $A9 \rightarrow A10$ and $A9 \rightarrow A11$. The data sharing by *A10* and *A11* is different from the data sharing by *A12* and *A13*. But when the adjusting behavior is applied, *A10*, *A11*, *A12,* and *A13* share the same data produced by *A6* and *A9*. The multidimensional delay counting function and the retiming depth of every node are shown in Fig. 4.7. The schedule vector (2,1) and the retiming base (1,-2) are calculated. Finally, the retiming function and the retimed MDFG is shown in Fig. 4.9. The maximum retiming depth equals to three, and the operand reutilizations in an iteration equal to nine, the maximum operand reutilizations in an iteration.

Actually, BROS sacrifices some performance, increased maximum retiming depth, to produce a schedule with well operand reutilizations. In the following, we give experimental

Node	Priority	MC(u)	Retiming Depth
A1	7	0	3
A2	6	0	3
A3	5	0	3
A4	3	1	2
A5	2	1	2
A6	1	2	1
A7	0	2	1
A8	0	2	1
A9	1	2	1
A10	0	3	0
A11	0	3	0
A12	0	3	0
A13	0	3	0
M1	8	0	3
M2	7	0	3
M3	6	0	3
M4	4	0	3

Fig. 4.7 The information of scheduling nodes

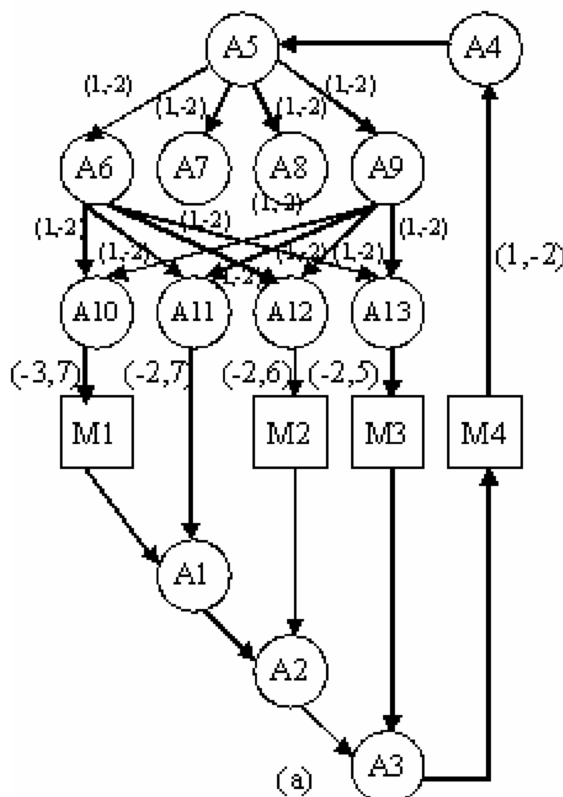
CS	Adder	Adder	Adder	Mult.
1	A6	A10		
2	A9	A11		
3	A7	A12		
4	A8	A13		
5				

(a)

CS	Adder	Adder	Adder	Mult.
1	A6	A10	A4	M1
2	A9	A11	A1	M2
3	A7	A12	A2	M3
4	A8	A13	A3	
5	A5			M4

(b)

Fig. 4.8(a) the schedule after allocation;
(b) the final schedule



r(M1)=r(M2)=r(M3)=r(M4)=3×(1,-2)
r(A1)=r(A2)=r(A3)=3×(1,-2)
r(A4)=r(A5)=2×(1,-2)
r(A6)=r(A7)=r(A8)=r(A9)=1×(1,-2)
r(A10)=r(A11)=r(A12)=r(A13)=0×(1,-2)

(b)

Fig. 4.9(a) The retimed MDFG; (b) the retiming function

results to show that BROS can produce a schedule with the performance near BRSM and operand reutilizations near LPLS.

4.4 Experimental Results

In this section, we first introduce the formula of executing the number of operand reutilizations, and then compare the execution time and number of operand reutilizations of five benchmarks scheduled by BROS, BRSM, RPUSM, and LPLS. We will give basic analysis of the comparison and conclude the advantage of BROS.

4.4.1 Evaluating Operand Reutilizations

The same as the formula (1), the total operand reutilizations can also be divided into three parts. For 2-dimensional loop, the three parts are the loop body, the prologue and epilogue inside the first level loop, and the prologue and epilogue out of the nested loop. Thus, based on similar concept, we can modify formula (1) to produce a formula of evaluating operand reutilizations as follows:

$$OR_A(m - s_2 \times d)(n - s_1 \times d) + (OR_B + OR_C)(s_1 \times m + s_2 \times n - s_1 \times s_2 - 2 \times d \times s_1 \times s_2) + OR_D \times s_1 \times s_2 \times d(d+1) \qquad \text{------ (2)}$$

where (s_1,s_2) is the schedule vector, d is the maximum retiming depth, OR_A is the number of operand reutilizations in an iteration after optimization, OR_D is the number of operand reutilizations in an iteration after applying "List Scheduling", OR_B, OR_C are the number of operand reutilizations of prologue and epilogue inside the first level loop, and m, n are the bounds of loop index. The equation $A(m - s_2 \times d)(n - s_1 \times d)$ represents total operand reutilizations in the loop body . The equation $(B + C)(s_1 \times m + s_2 \times n - s_1 \times s_2 - 2 \times d \times s_1 \times s_2)$

Table 4.1 The schedule vector and maximum retiming depth

		Floyd Steimberg	IIR Section	Transmission Line	DFT	2-D Filter
BROS	Schedule vector	(2,1)	(1,1)	(1,0)	(1,1)	(1,1)
	Maximum retiming depth	3	2	2	1	1
	Static schedule length	5	4	4	3	9

represents total operand reutilizations in the prologue and epilogue inside the first level loop. The equation $D \times s_1 \times s_2 \times d(d+1)$ represents total operand reutilizations in the prologue and epilogue out of the nested loop. Following, we will compare operand reutilizations and execution time of five benchmarks after schedule by BROS, BRSM, RPUSM, and LPLS.

4.4.2 Preliminary Performance Evaluations

We use five benchmarks to evaluate the effect of BROS. Operand reutilizations of five benchmarks after scheduled by LPLS is a reference to see the effect of BROS on operand reutilizations. Execution time of benchmarks after scheduled by BRSM and RPUSM is another reference to see the effect of BROS on performance.

The five benchmarks are the Floyd-Steimberg, IIR Section, Transmission Line, a Discrete Fourier Transform (DFT) , and a 2-D Filter , as shown in Appendix A . These benchmarks contain operations with common operands and are all 2-dimensional loops. Loop indexes vary from 10 to 50 with various combinations for each benchmark. The available number of functional units for each benchmark is shown in Table 3.1. After scheduled by BROS, the schedule vector and the maximum retiming depth are shown in Table 4.1. The operand reutilizations and execution time of these five benchmarks scheduled by these four methods are shown in Fig. 4.10 to Fig. 4.14.

In the Fig. 4.10, BROS produces a schedule with operand reutilizations near LPLS and

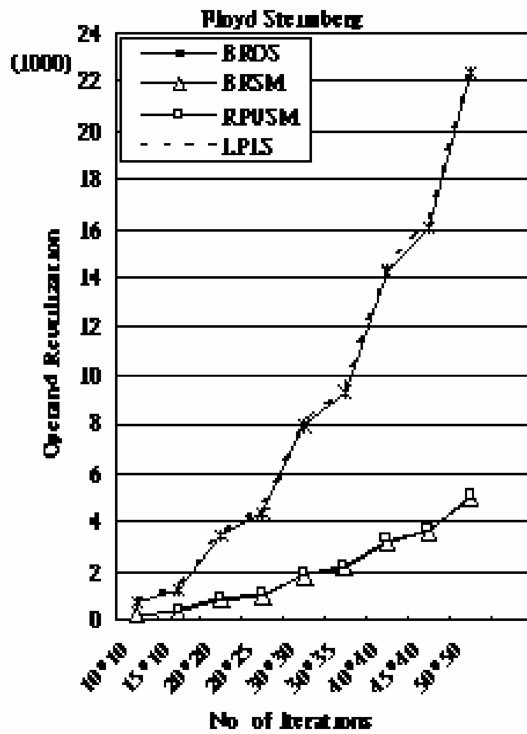
the performance better than that of LPLS and PRUSM. The difference of operand reutilizations between LPLS and BROS is a constant value. This is because BROS can't bind those nodes outside the nested loop and the number of those nodes won't change while number of iterations changes. In Fig. 4.11, it's a similar result.

In Fig. 4.12, BROS produces a schedule with operand reutilizations more than that of BRSM and RPUSM, and the performance similar. The difference of number of operand reutilizations between LPLS and BROS is getting bigger and bigger. Besides nodes outside the nested loop can't be bound by BROS. Another reason is that for this benchmark, an operand sharing set with eight nodes exists, but the maximum continuous cycles equals to minimum static schedule length, four. It has to be split into two subsets. One possible operand reutilization in an iteration is decreased. Thus, while number of iterations gets bigger , the difference also becomes bigger.

In Fig. 4.13, BROS produces a schedule with operand reutilizations more than that of RPUSM and the performance is the same as BRSM. The difference of number of operand reutilizations between BROS and LPLS is because two unnecessary nodes exist. Thus, two possible operand reutilizations in an iteration are decreased. In this benchmark, (1,0) is the schedule vector. Thus, there is no node outside the nested loop.

In Fig. 4.14, the performance of BROS is better than that of RPUSM and operand reutilizations are more than that of BRSM and RPUSM. The difference of number of operand reutilizations between BROS and LPLS is because nodes outside the nested loop can't be bound by BROS. Besides, there is one node belonging to two SSs, and they can be merged into one big SS. But BROS doesn't. Thus, one possible operand reutilization in an iteration is decreased.

From the five benchmarks, we find that, BROS is a good scheduling method for performance and power consumption. It has the advantage of BRSM , good performance, and



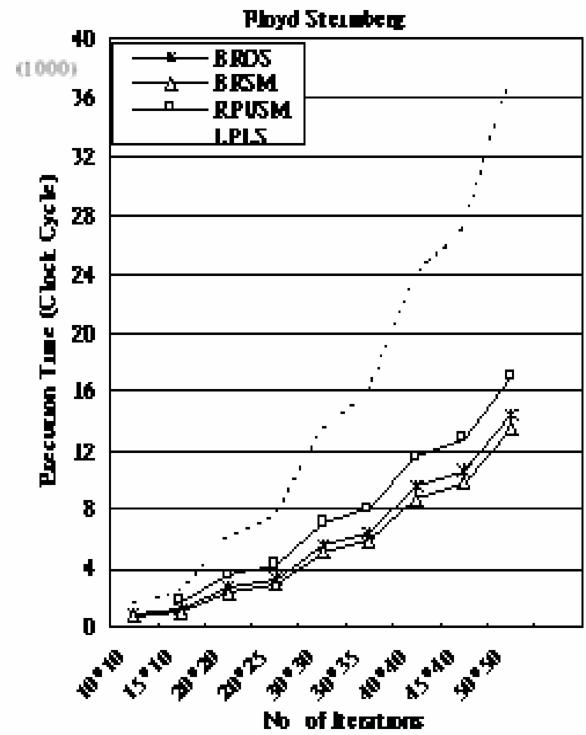
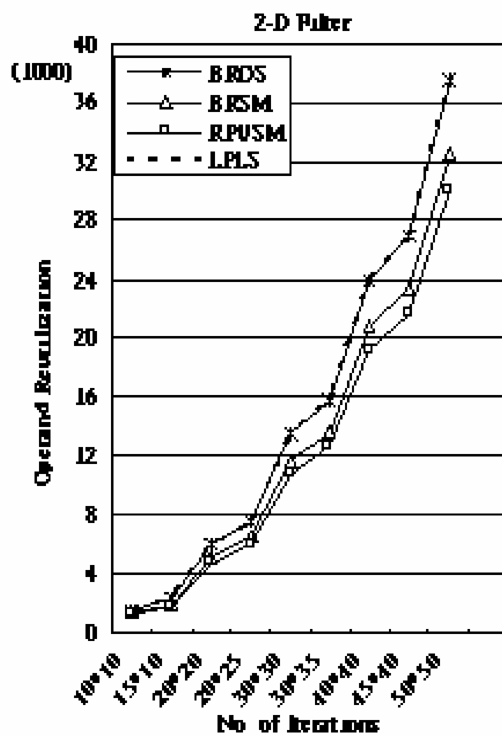


Fig. 4.10 Operand reutilizations and execution time of Floyd-Steimberg



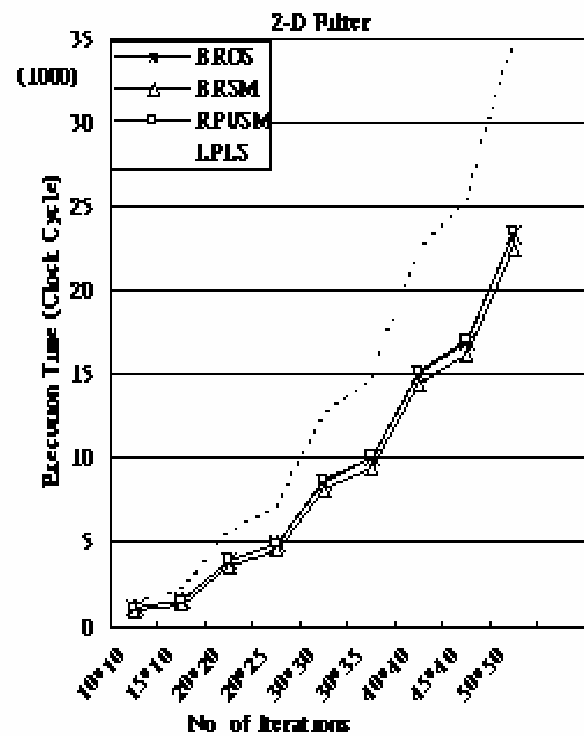
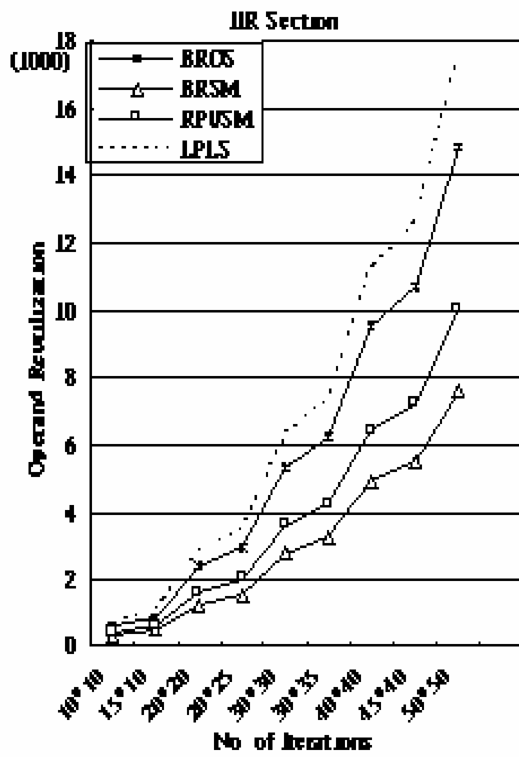


Fig. 4.11 Operand reutilizations and execution time of 2-D Filter



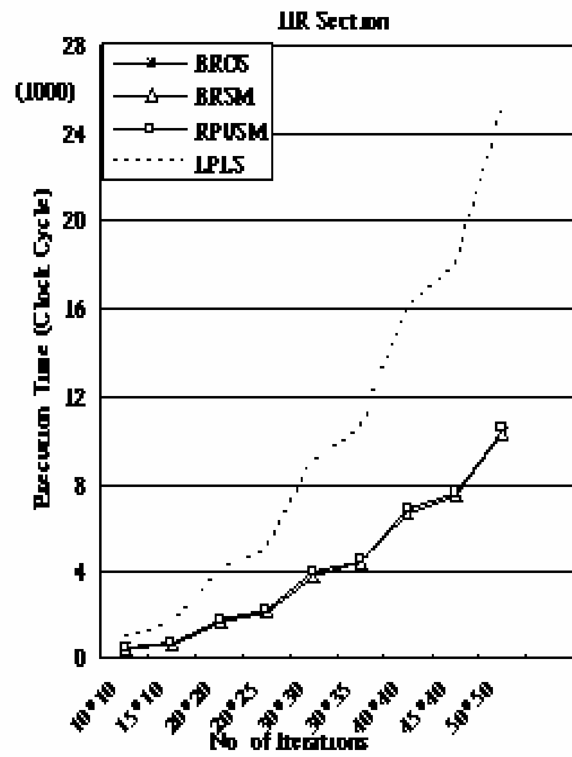
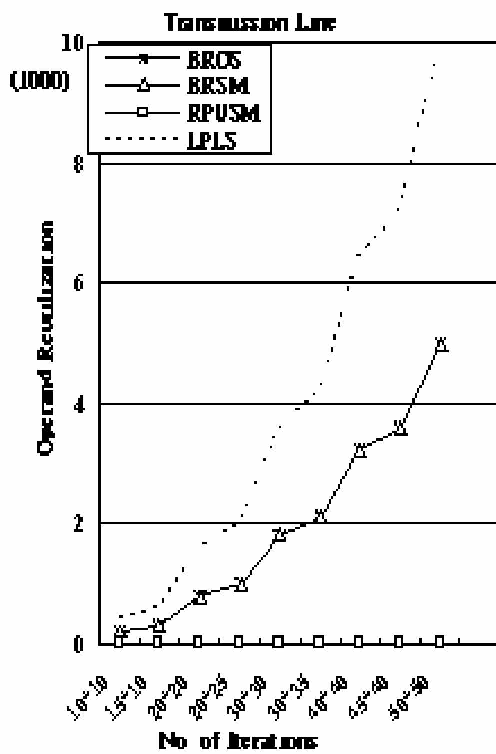


Fig. 4.12 Operand reutilizations and execution time of IIR Section



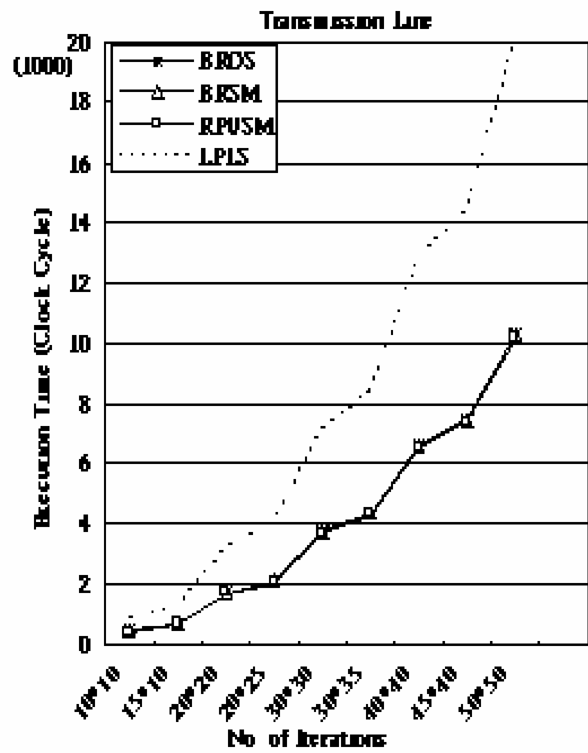


Fig. 4.13 Operand reutilizations and execution time of Transmission Line

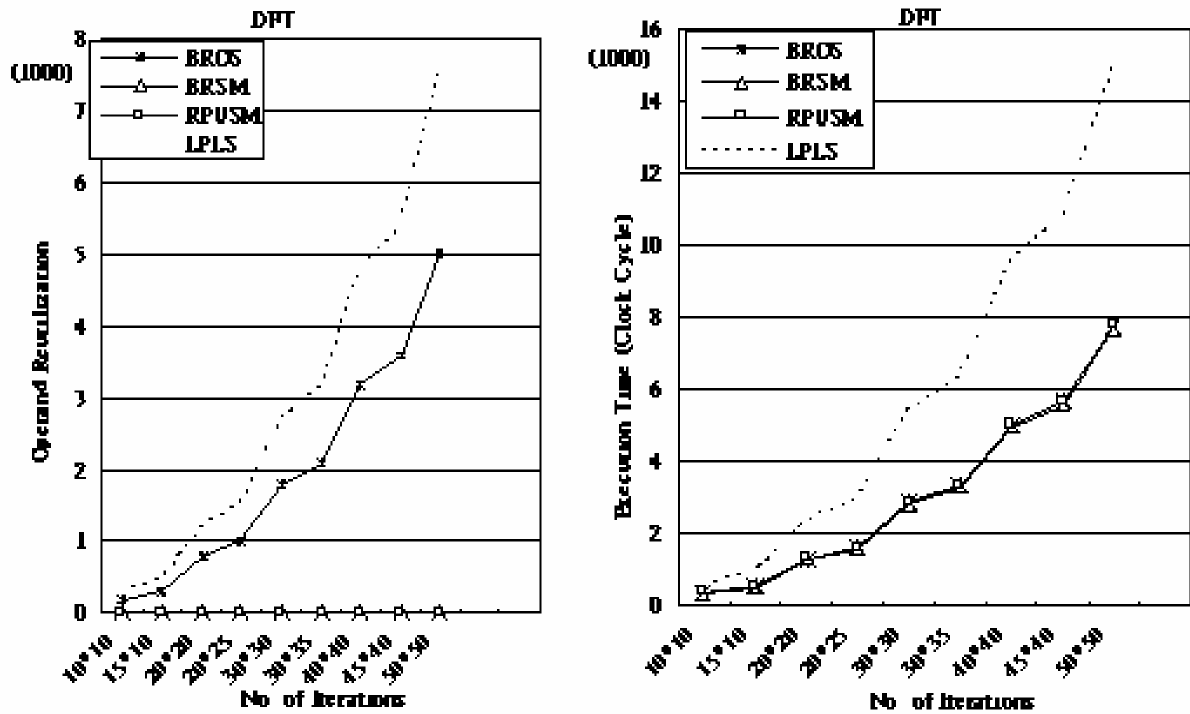


Fig. 4.14 Operand reutilizations and execution time of DFT

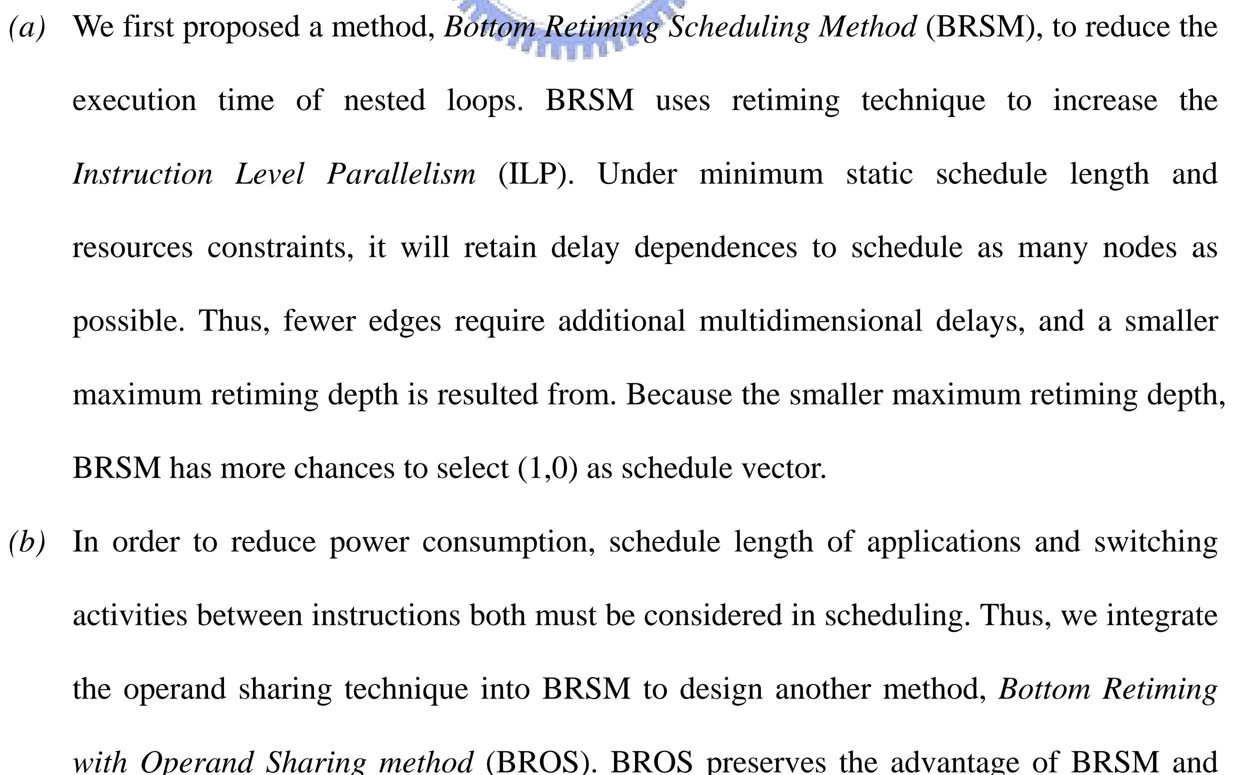
the advantage of the operand sharing technique, well operand reutilization. Although the effect of BROS on operand reutilizations is not as good as LPLS, BROS can achieve similar effect if no necessary nodes exist and operand sharing sets isn't split. Performance is the main shortcoming of LPLS. Although BROS sacrifices some performance to get better operand reutilizations than that of BRSM and RPUSM, BROS still achieves the performance very close to them, in some cases even better than RPUSM. Thus, BROS is a good scheduling method to find a good performance and high operand reutilizations schedule.

Chapter 5. Conclusion and Future Work

In this thesis, we have designed a retiming based scheduling method BRSM to reduce the execution time of nested loops. Then, we integrate the operand sharing technique into BRSM to produce another method BROS which reduces the execution time and increases operand reutilizations of nested loops. The experimental results have shown the effectiveness of the two methods. Finally, we will conclude our thesis and propose some future work for our research.

5.1 Conclusion

Because portable devices become popular, high performance DSP used in such devices needs to be processed with not only high data throughout but also low power consumption. For these two issues, we have proposed two methods. In summary, we give the following conclusions:



(a) We first proposed a method, *Bottom Retiming Scheduling Method* (BRSM), to reduce the execution time of nested loops. BRSM uses retiming technique to increase the *Instruction Level Parallelism* (ILP). Under minimum static schedule length and resources constraints, it will retain delay dependences to schedule as many nodes as possible. Thus, fewer edges require additional multidimensional delays, and a smaller maximum retiming depth is resulted from. Because the smaller maximum retiming depth, BRSM has more chances to select (1,0) as schedule vector.

(b) In order to reduce power consumption, schedule length of applications and switching activities between instructions both must be considered in scheduling. Thus, we integrate the operand sharing technique into BRSM to design another method, *Bottom Retiming with Operand Sharing method* (BROS). BROS preserves the advantage of BRSM and

the operand sharing technique, such as a smaller maximum retiming depth and high operand reutilizations.

(c) In order to evaluate the effectiveness of BRSM and BROS, we use two formulas to calculate the execution time and total operand reutilizations for several benchmarks. From the results, we find that BRSM is a good scheduling method which makes those benchmarks having shorter execution time compared with RPUSM and PUSM. We also find that BROS is suitable for low power and high performance scheduling. BROS can produce similar execution time and much more operand reutilizations than BRSM.

5.2 Future Work

In addition to our present research on BROS, there are still some issues in the research that can be improved in the future.

(a) There are some tools for estimating power consumption, such as SPA [22] and SLS [23]. In our experiments, we use the number of operand reutilizations to compare the effect on reducing switching activities. In order to precisely evaluate the effect of BROS on power consumption, we can try to modify these tools to evaluate the effect of BROS.

(b) After scheduling, if node u belonging to some SS has different operands from other nodes in the same SS, u is an unnecessary node. Unnecessary nodes will decrease operand reutilizations. If inter-iteration operand reutilizations are considered, the decreased operand reutilizations may possibly be regained. Thus, inter-iteration operand reutilizations can improve the number of operand reutilizations of BROS.

Bibliography

[1] E. Musoll and J. Cortadella, “High-level synthesis technique for reducing the activity of functional units,” in *Proc. of International Symposium on Low Power Design*, pp. 99-104, 1995.

[2] E. Musoll and J. Cortadella, “Scheduling and resource binding for low power,” In *Proc. of International Symposium on System Synthesis*, pp. 104-109, 1995.

[3] K. choi and A. Chatterjee, “Efficient instruction-level optimization methodology for low-power embedded systems,” in *Proceedings of the IEEE International Symposium on System Synthesis*, Oct. 2001, pp. 147-152..

[4] N.L. Passo, 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. ICCAD-95. Digest of Technical Papers., pp. 588-591, November 1995.

[5] N.L. Passo, 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.

[6] 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.

[7] P. Held, P. Dewilde, E. Deprettere, and P. Wielage, ”HIFI: From parallel algorithm to fixed-size VLSI processor array,” in *Application-Driven Architecture Synthesis*, F. Catthoor and L. Svensson, Eds. Norwell, MA: Kluwer Academic, 1993, pp. 71-94.

[8] R. Gnanasekaran, ”2-D filter implementation for real-time signal processing,” *IEEE Trans. Circuits Syst.*, vol. 35, no. 5, pp. 587-590, May 1988.

[9] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hell, 1984.

[10] C. Lee, J.-K. Lee, and T. Hwang, "Compiler optimization on instruction scheduling for low power," in *Proceedings of the IEEE International Symposium on System Synthesis*, Sept. 2000, pp. 55-60.

[11] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and low-power scheduling techniques for embedded dsp software," in *Proceedings of the IEEE International Symposium on System Synthesis*, Sept. 1995, pp. 110-115.

[12] D. Kim, D. Shin, and K. Choi, "Low power of linear systems: A common operand centric approach," In *Proc. of the IEEE/ACM International symposium on Low Power Design*. pages 225-230, August 2001.

[13] T. Z. Yu, F. Chen, and E. H.-M. Sha, "Loop scheduling algorithms for power reduction," In *Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 3073-3076, May 1998.

[14] Z. Shao, Q. Zhuge, Y. Zhang and E. H.-M. Sha, "Efficient Scheduling for Low-Power High-Performance DSP Applications," in *The 2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing in conjunction with The 12th International Conference on Parallel Architecture and Compilation Techniques (PACT 2003),* New Orleans, Louisiana, Sept. 2003.

[15] L.-F. Chao, A. LaPaugh, and E.H.-M. Sha, "Rotation Scheduling: A Loop Pipelining Algorithm," *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 566-572, Dallas, Tex., June 1993.

[16] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. M. Rabaey, and R. W. Brodersen, "Optimizing power using transformation," *IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems*, 14(1):12-31, Jan. 1995.

[17] L.-F. Chao and E.H.-M. Sha, “Static Scheduling of Uniform Nested Loops,” *Proc. Seventh Int’l Parallel Processing Symp.*, pp. 1,421-1,424, Newport Beach, Calif., Apr. 1993.

[18] D.A. Schwartz, “Cyclo-Static Realizations, Loop Unrolling and CPM: Optimal Multiprocessor Scheduling,” technical report, Grorgia Inst. of Technology, School of Electrical Eng., 1987.

[19] L. Kruse, E. Schmidt, g. Jochens, A. Stammermann, A. Schulz, E. Macii, and W. Nebel, “Estimation of lower and upper bounds on the power consumption from schedule data flow graphs,” *IEEE Trans. on VLSI Systems*, 9(1):3-14, Feb. 2001.

[20] A. Raghunathan and N. Jha, “Behavioral synthesis for low power,” In *Proc. of the Int. Conf. on computer Design*, pages 318-322, Oct. 1994.

[21] A. Dasgupta and R. Karri, “Simultaneous scheduling and binding for power minimization during microarchitectural synthesis,” In *Int. Symp. on Low Power Design*, pages 69-74, Apr. 1995.

[22] P. Landman and J. M. Rabaey, “Activity-sensitive architectural power analysis,” *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 571-587, June 1996.

[23] A. van Gerenden, “SLS: An efficient switch-level timing simulator using min-max voltage waveforms,” In *Proc. VLSI 89 Conf.*, pages 79-88, Aug. 1989.

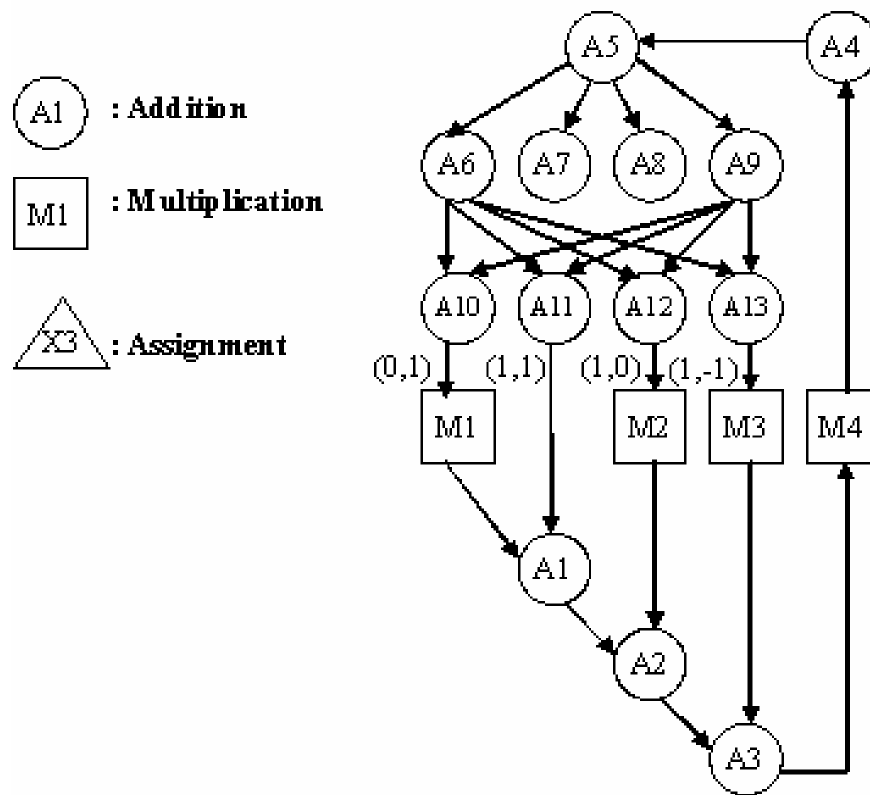
[24] Y. C. Hsu and Y. L. Jeang. “Pipeline Scheduling Techniques in High-Level Synthesis,” *ASIC Conference and Exhibit*, 1993. *Proceedings.* , *Sixth Annual IEEE International*, pp. 396-403, Sept. 1993.

[25] Tsing-Fa Lee, Allen C-H. Wu, Daniel D. Gajski and Youn-Long Lin, “An effective methodology for functional pipelining,” *Proceedings of IEEE International Confefence On Computer-Aided Design*, pp. 230-233, November 1992.

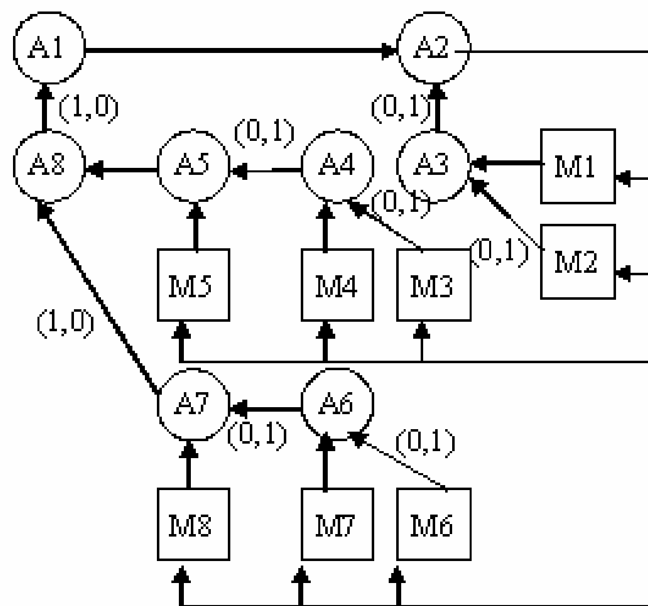
[26] Ching-Yi Wang and Keshab K. Parhi, “ High level DSP synthesis using the MARS design system,” *IEEE ISCAS*, pp. 164-167, 1992.

[27] C. E. Leiserson and J. B. Saxe, “Retiming synchronous circuitry,” Algorithmica, vol. 6, pp. 5-35, 1991.

Appendix A



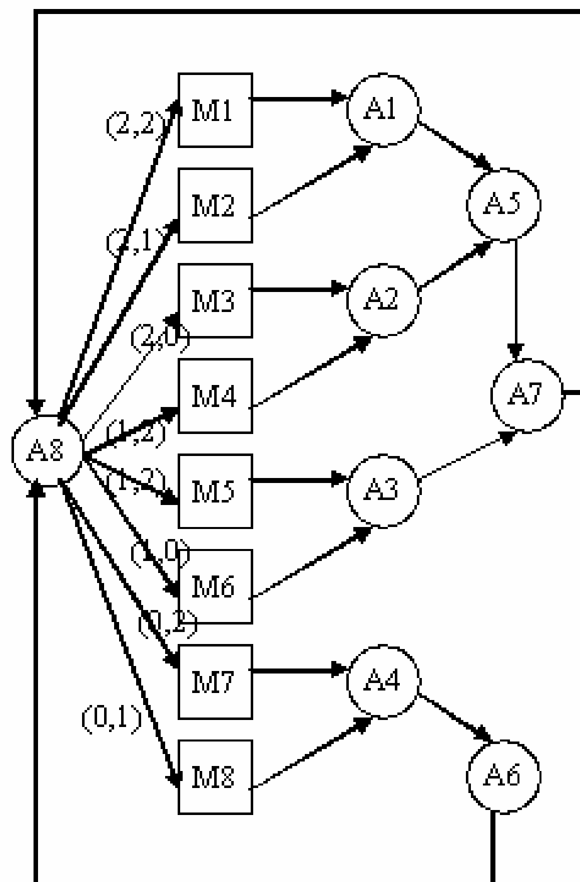
Floyd-Steimberg



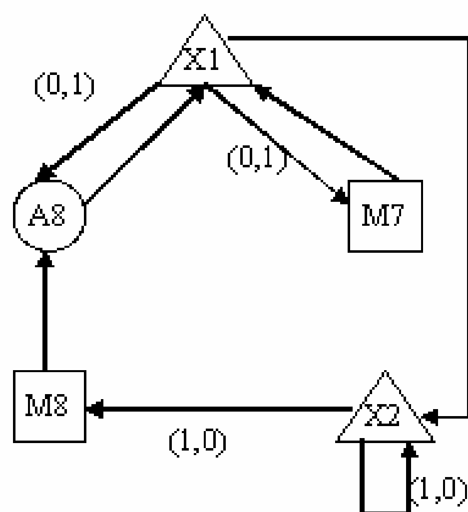
IIR Section

Transmission Line

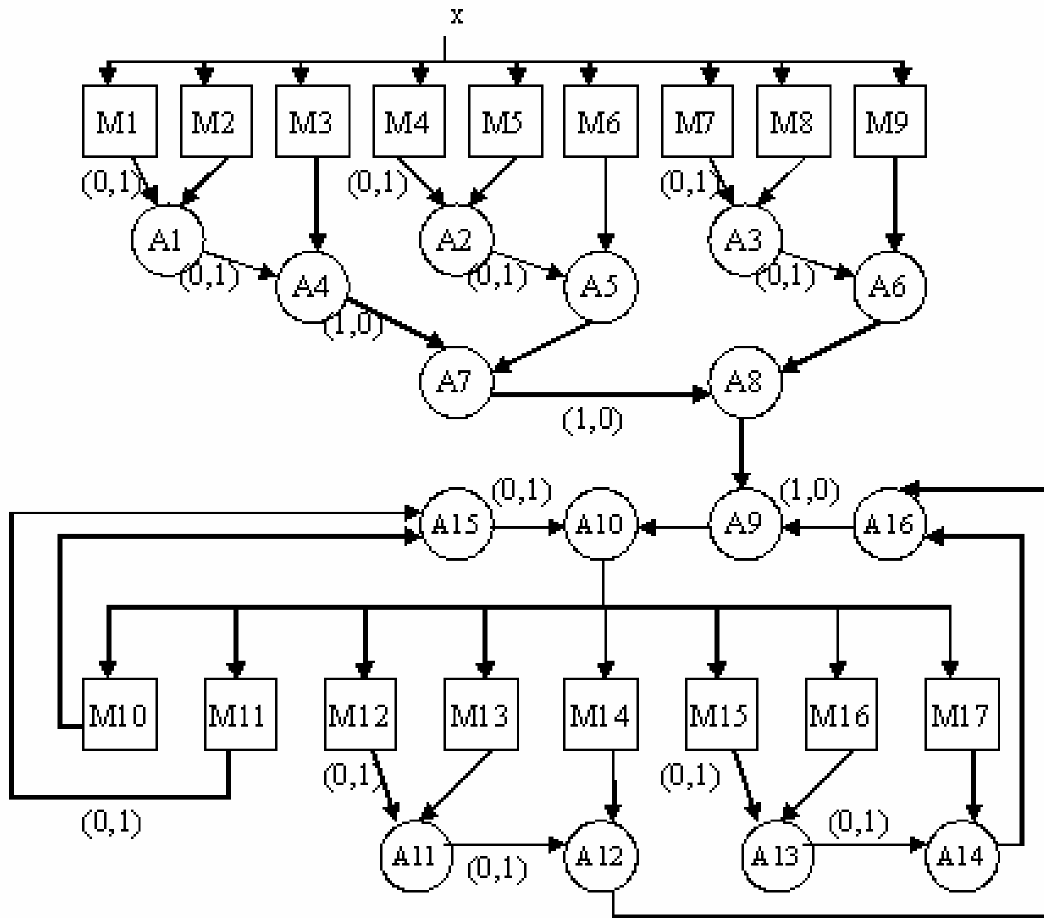
DFT



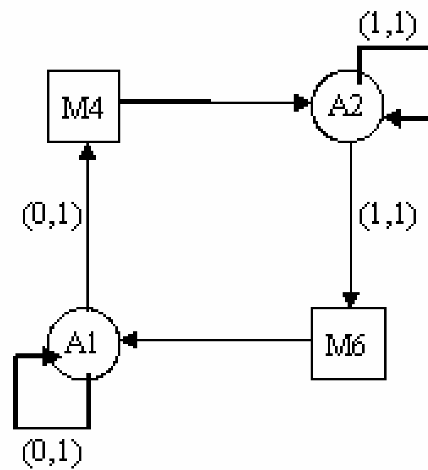
IIR Filter



Forward Substitution



2-D Filter



THC Solver