# REGISTER ALLOCATION FOR OPTIMAL LOOP SCHEDULING

*by*

*Qi Ning*

School of Computer Science
McGill University
Montréal, Québec
Canada

May 1993

A DISSERTATION
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
OF MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY

ISBN 0-315-87946-7

Canada

# Abstract

One of the major challenges in designing optimizing compilers, especially for scientific computation, is to take advantage of the parallelism in loops in order to obtain maximum speedup on parallel computer architectures. Optimal loop scheduling is therefore one of the most important topics studied by many computer scientists. However how to allocate minimum number of registers to support optimal loop scheduling for parallel architectures is less understood. In this thesis, we propose a simultaneous scheduling and register allocation approach for a parallelizing compiler, which will find one among all the time-optimal periodic schedules that uses minimum amount of registers. We prove that the general problem of finding such an optimal scheduling together with register allocation is NP-complete. Then we propose a practical approach to divide the register allocation problem into two steps. The first step solves a minimum buffer allocation problem, which will find a time-optimal periodic schedule using minimum number of buffers. We give a polynomial time algorithm to solve this problem. The second step analyzes the live ranges of the variables and uses coloring algorithms to reduce the register requirement by sharing. The algorithm has been implemented and used to test selected loops in benchmark programs. Testing results are reported in this thesis.

In order to allocate enough memory spaces to support optimal dynamic schedules, we propose a cycle balancing scheme that allocates buffers to the arcs of the dataflow graph representing a loop, so that it can allow a loop being scheduled dynamically to achieve maximum speedup. We show how to formulate the problem into an integer programming problem. Practical polynomial time solution algorithms are given.

# Résumé

Un défi important dans le développement de compilateurs, spécialement dans le cas des calculs scientifiques, est d'exploiter au maximum le parallélisme présent dans les boucles de façon à obtenir une plus grande vitesse d'exécution sur les ordinateurs parallèles. L'ordonnancement optimal des instructions dans les boucles est donc un sujet très important étudié par de nombreux chercheurs. Par contre, le problème de déterminer le nombre minimum de registres nécessaire pour supporter un ordonnancement optimal de telles boucles sur des machines parallèles est nettement moins bien compris. Dans cette thèse, nous proposons une approche réalisant simultanément l'ordonnancement des instructions et l'allocation des registres, approche qui permet d'identifier, parmi tous les ordonnancements périodiques optimaux, celui qui minimisera l'utilisation des registres. Nous prouvons que le problème général de trouver un ordonnancement optimal tout en optimisant l'allocation des registres est NP-complet. Ensuite, nous proposons une approche pratique qui décomposera le problème de l'allocation des registres en deux étapes. La première étape trouve une solution a un problème d'allocation minimum d'espace tampon, trouvant ainsi un ordonnancement périodique optimal utilisant le nombre minimum de tampons. Nous donnons un algorithme polynomial permettant de résoudre ce problème. La deuxième étape analyse ensuite les variables "vivantes" et utilise des algorithmes de coloration pour réduire les besoins en registre par l'intermédiaire d'un partage de ces registres.

Dans le but d'allouer suffisamment d'espace mémoire pour supporter des ordonnancements dynamiques optimaux, nous proposons aussi une approche de "balancement de cycle", approche qui vise à allouer l'espace tampon, sur les arcs d'un graphe

de flux de données représentant une boucle, de façon à ce que l'allocation obtenue permette un accroissement maximum de la vitesse d'exécution. Nous montrons comment formuler ce problème via un problème de programmation linéaire entière. Ensuite, nous donnons un algorithme pratique permettant de résoudre ce problème, algorithme de complexité polynomiale.

# Statement of Originality

Here, we summarize the original contributions of this dissertation:

- We prove that the problem of minimum register allocation to support time-optimal scheduling is NP-complete: Theorem 3.2.1 and Theorem 3.3.1 in Chapter 3.
- We formulate the simultaneous Optimal Scheduling and Buffer Allocation (OSBA) problem and develop a mathematical model for it: Section 4.3 in Chapter 4.
- We propose an efficient polynomial time solution algorithm for OSBA problem: Section 4.4 in Chapter 4. This method is combined with a graph coloring algorithm to map buffers to physical registers: Section 4.7 in Chapter 4.
- We propose and formulate the Cycle Balancing Scheme (CBS) of allocating buffers to arcs of a dataflow graph in order to support run-time optimal schedules: Section 5.4 in Chapter 5.
- We provide a polynomial algorithm that can solve the linear version of the Cycle Balancing problem: Section 5.5 in Chapter 5.
- We show that the Cycle Balancing problem has the Totally Dual Integrality (TDI) property, which allows it to be solved by linear programming when the right-hand-sides are rounded to integer ceilings: Section 5.6 in Chapter 5.

# Acknowledgments

First I would like to express my deepest thank and appreciation to Professor Guang R. Gao. During the years under his supervision, I have obtained enormous professional and financial support I needed to complete my Ph.D program. I have always enjoyed his numerous seminars, challenging courses and stimulating discussions.

Secondly I would like to thank Professor David Avis who co-supervised me, and who pushed me to the "system side" of computer science at the beginning of my Ph.D program at McGill. Without his long-sighted view, I would have been working on a totally different subject.

Next I wish to acknowledge the help and support received from the other ACAPS group members at McGill. Especially I would like to thank Professor Laurie Hendren. She has provided me with so many invaluable comments which have improved my representation enormously. I thank V.C. Sreedhar, Philip Wong, Robert Yates, Kevin Theobald, Chandrika Mukerji, Erik Altman, Cecile Moura and Justiani Dharmas for the useful discussions that in many ways improved my work. I would also like to thank the people who has worked as the system managers in the ACAPS lab for their constant effort to improve and maintain a comfortable system: Robert Yates, V.C. Sreedhar and Chris Donawa.

During my Ph.D study at McGill, my wife, Wenqun Mao, who is also a graduate student, has given me tremendous help and love. She has taken great effort to balance between her own studies and taking care of our son Alex and house work. I express my deepest appreciation of my wife. I greatly thank my mother-in-law, Aiyu Han,

who has contributed so much of her life taking care of my son Alex. Finally I would like to thank my parents, Shiguang Ning and Romei Wang, for their love and support I have received during all my life.

# Dedication

*to Wenqun and Alex.*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In this chapter we give an introduction of the subjects to be studied in this dissertation. Problem statements are informally given as well as the motivations.

## 1.1 Introduction

High performance parallel computer architectures that exploit the fine-grained instruction level parallelism, like Very Long Instruction Word (VLIW) and Superscalar architectures, are designed to issue multiple instructions in a single clock cycle. All sorts of parallelism have to be exploited in order to make the most efficient use of the parallel hardware available in such architectures, and to achieve the maximum speedup of user programs. Since loops are the most time consuming parts in a program, the efficient exploitation of fine-grain parallelism in loops has been a major challenge in the design of optimizing compilers for high-performance computer architectures.

Software pipelining has been proposed as one of the most important fine-grain loop scheduling methods. It determines a parallel schedule which may overlap instructions of a loop body from different iterations. Software pipelining can be applied to high-performance pipelined processor architectures, as well as Superscalar and VLIW architectures [2, 3, 4, 29, 30, 56, 70, 76, 78].

Although much progress has been made in finding time-optimal schedules for software pipelining of loops, to determine an instruction schedule and a register allocation simultaneously is less understood and remains an open problem. In terms of instruction scheduling for RISC processor architectures, it is well recognized that performing register allocation before the instruction scheduling (*postpass scheduling*) may introduce new constraints due to the reuse of registers, which may limit possible reordering and parallelism of the instructions, as reported in [47, 50]. On the other hand, if the instruction scheduling is done before (and independent of) register allocation (*prepass scheduling*), more registers than necessary may be needed, which may cause unnecessary *register spills* and severely degrade the performance of the resulting code. Warren has described a technique for the IBM RS/6000 superscalar workstation which applies instruction scheduling twice: once before and once after the register allocation [78]. Although it will be better than the one pass approach, it does not always do a good job. We believe that it also lacks the foundation to be

generalized to other architectures.

The objective of this thesis is to develop a unified scheduling-allocation framework to determine a scheduling and a register allocation simultaneously. We want the schedule to be time-optimal. We also want the register allocation to use minimum number of registers to support such time-optimal schedules. Our framework is different from the conventional sequential approach, which tries to allocate minimum number of registers for a fixed sequential schedule. For example, many of these register allocation algorithms are based on the coloring of *interference graphs* representing overlapping relations of the live ranges of program variables given a sequential execution schedule [1, 16, 15]. In that approach a schedule is fixed so that the meaning of live ranges of variables is well defined. However a fixed schedule may not use the minimum amount of registers necessary. In this thesis we will show that to find an optimal scheduling and register allocation simultaneously is NP-complete even on a machine with infinite computing resources. However, inspired by a seemingly different research area from compiling, i.e., the earlier work on acyclic dataflow graph balancing by allocating storage buffers to data channels to support maximum computation rate [37, 39, 40], we propose an approach which divides the simultaneous scheduling and register allocation problem into two steps. The first step is to allocate minimum number of buffers to variables to support a maximum speedup schedule. Then the second step analyzes the live ranges of the buffers and uses a coloring algorithm to map the buffers to physical registers. The buffers allow the results produced by instructions to be retained in registers for several iterations, which makes it possible to start a new iteration before the previous iterations finish. The class of schedules we will consider in our first step is called periodic schedules. They are formally defined in Section 2.7 in the next chapter. The two steps will be explained more later in the section. Let us first justify why we choose periodic scheduling as our class of schedules. Periodic scheduling attracts our attention because it shows many properties listed below which are particularly suitable for a compiler to do register allocation:

- It allows the iterations to be software pipelined, i.e. execution of different iterations can be overlapped, such that parallelism can be fully exploited and

optimal speedup can be achieved.

- It is simple and concise because we can describe such a schedule by a very small number of parameters.
- It is regular because the timings of different iterations show a strong regular pattern, which allows us to generate very compact code for the loop.

In the first phase of our simultaneous scheduling and register allocation scheme, we try to find a schedule among all time-optimal periodic schedules that uses minimum number of buffers. The buffers are allocated to individual variables defined in the loop and can be thought as virtual registers. Buffers do not overlap with each other, which let us solve the minimum buffer allocation problem in polynomial time. The idea in the first step comes from the the early research work on balancing acyclic dataflow graphs by allocating buffers to data channels [37, 39, 40] to achieve maximum software pipelining effect. In this thesis we not only generalize the idea to loops which may contain dependence cycles, we also solve the problem of code generation for von Neumann architectures that does not exist in the dataflow study. In the second phase, we have fixed a schedule produced in the first phase, which is time optimal and uses the minimum number of buffers. Therefore we can apply the traditional coloring algorithms to allow variables to share the physical registers.

We will consider the problem of allocating buffer storage to support maximum rate computation of loops represented by dataflow graphs, to support a more general class of dynamic scheduling schemes, as compared to static periodic schedules for compilers. The work is a direct generalization of the previous work on acyclic dataflow graphs [37, 39, 40]. We proposed a *cycle balancing* scheme which will allocate buffers to balance all the cycles in a dataflow graph. The effect of cycle balancing is to allocate minimum number of buffers to support at least one optimal schedule at run-time. Our method is not only useful for generating instruction scheduling and buffer allocation for a compiler, it can also be applied to solve problems in other fields, like digital signal processing, self-timed processor arrays, discrete-event systems and timed petri-nets, etc.

## 1.2 Organization of the Thesis

This section describes the organization of the thesis. We provide the notations and definitions in Chapter 2. However only general terms are defined there so that a reader should feel comfortable to read most of the rest chapters. More specific definitions are defined in later chapters where they are used.

In Chapter 3 we prove that the simultaneous optimal scheduling and register allocation problem for loops is NP-complete even under an idealized parallel architecture. Therefore we know the complexity of our problem in its most general form.

In Chapter 4 we propose a two-step approach to solve the simultaneous optimal scheduling and register allocation problem. The first step is called the Optimal Scheduling and Buffer Allocation (OSBA) problem. We present a polynomial time algorithm to solve it. The second step is to investigate the possibility to share the buffers among different variables. Hence we have found a schedule in the first step, now we only need to analyze the live ranges of the variables and use a coloring algorithm to color the ranges so that if two ranges do not overlap, their buffer entries can be shared. We also propose code generation schemes to support our register allocation method. Our algorithm has been implemented. Testing results about loops selected from benchmarks are also reported in this chapter.

In Chapter 5 we propose a cycle balancing scheme that will allocate buffers to arcs of a dataflow graph so that run-time optimal schedules can be supported.

Chapter 6 contains a brief summary of the achievements in the thesis and a discussion of the future research directions.

# Chapter 2

# Background and Terminology

In this chapter, we provide some background material and define the notations used in later chapters. However only those that are "more" important are given here. Loop models and their data dependence graph representation are defined here. Maximum computation rate and the periodic scheduling scheme that can achieve optimal rate are introduced. Targeting architecture models are briefly defined. Some linear programming and integer programming backgrounds are also provided.

## 2.1 Introduction

In this chapter, we provide the definitions and notations used in subsequent chapters. However not all the definitions and notations used in later chapters are defined here. For example if a definition is very specific to a solution method in a chapter and is not related to other concepts in the thesis, then we will give that definition in that chapter at the place it is used. I hope that this will help the readers to get through this chapter as fast as possible without worrying too much about the definition details.

We first introduce some set theoretic and graph theoretic terminology as the basis of our mathematical notations. Then we introduce the generic loop model we will concentrate on in this thesis. We will define the notion of a data dependence graph which is a form of program representation using graph theoretic terms.

Once we establish our mathematical representation of the program structures, we then consider the scheduling problem for the instructions. An instruction may mean a statement if the program is represented in a high level programming language, or it may mean an assembly level instruction if a low level representation is chosen. We will first look at the structures in a data dependence graph that will limit the maximum speedup of parallel executions of a loop. We will define what is a maximum computation rate of a given loop. We will review the previous results concerning the maximum computation rate of a loop. Then we introduce the periodic scheduling scheme for loops, which can achieve the maximum computation rate. This is the basis of the scheduling scheme for which our register allocation scheme will support. The general simultaneous scheduling and register allocation problem is formally studied in Chapter 3 and Chapter 4.

We then discuss the superscalar and VLIW architecture models as representative target architectures on which our scheduling and register allocation scheme is most useful. Dataflow architectures will be formally introduced later in Chapter 5.

We will also introduce the basics of linear and integer programming which we will use extensively in the later chapters to solve our optimization problems.

## 2.2 Notations on Sets

A set is collection of elements which have some common features. If an element $a$ belongs to a set $A$, then we say, $a$ is in $A$ or $A$ contains $a$, denoted by $a \in A$. We use the notation, $A \in a$, to indicate all the sets $A$ containing $a$. We will use $Z$ to indicate the set of integers, i.e.

$$Z = \{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\}.$$

The set of positive integers is indicated by $Z_+$:

$$Z_+ = \{1, 2, 3, \cdots\}.$$

The following operations are defined on sets:

**Union** $A \bigcup B$ of two sets $A$ and $B$:

$$A \bigcup B = \{a; a \text{ belongs to } A \text{ or belongs to } B.\ \}$$

**Intersection** $A \bigcap B$ of two sets $A$ and $B$:

$$A \bigcap B = \{a; a \text{ belongs to } A \text{ and also belongs to } B.\ \}$$

**Cardinality** $|A|$ of a finite set $A$ is the number of elements contained in it:

$$|A| = \text{ number of elements in } A.$$

If a set contains an infinite number of elements, then its cardinality is simply defined as infinity.

## 2.3 Graph Theoretic Terminology

We will use directed graphs to represent the data dependences of computer programs. When graphs are used to represent the real world applications, they are often given

special names. This is also true in this thesis. If graphs are used to represent data dependences, then they are called data dependence graphs (DDG). When graphs are used to represent data flow computations, they are called dataflow graphs.

In this section we give the basic definitions about the general graphs. Data Dependence Graphs (DDG) and dataflow graphs will be introduced later. Most of the graph theoretic terminology has been adapted from [58].

**Definition 2.3.1** *A directed graph or multigraph $G = (N, A)$ consists of a set $N$ of nodes and a set $A$ of arcs, where $N = \{n_1, n_2, \cdots, n_{|N|}\}$ and $A = \{e_1, e_2, \cdots, e_{|A|}\}$. Each arc $e_j$ consists of two nodes: $e_j = (n_h, n_k)$, where node $n_h$ is called the tail of $e_j$, and node $n_k$ is called the head of $e_j$. The direction of the arc $e_j = (n_h, n_k)$ is from the tail $n_h$ to the head $n_k$. Multiple arcs between a pair of nodes are possible. It is also possible that the tail and the head of an arc are the same node. In that case the arc is called a self-loop.*

The definition of directed graphs will be refined to accommodate to our applications. Nodes and arcs can be annotated with one or more labels to carry some physical information. However we will also use undirected graphs in later chapters, mainly in the proofs and transformations in order to obtain the solutions. So we give the definition of undirected graphs here. In the following definition we use the words "vertex" and "edge" to distinguish them from the directed graph case.

**Definition 2.3.2** *A undirected graph $H = (V, E)$ consists of a set $V$ of vertices and a set $E$ of edges, where $V = \{v_1, v_2, \cdots, v_{|V|}\}$, and $E = \{e_1, e_2, \cdots, e_{|E|}\}$. Each edge $e_j$ consists of two vertices: $e_j = (n_h, n_k)$, where $n_h, n_k$ are called the end vertices of edge $e_j$. There is no direction associated with the edge $e_j$.*

If we do not mention explicitly whether a graph is directed or undirected, then we mean it is directed. When we use undirected graphs we will always have the adjective "undirected" before the word "graph".

In practice, we often have additional information associated with the nodes and arcs of the graph. We called such graphs with additional information weighted graphs.

**Definition 2.3.3** *A weighted graph $G = (N, A; w_1, w_2, \cdots)$ consists of a graph $(N, A)$ and one or more labels $w_1, w_2, \cdots$ defined on the nodes and/or the arcs.*

Similar situation also applies to undirected graphs. Next we give the definitions of some common structures in graphs.

**Definition 2.3.4** *Given a graph $G = (N, A)$.*

- *A path $P$ in $G$ is a sequence of arcs: $\{e_{j_1}, e_{j_2}, \cdots, e_{j_k}\}$, such that the head of $e_{j_h}$ is the tail of $e_{j_{h+1}}$, for $h = 1, \cdots k-1$. The tail of $e_{j_1}$ is also called tail of the path and the head of $e_{j_k}$ is called the head of the path.*
- *A path is called a cycle if the tail and the head of the path are the same. The set of all cycles in $G$ is denoted by $C(G)$.*
- *Given an arc $(n_i, n_j) \in A$, $n_i$ is called an immediate predecessor of $n_j$. Similarly, node $n_j$ is called an immediate successor of $n_i$.*
- *For any given node $n_i$, we use $\delta^+(n_i)$ to indicate the set of out-going arcs from $n_i$, i.e.*

$$\delta^+(n_i) = \{(n_i, n_j);\ \textit{such that } (n_i, n_j) \in A\}.$$

*We use $\delta^-(n_i)$ to indicate the set of in-coming arcs of $n_i$, i.e.*

$$\delta^-(n_i) = \{(n_j, n_i);\ \textit{such that } (n_j, n_i) \in A\}.$$

- *Given a node $n_i$, its out-degree is defined to be the number of arcs in $\delta^+(n_i)$, and its in-degree is defined to be the number of arcs in $\delta^-(n_i)$.*

## 2.4 Loop Model

In this thesis we focus on the class of inner-most *do loops*, or *while* loops that can be transformed to do loops, containing no conditional tests in the loop body. Several

techniques have been invented to eliminate conditional tests at compile time. For instance, [5] proposed a method to convert control flow dependences into dataflow dependences. Hardware supported schemes also exist that uses "predicated" instructions [71, 72] to allow a compiler to schedule a conditional as a non-conditional, and nullify the untaken branch at run-time. Hierarchical reduction [56] is an approach that collapses a conditional test and its branches into a single node so that it represents the longest path for the collapsed structure. Therefore our focus of loops does not limit generality of our methods presented in this thesis.

We have chosen a high level representation of the loops for the purpose of easy illustration. The methods developed in this thesis can also be applied to intermediate or lower level representations, such as three address code or assembly code etc. Section 4.10 gives an example in assembly level representation to show our method. The generic form of the loops under consideration is:

**for** $i = 1$ **to** $U$ **do**
$S_1$
$S_2$
$\vdots$
$S_q$
**enddo**

We assume that the generic form has the following properties:

1. Each $S_j$ is an assignment instruction of the form:

$$S_j : \quad x = E$$

where $x$ is a variable and E an expression. Variable $x$ can be either a scalar variable or an element of an array variable. Expression $E$ does the arithmetic, logic, relational and other simple operations (like shifting bits, etc). In a low level representation, $E$ is an instruction which involves at most two operands. So it is one of the forms: "*opcode operand1*" or "*operand1 opcode operand2*",

where *operand1* and *operand2* could be either scalar variables or elements of an array. Therefore all the following are possible formats of instructions in our model:

$$x = -y$$
$$a[i] = y$$
$$x = a[i]$$
$$x = y + z$$
$$x = y * a[i]$$
$$x = a[i] - b[i-1]$$
$$a[i] = x + y$$
$$a[i] = x * c[i-3]$$
$$a[i] = b[i] * c[i-2]$$

2. The loop bounds need not to be constants. We assume that the number of iterations of a loop is unbounded in this thesis to simplify the problem representation.

3. The loop may contain loop-carried dependences [6]. Loop-carried dependences are those that across iterations. They make the situation more difficult for the scheduling of instructions across iterations. We will show later that if the loop-carried dependences form dependence cycles, then the maximum computation rate is bounded by a parameter determined by these cycles.

4. The data dependences between instructions are only flow-dependences. Other kinds of dependences like anti-dependences and output-dependences are caused by memory contention and can be eliminated by renaming techniques [10]. It should be pointed out that renaming of array variables may involve copying overhead. However our techniques in this these can be easily extended to handle anti- and output-dependences. In the literature, flow-dependences are also called the true dependences, which represent the data (or information) flow

along the computational paths. In this thesis we simple call the flow dependence "data dependence" or "dependence", whenever it is clear from the contents. The terminology we use is adapted from [10].

5. We assume that any of the dependence distances [10] between any pair of instructions is constant (i.e. independent of the iteration index $i$). This restriction is necessary for a compile time static scheduler. On the other hand, if some dependences are not independent of the iteration index, we can take the conservative approach and assume the shortest distance as a constant for all iterations.

A typical loop confined to the above properties is shown in (2.1):

| | |
|---|---|
| $L$ : | **for** i = 1 **to** 100 **do** |
| $s_1$ : | $X = X + c[i-1];$ |
| $s_2$ : | $a[i] = X + b[i-2];$ |
| $s_3$ : | $b[i] = a[i] * F;$ |
| $s_4$ : | $c[i] = a[i]/b[i];$ |
| | **enddo;** |

(2.1)

In iteration $i$, instruction $s_1$ reads two operands $X$ and $c[i-1]$. Both of them are produced in the previous iteration. Actually $X$ is produced by $s_1$ itself in the previous iteration. Therefore there is a data dependence from $s_1$ to itself, and the dependence distance is one iteration. The other operand $c[i-1]$ is produced by $s_4$ in the previous iteration. Therefore there is a data dependence of distance 1 from $s_4$ to $s_1$.

Now let us consider instruction $s_2$. Its two operands are $X$ and $b[i-2]$. The first operand $X$ comes from $s_1$ in the same iteration. Therefore there is a data dependence from $s_1$ to $s_2$ with dependence distance 0. The other operand $b[i-2]$ comes from $s_3$ in iteration $i-2$. Hence there is a data dependence from $s_3$ to $s_2$ with dependence distance 2.

There are many research results on how to automatically detect the data dependence information for general loops for a compiler [10, 79, 80]. It is not our goal to describe these methods here. We only assume that some tool can provide us such dependence information. In the next section we will introduce the data dependence graphs to represent the loops in abstract structures.

## 2.5 Data Dependence Graphs

When considering the scheduling and register allocation problems, the actual computation performed is not important. Only the data dependence and delay information of the instructions are important to us. For a program representation, we choose to use the *data dependence graph* (DDG) [10] annotated with dependence distance and instruction delay information, since it is simple and contains enough information for our scheduling and register allocation purpose.

**Definition 2.5.1** *Given a generic loop, the corresponding Data Dependence Graph (DDG) is a weighted directed graph* $G = (N, A; m, d)$, *with the following interpretations:*

1. $N$ *is the set of nodes, each representing an instruction in the loop body.*
2. $A$ *is the set of arcs, each representing a data dependence between a pair of instructions. That is to say, if node* $n_k$ *reads an operand produced by another node* $n_h$, *then the arc* $(n_h, n_k)$ *is in* $A$.
3. $m = \{m_{hk};\ (h,k) \in A\}$ *is the dependence distance vector defined on the arc set* $A$, *such that* $m_{hk}$ *is a nonnegative integer which indicates the iteration distance of the dependence* $(h,k)$. *If the dependence* $(h,k)$ *does not cross iterations, then* $m_{hk} = 0$. *If* $m_{hk} > 0$ *then* $(h,k)$ *is called a loop-carried dependence.*
4. $d = \{d_h;\ h \in N\}$ *is the delay vector of instructions defined on the node set* $N$. $d_h$ *is the number of clock cycles needed to complete one execution of the instruction*

*h. Although the delay vector is given without specifying an architecture, it is assumed that the functional units are pipelined and are hardware hazard-free.*

The DDG of the example loop $L$ in (2.1) in Section 2.4 is shown in Figure 2.1. We assume in this thesis that the delay of (floating point) Addition is 1 clock cycle, the delay of Multiplication is 2 and the delay of Division is 17.

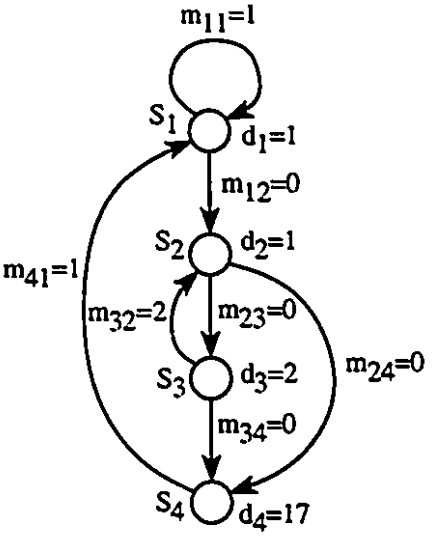



Figure 2.1: Data dependence graph of the example loop $L$.

Although the delays are associated with the nodes of the DDG, it is easy to generalize them to the arcs. One can simply define the delays of the output arcs of a node $n_i$ to be the delay of the node $n_i$.

**Definition 2.5.2** *Given a DDG $G = (N, A; m, d)$, we define a new delay vector $l$ on the arc set $A$:*

$$l_e = d_h, \text{ if } e = (h, k) \in A.$$

Very often we will consider the total delay and total dependence distance along a path or a cycle in the DDG. Now we give some short hand notations for them.

**Definition 2.5.3** *Given a DDG $G = (N, A; m, d)$. Let $P$ be a path or a cycle in $G$.*

- *We define the delay $D(P)$ of $P$ to be the sum of the delays of all the nodes in $P$:*

$$D(P) = \sum_{h \in P} d_h.$$

- *We define the dependence distance $M(P)$ of $P$ to be the sum of the dependence distances of all the arcs in $P$:*

$$M(P) = \sum_{(h,k) \in P} m_{hk}.$$

## 2.6 Computation Rate

A schedule of instructions is a function from the domain of instructions of a program to the integers representing time clock cycles. If a computer architecture has no limitation on the number of processing units and other resources, then the instructions can be scheduled as early as possible subjected only to data dependences in the program. Furthermore, if there are no data dependence cycles, then all the instructions of all the iterations can be scheduled in parallel and finished within $L$ clock cycles, where $L$ is the length of the longest dependence path of the DDG, which is independent of the iteration bounds. On the other hand if data dependences form cycles in the DDG, then subsequent iterations can only be scheduled with a certain delay from the previous iterations. Therefore instructions can only be scheduled at a certain finite rate. Different instructions may have a different computation rate depending on the cycle or cycles in which they are located. However when the throughput of iterations is considered, the rate is determined by the slowest rate of the instructions in the loop body, because an iteration is considered finished only when all the instructions in it are finished. Now we give the formal definition of the computation rate of an instruction and of the whole loop.

**Definition 2.6.1** *Given a schedule of a loop, the computation rate of an instruction $h$ is the average number of executions over one unit of time observed during a long*

*period of time. The computation rate R of a loop is the slowest of the computation rates of the instructions in its loop body. The maximum computation rate of the loop is the maximum of computation rates obtainable over all feasible schedules.*

Reiter [73] proved a theorem about the maximum computation rate of any given loop. Before we state the theorem, we give some new definitions to simplify the notations.

**Definition 2.6.2** *Given a DDG $G = (N, A; m, d)$. Let $C$ be a cycle in $G$: $C \in \mathcal{C}(G)$. The balancing ratio $B(C)$ of $C$ is defined by:*

$$B(C) = \frac{M(C)}{D(C)} = \frac{\sum_{(h,k)\in C} m_{hk}}{\sum_{h\in C} d_h}.$$

*A cycle $C^*$ in $G$ is called critical if it has the smallest balancing ratio among all the cycles in $G$, i.e.*

$$B(C^*) = \min\{\frac{M(C)}{D(C)},\ \forall C \in \mathcal{C}(G)\}$$

Now we are ready to give the maximum rate theorem by Reiter.

**Theorem 2.6.1 (Reiter 1968 [73])** *The maximum (achievable) computation rate $R$ of a given loop is equal to the balancing ratio of the critical cycle(s) in the DDG of a loop, i.e.*

$$R = \min\{\frac{M(C)}{D(C)},\ \forall C \in \mathcal{C}(G)\}$$

## 2.7 Scheduling Schemes

Instruction scheduling for loops exploits the following special properties:

- The instructions are going to be executed repeatedly.

- The dependence relations do not change from iterations to iterations.

These properties allow the scheduler to have more opportunity to find optimal schedules by exploiting the repeating behavior. In this sense, scheduling problems for loops are different from the traditional scheduling theory which focuses on single pass schedules [19, 18, 12].

Cytron [21] proposed the do-across method, which tried to find a minimum constant delay between consecutive iterations so that subsequent iterations may start before the previous iterations finish. However, Cytron's do-across assumed a fixed sequential ordering of the instructions in an iteration, which limited the exploitation of parallelism considerably.

Aiken and Nicolau [3, 4] developed a scheduling scheme called percolation. Percolation is a set of rules to move instructions in the dataflow or control flow graph so that the semantics are preserved. They applied percolation to the inner-most loops to obtain parallel scheduling. The algorithm emulates the execution of the loop by virtually unrolling the loop unbounded number of times, and schedules the instructions as early as possible, until a periodic pattern can appear, which often consists of instructions from a number of consecutive iterations. This technique is called *OPTimal loop parallelization (OPT)* (a particular case of *Perfect Pipelining*). However there is a problem with this approach: for loops with multiple *critical cycles*, no polynomial time bound is known to occur for such a periodic pattern, caused by the reason that the patterns may be exponential in size under the earliest firing rule [67]. Furthermore, the register allocation problem was not considered either.

Ebcioglu et al [29, 28, 30] proposed several refinements based on Aiken and Nicolau's algorithms. However their scheduling scheme was still heuristic and the register allocation problem was not considered.

Lam [56] proposed a scheduling scheme which used the name *software pipelining*. In the method, both the data dependences and the number of processing units were considered as fixed parameters. So Lam claimed the scheduling problem being NP-complete. We notice here that if the number of processing units are assumed infinite,

then the scheduling problem is not NP-complete and an optimal schedule can be found in polynomial time [73, 77].

Hence, in this thesis, we will consider a scheduling scheme, in which only data dependences are considered as constraints. This assumption allows us to achieve optimal speedup allowed by a program. We will investigate that under this assumption, how many functional units and registers are enough to support most of the typical loops in benchmarks. Furthermore Lam's scheduling method did not take into account the register allocation problem, while ours will.

Given a loop with its DDG G=(N, A; m, d), defined in Section 2.5, let us examine what is the minimum requirement for a schedule to be feasible in the sense of not violating any dependence relations.

**Definition 2.7.1** *For a given DDG G=(N, A; m, d), we use $t_h(i)$ to indicate the time when node h in iteration i is scheduled. Then a schedule t is feasible if and only if the following is true:*

$$t_h(i) + d_h \leq t_k(i + m_{hk}),\ \forall (h,k) \in A.$$

That is to say, because of the dependence distance $m_{hk}$, the consumer node $k$ in iteration $i + m_{hk}$ can only be scheduled after the producer node $h$ in iteration $i$ finishes. Many kinds of schedules can be feasible, but not all of them are easy to work with from the perspective of a compiler. In the following we introduce the periodic scheduling scheme that shows strong regularity and can achieve optimal rate.

Intuitively a scheduling for a loop is periodic if all the instructions in all iterations are scheduled with a fixed period. Formally we give the following definition.

**Definition 2.7.2** *A schedule for a given loop is called periodic with period P if for any two consecutive iterations i and $i + 1$, we have*

$$t_h(i+1) - t_h(i) = P,\quad \forall h \in N.$$

Hence for a periodic schedule, the second iteration is just a repeat of the first iteration after $P$ clock cycles, and the third iteration is a repeat of the first iteration after $2P$ clock cycles, etc. This means that the timing of the instructions in the first iteration plus the constant period $P$ will determine the whole schedule. Therefore a periodic schedule is very simple to describe and still powerful enough to satisfy the requirement of producing a good parallel code at compile time, which is addressed in Section 4.6 in Chapter 4. This is a main reason why we choose periodic scheduling as our starting point of doing register allocation.

For the reason of easy reference, we give the following definition:

**Definition 2.7.3** *Let $t$ be a periodic schedule of a loop. We use $t_h$ to indicate the scheduling time of instruction $h$ in the first iteration. With this notation, instruction $h$ in iterations 2, 3, ... are scheduled at clock cycles $t_h + P, t_h + 2P, \cdots$, where $P$ is the period.*

With the above notation, we list the following properties of a periodic schedule:

1. If the period is $P$, then the computation rate is $\frac{1}{P}$. Therefore the minimum period corresponds to the maximum computation rate.

2. The schedule is feasible according to Definition 2.7.2 if and only if

$$t_h + d_h \leq t_k + m_{hk} \cdot P, \quad \forall (h,k) \in E. \tag{2.2}$$

## 2.8 Architecture Models

In this section we outline our target computer architectures on which our techniques are most useful. These include two classes of architectures: one class represented by the *Very Long Instruction Word* (VLIW) and *Superscalar* architectures. The other class is dataflow architectures which will be introduced in Chapter 5. We will

only give very brief and high level descriptions of these architectures. Details of the architectures that are not relative to our scheduling and register allocation problems are not mentioned.

Both VLIW and Superscalar architectures [34, 51, 57, 52] are designed to exploit parallelism at the fine-grain instruction level. Their main difference is that superscalar architectures assume a sequential machine level language and depend on a dynamic scheduler to find the parallelism in a window of instructions. VLIW machines need very wide instructions each of which contains independent and parallel subinstructions. With this difference, the superscalar architecture can run codes compiled for a sequential machine without any change, while for a VLIW architecture the sequential code has to be recompiled by a compiler that can generate Very Long Instructions. However, since a superscalar machine uses a dynamic scheduler which can only look at a small window of instructions to find parallel instructions, it is conceivable that a VLIW machine can exploit more parallelism by advanced compiling techniques. A good compiler on a superscalar machine should also exploit parallelism so that the dynamic scheduler can find parallel instructions easily or trivially in consecutive sequence of instructions.

Now we describe the two architectures in very brief terms, but the description will be enough for our scheduling and register allocation scheme.

**VLIW Architecture [34, 51]** : A Very Long Instruction Word (VLIW) architecture has several, say p, tightly coupled processing units. Each processing unit may be pipelined. We assume that the pipeline has no hardware structural hazard. All the processing units are equivalent in execution functions. All processing units share a common main memory and a common register file and probably a common data cache and/or a common instruction cache. At any given time unit, a Long Instruction (LI) containing at most p instructions can be fetched and the p instructions can be processed simultaneously on the p processing units. Each instruction can be considered equivalent to be an assembly instruction on a sequential von Newmann machine. The Instruction Set

is RISC like and the only instructions which can access data from and to the main memory are LOAD and STORE. Other instructions take their operands from registers.

**Superscalar Architecture [52]** : A Superscalar architecture is very much like a VLIW architecture if only hardware organization is considered, i.e., it has p functional units, each of them can be pipelined. All of the functional units share a common memory, a common register file, a common instruction cache and a common data cache. However there are no Long Instructions in superscalar machines. Instructions on a superscalar architecture may be considered equivalent to the instructions on a sequential von Newmann architecture machine. A dynamic scheduler unit is used to select parallel instructions. At any given time unit, the dynamic scheduler will look at a window of instructions and choose among them at most p instructions so that they can be processed simultaneously by the p processing units. As in a VLIW architecture, a superscalar architecture also has a RISC like instruction set, in which the only instructions which can access data from the main memory are LOAD and STORE.

To simplify the problem, we choose an idealized architecture in which the number of processing units is assumed to be infinite. With this idealized assumption, we can always achieve maximum computation rate allowed by the data dependences of the loops since hardware puts no limits on the speed. If the number of processing units are assumed fixed at the beginning, just to determine the maximum possible computation rate would be NP-complete, which will complicate our problem tremendously. If the code, generated with the idealized assumption, can not be directly used on a fixed processor machine, mapping techniques are needed to convert an idealized code to the real machine.

## 2.9 Linear and Integer Programming

We briefly review some of the main results in the theory of linear and integer programming in this section. Detailed theory can be found in the excellent books by Chvatal [17] and Schrijver [74]. The following optimization problem with a linear objective function and a set of linear constraints is called a linear programming problem, where $c_i$, $b_j$, $a_{ij}$, $i = 1, \cdots, n$, $j = 1, \cdots, m$ are rational numbers.

$$\min \; c_1x_1 + c_2x_2 + \cdots + c_nx_n \tag{2.3a}$$

subject to

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \geq b_1, \tag{2.3b}$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \geq b_2, \tag{2.3c}$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \geq b_m, \tag{2.3d}$$

$$x_i \geq 0, \; \forall i \in [1..n] \tag{2.3e}$$

We can use vectors and matrices to obtain a more compact form of the linear programming problem:

$$\min \; cx$$

subject to

$$Ax \geq b$$

$$x \geq 0$$

The linear programming problem in (2.3) or its compact form is called the primal problem. Correspondingly it has dual problem:

$$\max \; b_1y_1 + b_2y_2 + \cdots + b_my_m \tag{2.4a}$$

subject to

$$a_{11}y_1 + a_{21}y_2 + \cdots + a_{m1}y_m \leq c_1 \tag{2.4b}$$

$$a_{12}y_1 + a_{22}y_2 + \cdots + a_{m2}y_m \leq c_2 \tag{2.4c}$$

$$\vdots$$

$$a_{1n}y_1 + a_{2n}y_2 + \cdots + a_{mn}y_m \leq c_n \tag{2.4d}$$

$$y_j \geq 0, \ \forall j \in [1..m] \tag{2.4e}$$

or in the compact matrix form:

$$\max \ yb$$

subject to

$$yA \leq c$$

$$y \geq 0$$

It is easy to see that the dual problem (2.4) is also a linear programming problem. If we try to write down the dual of (2.4), we obtain the original problem (2.3). This duality of linear programming plays a very important role in its solutions. The following theorem states the relationship between the primal and the dual.

**Theorem 2.9.1** *The primal linear programming (2.3) has a finite optimal solution if and only if the dual (2.4) has a finite optimal solution. If the primal-dual pair has optimal solutions, then their objective values are equal.*

There are many algorithms for solving linear programming problems. They can be basically divided into three categories. One category is the simplex method (see for example [17]). The method does not give a polynomial time algorithm. Theoretically it is an exponential algorithm in the worst case. But in practice the speed is very fast for the majority of application problems. Another category is the ellipsoid method invented by Khachian [55]. This method gives the first polynomial time algorithm

for solving linear programming problems. But the complexity of the algorithm is quite high, and its speed is not at all comparable with the simplex method. The last category is the interior point method (also called projective method) first invented by Karmarkar [54]. This method also gives a polynomial time algorithm for solving linear programming problems. The complexity of the method is better than that of the ellipsoid method. Its practical computation speed is much better than the ellipsoid method. However, the simplex method is still much faster than the other two methods for small size problems, which include the problems we are dealing in this thesis.

On the other hand, real application problems often show special structures which may make them easier to solve than the general linear programming problem. Minimum cost flow is such a special class of applications for which much faster algorithms ($O(n^3)$ or better) exist. Another special property of the constraint matrix of the minimum cost flow problem is that if the cost coefficients $c$ and the bounds $b$ in (2.3) are all integers, then an integral optimal solution $x$ can always be found. This is particularly helpful if integer solutions are preferred. We will introduce the property which the constraint matrix of the minimum cost flow problem has in the next section.

An integer programming problem is a linear programming problem (2.3) plus the requirement that the variables $x_i$ must be integers. Integer programming problems are usually much harder than their linear versions. The general integer programming problem is NP-complete. However some particularly structured problems do have polynomial time solutions, like the minimum cost flow problem. For hard integer programming problems, one of the heuristic approaches is first to solve the linear version of the problem, then try to round the obtained fractional solution to integers. Often the linear version of an integer programming problem is called the linear relaxation problem, because it relaxes the integer requirement on the variables. In the next section we introduce a property which can guarantee a linear programming solution to be integral, therefore is very important to solve integer programming problems.

## 2.10 Totally Unimodular Matrices

In this section we introduce *the total unimodularity* property on matrices which has a very important role in solving integer programming problems.

**Definition 2.10.1** *A rational matrix $A$ is totally unimodular (TUM) if the determinant of each square submatrix of $A$ is either 0, or +1, or -1.*

Since each entry of the matrix $A$ can be considered as a square submatrix, a immediate consequence is that each entry of the matrix $A$ must be either 0, or +1, or -1.

The following theorem illustrates the importance of the TUM property. It implies that if a constraint matrix is TUM, it suffices to solve the linear programming to find an integer valued solution.

**Theorem 2.10.1** *Let $A$ be a totally unimodular matrix, and let $b$ and $c$ be integral vectors. Then both the primal and dual linear programming problems*

$$\min\{cx|Ax \geq b, x \geq 0\} = max\{yb|yA \leq c, y \geq 0\}$$

*have integral optimum solutions.*

However only the simplex method can guarantee to find such an optimal integral valued solution. Neither the ellipsoid method nor Karmarkar's method can guarantee to produce an integral valued optimal solution even if the constraint matrix is TUM, because the solutions obtained from these two methods are usually not the vertices of the polyhedron defined by the constraints.

# Chapter 3

# NP-Completeness Results

In this chapter we prove that the minimum register allocation problem to support a time-optimal parallel schedule among all such schedules, even on an idealized parallel computer architecture, is NP-complete. The result holds for both the acyclic case (straight-line codes) and the cyclic case (loops).

## 3.1 Introduction

In this chapter we prove that the minimum register allocation problem to support a time-optimal parallel schedule among all such schedules, even on an idealized parallel computer architecture, is NP-complete. The result holds for both the acyclic case (straight-line codes) and the cyclic case (loops).

There are many versions of register allocation problems, both for sequential and for parallel computer architectures [45]. In [75], Sethi proved that the minimum register allocation problem while allowing instruction reordering for an acyclic graph on a sequential computer architecture is NP-complete. Garey et al [46] proved an NP-completeness result for a loop with a fixed sequential schedule. In this chapter we consider the register allocation problem for a parallel architecture without fixing a particular schedule. That is to say, we want to find an optimal feasible parallel schedule which uses minimum number of registers among all such schedules. Our results in this chapter generalize the results mentioned above. We will prove the following two NP-completeness results:

1. The minimum register allocation problem to support an optimal schedule for a program with an acyclic DDG representation on an idealized parallel computer architecture is NP-complete.

2. The minimum register allocation to support an optimal rate schedule for a loop with a cyclic DDG representation on an idealized parallel computer architecture is also NP-complete.

In the next section we give the problem definition and prove an NP-completeness result for straight-line code to prepare for the result on loops. Then in Section 3.3, we prove our main result of the chapter that simultaneous optimal scheduling and register allocation is NP-complete.

## 3.2 Case of Acyclic DDG

Let us first state the register allocation problem as a decision problem. Then we formulate a more restricted version. We will actually prove the restricted version of the problem is NP-complete.

---

**Parallel Register Allocation Problem (PRAP):**

**Given:** An acyclic DDG $G = (N, A; m = 0, d = 1)$ representing a straight line code. Assume that there are infinite number of processing units. Let $R$ be the number of available registers.

**Question:** Let $L$ be length of the longest path of G. Is there a schedule of G which finishes within $L$ clock cycles and uses at most $R$ registers?

---

The restricted version of the problem is to limit the DDG to have only one source node and one sink node. This is formally stated below.

---

**Restricted Parallel Register Allocation Problem (R-PRAP):**

**Given:** A acyclic DDG $G = (N, A; m = 0, d = 1)$ representing a straight line code in which it has only one node having no input arcs and only one node having no output arcs. Assume that there are infinite number of processors. Let $R$ be a given positive integer.

**Question:** Let $L$ be the length of the longest path of G. Is there a schedule of G which finishes within $L$ clock cycles and uses at most $R$ registers?

---

**Theorem 3.2.1** *R-PRAP is NP-complete. Therefore PRAP is also NP-complete.*

**Proof:** We show that the *Vertex Cover* problem [45] can be reduced to R-PRAP. Let us first state the definition of the Vertex Cover problem.

**Vertex Cover problem (VC):**

**Given:** A undirected graph $H = (V, E)$ where $V$ is the set of vertices and $E$ the set of edges in $H$. Let $K$ be a positive integer.

**Question:** Is there a vertex cover $V' \subset V$ of G whose size is exactly $K$, i.e., $|V'| = K$, such that $\forall e \in E$, at least one of the two end vertices of $e$ is in $V'$?

Let the given undirected graph $H$ have the following sets of vertices and edges:

$$V = \{v_1, v_2, \cdots, v_n\},$$

$$E = \{e_1, e_2, \cdots, e_m\},$$

where an edge $e_k = (v_i, v_j)$ for some indices $i$ and $j$. Here we assume that there are $n$ vertices and $m$ edges in $H$. We also assume that there are no isolated vertices in the given undirected graph $H$ because they can be very easily treated.

Next we construct an instance of the R-PRAP from the given instance of the VC. We will adopt some convention on the terminology. In the VC problem we are given an undirected graph, so we use the words *vertex* and *edge* to describe the objects for the instance of VC. We use the words *node* and *arc* for the instance of R-PRAP since we have a directed graph.

In our construction, we will design a component for each vertex and a component for each edge. We will call them vertex components and edge components. They will be connected together (serially) to get a larger construct. Then we will append other structures to it to form the whole instance of R-PRAP. In the following we often use the term $a$-nodes to indicate the set of nodes labeled with $a$, etc.

We will first describe all the components individually: initialization component, vertex components, edge components, control $c_2$-$c_3$-$c_4$ component and bookkeeping

chain. Then we will formally define their internal structures and show how they are connected.

The initialization component is shown in Figure 3.1. It consists of a starting node $s$ of the whole graph, $7n + 2m$ initialization nodes:

$$\{r_1, \cdots, r_{2n}\}, \{cz_1, \cdots, cz_n\}, \{cv_1, \cdots, cv_n\}, \{cu_1, \cdots, cu_n\}, \{d_1, \cdots, d_n\}, \{f_1, \cdots, f_n\},$$

$$\{g_1 \cdots, g_m\}, \{h_1, \cdots, h_m\},$$

and a control node $c_1$. Starting node $s$ is connected *to* each of the initialization nodes and control node $c_1$ is connected *from* each of the initialization nodes. The initialization nodes will also be connected to other nodes in the other components, which will be shown later. Control node $c_1$ will be connected to the starting node $s_1$ of the first vertex component described next.

For each vertex $v_i$, the corresponding vertex component is shown in Figure 3.2. In the figure, the shaded nodes should not be considered as part of the vertex component, they are designed for initialization purpose. There are also other unshown arcs from initialization $g$-nodes and $h$-nodes to the $v$-node and $u$-node in the vertex component. The unshown initialization $g$-nodes and $h$-nodes are designed for the edge components and they are in 1-to-1 correspondence with the edge components. They will be shown when we give the construction of the edge components later, while each shown initialization node in the vertex component is design particularly for this vertex component.

The $s_i$ node is called the *starting* node of the component and node $w_i$ is the *ending* node of the component. Nodes $v_i$ and $u_i$ both correspond to the vertex $v_i$ in the VC instance. Later we will see that at most one of the two nodes $v_i$ and $u_i$ can be scheduled during the first phase of the vertex component scheduling. The case that $v_i$ is scheduled in the first phase corresponds to the case where vertex $v_i$ is in the vertex cover. The case that $u_i$ is scheduled in the first phase corresponds to the case where vertex $v_i$ is not in the vertex cover. We also call the set of nodes $\{a_{i,1}, \cdots, a_{i,n-i+1}\}$ the $a$-branch. Similarly the set $\{b_{i,1}, \cdots, b_{i,n-i+1}\}$ is called the $b$-branch.

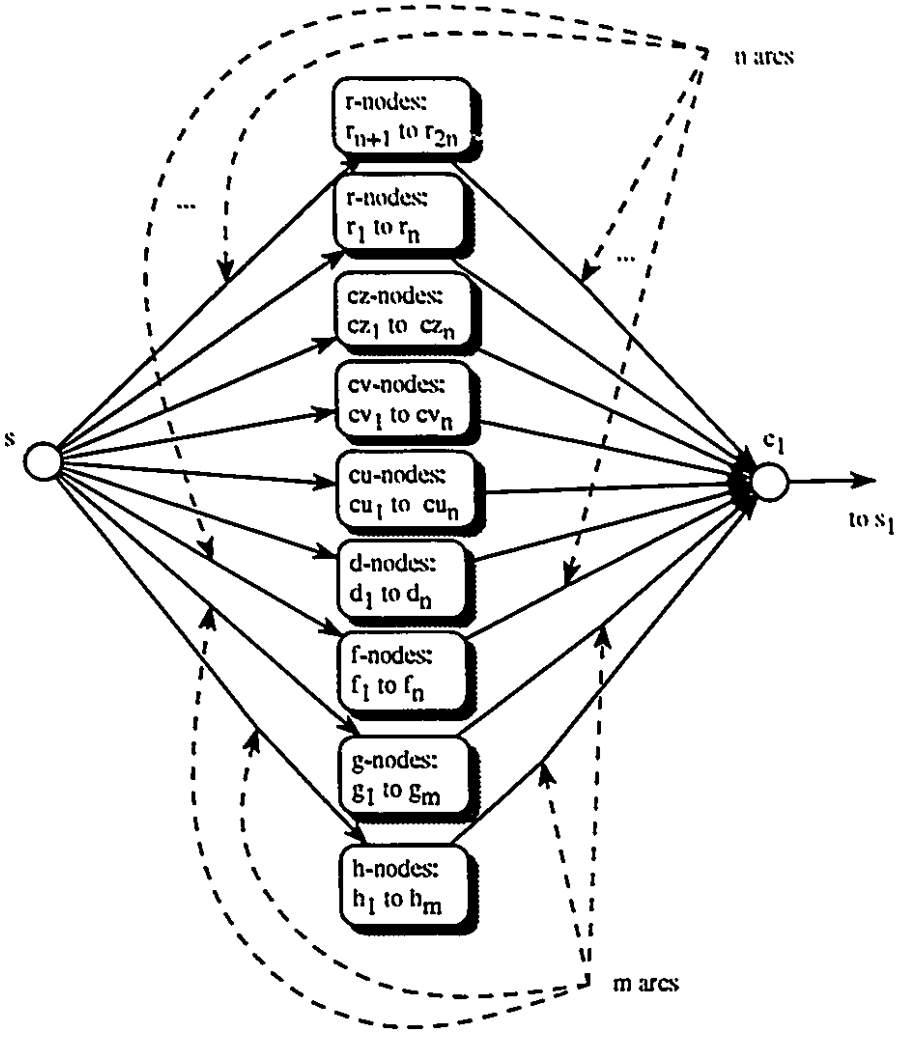



Figure 3.1: The initialization component.

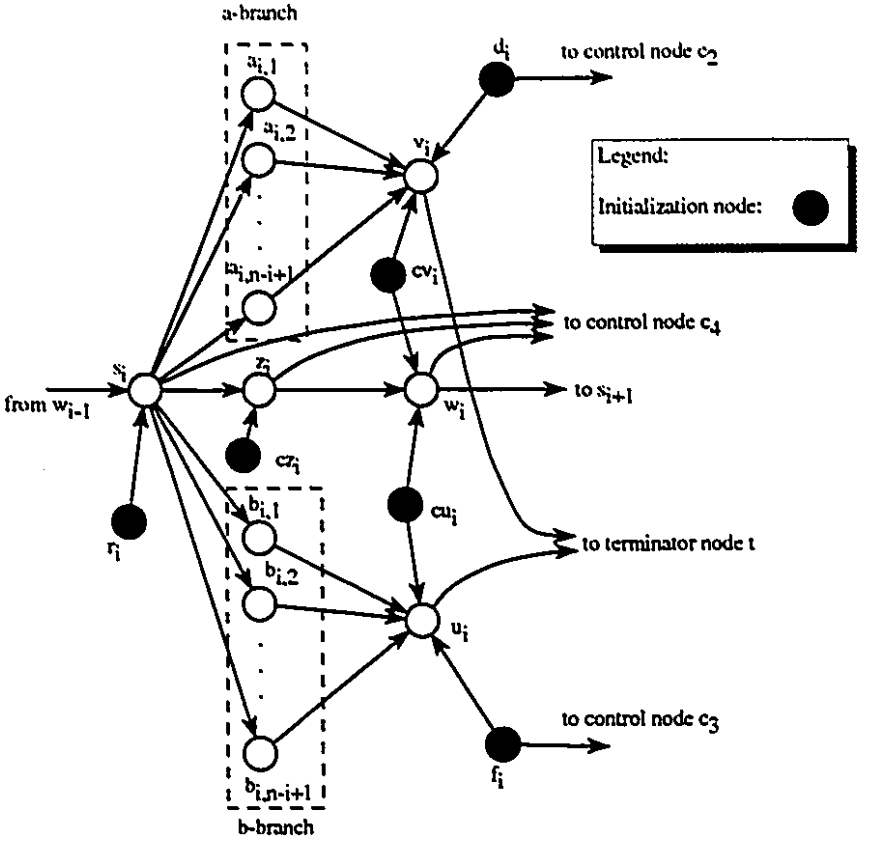



Figure 3.2: Vertex component for vertex $v_i$.

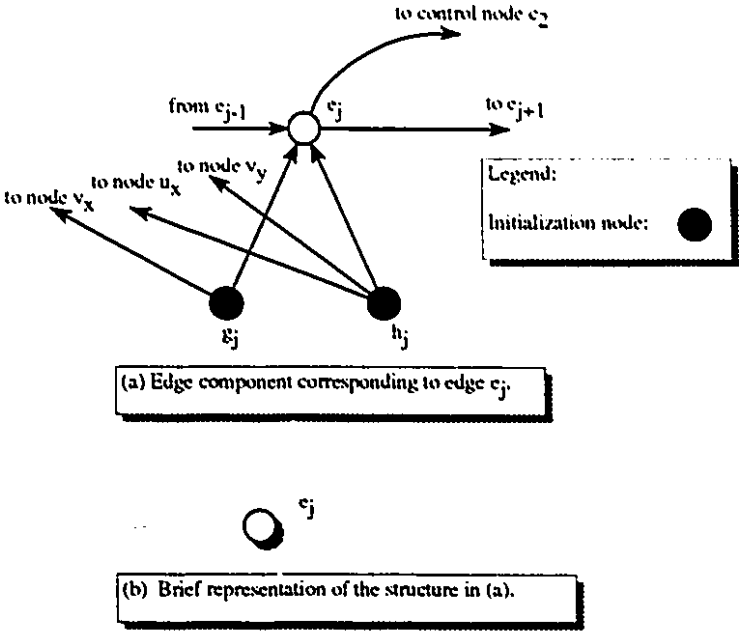



Figure 3.3: Edge component for edge $e_j$.

For each edge $e_j = (v_x, v_y)$ in the instance of VC, its corresponding edge component is shown in Figure 3.3. The two nodes $g_j$ and $h_j$ are connected to the nodes in the corresponding vertex components of the two end-vertices $v_x$ and $v_y$ of edge $e_j$. There are three such arcs, one from $g_j$ to $v_x$: $(g_j, v_x)$ and two from $h_j$ to $u_x$ and $v_y$: $(h_j, u_x), (h_j, v_y)$. They are so connected to ensure that exactly one of $g_j$ and $h_j$ can free its register, i.e. $e_j$ is the last unscheduled successor of either $g_j$ or $h_j$, when we schedule node $e_j$ in the edge component.

Then we connect the vertex components and edge components sequentially (see Figure 3.5). After that we will add some more control nodes in our construction of the instance for R-PRAP.

The component containing control nodes $c_2, c_3$ and $c_4$ is described in Figure 3.4. Control nodes $c_2$ and $c_3$ will ensure that the vertex cover contains exactly $K$ vertices. Control node $c_4$ has the effect that all the registers on the nodes connected to it can not be freed until $c_4$ is scheduled, because it is the last successor of such nodes. The

purpose of $c_4$ is to free enough registers so that the last phase of scheduling can do its bookkeeping work.

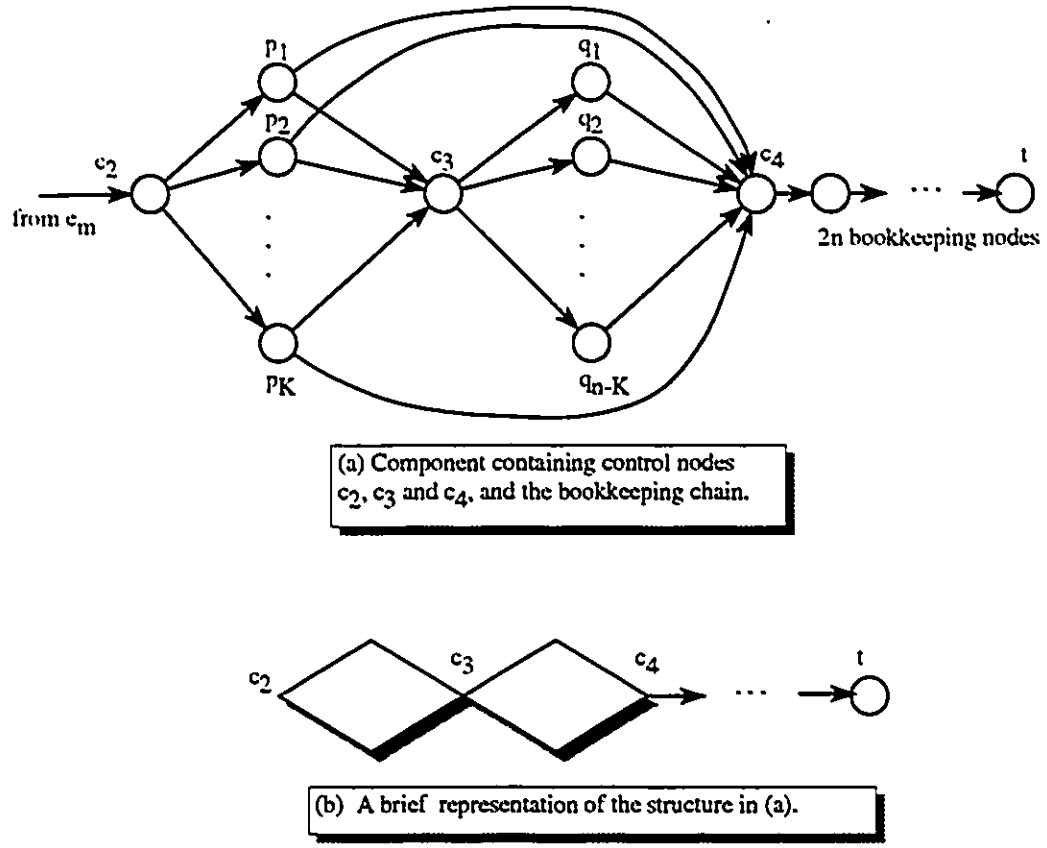



Figure 3.4: Component for control nodes $c_2, c_3$ and $c_4$, and the bookkeeping chain of $2n$ nodes.

After control node $c_4$ we append a chain of $2n$ nodes to do the bookkeeping work. The last node in the chain is called $t$ (terminator). See Figure 3.4.

These are all the components for the instance of the R-PRAP. The overall construction for the instance of R-PRAP is illustrated in Figure 3.5. Note that not all the arcs have been shown in Figure 3.5 for simplicity. It is easy to check that only the last $t$ has no output arc and only the starting node $s$ has no input arc.

Now we give the formal construction of the instance for R-PRAP:

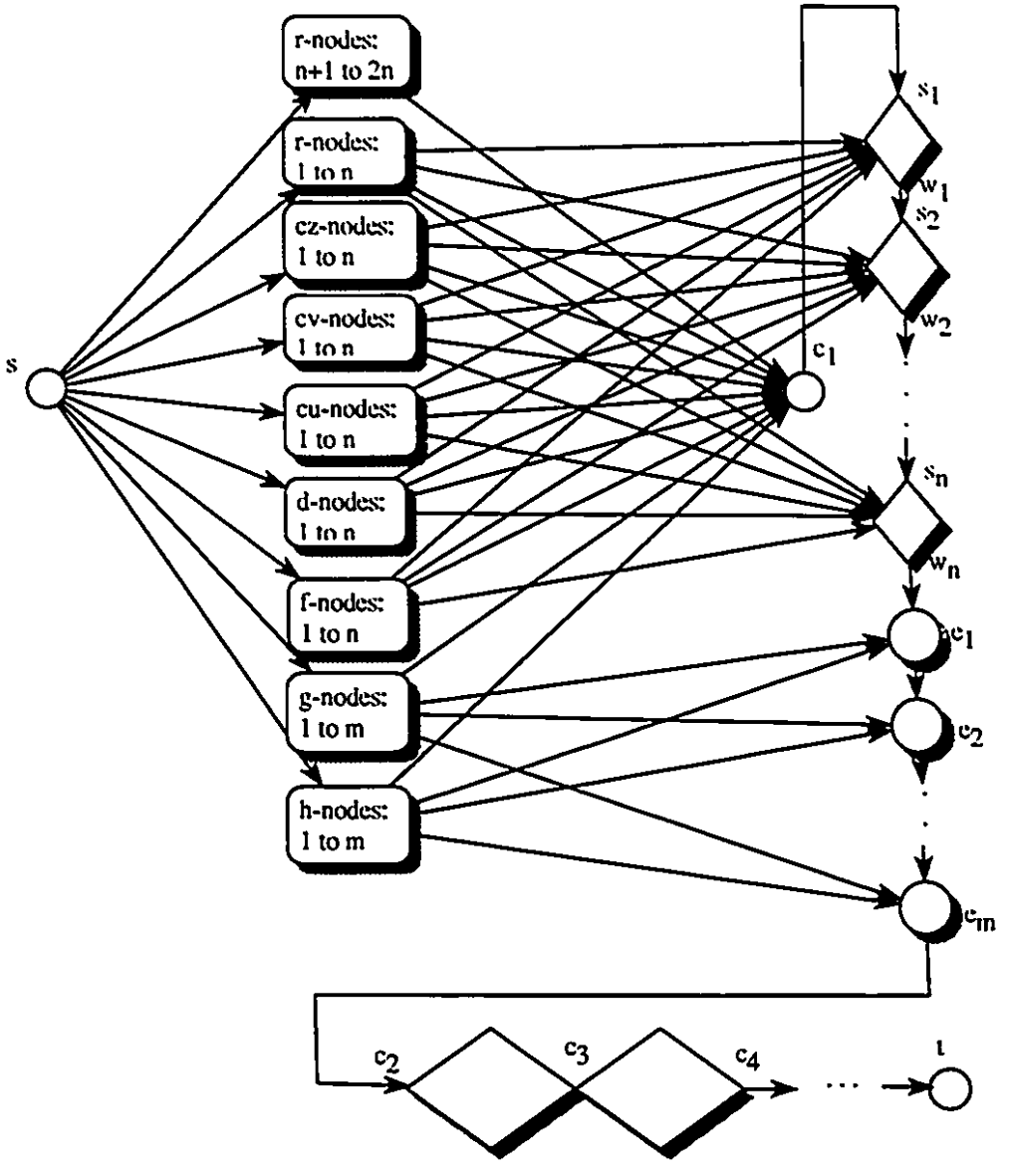



Figure 3.5: Overall structure of the construction for the instance of R-PRAP.

- Initialization Component (Figures 3.1, 3.5):

  1. A starting node $s$.
  2. Initialization nodes:

$$r_1, r_2, \cdots, r_n, \ r_{n+1}, r_{n+2}, \cdots, r_{2n},$$

$$cz_1, cz_2, \cdots, cz_n,$$

$$cv_1, cv_2, \cdots, cv_n,$$

$$cu_1, cu_2, \cdots, cu_n,$$

$$d_1, d_2, \cdots, d_n, \ f_1, f_2, \cdots, f_n,$$

$$g_1, g_2, \cdots, g_m, \ h_1 h_2, \cdots, h_m,$$

  where $r_1$ through $r_n$ are designed to supply registers to the starting nodes $s_1, \cdots, s_n$ of the vertex components. $r_{n+1}$ through $r_{2n}$ are designed to supply registers to the $a$-branches or the $b$-branches in the vertex components. $cz_1, \cdots, cz_n$ are designed to supply registers to $z_1, \cdots, z_n$ in the vertex components. $cv_1, \cdots, cv_n$ are designed to supply registers to $v_1 \cdots, v_n$ in the vertex components. $cu_1, \cdots, cu_n$ are designed to supply registers to $u_1 \cdots, u_n$ in the vertex components. $d_1, \cdots, d_n$ and $f_1, \cdots, f_n$ are designed to control of the size of the vertex cover. $g_1, g_2, \cdots, g_m$ and $h_1, h_2, \cdots, h_m$ are designed to make sure that the vertex cover indeed covers all the edges. Both the $g$-nodes and the $h$-nodes are in 1–to–1 correspondence with the $m$ edges in the VC instance.

  3. The control node $c_1$ which is designed to force all the initialization nodes get scheduled before $c_1$.
  4. There is an arc from the starting node $s$ to each of the initialization nodes:

$$(s, r_i), \text{ for } i = 1, 2, \cdots, 2n,$$

$$(s, cz_i), (s, cv_i), (s, cu_i), (s, d_i), (s, f_i), \text{ for } i = 1, 2, \cdots, n,$$

$$(s, g_j), (s, h_j), \text{ for } j = 1, 2, \cdots, m.$$

5. There is an arc from each of the initialization nodes to control node $c_1$.

$$(r_i, c_1), \text{ for } i = 1, 2, \cdots, 2n.$$

$$(cz_i, c_1), (cv_i, c_1), (cu_i, c_1), (d_i, c_1), (f_i, c_1), \text{ for } i = 1, 2, \cdots, n.$$

$$(g_j, c_1), (h_j, c_1), \text{ for } j = 1, 2, \cdots, m.$$

- Vertex Components (Figure 3.2):

1. For each vertex $v_i$ in the VC instance, we have the following vertex component: starting node $s_i$, $n-i+1$ nodes in the $a$-branch: $a_{i,1}, \cdots, a_{i,n-i+1}$, $n-i+1$ nodes in the $b$-branch: $b_{i,1}, \cdots, b_{i,n-i+1}$, the ending node $w_i$ and three other nodes $z_i, v_i, u_i$.
2. There is an arc from $s_i$ to each of the $a$-nodes and each of the $b$-nodes: $(s_i, a_j), (s_i, b_j)$, for $j = 1, 2, \cdots, n-i+1$.
3. There is an arc from $s_i$ to $z_i$: $(s_i, z_i)$.
4. There is an arc from the initialization node $r_i$ to $s_i$: $(r_i, s_i)$. $s_i$ is also connected to other nodes outside of this vertex component, we will describe those arcs when those nodes are described later.
5. There is an arc from each of the $a$-nodes to $v_i$: $(a_j, v_i)$, for $j = 1, 2, \cdots, n-i+1$.
6. There is an arc from each of the $b$-nodes to $u_i$: $(b_j, u_i)$, for $j = 1, 2, \cdots, n-i+1$.
7. There is an arc from $z_i$ to the ending node $w_i$: $(z_i, w_i)$. And there is an arc from the initialization node $cz_i$ to $z_i$: $(cz_i, z_i)$.
8. There are two arcs $(cv_i, v_i)$ and $(cv_i, w_i)$ from the initialization node $cv_i$ to $v_i$ and $w_i$.
9. There are two arcs $(cu_i, u_i)$ and $(cu_i, w_i)$ from the initialization node $cu_i$ to $u_i$ and $w_i$.
10. There is an arc from initialization node $d_i$ to $v_i$: $(d_i, v_i)$ and there is an arc from initialization node $f_i$ to $u_i$: $(f_i, u_i)$.

11. There are also arcs from initialization nodes $g_j, h_j$ to $v_i, u_i$. Now we describe their exact connections. Consider an edge $e_j$ in the instance of VC. If vertex $v_i$ is one of the end-vertices of $e_j$, then the following three arcs are added: $(g_j, v_i), (h_j, u_i)$ and $(h_j, v_i)$.
12. There is an arc from the ending node $w_i$ of the $i^{th}$ vertex component to the starting node $s_{i+1}$ of the $(i+1)^{th}$ vertex component: $(w_i, s_{i+1})$, for $i = 1, 2, \cdots, n-1$.
13. There is an arc from control node $c_1$ to the starting node $s_1$ of the first vertex component: $(c_1, s_1)$.

- Edge Components (Figure 3.3):

1. For each edge $e_j$ in the VC instance, we have an edge component which obtains a single node also named $e_j$.
2. There is an arc from the initialization node $g_j$ to $e_j$: $(g_j, e_j)$ and there is an arc from the initialization node $h_j$ to $e_j$: $(h_j, e_j)$.
3. There is an arc from $e_j$ to $e_{j+1}$: $(e_j, e_{j+1})$, for $j = 1, 2, \cdots, m-1$.
4. There is an arc from the ending node $w_n$ of the last vertex component to the first edge component $e_1$: $(w_n, e_1)$.

- Component containing Control Nodes $c_2, c_3, c_4$ (Figure 3.4):

1. We add three new control nodes $c_2, c_3, c_4$.
2. We add $K$ $p$-nodes and $n-K$ $q$-nodes:

$$p_1, p_2, \cdots, p_K,$$

$$q_1, q_2, \cdots, q_{n-K}.$$

3. There is an arc from $c_2$ to each of the $p$-nodes: $(c_2, p_i)$, for $i = 1, \cdots, K$.
4. There is an arc from each of the $p$-nodes to control node $c_3$: $(p_i, c_3)$, for $i = 1, \cdots, K$.

5. There is an arc from $c_3$ to each of the $q$-nodes: $(c_2, q_i)$, for $i = 1, \cdots, n-K$.
6. There is an arc from each of the $q$-nodes to control node $c_4$: $(q_i, c_4)$, for $i = 1, \cdots, n-K$.
7. There is an arc from each of the $p$-nodes to $c_4$: $(p_i, c_4)$, for $i = 1, \cdots, K$.
8. There is an arc from the last edge component $c_m$ to $c_2$: $(c_m, c_2)$.
9. There is an arc from each of initialization nodes $d_i$ to $c_2$: $(d_i, c_2)$, for $i = 1, \cdots, n$.
10. There is an arc from each of the initialization nodes $f_i$ to $c_3$: $(f_i, c_3)$, for $i = 1, \cdots, n$.
11. There arc three more arcs from each vertex component to $c_4$: $(s_i, c_4)$, $(z_i, c_4)$, $(w_i, c_4)$, for $i = 1, \cdots, n$.

- Bookkeeping Chain (Figures 3.4, 3.5):

1. We add a chain of $2n$ nodes which terminates at node $t$.
2. There is an arc from the last edge component $c_m$ to the first node in the bookkeeping chain.
3. There are two arcs from each vertex component to the terminating node $t$: $(v_i, t), (u_i, t)$, for $i = 1, \cdots, n$.

Each node has a delay of 1 clock cycle. This completes the construction of the instance for the R-PRAP. It is easy to see that the length of the longest path in the constructed DDG is $5n + m + 7$. One of the path is $s \rightarrow r_1 \rightarrow c_1 \rightarrow s_1 \rightarrow z_1 \rightarrow w_1 \rightarrow \cdots \rightarrow s_n \rightarrow z_n \rightarrow w_n \rightarrow c_1 \rightarrow \cdots \rightarrow c_m \rightarrow c_2 \rightarrow p_1 \rightarrow c_3 \rightarrow q_1 \rightarrow c_4 \rightarrow$ bookkeeping chain of $2n$ nodes.

We show in the remaining part of the proof that the given instance of VC has a vertex cover $V'$ of size $K$ if and only if the constructed instance for R-PRAP can be scheduled in $L = 5n + m + 7$ time units and uses at most $R = 7n + 2m$ registers. When a node $i$ is scheduled at the some time instance, node $i$ can release the registers

on its predecessors if the values in those registers will no longer be used after that time instance, and node $i$ itself will use one register to hold its own computed value. A node can not be scheduled if either one of its predecessors has not been scheduled yet, or even if all its predecessors have been scheduled, no free register is available and it can not release any register on its predecessors.

**The "If" part.** Suppose that VC has a vertex cover $V'$ of size $K$.

In the first time step the starting node $s$ is scheduled. In the second time step we schedule the $7n + 2m$ nodes:

$$r_1, \cdots, r_n, \ r_{n+1}, \cdots, r_{2n},$$

$$cz_1, \cdots, cz_n, \ cv_1, \cdots, cv_n, \ cu_1, \cdots, cu_n,$$

$$d_1, \cdots, d_n, \ f_1, \cdots, f_n,$$

$$g_1, \cdots, g_m, \ h_1, \cdots, h_m.$$

Their results are put into the $7n + 2m$ available registers. At the third time step, the only node we can schedule is the control node $c_1$ which will free the $n$ registers allocated to $r_{n+1}, \cdots, r_{2n}$, because $c_1$ is the only successor of each of these nodes. At the fourth time step we schedule the starting node $s_1$ of the vertex component corresponding to $v_1$, which frees the register on $r_1$. The result of $s_1$ can be put into the register allocated for $r_1$. Therefore after $s_1$ is scheduled there are $n$ free registers first allocated to $r_{n+1}, \cdots, r_{2n}$ which are not connected to other nodes in the rest of the graph. At the fifth time step, $z_1$ and one branch of $\{a_{1,1}, \cdots, a_{1,n}\}$ or $\{b_{1,1}, \cdots, b_{1,n}\}$ will be scheduled. If vertex $v_1$ is in the vertex cover $V'$ then we schedule the $a$-branch. If vertex $v_i$ is not in the vertex cover $V'$, we schedule the $b$-branch. Without loss of generality, we can assume that vertex $v_1$ is in the vertex cover $V'$: $v_1 \in V'$. The result of $z_1$ can be put into the register allocated to $cz_1$ and the results of the $a$-branch nodes are put into the $n$ free registers. At the sixth time step, we schedule $w_1$ and $v_1$, which will free the registers on $cv_1$ and on $\{a_{1,1}, \cdots, a_{1,n}\}$. The result of $w_1$ can be put into the register of $cv_1$ and the result of $v_1$ will occupy one of the $n$ free registers. The registers on $s_1, z_1, w_1, v_1$ can not be freed during this first phase of the scheduling

of the vertex and edge components because they are connected to the control node $c_1$ or the terminating node $t$ which are far away on the longest dependence path. So when $w_1$ has been scheduled there are $n-1$ free registers which are not enough to schedule the $b$-branch in the same vertex component. At the next (seventh) time step we schedule the next vertex component corresponding to $v_2$ and there are $n-1$ free registers at the beginning of the schedule of that component.

In general, just before we schedule the vertex component corresponding to $v_i$, there are $n-i+1$ free registers. So when we schedule $s_i$ at the $(3i+1)^{th}$ time step, its result can be put into the register allocated to $r_i$ and there are still $n-i+1$ free registers. At the $(3i+2)^{th}$ time step we schedule node $z_i$ and one of the branches $\{a_{i,1},\cdots,a_{i,n-i+1}\}$ or $\{b_{i,1},\cdots,b_{i,n-i+1}\}$. We choose to schedule the $a$-branch if vertex $v_i$ is in the vertex cover $V'$. Otherwise we choose to schedule the $b$-branch. Assume that vertex $v_i$ is not in the vertex cover. So we choose to schedule the $b$-branch. The results of the $n-i+1$ nodes $b_{i,1},\cdots,b_{i,n-i+1}$ are put into the $n-i+1$ free registers and the result of $z_i$ can be put into the register allocated for $cz_i$. At the next $(3i+3)^{th}$ time step we schedule nodes $w_i$ and $u_i$. The result of $w_i$ can be put into the register allocated to $cu_i$ and the result of $u_i$ will occupy one of the $n-i+1$ registers previously allocated to the $b$-nodes. Since $s_i, w_i$ and $u_i$ are connected to control node $c_i$ or the terminator node $t$, their registers can not be freed before $c_i$ is scheduled.

When the last vertex component corresponding to $v_n$ has been scheduled, that is, when $w_n$ has been scheduled, there are no free registers.

After $w_n$ has been scheduled, we now consider the edge components. At the $(3(n+1)+1)^{th}$ time step, we can schedule $e_1$. Its result can be put into one of the registers allocated for $g_1$ and $h_1$. Which registers can be used to hold the result of $e_1$ will become clear after we state the general case below.

In general, at the $(3(n+1)+j)^{th}$ time step, we schedule node $e_j$. Its result will be put into one of the registers allocated to $g_j$ or $h_j$. To see which one can be used let us assume that $e_j=(v_x,v_y)$, and the three arcs from $g_j$ and $h_j$ to the vertex components are: $(g_j,v_x),(h_j,u_x)$ and $(h_j,v_y)$.

**Case 1:** Vertex $v_x$ is in the vertex cover, then by the scheduling strategy for the vertex components we have scheduled $v_x$ when we went through the vertex components. Therefore when $e_j$ is scheduled, the register on $g_j$ can be freed so that it can be used to hold the result of $e_j$.

**Case 2:** Vertex $v_x$ is not in the vertex cover, then $v_y$ must be in the vertex cover because the vertex cover should cover edge $e_j$. Since by the strategy on the scheduling of the vertex components, we have scheduled $u_x$ and $v_y$. Hence when $e_j$ is scheduled, the register on $h_j$ can be freed to hold the result of node $e_j$.

When all the edge components have been scheduled, we schedule the control node $c_2$ at time step $3(n+1)+m+1$. $c_2$ can free $K$ registers on the $d$-nodes belonging to the vertex components which correspond to the vertices in the vertex cover. Therefore after $c_2$ is scheduled, we can schedule the $K$ $p$-nodes following it. These $p$-nodes use all the $K$ free registers to hold their results until control node $c_4$ is scheduled. In the next time step we schedule the control node $c_3$ which will free $n-K$ registers from the $f$-nodes in the vertex components. Then we are able to schedule the $n-K$ $q$-nodes following it. Then we schedule $c_4$ at time step $3(n+1)+m+5$ which will free all the registers on the $w$-nodes, $z$-nodes, $p$-nodes and $q$-nodes. Therefore we now have at least $3n$ free registers. At the next $2n$ time steps, we schedule the next $2n$ nodes in the bookkeeping chain. While during the same $2n$ time step period, we can use the free registers to schedule the remaining branches left in the $n$ vertex components.

This gives a schedule for the constructed instance of R-PRAP within the longest path length time $3(n+1)+m+5+2n-1 = 5n+m+7$ and uses at most $7n+2m$ registers.

**The "Only If" part.** Now we show the reverse, that is, if we can schedule the constructed instance of R-PRAP within the length of the longest dependence path and uses at most $7n+2m$ registers, then we can find a vertex cover $V'$ of size $K$ for the VC instance.

Since $s$ is the only node which has no input arc, it must be scheduled in the first time step. In order for control node $c_1$ to be scheduled, all the initializing nodes:

$$r_1, \cdots, r_n, r_{n+1}, \cdots, r_{2n},$$

$$cz_1, \cdots, cz_n, cv_1, \cdots, cv_n, cu_1, \cdots, cu_n,$$

$$d_1, \cdots, d_n, f_1, \cdots, f_n,$$

$$g_1, \cdots, g_m, h_1, \cdots, h_m,$$

must be scheduled before $c_1$ because they all have arcs directed to $c_1$. These initializing nodes use all the $7n + 2m$ registers available. In order for the whole schedule to be finished within the length of the longest dependence path, they must be scheduled in the same time step because each of them is on some longest path. Following the initializing nodes, one must have scheduled control node $c_1$, since it is the only node that can be scheduled at that moment. Note that $c_1$ can free $n$ registers on $r_{n+1}, \cdots, r_{2n}$ because $c_1$ is their last successor. Next time $s_1$ is the only node that can be scheduled. Note that $s_1$ does not reduce the number of free registers because it can free a register on $r_1$. So after $s_1$ is scheduled, there are exactly $n$ free registers that can be used in the next time step. In the next two time steps, $z_1, w_1$ must be scheduled because they are on a longest path. In addition to $z_1, w_1$, at most one branch of the $a$-nodes or $b$-nodes plus either $v_1$ or $u_1$ (but not both) could been scheduled since there are only $n$ free registers. Node $z_1$ occupies one register until control node $c_4$ is scheduled, but it also frees a register on $cz_1$. Node $w_1$ will occupy a register until control node $c_4$ is scheduled. Now if $v_1$ (or $u_1$) is scheduled, it will also occupy a register until the terminate node $t$ is scheduled. But since $v_1$ and $w_1$ are both scheduled, the register on $cv_1$ can be freed. Therefore the number of free registers available before $s_2$ is scheduled is $n - 3 + 2 = n - 1$. If neither $v_1$ nor $u_1$ is scheduled, then $w_1$ will use one free register. So the number of free registers after $w_1$ is scheduled is $n - 1$.

In general, suppose that we are at the moment that $s_i$ is scheduled and there are $n - i + 1$ free registers just before $s_i$ is scheduled. After $s_i$ is scheduled its result

will occupy one register but it also frees the register on $r_i$. Since there are $n-i+1$ free registers for the next time step, only $z_i$ and at most one of the two branches $\{a_{i,1}, \cdots, a_{1,n-i+1}\}$ or $\{b_{i,1}, \cdots, b_{i,n-i+1}\}$ could possibly be scheduled. And the next time step, only $w_i$ and possibly one of $v_i$ or $u_i$ can be scheduled. If none of $v_i$ or $u_i$ is scheduled, $w_i$ must occupy one of the free registers so that the number of free registers (after $w_i$ being scheduled) is reduced to $n-i$. On the other hand if one of $v_i$ (or $u_i$) is scheduled, then although it will occupy one register, the register on $cv_i$ (or $cu_i$) can be freed, so still we have $n-i$ free registers.

When the last vertex component has been scheduled, i.e., $w_n$ has been scheduled, there can not be any free register left. In order to be able to schedule edge component $e_1$, there must be a register on one of $g_1$ or $h_1$ that can be freed by scheduling $e_1$. This is true only if one of the end vertices of $e_1$ is scheduled before. Generally we are able to schedule edge component $e_j$ only if one of the registers on either $g_j$ or $h_j$ can be freed by scheduling $e_j$. Let us assume that edge $e_j = (v_x, v_y)$ and the three arcs from $g_j$ and $h_j$ to the vertex components are: $(g_j, v_x), (h_j, u_x)$ and $(h_j, v_y)$. In order for the register on $g_i$ to be released when scheduling $e_j$, $v_x$ must have been scheduled before. In order for the register on $h_j$ to be released when scheduling $e_j$, $u_x$ and $v_y$ must have been scheduled before. In both of these two cases we see that at least one of $v_x$ and $v_y$ is scheduled. The one that is scheduled will be put into the vertex cover set $V'$ for the VC instance. Since $e_j$ can be any edge, the $V'$ we constructed is indeed a vertex cover for all the edges in the VC instance. Later when we go on to the control nodes $c_2, c_3$ and $c_4$ we will see that such defined vertex cover $V'$ contains exactly $K$ vertices.

After the last edge component $e_m$ is scheduled there are still no free registers. In order to schedule control node $c_2$ and its $K$ following nodes $p_1, \cdots, p_K$, there must be $K$ free registers to hold the results of $p_1, \cdots, p_K$. These free registers can only come from the $d$-nodes which have the $v$-nodes as successors as well. Hence for the $d$-nodes to release $K$ registers when scheduling $p_1, \cdots, p_K$, at least $K$ of the $v$-nodes must have been already scheduled. For the same reason, when scheduling the control node $c_3$ and its $n-K$ following nodes $q_i, \cdots, q_{n-K}$, there are must be at least $n-K$ $u$-nodes

that have been scheduled. Since only one of the branches in the vertex components could have been scheduled so far, $c_2$ and $c_3$ guarantee that the number of the e-nodes which have been scheduled is exactly $K$. In another word, the vertex cover $V'$ contains exactly $K$ vertices, which is exactly what we wanted to prove. □

## 3.3 Loop Version

In the last section we proved that the register allocation problem for the acyclic case is NP-complete even for a restricted class of programs. In this section, we give the definition of the register allocation for loops, and prove that it is also NP-complete.

---

**Parallel Register Allocation for Loops (PRAL):**

**Given:** A DDG $G = (N, A; m, d = 1)$ representing a loop, where N is the node set, A is the arc set, $m = \{m_{ij}, (i,j) \in A\}$ is the dependence distance vector with $m_{ij} > 0$ meaning a loop-carried dependence. Assume that there are infinite number of processing units. Let $R$ be a given positive integer.

**Question:** Let $P$ be a given feasible computation rate. Is there schedule of the DDG which will run at a rate $P$ for the iterations and uses at most $R$ registers?

---

**Theorem 3.3.1** *PRAL is NP-complete.*

**Proof:** From Theorem 3.2.1, we know that R-PRAP is NP-complete. Here we show that R-PRAP can be reduced to PRAL.

Suppose that an instance $D$ of the R-PRAP is given. Let $Q$ be a longest path in $D$. Let $L$ be the length of $Q$. Let $s$ be the starting node and $t$ the end node of $Q$. Then $s$ must be the node without input arc and $t$ must be the node without output

arc, otherwise $Q$ would not be the longest path in $D$. Add an arc from $t$ to $s$ to form an instance $G$ of PRAL, where the newly added arc is a loop-carried dependence and has $m_{ts} = 1$ and $d_{ts} = 1$. Let the number of available registers R in PRAL be the same as in the given instance of R-PRAP.

It is easy to see that all the cycles in the newly formed G must pass through the new arc (t,s). Therefore the optimal computation rate of $G$ is $\frac{1}{L}$, where $L$ is the length of $Q$. We choose this rate for the instance of PRAL.

Now we show that the instance of the R-PRAP has a solution then the constructed instance of PRAL also has a solution. Let us note that $s$ must be scheduled at time 1 and $t$ must be scheduled at time $L$ in the R-PRAP schedule. Therefore if we use the schedule of R-PRAP and repeat it after every $L$ cycles, then we obtain a schedule for the loop G and at the same time the register allocation can be used by such a schedule for PRAL as well. Hence PRAL can use $R$ registers.

For the reverse direction of the proof, we show next that if the constructed instance of PRAL has a solution then the given instance of the Restricted R-PRAP also has a solution. Consider an arbitrary iteration of the constructed instance of PRAL. Since $t$ is the only node without output arc in R-PRAP, and by the structure of the constructed instance of PRAL, all the instructions of an iteration in the instance of PRAL must be finished before the last instruction $t$ in the same iteration. Since the rate is $\frac{1}{L}$, an iteration of PRAL must finish in $L$ cycles. Note that $s$ is the only node in the R-PRAP that has no input arc. Hence $s$ must be the only first instruction to be scheduled before all the other instructions in the same iteration get scheduled. Therefore any given iteration $i$ must be finished between time steps $(i-1) * L + 1$ and $i * L$ inclusive. Thus the scheduling of one iteration in the PRAL is also a valid scheduling for the R-PRAP which uses the same amount of registers.

Hence we have shown that the given instance of R-PRAP has a solution if and only if the constructed instance of PRAL has a solution. This completes the proof of the theorem. □

## 3.4 Summary

In this chapter we proved that the minimum register allocation problem of an acyclic loop is NP-complete if instructions can be arbitrarily moved and restricted only by data dependences, even if an idealized infinite computer architecture is provided which can always support time-optimal schedules. We also showed that the same minimum register allocation problem for inner most loops with dependence cycles is NP-complete with the same architecture assumption. Therefore the register allocation for a parallel architecture is hard in the sense that no polynomial time algorithms could likely be found. The hardness of the problem is caused by the sharing of physical registers if variable live ranges do not overlap. However if the sharing of the registers by different variables is limited, the restricted situation can give a polynomial time solvable problem. In next chapter, we will show that we can allocate minimum number of buffers to variables in polynomial time, and we will show how to use coloring method to map the buffers to registers. In Chapter 5 we show that if we only allow the registers to be shared by the instructions on a chain, then an optimal allocation of registers can be done in polynomial time to support maximum computation rate.

# Chapter 4

# Register Allocation

The objective of this chapter is to develop a unified framework to do scheduling and register allocation simultaneously to support time-optimal software pipelining on superscalar-like architectures. Our register allocation approach for software pipelining is solved in two steps. The first step determines the time-optimal schedule and allocates symbolic registers organized as FIFO buffer queues, one queue for each variable defined in the loop. We show that the minimum buffer allocation and the time-optimal scheduling problem can be formulated together as an integer programming problem, called *Optimal Scheduling and Buffer Allocation* (OSBA) problem, which has a polynomial time solution. The second step is to map the symbolic registers of the FIFO buffers into physical registers. Since a time-optimal schedule is derived from the solution of the OSBA problem, a coloring algorithm can be applied to minimize the number of physical registers required to implement the buffers. Code generation schemes with or without special hardware support are discussed.

## 4.1 Introduction

In the previous chapter we have proved that the minimum register allocation problem to support time optimal scheduling is NP-complete in its general form, even if an idealized architecture model is assumed. However simultaneous scheduling and register allocation is more important in parallel instruction scheduling than its sequential counterpart, because as the schedule gets more parallel, it consumes more registers than the sequential execution model. Therefore "bad" optimal parallel schedule could consume substantially more registers than necessary.

The traditional register allocation approach assumes that a fixed schedule is given. Then the allocator tries to allocate a minimum number of registers to support the given schedule. Usually the schedule is produced by optimizing the code so that minimum delay is introduced due to various conflicting data or resource dependences. However the schedule produced has no control over the use of registers. In another word, it totally depends on the register allocator to calculate how many registers are going to be used. Therefore the schedule produced may use more than it really needs. If the number of available registers is not enough, spill code must be introduced, which will change the schedule. So the traditional approach will either try to minimize the spill code, or after the spill code is introduced, the code will be sent back to the scheduler to do another phase of scheduling and to do register allocation again. This process may be repeated several times until the schedule and register allocation are acceptable.

However this traditional approach does not have a theoretic foundation that will point out when it should do this and when it should do that, and what result we can expect.

Despite the NP-completeness results in Chapter 3, the objective of this chapter is to develop a unified scheduling-allocation framework to determine a scheduling and a register allocation simultaneously. This is quite different from the conventional approach, which is to minimize the number of registers under a given schedule. For example, many of these register allocation algorithms are based on the coloring of

*interference graphs* representing overlapping relations of the live ranges of program variables given by a sequential execution schedule [16, 15, 1].

The method in this chapter is also different from other work on register allocation for loop variables with or without software pipelining. In particular, the current method generalizes the work by Callahan, Carr and Kennedy on *scalar replacement* as a register allocation method for subscript variables [13], the work by Lam on *modular variable expansion* for software pipelined loops [56], and the work by Rau et. al. on register allocation for modulo scheduled loops [72]. A comparison with related work is outlined in Section 4.11.

This chapter proposes a framework in which register allocation for software pipelining is solved in two steps.

**Step 1: Optimal Scheduling and Buffer Allocation:** The first step determines the time-optimal schedule for a software pipelined loop and allocates symbolic registers organized as FIFO buffer queues, one buffer queue for each variable defined in the loop. Intuitively, such a buffer queue is used to "extend" the lifetime of the corresponding loop variable generated in successive iterations, permitting multiple iterations to be overlapped in concurrent executions. It is shown that the minimum buffer allocation and the time-optimal scheduling problem can be formulated together as an integer programming problem called the *Optimal Scheduling and Buffer Allocation* (OSBA) problem. An efficient polynomial time solution is presented based on a transformation of the OSBA problem into *min-cost flow problem.*

**Step 2: Mapping buffers to physical registers:** The second step is to map the symbolic registers of the buffers into physical registers. Since a schedule is derived from the solution of the OSBA problem, a coloring algorithm can be applied to minimize the number of physical registers required to implement the buffers. In particular, a recently proposed method based on coloring of *cyclic interval graphs* [49] can be applied. Code generation schemes with or without special hardware support are discussed.

The method developed in this chapter is applicable to machines such as VLIW (Very Long Instruction Word) [34], superscalar [52] and superpipelined architectures [53].

Since the register coloring method is well understood, our discussion centers on the buffer allocation step and the code generation schemes. Some preliminary results also appeared in the paper [64]. We organize the subsequent sections as follows. In Section 4.2 we present a simple example loop to motivate the concept of buffers and the two step approach for concurrent scheduling and register allocation. In Section 4.3 we formulate the Optimal Scheduling and Buffer Allocation (OSBA) problem. A polynomial time solution is stated in Section 4.4. We return to our motivating example in Section 4.5 to illustrate the formulation and solution of the corresponding OSBA problem. In Section 4.6, we propose two schemes of code generation for our (OSBA) scheme. One of the schemes is to shift the registers to simulate the effect of a FIFO (First-In-First-Out) buffer. This scheme is called *Access Stationary Code* (ASC) where the accessing mode is fixed. The other scheme uses self-modifying code (if hardware supports it) to eliminate the burden of register shifting. This second approach is called *Data Stationary Code* (DSC) where the data are not moved but they have to be written into different locations for different iterations. In Section 4.7, we show how to reduce the register requirement further through coloring technique. This is the second phase in our register allocation scheme. In Section 4.8 we consider two special cases. We show that Callahan et al's result [13] can be viewed as a special case of the OSBA formulation. We will also point out that if a loop contains no loop-carried dependences, then the OSBA problem is easier to solve. So it generalizes our earlier results in [63]. In Section 4.9 we give some experimentation results on some benchmarks. In Section 4.10 we apply the OSBA scheme to an example taken from [72]. In Section 4.11 we compare our approach with related work.

## 4.2 Motivation

In this section, we motivate our approach by studying register allocation for a simple loop $L_1$ under software pipelining. Although a high level language representation of the loop is chosen here, it is intended only to give a simple description of our technical framework. There is no difficulty in applying our framework to a lower level representation of the code. As shown below, loop $L_1$ contains three instructions in its body.

| | |
|---|---|
| $L_1$ : | **for** i = 1 **to** n **do** |
| $s_1$ : | $a[i] = X + c[i-2]$; |
| $s_2$ : | $b[i] = a[i] * F$; |
| $s_3$ : | $c[i] = a[i] + b[i]$; |
| | **enddo;** |

(4.1)

$L_1$ contains a *loop-carried dependence* of distance 2 from $s_3$ to $s_1$. For instance, the value $c[i]$ generated by $s_3$ in iteration $i$ is only used two iterations later by statement $s_1$ in iteration $i+2$. The other data dependences in the loop are all within the same iteration. For instance, $s_2$ reads $a[i]$ which is produced by $s_1$ in the same iteration. And $s_3$ reads $a[i]$ and $b[i]$ produced by $s_1$ and $s_2$ in the same iteration. The DDG of $L_1$ is shown in Figure 4.1

Under software pipelining, the iterations are permitted to overlap so that the subsequent iterations may start before the previous iterations finish. Since there exist loop-carried dependences, we have to work out a proper initiation delay interval $P$ between successive iterations so that when the next iteration starts $P$ clock cycles after the previous iteration started, no loop-carried dependences are violated.

In our example, assuming that the delay for Add is 1 clock cycle and the delay for Multiply is 2 clock cycles, then a delay of $P = 2$ clock cycles between the starting times of two consecutive iterations is optimal in the sense that the scheduled loop achieves the maximum computation rate. A possible maximum computation rate

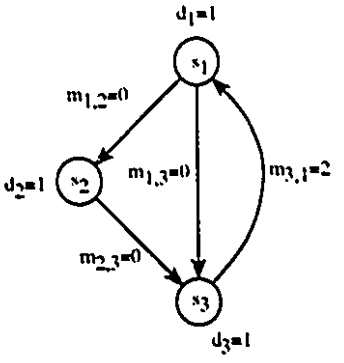



Figure 4.1: Data dependence graph of the example loop $L_1$.

schedule is shown below, in which we have used the second index to indicate the iteration, i.e. $s_{2,1}$ means node $s_2$ in iteration 1:

| | iteration 1 | iteration 2 | iteration 3 | iteration 4 |
|---|---|---|---|---|
| 0 | $s_{1,1}$: $a[1]=X+c[-1]$ | | | |
| 1 | $s_{2,1}$: $b[1]=a[1]*F$ | | | |
| 2 | | $s_{1,2}$: $a[2]=X+c[0]$ | | |
| 3 | $s_{3,1}$: $c[1]=a[1]+b[1]$ | $s_{2,2}$: $b[2]=a[2]*F$ | | |
| 4 | | | $s_{1,3}$: $a[3]=X+c[1]$ | |
| 5 | | $s_{3,2}$: $c[2]=a[2]+b[2]$ | $s_{2,3}$: $b[3]=a[3]*F$ | |
| 6 | | | | $s_{1,4}$: $a[4]=X+c[2]$ |
| 7 | | | $s_{3,3}$: $c[3]=a[3]+b[3]$ | $s_{2,4}$: $b[4]=a[4]*F$ |
| 8 | | | | |
| 9 | | | | $s_{3,4}$: $c[4]=a[4]+b[4]$ |

(4.2)

Note that iteration 3 starts after iteration 1 produces $c[1]$, and iteration 4 starts after iteration 2 produces $c[2]$, etc. So the loop-carried dependence is not violated. The schedule exploits parallelism since there are two instructions scheduled in parallel at clock cycle 3 (5, 7 etc.). Under the above time-optimal schedule, $s_1$ is executed twice before its successor $s_3$ is scheduled for the first time in clock cycle 3. Hence, in order to support the schedule, conceptually it is natural to provide a storage buffer of size more than one between the generator $s_1$ and its successors $s_2, s_3$. And to enforce the correct order of the values produced, the buffers should behave like a first-in-first-out (FIFO) queue.

Let us examine this in some detail. Suppose that a FIFO buffer queue of two symbolic registers $\{a_0, a_1\}$ is allocated to $a$, such that $a_0$ is the tail and $a_1$ is the head. Each of the other variables is allocated a buffer of size 1, which is a single register. For convenience we will use the variable name as its register allocated if the buffer size is one. When a[0] is produced by $s_1$ in clock cycle 0, it is written into the tail $a_0$ of the buffer queue. At clock cycle 2, a new iteration starts and $a$[2] is produced before $a$[1] is consumed at clock cycle 3. At this moment, the new value can not be written into $a_0$, otherwise it would have overwritten a value ($a$[0]) which is still needed in clock cycle 3.

Now since we allocated two registers and organized them as a FIFO buffer queue, we can continue to write to the queue at the tail at clock cycle 2. But before we do that we have to assume that the queue has a mechanism to shift its contents towards its head so that the tail ($a_0$) is ready to get a new value. In our case, we can assume that the old value in $a_0$ is shifted to $a_1$ at beginning of clock cycle 2. So at clock cycle 2, $a$[1] is in $a_1$ and a[2] is in $a_0$.

At the code generation phase, we should generate appropriate code to implement the FIFO addressing mechanism. For example, to implement the FIFO buffer queue mechanism allocated for $a$, we have to figure out how the successors obtain the correct values of a[i] at the buffer. The following pseudo code may be generated to ensure the correct accesses of the buffer queue positions, in which the "b-shift" means *buffer shift.*

| | iteration 1 | iteration 2 | iteration 3 | iteration 4 |
|---|---|---|---|---|
| 0 | b-shift: $a_1 = a_0$<br>$s_{1,1} : a_0 = X + c$ | | | |
| 1 | $s_{2,1} : b = a_0 * F$ | | | |
| 2 | | b-shift: $a_1 = a_0$<br>$s_{1,2} : a_0 = X + c$ | | |
| 3 | $s_{3,1} : c = a_1 + b$ | $s_{2,2} : b = a_0 * F$ | | |
| 4 | | | b-shift: $a_1 = a_0$<br>$s_{1,3} : a_0 = X + c$ | |
| 5 | | $s_{3,2} : c = a_1 + b$ | $s_{2,3} : b = a_0 * F$ | |
| 6 | | | | b-shift: $a_1 = a_0$<br>$s_{1,4} : a_0 = X + c$ |
| 7 | | | $s_{3,3} : c = a_1 + b$ | $s_{2,4} : b = a_0 * F$ |
| 8 | | | | |
| 9 | | | | $s_{3,4} : c = a_1 + b$ |

(4.3)

Let us see how the successors $s_2, s_3$ of $s_1$ access the buffer queue of $a$. Since $s_{2,1}$ is scheduled at clock cycle 1, when it reads the value of $a[1]$, we can assume it is still in register $a_0$. Therefore, the actual code for $s_{2,1}$ in iteration 1 could be $b = a_0 * F$. On the other hand, if we assume that the queue shifts its contents at clock cycle 2, then at clock cycle 3 when $s_{3,1}$ is executed, the value of $a[1]$ is shifted to $a_1$ in the queue, hence $s_{3,1}$ should read from $a_1$. Therefore the code for $s_{3,1}$ should be $c[1] = a_1 + b$. Hence $s_{2,i}$ always reads from the tail $a_0$ and $s_{3,i}$ always reads from the head $a_1$. This phenomenon is caused by the different scheduled timings of the successors. We call such a buffer queue *multiple-head* FIFO queue because the successors read the contents of the queue at different places. The multiple-head buffer queue is illustrated in Figure 4.2.

The concept of FIFO buffer plays an important role here, as it captures the notion of lifetime of a loop variable "extended" into successive iterations. In Section 4.3, we formulate, as the first step of our method, the optimal scheduling and buffer allocation problem by relating the schedule of each producer and its successors to the size of the corresponding FIFO buffer queue allocated to the generator. The solution of the problem is a time-optimal schedule which uses minimal number of buffers among all

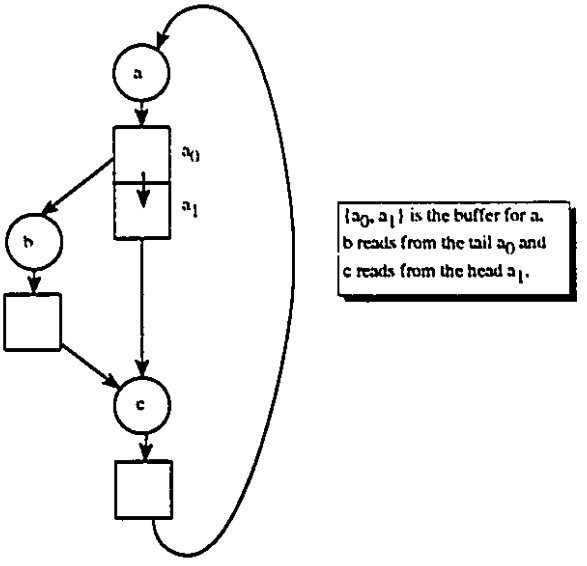



Figure 4.2: A multiple-head buffer.

time-optimal periodic schedules. In our example, the repeating pattern between clock cycles 2 and 3 in (4.3) is, in fact, the time-optimal schedule for software pipelining we expect to derive.

So far, we are assuming that the buffers are represented by symbolic registers. We notice that the buffers allocated to individual instructions can share the same physical registers if their live ranges in the produced schedule are non-overlapping. In our example loop $L_1$ and its given schedule (4.3), the live ranges of the variables in the repeating pattern are indicated in Figure 4.3. Since the live ranges of loop invariants $F$ and $X$ are the whole range of the loop, they are not shown in Figure 4.3. As shown in Figure 4.3, the live range of $c$ does not intersect with the live range of $a_1$. So they can share the same physical register. Thus, the mapping of buffers to physical registers (step 2 of our method) is similar to the traditional register allocation problem. In Section 4.7, we describe how to apply the register allocation method based on cyclic interval graphs for this step. After this procedure the code does not need the extra register $c$, which is replaced by $a_1$, as shown in (4.4) where the symbolic register names now represent physical registers.

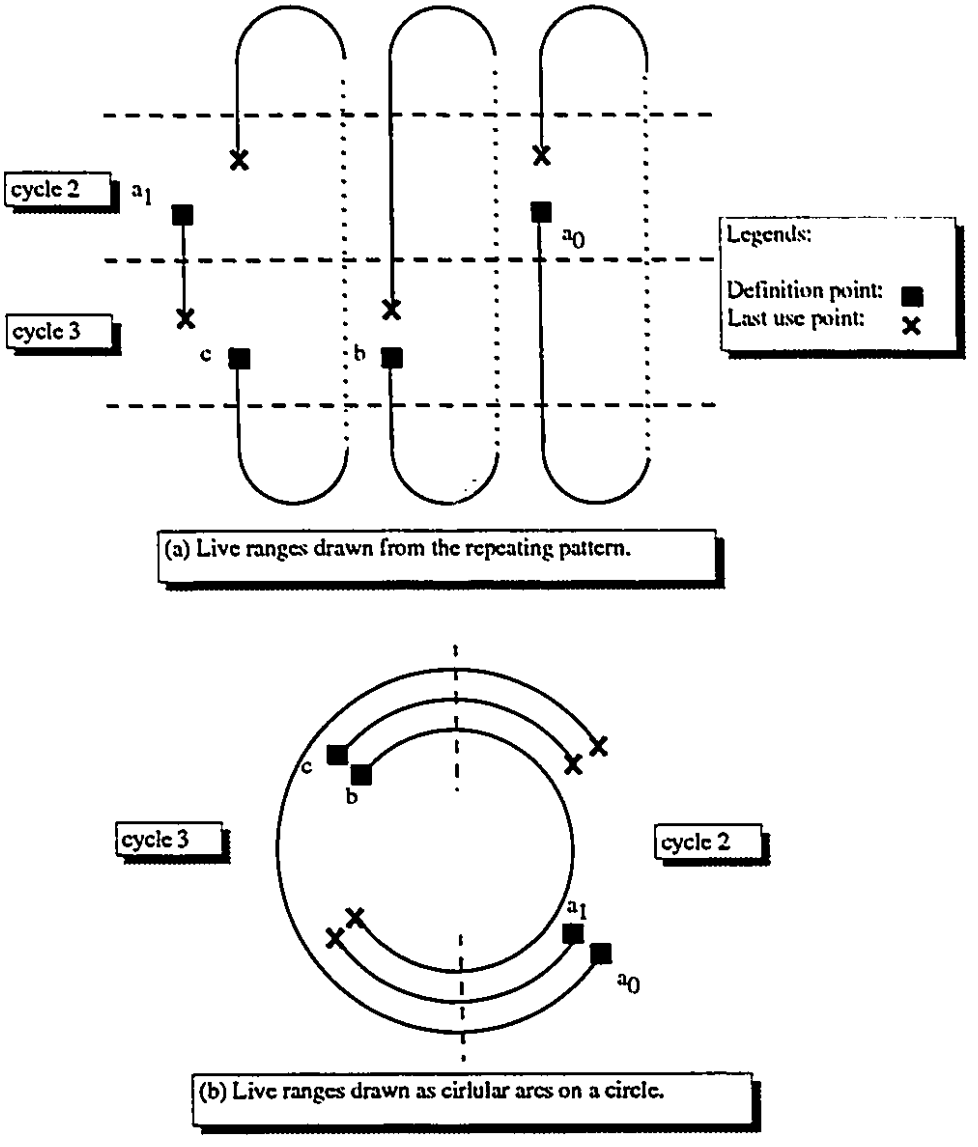



Figure 4.3: The live ranges of the variables for code generated by the ASC scheme.

| | iteration 1 | iteration 2 | iteration 3 | iteration 4 |
|---|---|---|---|---|
| 0 | b-shift: $a_1 = a_0$<br>$s_{1,1} : a_0 = X + a_1$ | | | |
| 1 | $s_{2,1} : b = a_0 * F$ | | | |
| 2 | | b-shift: $a_1 = a_0$<br>$s_{1,2} : a_0 = X + a_1$ | | |
| 3 | $s_{3,1} : a_1 = a_1 + b$ | $s_{2,2} : b = a_0 * F$ | | |
| 4 | | | b-shift: $a_1 = a_0$<br>$s_{1,3} : a_0 = X + a_1$ | |
| 5 | | $s_{3,2} : a_1 = a_1 + b$ | $s_{2,3} : b = a_0 * F$ | |
| 6 | | | | b-shift: $a_1 = a_0$<br>$s_{1,4} : a_0 = X + a_1$ |
| 7 | | | $s_{3,3} : a_1 = a_1 + b$ | $s_{2,4} : b = a_0 * F$ |
| 8 | | | | |
| 9 | | | | $s_{3,4} : a_1 = a_1 + b$ |

(4.4)

## 4.3 Formulation of the OSBA Problem: Step 1

In this section we give a mathematical formulation of the simultaneous scheduling and buffer allocation problem. Suppose that we are given a DDG $G = (N, A; m, d)$ representing an inner-most loop, where $N$ is the node set, $A$ is the dependence arc set, $m$ is the dependence distance vector on $A$ and $d$ is the delay vector on $N$. In general, there are many time-optimal periodic schedules. One of our goals in this thesis is to find the best schedule $t_i$ such that it computes the loop at the optimal computation rate and will need the least number of registers. In this section we look at the problem of providing the minimum number of buffers so that successive iterations can be initiated at the desired optimal rate. We do not assume that a fixed schedule is given. Instead we will find one that can achieve both time-optimality and space-optimality. The time-optimal property is enforced by using an optimal period while not fixing the timings of the individual nodes.

A schedule may produce different numbers of results at different time instances

during the execution. So our first objective is to minimize the number of buffers required at different time instances during the execution.

We will allocate a set of buffers for each node in DDG, so that the buffers are organized as a FIFO queue and they can retain the results for several iterations before the consumers read these results. Let $i$ be a node in DDG which will produce result data. We want to know how many buffers we should allocate to a node $i$. This number depends on the timings of $i$'s successors and also on the reservation scheme for the registers. Here we take a conservative assumption that a register is reserved at the issue time of the instruction. However our analysis can be applied to other reservation schemes with only minor modification. For instance, if we assume that a register is reserved only at the output stage of the pipeline, then a modified formulation is shown in Appendix A.

Let us consider one node $j$ of such successors, so that $(i, j)$ is an arc from node $i$ to node $j$.

Recall from Chapter 2 Section 2.7 that we use $P$ to represent the period of a periodic schedule. For node $i$, its scheduled time in the first iteration is indicated by $t_i$. A periodic schedule is one such that the scheduled times of the node $i$ in iterations 2, 3, $\cdots$ etc, are $t_i + P$, $t_i + 2P$, $\cdots$ etc. A periodic schedule is feasible if and only if it satisfies (2.2) in Section 2.7.

When node $i$ is scheduled for the first time, it is $t_i$. If the dependence distance between $i$ and $j$ is $m_{ij}$, then the result value produced by node $i$ will be consumed by node $j$ in iteration $m_{ij}$ at time $t_j + Pm_{ij}$. Therefore the live time span of the result value is at least $t_j + Pm_{ij} - t_i$. During this period of time, node $i$ will be scheduled every $P$ clock cycles. All these new results produced by node $i$ have to be saved in buffers so that node $j$ in later iterations can read them. Hence the number $b_i$ of buffers for node $i$ should satisfy the following inequality, in which $j$ belongs to the set of immediate successors of node $i$:

$$b_i \geq \frac{t_j + Pm_{ij} - t_i}{P}, \forall (i, j) \in \delta^+(i). \tag{4.5}$$

Now we combine the timing constraints (2.2) for feasible schedules stated in Section 2.7 and the buffer size constraints in (4.5). Together these two sets of constraints define all the feasible schedules with optimal speedup and all supporting buffer allocation schemes. Then we want to minimize the total sum of buffers among all these feasible schedules. Putting all together, we obtain the following integer programming problem:

$$\min \sum_{i \in N} b_i \tag{4.6a}$$

subject to (4.6b)

$$b_i \geq \frac{t_j - t_i}{P} + m_{ij}, \ \forall (i,j) \in A \tag{4.6c}$$

$$t_j \geq t_i + d_i - P m_{ij}, \forall (i,j) \in A \tag{4.6d}$$

$$t_i, b_i \ \text{ integers}, \quad \forall i \in N. \tag{4.6e}$$

In the following we rewrite the above formulation (4.6) so that all the variables appear on the left sides of the inequalities. We name it as *Optimal Schedule and Buffer Allocation (OSBA) Problem.*

---

**Optimal Scheduling and Buffer Allocation (OSBA) Problem:**

$$\min \sum_{i \in N} b_i \tag{4.7a}$$

subject to (4.7b)

$$P b_i + t_i - t_j \geq P m_{ij}, \forall (i,j) \in A \tag{4.7c}$$

$$t_j - t_i \geq d_i - P m_{ij}, \forall (i,j) \in A \tag{4.7d}$$

$$t_i, b_i \text{ integers}, \quad \forall i \in N. \tag{4.7e}$$

---

## 4.4 Solution of the OSBA Problem

In previous section we obtained an integer programming formulation (4.7) of the OSBA problem. In this section we investigate its solution.

In (4.7e), $b_i$ is required to be an integer. However the coefficient before $b_i$ in constraint (4.7c) is the period $P$ which in general can be greater than 1. Therefore that creates some difficulty when we want to solve (4.7) directly to obtain integer solutions. To overcome the difficulty, we do a variable substitution:

$$b'_i = Pb_i \tag{4.8}$$

and transform the formulation (4.7) into the following form:

---

**OSBA Problem with Variable Substitution (4.8)**

$$\min \sum_{i \in N} b'_i \tag{4.9a}$$

subject to

$$b'_i + t_i - t_j \geq Pm_{ij}, \forall (i,j) \in A \tag{4.9b}$$

$$t_j - t_i \geq d_i - Pm_{ij}, \forall (i,j) \in A \tag{4.9c}$$

$$b'_i, t_i \text{ integers}, \quad \forall i \in N \tag{4.9d}$$

---

In the next subsection we prove that the constraint matrix in (4.9) is totally unimodular, which enables us to solve the integer programming problem (4.9) as a linear programming problem. Then in Subsection 4.4.2 we present an efficient algorithm based on a transformation to the minimum cost flow problem.

### 4.4.1 Totally Unimodular Constraint Matrix

In this subsection, we prove that the constraint matrix in (4.9) is totally unimodular, a concept defined in Section 2.10. In order to prove that, we need to give some definitions.

**Definition 4.4.1** *The Out-incidence matrix $U^+$ of (directed) graph $G$ is a matrix with the rows indexed by arcs and the columns indexed by nodes and the entries defined by*

$$u^+_{ei} = +1, \forall n \in N, \forall e \in A \text{ and } i \text{ is the tail of arc } e.$$

*The In-incidence matrix $U^-$ of graph $G$ is a matrix with rows indexed by arcs and the columns indexed by nodes and the entries defined by*

$$u^-_{ej} = -1, \forall n \in N, \forall e \in A \text{ and } j \text{ is the head of arc } e.$$

With these definitions, we can see that each row of $U^+$ (or $U^-$) has exactly one $+1$ ($-1$) at the column indexed by the tail (head) of the arc. And for each column of $U^+$ (or $U^-$), the number of $+1$'s ($-1$'s) is the out-degree (in-degree) of the node indexing that column.

**Definition 4.4.2** *The usual incidence matrix of graph $G$ is just*

$$U = U^+ + U^-.$$

So for a row in $U$ not indexed by a self-loop it has one $+1$ and one $-1$ at the columns indexed by the tail node and head node of that arc, respectively. If a row in $U$ is indexed by a self-loop, then the row contains 0 values only.

Testing whether a given matrix having the *Total Unimodularity* (TUM) property is not easy in general although sophisticated procedures have been developed to do this in polynomial time [74]. There are several known necessary and sufficient conditions in the literature for testing the TUM property. Here we only list one which we will use later in our proof.

**Definition 4.4.3** *A submatrix of a $\{0, \pm 1\}$ matrix is called Eulerian if the sum of the entries in each row and in each column of the submatrix is even.*

**Theorem 4.4.1 (Camion (1965) [14])** *A $\{0, \pm 1\}$ matrix is totally unimodular if and only if the sum of entries in each Eulerian submatrix can be divided by 4.*

Now we are ready to prove our theorem.

**Theorem 4.4.2** *The constraint matrix in (4.9) is totally unimodular, that is to say, each of its square submatrices has a determinant equal to either 0 or 1 or $-1$.*

**Proof:** The constraint matrix in (4.9) has very strong relation with the incidence matrix $U$ of the graph. Actually the constraint matrix is the following matrix:

$$\begin{bmatrix} U^+ & U \\ O & -U \end{bmatrix} \tag{4.10}$$

where $O$ is a submatrix of proper size in which all entries are zero, $U^+$ is the Out-incidence matrix and $U$ the incidence matrix.

Although it is well known that the incidence matrix of a directed graph is totally unimodular, it is not generally true that the combination of totally unimodular matrices preserves the total unimodularity property. However, we show next that the constraint matrix of (4.9) is totally unimodular. In fact, we will show that the condition in Theorem 4.4.1 is satisfied.

Let us index the constraint matrix (4.10) in the following way: the first $|N|$ columns are indexed by $v'_1, v'_2, \cdots, v'_{|N|}$, and the remaining $|N|$ columns are indexed by $v_1, v_2, \cdots, v_{|N|}$, where $|N|$ is the number of nodes in the graph; the first $|A|$ rows are indexed by $e'_1, e'_2, \cdots, e'_{|A|}$, and the next $|A|$ rows are indexed by $e_1, e_2, \cdots, e_{|A|}$, where $|A|$ is the number arcs in the graph. Here we assume that $v'_i$ and $v_i$ represent the same node $i$ in the DDG representing the loop. A similar assumption for arcs is also true.

Therefore the $v'$'s index the columns of the submatrix $\begin{pmatrix} U^+ \\ O \end{pmatrix}$ in the constraint matrix (4.10). Similarly, $v$'s index the columns of the submatrix $\begin{pmatrix} U \\ -U \end{pmatrix}$, $e'$'s index the rows of the submatrix $(U^+ \;\; U)$, and $e$'s index the rows of the submatrix $(O \;\; -U)$.

Let $H$ be an arbitrary Eulerian submatrix of the constraint matrix (4.10). In general, we can assume that some rows of $H$ are indexed by some $e'$'s and some $e$'s. Similarly, we can assume that some columns of $H$ are indexed by some $v'$'s and some $v$'s.

Since each row of the constraint matrix contains at most two $+1$'s and one $-1$, any row of $H$ containing a non-zero element must contain either $+1, +1$ or $+1, -1$, by the Eulerian condition. The rows of the latter case sum to zero and can be removed from further consideration since their sum is divisible by 4.

The remaining non-zeros in $H$ are all $+1$'s and each row contains exactly two $+1$'s because the sum of each row has to be even. The columns containing these two $+1$'s must be labeled by pair $v'_i, v_i$ corresponding to the same node $i$. These two columns are identical under our assumption that all the $-1$'s have been removed from $H$. Since the sum of each of these two columns is divisible by 2, the sum of these two columns is divisible by 4.

Since we have counted all the non-zero entries in $H$, we conclude that the sum of all the entries in $H$ can be divided by 4. Hence by Camion's Theorem 4.4.1, the constraint matrix of (4.9) is totally unimodular. □

The right hand sides of (4.9) are all integers. From linear programming theory [74] if the constraint matrix is totally unimodular and the right hand sides are integral, then the integer programming problem can be solved as a linear programming problem, in which no integer constraints have been put on the variables, and the optimal solution is guaranteed to be integral. Thus we obtain the following corollary:

**Corollary 4.4.1** *When (4.9) is solved as a linear programming problem (dropping the integer requirement), the optimal solution obtained is always integral, i.e. it is the integer programming solution of (4.9).*

By Corollary 4.4.1, to solve (4.9), we can use general linear programming algorithms like the simplex method [17], or the ellipsoid method [55] or the interior point

method [54]. But the simplex method is not a polynomial time algorithm although in practice it runs very fast. The ellipsoid method and the interior point method are polynomial algorithms but are slow for small size problems in practice and have time complexities in the order of $O(|N|^5)$. We present an more efficient algorithm next, which reduces the problem to a minimum cost flow problem on a network, and which can be solved by an $O(|N|^3 \log |N|)$ algorithm.

### 4.4.2 More Efficient Algorithm for Solving OSBA

In this subsection, we show a more efficient algorithm to solve the OSBA problem after variable substitution, which is (4.9). The algorithm is a number of transformations of the problem to the minimum cost flow problem. Since the minimum cost flow problem can be solved more efficiently by the so-called combinatorial algorithms, this will imply that our original (4.9) can also be solved more efficiently.

Let us first write down the linear programming dual of (4.9):

$$\max \sum_{(i,j)\in A} \{(Pm_{ij}\lambda_{ij} + (d_i - Pm_{ij})\pi_{ij}\} \tag{4.11a}$$

subject to

$$\sum_{(i,j)\in\delta^+(i)} \lambda_{ij} = 1, \qquad \forall i \in N \tag{4.11b}$$

$$\sum_{(i,j)\in\delta^+(i)} (\lambda_{ij} - \pi_{ij}) - \sum_{(j,i)\in\delta^-(i)} (\lambda_{ji} - \pi_{ji}) = 0, \forall i \in N \tag{4.11c}$$

$$\lambda_{ij} \geq 0, \ \pi_{ij} \geq 0, \qquad \forall (i,j) \in A \tag{4.11d}$$

where $\delta^+(i), \delta^-(i)$ are the sets of out-going and in-coming arcs of $i$, respectively.

If we reorganize the variables in the objective function, then it can be written as:

$$\sum_{(i,j)\in A} \{Pm_{ij}\lambda_{ij} + (d_i - Pm_{ij})\pi_{ij}\} \tag{4.12a}$$

$$= \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{(i,j)\in A} d_i \pi_{ij} \tag{4.12b}$$

$$= \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} \sum_{(i,j)\in\delta^+(i)} d_i \pi_{ij} \tag{4.12c}$$

$$= \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{(i,j)\in\delta^+(i)} \pi_{ij} \tag{4.12d}$$

With the new form of the objective function, the dual problem (4.11) can be written in the following form:

$$\max \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{(i,j)\in\delta^+(i)} \pi_{ij} \tag{4.13a}$$

subject to

$$\sum_{(i,j)\in\delta^+(i)} \lambda_{ij} = 1, \qquad \forall i \in N \tag{4.13b}$$

$$\sum_{(i,j)\in\delta^+(i)} (\lambda_{ij} - \pi_{ij}) - \sum_{(j,i)\in\delta^-(i)} (\lambda_{ji} - \pi_{ji}) = 0, \quad \forall i \in N \tag{4.13c}$$

$$\lambda_{ij} \geq 0, \ \pi_{ij} \geq 0, \qquad \forall (i,j) \in A \tag{4.13d}$$

Now we do a variable substitution for formulation (4.13):

$$f_{ij} = \pi_{ij} - \lambda_{ij}, \ \forall (i,j) \in A. \tag{4.14}$$

With this variable substitution, the objective function in (4.13a) becomes:

$$\sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{(i,j)\in\delta^+(i)} \pi_{ij} \tag{4.15a}$$

$$= -\sum_{(i,j)\in A} Pm_{ij} f_{ij} + \sum_{i\in N} d_i \sum_{(i,j)\in\delta^+(i)} (f_{ij} + \lambda_{ij}) \tag{4.15b}$$

$$= -\sum_{(i,j)\in A} Pm_{ij} f_{ij} + \sum_{i\in N} \sum_{(i,j)\in\delta^+(i)} d_i f_{ij} + \sum_{i\in N} d_i \sum_{(i,j)\in\delta^+(i)} \lambda_{ij} \tag{4.15c}$$

$$= -\sum_{(i,j)\in A} (Pm_{ij} - d_i) f_{ij} + \sum_{i\in N} d_i \tag{4.15d}$$

Note that the last term in (4.15d) is a constant and can be discarded from the objective function. Then the formulation (4.13) becomes the following with the variable substitution (4.14):

$$\max \; -\sum_{(i,j)\in A} (Pm_{ij} - d_i) f_{ij} \tag{4.16a}$$

subject to

$$\sum_{(i,j)\in\delta^+(i)} (\pi_{ij} - f_{ij}) = 1, \quad \forall i \in N \tag{4.16b}$$

$$\sum_{(i,j)\in\delta^+(i)} f_{ij} - \sum_{(j,i)\in\delta^-(i)} f_{ji} = 0, \forall i \in N \tag{4.16c}$$

$$f_{ij} \geq -1, \; \pi_{ij} \geq 0, \qquad \forall (i,j) \in A \tag{4.16d}$$

Formulation (4.16) is not yet a minimum cost flow problem. Later we show that (4.16) can be further reduced to a minimum cost flow problem. Now let us notice that in (4.16) the objective function does not contain variables $\pi_{ij}$'s, which only appear in constraints (4.16b). Therefore we can simplify constraints (4.16b) so that the variables $\pi_{ij}$'s do not appear in any constraints, that is, we want to eliminate them.

**Lemma 4.4.1** *In formulation (4.16), the constraints (4.16b) can be replaced by the following equivalent constraints:*

$$\sum_{(i,j)\in\delta^+(i)} f_{ij} \geq -1, \; \forall i \in N. \tag{4.17}$$

**Proof:** By moving the terms, (4.16b) can be rewritten as:

$$\sum_{(i,j)\in\delta^+(i)} f_{ij} = \sum_{(i,j)\in\delta^+(i)} \pi_{ij} - 1 \tag{4.18}$$

Since $\pi_{ij}$'s are non-negative variables, it is immediate to see that if $f_{ij}$'s satisfy (4.18), then they must also satisfy (4.17).

To show the reverse, let us assume that $f_{ij}$'s satisfy (4.17). We want to find a set of values for the $\pi_{ij}$'s so that they satisfy (4.18). For any given node $i$, we choose a fixed out-coming arc: $(i, j_i) \in \delta^+(i)$. Then we define:

$$\pi_{ij_i} = \sum_{(i,j)\in\delta^+(i)} f_{ij} - 1, \tag{4.19a}$$

$$\pi_{ij} = 0, \ \forall (i,j) \in \delta^+(i) - \{(i,j_i)\}. \tag{4.19b}$$

Then by (4.17) and the above definition, we have:

$$\pi_{ij} \geq 0, \ \forall (i,j) \in A.$$

By (4.19a) we have:

$$\sum_{(i,j)\in\delta^+(i)} \pi_{ij} = \pi_{ij_i} + \sum_{(i,j)\in\delta^+(i)-\{(i,j_i)\}} \pi_{ij} = \sum_{(i,j)\in\delta^+(i)} f_{ij} - 1,$$

which is exactly (4.18). Thus we prove the equivalence. □

Hence formulation (4.16) is equivalent to the following (4.20) in which $\pi_{ij}$'s do not appear.

$$\min \sum_{(i,j)\in A} (Pm_{ij} - d_i) f_{ij} \tag{4.20a}$$

subject to

$$\sum_{(i,j)\in\delta^+(i)} f_{ij} \geq -1, \qquad \forall i \in N \tag{4.20b}$$

$$\sum_{(i,j)\in\delta^+(i)} f_{ij} - \sum_{(j,i)\in\delta^-(i)} f_{ji} = 0, \forall i \in N \tag{4.20c}$$

$$f_{ij} \geq -1, \qquad \forall (i,j) \in A \tag{4.20d}$$

Formulation (4.20) can be thought as a variant of the traditional network flow problem [35, 74]. The first set of constraints (4.20b) gives a lower limit on the sum of output flow for each node. The second set of constraints (4.20c) is the conservation law for the flow meaning that the flow coming into a node must equal to the flow

coming out of that node. If the first set of constraints (4.20b) have not appeared in (4.20), then it is the ordinary minimum cost network flow problem. We will show that how we can split the nodes in the graph to make the current formulation fit into the ordinary minimum cost flow problem.

Actually, we can replace each node $i$ in the original graph by two nodes $i'$ and $i''$. The original input arcs to node $i$ are now directed to node $i'$. The original output arcs from node $i$ are now going out from node $i''$. We also add a new arc from node $i'$ to node $i''$. See Figure 4.4 for the illustration of splitting a node $i$.

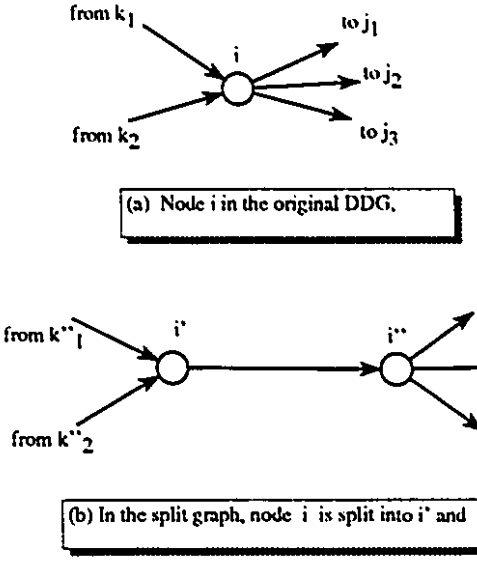



Figure 4.4: How node $i$ is split into $i'$ and $i''$.

Now consider the ordinary minimum cost flow problem on the result graph. Let $N'$ be the set of $i'$ nodes and $N''$ be the set of $i''$ nodes. We use $A'$ to denote the set of arcs in the result graph.

It is easy to see that the following minimum cost flow problem (4.21) is equivalent to (4.20).

$$\min \sum_{(u,v)\in A'} d'_{uv} f'_{uv} \tag{4.21a}$$

subject to

$$\sum_{(u,v)\in\delta^+(u)} f'_{uv} - \sum_{(v,u)\in\delta^-(u)} f'_{vu} = 0, \forall u \in N' \cup N'' \tag{4.21b}$$

$$f'_{uv} \geq -1, \qquad \forall (u,v) \in A'. \tag{4.21c}$$

where we define the cost coefficients in the objective function by:

$$d'_{uv} = \begin{cases} Pm_{ij} - d_i, & \text{if } u = i'' \in N'' \text{ and } v = j' \in N' \text{ and } (i,j) \in A, \\ 0, & \text{if } u = i' \in N' \text{ and } v = i'' \in N''. \end{cases}$$

**Lemma 4.4.2** *Formulation (4.21) and formulation (4.20) are equivalent, that is, given an optimal solution $\{f'_{uv}\}_{(u,v)\in A'}$ of (4.21) then the $\{f_{ij}\}_{(i,j)\in A}$ defined by the following formula is an optimal solution of (4.20):*

$$f_{ij} = f'_{uv}, \text{ if } u = i'' \in N'' \text{ and } v = j' \in N' \text{ and } i \neq j,$$

*Similarly, given an optimal solution $\{f_{ij}\}_{(i,j)\in A}$ of (4.20), the the following defined $\{f'_{uv}\}_{(u,v)\in A'}$ is an optimal solution of (4.21):*

$$f'_{uv} = \begin{cases} f_{ij}, & \text{if } u = i'' \in N'' \text{ and } v = j' \in N' \text{ and } (i,j) \in A, \\ \sum_{(i,j)\in\delta^+(i)} f_{ij}, & \text{if } u = i' \in N' \text{ and } v = i'' \in N''. \end{cases}$$

The proof of the lemma is straightforward, and is omitted.

It is well known that the minimum cost flow can always obtain an optimal integer flow if all the capacity constraints on the arcs are integral [74]. The capacity constraints on the arcs in (4.20d) and (4.21c) are integral, therefore they have optimal integral solutions. Actually the efficient out-of-kilter algorithm (see [58]) and its variants will give such an optimal integer solution when they are applied to (4.21).

**Theorem 4.4.3** *The problem (4.9) can be solved in $O(|N|^3 \log |N|)$ time, where $|N|$ is the number of nodes in the graph representing the loop.*

**Proof:** Recall that (4.11) is the dual of (4.9). Therefore it is equivalent to solve either of them. We have a series transformations to transform (4.11) to (4.20). Lemma 4.4.2 established the equivalence between (4.20) and (4.21). Therefore (4.11) is equivalent to (4.21) which is a minimum cost flow problem. Using the Out-of-Kilter method in [58] to solve the minimum cost flow problem, the algorithm for our case has a complexity $O(|N|^3 \log |N|)$. Hence our transformation procedures never use more than $O(|N|^3 \log |N|)$ time, we can conclude here that (4.11) can be solved in $O(|N|^3 \log |N|)$ time. □

### 4.4.3 Back Substitution

The $t_i$ variables give the optimal schedule of the nodes. The $b'_i$ variables have to be substituted back by the formula:

$$b_i = \frac{b'_i}{P}, \ \forall i \in N. \tag{4.22}$$

However such $b_i$'s may not always be integral since we have done a divide operation in (4.22). If we simply round the $b_i$'s to their integral ceilings, a suboptimal solution may result. We can solve this problem by noticing that by this time the schedule is already produced. By fixing the schedule to be the $t_i$'s produced by the solution of OSBA, we can use the lower bound in (4.5) and take the integer ceiling of it to obtain the integer lower bound on $b_i$. Thus we use the following formula to obtain the integer value for $b_i$:

$$b_i = \max \{ \lceil \frac{t_j - t_i}{P} \rceil + m_{ij}, \ \forall (i,j) \in \delta^+(i) \}, \ \forall i \in N. \tag{4.23}$$

## 4.5 Example Continued

In this section, we apply our OSBA procedure to the example loop given in Section 4.2. The loop is given in (4.1). Its data dependence graph is shown in Figure 4.1. The number beside an arc is its dependence distance.

There are two directed cycles in the data dependence graph. One cycle $C_1$ is $s_1, \rightarrow, s_2, \rightarrow, s_3, \rightarrow, s_1$. The other cycle $C_2$ is $s_1, \rightarrow, s_3, \rightarrow, s_1$. The length (sum of delays on the nodes along the cycle) of $C_1$ is 4 and the sum of its distances is 2. The length of $C_2$ is 2 and the sum of its distances is 2. Therefore the B-ratios of the two cycles are $\frac{1}{2}$ and 1. Hence the critical B-ratio is $\frac{1}{2}$ which also gives the optimal period P(=2) for our periodic scheduling.

The OSBA problem formulation (4.7) for loop $L_1$ is in (4.24).

$$\begin{aligned} & \min\ b_1 + b_2 + b_3 \\ \text{subject to} & \\ & 2b_1 + t_1 - t_2 \geq 0 \\ & 2b_1 + t_1 - t_3 \geq 0 \\ & 2b_2 + t_2 - t_3 \geq 0 \\ & 2b_3 + t_3 - t_1 \geq 4 \\ & t_2 - t_1 \geq 1 \\ & t_3 - t_1 \geq 1 \\ & t_3 - t_2 \geq 2 \\ & t_1 - t_3 \geq -3 \end{aligned} \tag{4.24}$$

Then we do variable substitution (4.8), and obtain the following formulation:

$$\begin{aligned} & \min\ b_1 + b_2 + b_3 \\ \text{subject to} & \\ & b'_1 + t_1 - t_2 \geq 0 \\ & b'_1 + t_1 - t_3 \geq 0 \\ & b'_2 + t_2 - t_3 \geq 0 \\ & b'_3 + t_3 - t_1 \geq 4 \\ & t_2 - t_1 \geq 1 \\ & t_3 - t_1 \geq 1 \\ & t_3 - t_2 \geq 2 \\ & t_1 - t_3 \geq -3 \end{aligned} \tag{4.25}$$

Solving (4.25) for loop L, we obtain the following scheduling and guidelines for the buffer sizes:

$$b'_1 = 4,\ b'_2 = 3,\ b'_3 = 2. \tag{4.26}$$

$$t_1 = 0,\ t_2 = 1,\ t_3 = 3, \tag{4.27}$$

If we round up the solution for the $b_i$'s by dividing 2 to the values in (4.26), we would end up with 5 registers. However if we use (4.23) to calculate the real need for the buffers that support the schedule, then we can have the following allocation:

$$b_1 = 2,\ b_2 = 1,\ b_3 = 1. \tag{4.28}$$

which uses only 4 buffers.

The actual schedule for the loop L is shown in (4.2). In (4.2), notice that by the time node $s_3$ is first scheduled, its predecessor node $s_1$ has been executed 2 times. That is why we allocated a FIFO buffer queue of size 2 for node $s_1$.

## 4.6 Code Generation

In this section we discuss the code generation problem based on our solution of the OSBA problem in Section 4.4.

The unique aspect of our code generation methods is that the buffer queue to each node has multiple heads. If we have allocated more than one buffer to a node, by organizing them as a FIFO queue we can make sure that the results are consumed in the same order as they are produced.

Conceptually, the new result produced by a node should always be written to the tail of the corresponding FIFO buffer queue, while the successors of the node should read the results at the proper places of the queue. It is possible that a successor should read the result from a place in the FIFO buffer queue other than the head. In other word, the queue should have multiple heads. The intuition is that these successor nodes are, in general, executing at different time instances in the software pipelined

schedule. And the buffer shifts its contents each time a new value is produced by the associated node. Therefore the successors need to read from different places of the queue. Hence, a FIFO with multiple heads is required. Such a multiple-head queue was illustrated in Figure 4.2 in Section 4.2.

In the rest of this section, we illustrate two schemes to generate code which implements the the FIFO buffer queue with multiple heads using registers. The tradeoff of dedicated hardware architecture support will also be discussed.

**Scheme I: Access Stationary Coding (ASC).** In this scheme, the FIFO buffer queue between a producer node and its successor nodes is directly accessed using fixed register assignment for the tail and the heads. This assignment is "stationary", and will remain the same during the entire execution. On the other hand, the data in the FIFO are explicitly shifted each time the producer node is writing a new value to its tail. The "shifting" can be realized by issuing multiple register move instructions, or by special architecture support for register shifts.

**Scheme II: Data Stationary Coding (DSC).** In this scheme, instead of letting the registers in a buffer shift their contents, we simply let the next iteration write to the next position in the corresponding FIFO buffer. Thus, data are kept *stationary*, while accesses to the registers of a FIFO buffer by the producer and successors are performed with the modulo addressing method. Similar code generation schemes can be seen in [72].

### 4.6.1 Scheme I: Access Stationary Coding

Under the Access Stationary Coding (ASC) scheme, the code generated with register shifting for our example loop $L_1$ is given in (4.29). In the table, at clock cycle 0 (or cycles 2, 4, 6, etc) the FIFO buffer of two registers allocated to node $s_1$ (for variable $a$) shifts its contents, and a new value is written into its tail $a_0$. We assume that at the beginning of the clock cycle, all the old contents are read off from the registers, and at

the end of the same clock cycle the new contents are written back into the registers. Therefore $a_1 = a_0, a_0 = X + c$ have the effect of shifting the old contents in $a_0$ to the register $a_1$ and the new result is written into the tail $a_0$ at the same clock cycle. It is safe to overwrite $a_1$ at this moment because the schedule and the supporting buffer allocation guarantee that the old contents of $a_1$ are no longer used. We can always align the shifting operation at the point when the corresponding instruction is issued.

| | iteration 1 | iteration 2 | iteration 3 | iteration 4 |
|---|---|---|---|---|
| 0 | b-shift: $a_1 = a_0$<br>$s_{1,1} : a_0 = X + c$ | | | |
| 1 | $s_{2,1} : b = a_0 * F$ | | | |
| 2 | | b-shift: $a_1 = a_0$<br>$s_{1,2} : a_0 = X + c$ | | |
| 3 | $s_{3,1} : c = a_1 + b$ | $s_{2,2} : b = a_0 * F$ | | |
| 4 | | | b-shift: $a_1 = a_0$<br>$s_{1,3} : a_0 = X + c$ | |
| 5 | | $s_{3,2} : c = a_1 + b$ | $s_{2,3} : b = a_0 * F$ | |
| 6 | | | | b-shift: $a_1 = a_0$<br>$s_{1,4} : a_0 = X + c$ |
| 7 | | | $s_{3,3} : c = a_1 + b$ | $s_{2,4} : b = a_0 * F$ |
| 8 | | | | |
| 9 | | | | $s_{3,4} : c = a_1 + b$ |

(4.29)

To ensure that the successors also read the correct results from the right places, we have to calculate the positions for them to read in the FIFO buffer queue. We have seen the use of multiple-head buffer in Section 4.2. Here we give a lemma to calculate the positions for the successors to access the data from the buffers:

**Lemma 4.6.1** *Let $(i, j)$ be a dependence arc in the DDG, that is to say, that node $i$ is the producer and node $j$ is a consumer (successor). The formula to calculate the position from which node $j$ should read in the FIFO buffer queue of node $i$ is:*

$$index_{ij} = \lceil \frac{t_j - t_i}{P} \rceil + m_{ij} - 1. \tag{4.30}$$

**Proof:** Node $i$ writes the result to the tail of its FIFO buffer at time $t_i + (K-1) * P$ in iteration $K$. This result will be read by node $j$ in iteration $K + m_{ij}$. Node $j$ in iteration $K + m_{ij}$ is scheduled at cycle $t_j + (K - 1 + m_{ij}) * P$. Therefore the time difference between the production and the consumption is:

$$[t_j + (K - 1 + m_{ij}) * P] - [t_i + (K-1) * P]$$

$$= t_j - t_i + m_{ij} * P.$$

During this time interval, there are

$$\lceil \frac{t_j - t_i + m_{ij} * P}{P} \rceil = \lceil \frac{t_j - t_i}{P} \rceil + m_{ij}$$

many register shiftings for the buffer queue allocated to node $i$. Also note that the above formula is independent of iteration $K$. Because the buffer queue for node $i$ is numbered from 0 at the tail end, hence node $j$ (in any iteration) should read from the buffer position indexed by the above formula minus 1, which is (4.30). □

As an example, we use formula (4.30) to calculate the positions node $s_2$ and node $s_3$ read from the FIFO buffer of size 2 allocated for node $s_1$. For node $s_2$, we have

$$index_{1,2} = \lceil \frac{t_2 - t_1}{P} \rceil + m_{1,2} - 1$$

$$= \lceil \frac{1 - 0}{2} \rceil + 0 - 1$$

$$= 0,$$

so node $s_2$ should read from $a_0$

For node $s_3$, we have

$$index_{1,3} = \lceil \frac{t_3 - t_1}{P} \rceil + m_{1,3} - 1$$

$$= \lceil \frac{3 - 0}{2} \rceil + 0 - 1$$

$$= 1,$$

so node $s_3$ should read from $a_1$.

If we examine the code in (4.29) line by line, we discover a repeating pattern of code from clock cycles 2 to 3, as shown below in (4.31).

| iteration i | iteration i+1 |
|---|---|
| | b-shift: $a_1 = a_0$<br>$s_{2,i+1} : a_0 = X + c$ |
| $s_{3,i} : c = a_1 + b$ | $s_{2,i+1} : b = a_0 * F$ |

(4.31)

We will use this repeating pattern as our new loop body. The original loop is now transformed into a new parallel loop body plus a prologue and an epilogue. The important fact is that the new loop body uses only of P(=2) clock cycles, which means that in every 2 clock cycles a new iteration will start. That is the optimal rate we can obtain. The new parallel loop is shown in (4.32), in which the $\|$ sign means "execute in parallel with".

| |
|---|
| prologue code:<br>$a_1 = a_0$<br>$a_0 = X + c$ |
| for i = 1 to n-P do<br>$a_1 = a_0 \,\|\, a_0 = X + c$<br>$c = a_1 + b \,\|\, b = a_0 * F$<br>enddo |
| epilogue code:<br>*noop*<br>$c = a_1 + b$ |

(4.32)

Generally, let

$$T = \max \{t_i;\ i \in N\},$$

then the pattern is formed from clock cycle $T - P + 1$ to clock cycle $T$.

So far, we have assumed that the register shifting operation can be implemented using register moves (copying) in conventional architectures. However, it is also possible that a processor architecture supports register shifting directly in hardware. Such

support allows the ALUs be devoted to other computation functions, thus improves the performance.

### 4.6.2 Scheme II: Data Stationary Coding

The Data Stationary Coding (DSC) scheme proposed here is intended to avoid register shifting in the previous ASC scheme. Instead of letting the registers to shift their contents, we simply let the next iteration write to the next position in the corresponding FIFO buffer. For the successor nodes, we can not simply use formula (4.30) to calculate the positions to read in the FIFOs. Instead we must use modulo addressing according to the following lemma:

**Lemma 4.6.2** *For a dependence arc (i,j), if in the current iteration node i is writing to position $Q_i$ (where $Q_i$ is the index) of its buffer queue, then node j in current iteration should read from position:*

$$index'_{ij} = (Q_i - m_{ij}) \ mod \ b_i, \tag{4.33}$$

*where $b_i$ is the buffer size.*

**Proof:** Suppose that the current iteration is $K$. Then node $i$ writes to the position $(K-1)$ mod $b_i$ since the data is not moved. In current iteration $K$, node $j$ should read the result produced by node $i$ in iteration $K - m_{ij}$. Therefore node $j$ in current iteration should read from position $(K - m_{ij} - 1)$ mod $b_i$. Substitute $K-1$ with $Q_i$, we obtain the formula (4.33). □

The code generated by using modulo addressing is shown in (4.34). For example, at clock cycle $P(i-1)$, we have the instruction $a_{(i-1) \bmod 2} = X + c$ for iteration $i$.

| clock cycle | iteration i |
|---|---|
| $P(i-1)$ | $s_{1,i} : a_{(i-1) \bmod 2} = X + c$ |
| $P(i-1)+1$ | $s_{2,i} : b = a_{(i-1) \bmod 2} * F$ |
| $P(i-1)+2$ | |
| $P(i-1)+3$ | $s_{3,i} : c = a_{(i-1) \bmod 2} + b$ |

(4.34)

Its expanded version is shown in (4.35).

| | iteration 1 | iteration 2 | iteration 3 | iteration 4 |
|---|---|---|---|---|
| 0 | $s_{1,1} : a_0 = X + c$ | | | |
| 1 | $s_{2,1} : b = a_0 * F$ | | | |
| 2 | | $s_{1,2} : a_1 = X + c$ | | |
| 3 | $s_{3,1} : c = a_0 + b$ | $s_{2,2} : b = a_1 * F$ | | |
| 4 | | | $s_{1,3} : a_0 = X + c$ | |
| 5 | | $s_{3,2} : c = a_1 + b$ | $s_{2,3} : b = a_0 * F$ | |
| 6 | | | | $s_{1,4} : a_1 = X + c$ |
| 7 | | | $s_{3,3} : c = a_0 + b$ | $s_{2,4} : b = a_1 * F$ |
| 8 | | | | |
| 9 | | | | $s_{3,4} : c = a_1 + b$ |

(4.35)

In (4.35) the repeating pattern is from clock cycle 2 to clock cycle 3, which is shown below in (4.36).

| iteration i | iteration i+1 |
|---|---|
| | $s_{1,i+1} : a_{i \bmod 2} = X + c$ |
| $s_{3,i} : c = a_{(i-1) \bmod 2} + b$ | $s_{2,i+1} : b = a_{i \bmod 2} * F$ |

(4.36)

We can see from (4.36) that the pattern derived by using the DSC scheme contains fewer instructions than that of the ASC scheme, which is due to the elimination of the register shifting operations. The new parallel loop body is shown in (4.37).

| prologue code |
|---|
| for i = 1 to n-P do<br>$a_{i \bmod 2} = X + c$<br>$c = a_{(i-1) \bmod 2} + b \parallel b = a_{i \bmod 2} * F$<br>enddo |
| epilogue code |

(4.37)

## 4.7 Reduce Register Requirement Further: Step 2

In Section 4.6 we showed how to generate code from a repeating pattern. That finishes the first step of our register allocation scheme. At this point, the FIFO buffer sizes and the schedule are all determined. However we still have the chance to share buffer elements if their live ranges do not overlap with each other for this fixed schedule. Hence the second step of our register allocation scheme is to apply the conventional coloring algorithm(s) [16, 15, 49] to further reduce the register requirement.

For each of the instructions that we allocated a buffer of size 1, the register may be thought as a symbolic register. Each such symbolic registers may be reused. For an instruction we allocated a buffer queue of size more than 1, only the head of the queue has the chance to be shared with other the buffers of the other nodes. Other elements of the buffer queue are live throughout the entire range of the repeating pattern, and therefore can not be shared with other buffers.

In our example loop $L_1$, suppose that we use the ASC scheme to generate code, then we choose the repeating pattern in (4.31), and draw the live-range diagram of the variables in Figure 4.3 in Section 4.2.

We can draw the interference graph according to the circular arcs in Figure 4.3, and color the interference graph with 3 colors. For instance, the following is a legal coloring with 3 colors: color_1 = $\{a_1, c\}$, color_2= $\{b\}$, color_3 = $\{a_0\}$. Therefore the actual number of registers required for the repeating pattern is 3 for these 3 colors, plus 2 extra for loop invariants $X$ and $F$, which totals 5 registers. In general we can use the coloring algorithms [16, 15, 49] to obtain the minimum number of registers used in the new loop body. In this section, we apply a recent method of cyclic interval graph coloring [49].

After the coloring algorithm is applied, the final code for the repeating pattern is shown in (4.38), in which $c$ is replaced by $a_1$ since they have the same color.

| prologue code |
|---|
| for i = 1 to n-P do<br>$a_1 = a_0 \parallel a_0 = X + a_1$<br>$a_1 = a_1 + b \parallel b = a_0 * F$<br>enddo |
| epilogue code |

(4.38)

The coloring algorithm can also be applied to the code produced by the DSC scheme. However the live ranges of the registers in a buffer of size more than one may last for several repeating patterns (new iterations) because there are no explicit register-shiftings. For instance, the live ranges of the variables in the code (4.37) generated by the DSC scheme are shown in Figure 4.5. In the picture no two variables can be colored the same color. Hence the code already uses minimum number of registers and we do not need to change the code again.

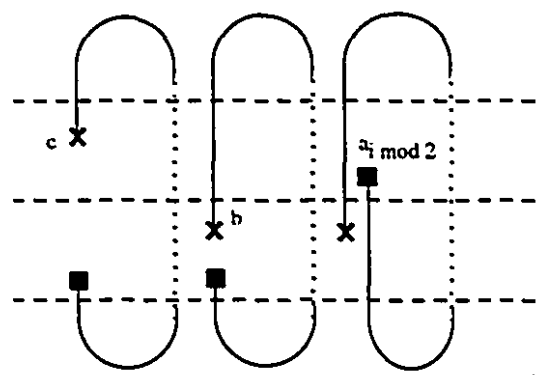



Figure 4.5: Live range intervals for code generated by DSC scheme.

## 4.8 Special Cases

In this section we look at two special cases of the OSBA formulation (4.7). The first special case is the result by Callahan et al's [13] for a fixed sequential program.

The second special case considers loops without loop-carried dependences [63], which makes the problem easier to solve than the general OSBA problem.

### 4.8.1 Callahan et al's Result

Callahan et al [13] considered the problem of using several registers for a subscripted variable to eliminate most of the loads and stores for that variable in a loop. They assume that the (sequential) execution order for the loop has already been fixed. Then they look at the dependence arcs outcoming from an instruction. They choose a number $\tau$ which is the longest dependence distance across iterations from this instruction. Then they allocate $\tau + 1$ registers to the instruction which writes to a subscripted variable. In this way they can eliminate the store of the subscripted variable in the current iteration and the load in the subsequent iterations which use this subscripted variable.

Now suppose that we are given a fixed execution order for a sequential architecture. We can still think that the schedule for the sequential machine is periodic because the iterations are executed in a periodic way. However in this case the period $P$ is the total execution time of a whole iteration. Therefore we have

$$t_i \leq P, \ \forall i \in N. \tag{4.39}$$

Thus the lower bound on the number of buffers for node $i$ becomes:

$$b_i \geq \lceil \frac{t_j - t_i}{P} \rceil + m_{ij}, \ \forall (i,j) \in \delta^+(i) \tag{4.40a}$$

$$\geq 1 + m_{ij}, \ \forall (i,j) \in \delta^+(i) \tag{4.40b}$$

which is exactly Callahan et al's formula to calculate the amount of registers needed to an instruction $i$.

In this sense our method is more general than theirs since we can also handle parallel schedules, and more importantly we do not fix the schedule so that best schedule will be found which uses minimum amount of registers.

### 4.8.2 Loops without Loop Carried Dependences

A lot of the inner most loops in the program does not contain any loop-carried dependence. Even if a loop contains a loop-carried dependence, dependence cycles may not occur. This section considers this situation and tries to exploit this special property to investigate its implications.

First of all, if no dependence cycle exists, all the iterations can be scheduled at the same starting time. The whole execution time is that for one iteration. This approach to scheduling is not practical for any configuration of hardware unless the loop has a small iteration upper bound. However if the loop has a very large iteration upper bound, one must control the amount of parallelism in the loop to fit the hardware configuration.

Note that since the DDG for a loop without loop-carried dependence has no cycles, the computation rate is not defined by Definition 2.6.1. But we can consider that the maximum computation rate is infinite. However we can still use a periodic schedule and choose a positive period for the iterations, to control the amount of parallelism. For example we can choose 1 as the period or the rate. Then every iteration will start one clock cycle after its previous iteration. If one iteration of the loop body does not have enough parallelism to fully utilize the processing units, a proper number of iterations can be unrolled first, and then follow the periodic scheduling scheme.

When the period and the rate are 1, the formulation of the OSBA problem becomes much simpler. Since period $P$ is equal to 1, the constraint matrix in the OSBA problem is a {0,1} matrix. So we do not need to substitute variables. Another not so trivial observation is that all the symbolic registers allocated as buffers are busy all the time without any empty time slots, which makes the second step of the register allocation framework redundant. Let $B_i$ be the memory space in which the output tokens of node $i$ can be stored.

**Lemma 4.8.1** *In an optimal storage allocation scheme the memory spaces allocated to two different nodes (where the two nodes can output their result tokens into) can*

*not overlap, that is,*

$$B_i \cap B_j = \emptyset, \ \forall i, j \in N, \ i \neq j. \tag{4.41}$$

**Proof:** Since we assume that the loop under consideration has no loop-carried dependence, our dataflow graph representing the loop has no (directed) cycle. Therefore once node $i$ is fired at time $t_i$ for the first time, it can be fired consecutively for the rest of the iterations, that is, $i$ will be fired at time $t_i + 1$ for the second iteration, at time $t_i + 2$ for the third iteration, etc. A similar argument is true for node $j$, that is, it will be fired at time $t_j, t_j + 1, t_j + 2, \cdots$. This means that the output tokens of a node will be produced at a rate of one token per cycle. The input tokens will be consumed by the node also at a rate of one token per cycle because the successor nodes will also be fired every cycle. Now the empty slot in $B_i$ will be filled up by the consecutive firings of node $i$ in the initial period and later on when a token in $B_i$ is consumed at some cycle, the slot can be filled up in the next cycle by the result token from node $i$ fired at that cycle. Therefore the output tokens of node $j$ can not be put into $B_i$, which means

$$B_i \cap B_j = \emptyset.$$

□

## 4.9 Experimentation Results

We have implemented our algorithm and used it to test some loops selected from benchmark programs. This section gives an overview of the implementation, and shows the experimentation results.

We want to test how many floating-point registers and floating-point functional units are needed to support our scheme for typical loops selected from a collection of benchmark loops. The loops tested are selected in Livermore Loops, SPEC benchmarks and Whetstone benchmarks. Since loops without loop-carried data dependences are easy to be parallelized, and relatively easy to be handled when register

allocations are concerned, we have restricted ourselves to the loops that contain loop-carried data dependences. Due to the limitation of our tools, we also restricted ourselves to loops containing no conditional tests in the loop body. This limitation can be eliminated if the conditionals have been dealt with before the code is passed to our scheduling and allocation program.

All the loops are written in Fortran. We have isolated each loop we selected in a single file. Then we manually rewrite a loop in order to make the loop in high level language in a form of the assembly or three address code so that the scheduling and register allocation produced are realistic. That includes:

- Break the long expressions into sequence of three address instructions, i.e., each instruction has at most two input operands and one output operand. Temporaries are generated if necessary.
- Load and Store instructions are also inserted when we break the long expressions. However we do not go to the details of computing the addresses of the array references since they are mainly integer operations.
- We have used assumptions in Table 4.1 for the execution delays of the instructions. They are typical numbers [51] found in available commercial or research machines like IBM RS/6000 [78] and Cydra 5 [72]. The delay for *Load* instruction may not be a constant depending on whether the load is a hit in cache or not. The number of cycles we listed in Table 4.1 should be understood as the delay of a hit in the second level cache.

To obtain the data dependence information between pairs of instructions of the loops, we have used Parafrase-2 [68, 48] developed at University of Illinois at Urbana-Champaign, which can process Fortran programs and generate all the dependences we need. For a loop, with the dependence information obtained from Parafrase-2, our program will do the following:

1. Compute the optimal period $P$ for this loop.

| instructions | clock cycle(s) |
|---|---|
| Add | 1 |
| Subtract | 1 |
| Negate | 1 |
| Multiply | 2 |
| Divide | 17 |
| Load | 13 |
| Store | 1 |

Table 4.1: Execution delays of the instructions.

2. Ask the user to change the period to a bigger period if he or she wants. The purpose of this is to let the user have the choice of making comparisons on the register usages for different periods.

3. Generate a schedule with period $P$ and an optimal buffer allocation simultaneously by solving the OSBA problem (4.7).

4. Using the generated schedule, compute the repeating pattern and the live range intervals of the buffers.

5. Use an optimal coloring algorithm to color the intervals. The reason that we choose to use an optimal coloring algorithm, which does backtracking and can run in exponential time in the worst case, is that we want to obtain exact results of the register usage. Actually the algorithm runs very fast in all of our testings where a loop body typically contains less than 30 floating-point variables.

6. Collect all the statistics of the schedule, which include: Number of registers used, Number of buffers allocated, Number of functional units needed to support the schedule and the Average buffer queue length for each variables.

We have applied our program to 22 loops selected. Figure 4.6 shows the number of total buffers and the number of total registers allocated to each loop. Figure 4.7

shows the average buffer queue length in each loop. In Figure 4.8 we show the number of functional units needed for each loop.

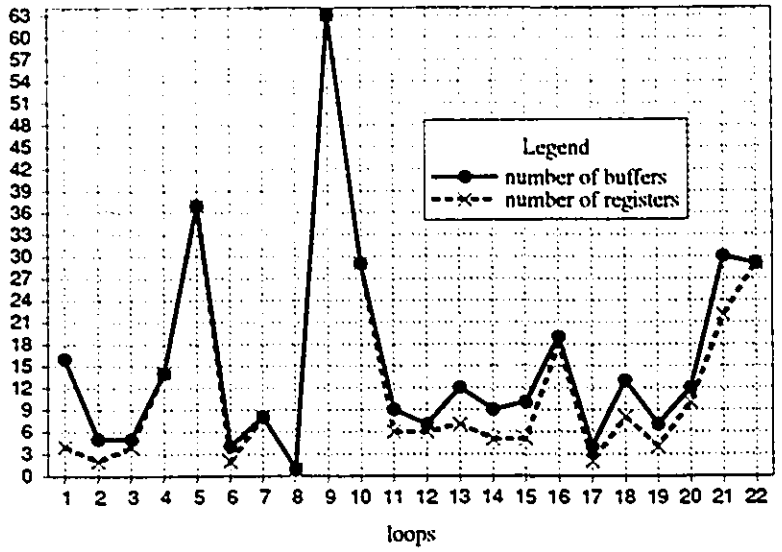



Figure 4.6: Buffers and registers allocated to each loop.

In Table 4.2, we show the averages of our experimental results. The average number of buffers allocated for each loop is 15.6. After the coloring step, the average actual register requirement is 13. The improvement of the coloring step over the buffers is about 16.6%. There are 72.2% of the loops actually used less than 16 registers. Furthermore, 90.9% of the loops used less than 32 registers.

The average functional units required is 2.73, which is obtained by counting all the instructions, including loads. So the number is a little over-estimated if the underlining architecture allows load instructions to by-pass the functional units. Actually, 17 out of 22 loops used less or equal to three functional units. That represents 77.3% of the loops selected. Only 22.7% of the loops used more than three functional units. These loops show much higher parallelism than the average. We have also tested to use a bigger period to control the amount of parallelism in a loop, and the test results do indicate that the register usage and functional units requirement are down.

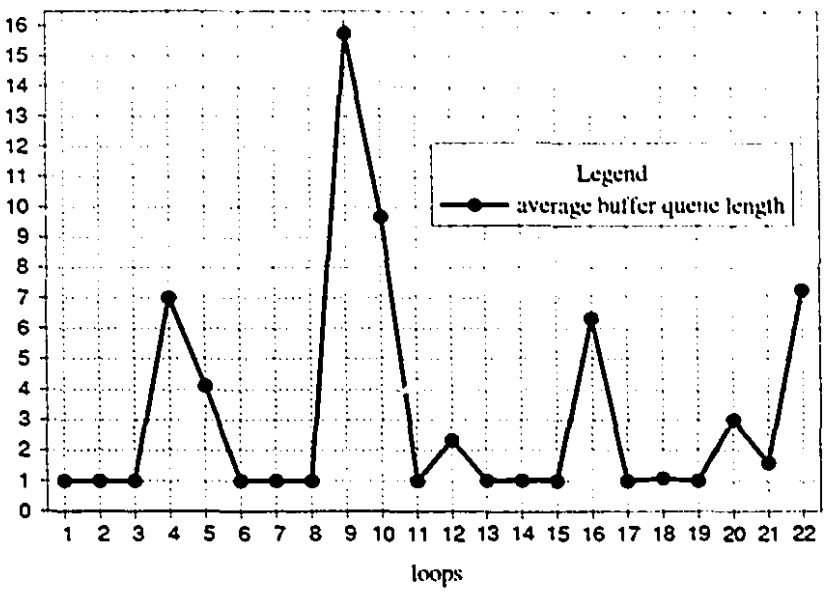



Figure 4.7: Average buffer queue length in each loop.

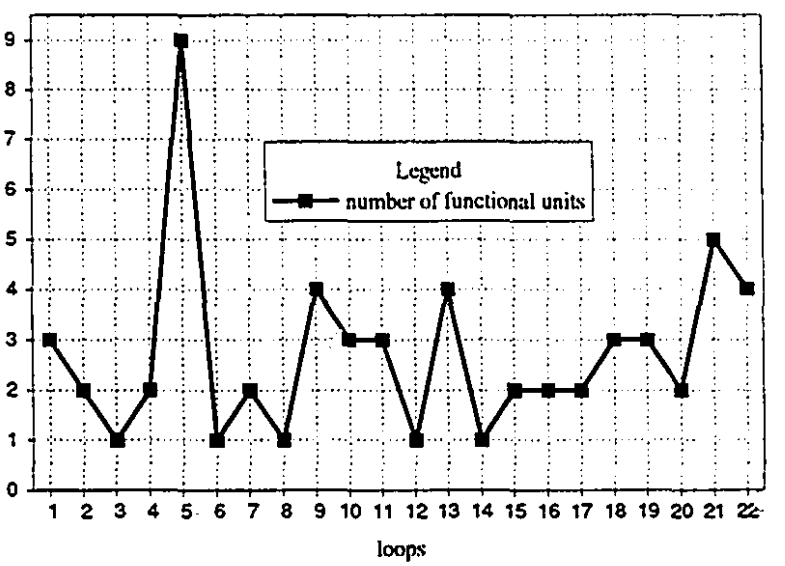



Figure 4.8: Number of functional units needed for each loop.

| | |
|---|---|
| number of loops tested | 22 |
| average number of buffers allocated for each loop | 15.6 |
| average number of registered allocated for each loop | 13 |
| average buffer queue length for each instruction | 3.19 |
| average number of functional units needed for each loop | 2.73 |

Table 4.2: Experimental Results.

## 4.10 The Example from Rau Et Al's Paper

In this section, we look at the example loop given in Rau Et Al's paper [72]. The loop is shown below in (4.42).

$$\begin{array}{|l|}\hline \text{for i = 1 to n do} \\ s = s + a[i] \\ a[i] = s * s * a[i] \\ \text{enddo} \\ \hline \end{array} \tag{4.42}$$

Its low level representation, like 3-address code, is shown below.

| | | |
|---|---|---|
| $a$ : | $vr33 = vr33 + vr32$ | % vr33 is address of a[i] % |
| $b$ : | $vr34 =$ load $m(vr33)$ | % vr34 = a[i]% |
| $c$ : | $vr35 = vr35 + vr34$ | % vr35 = s % |
| $d$ : | $vr36 = vr35 * vr35$ | |
| $e$ : | $vr37 = vr36 * vr34$ | % vr37 = new a[i] % |
| $f$ : | store$(vr37, m(vr33))$ | |
| | branch to $a$ if $i \leq n$ | |
| | enddo | |

(4.43)

We focus on the low level representation in this section. The data dependence graph of the low level presentation is shown in Figure 4.9. The delay for Add and Store is 1, the delay for Multiply is 2 and delay for Load is 13.

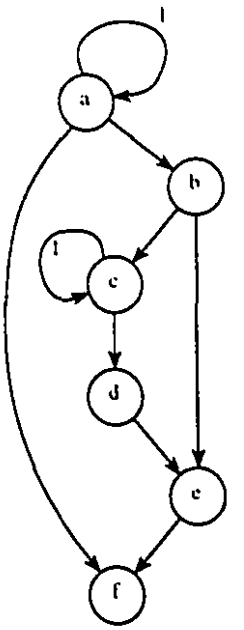



Figure 4.9: Data dependence graph of the low level code of Rau's example.

There are two directed cycles in the dependence graph, which are self-loops from node $a$ to $a$ and from node $c$ to $c$. The B-ratio of it is 1. Therefore we can generate a schedule with period $P = 1$. However since Rau et al used 2 as their period in [72], we will also use 2 as our period for generating the schedule and the register allocation. The OSBA formulation of the low level representation is the following:

$$\min b_a + b_b + b_c + b_d + b_e + b_f$$

subject to

$$\begin{aligned}
&2b_a + t_a - t_b \geq 0\\
&2b_a + t_a - t_f \geq 0\\
&2b_a + t_a - t_a \geq 2\\
&2b_b + t_b - t_c \geq 0\\
&2b_b + t_b - t_e \geq 0\\
&2b_c + t_c - t_d \geq 0\\
&2b_c + t_c - t_c \geq 2\\
&2b_d + t_d - t_e \geq 0\\
&2b_e + t_e - t_f \geq 0\\
&t_b - t_a \geq 1\\
&t_f - t_a \geq 1\\
&t_a - t_a \geq -1\\
&t_c - t_b \geq 13\\
&t_e - t_b \geq 13\\
&t_d - t_c \geq 1\\
&t_e - t_c \geq -1\\
&t_c - t_d \geq 2\\
&t_f - t_e \geq 2
\end{aligned} \tag{4.44}$$

After variable substitution (4.8), we have:

$$\min\ b_a + b_b + b_c + b_d + b_e + b_f$$

subject to

$$\begin{aligned}
& b'_a + t_a - t_b \geq 0 \\
& b'_a + t_a - t_f \geq 0 \\
& b'_a + t_a - t_a \geq 2 \\
& b'_b + t_b - t_c \geq 0 \\
& b'_b + t_b - t_e \geq 0 \\
& b'_c + t_c - t_d \geq 0 \\
& b'_c + t_c - t_c \geq 2 \\
& b'_d + t_d - t_e \geq 0 \\
& b'_e + t_e - t_f \geq 0 \\
& t_b - t_a \geq 1 \\
& t_f - t_a \geq 1 \\
& t_a - t_a \geq -1 \\
& t_c - t_b \geq 13 \\
& t_e - t_b \geq 13 \\
& t_d - t_c \geq 1 \\
& t_c - t_c \geq -1 \\
& t_e - t_d \geq 2 \\
& t_f - t_e \geq 2
\end{aligned} \tag{4.45}$$

By solving (4.45) we obtain the following solution:

$$t_a = 0,\ t_b = 1,\ t_c = 14,\ t_d = 15,\ t_e = 17,\ t_f = 19; \tag{4.46}$$

$$b'_a = 19,\ b'_b = 16,\ b'_c = 2,\ b'_d = 2,\ b'_e = 2,\ b'_f = 0. \tag{4.47}$$

Instruction $f$ is not allocated a buffer because it is a "Store" instruction. With the technique (4.23) in Section 4.4, we obtain the following buffer allocation:

$$b_a = 10,\ b_b = 8,\ b_c = 1,\ b_d = 1,\ b_e = 1,\ b_f = 0. \tag{4.48}$$

The second step is to analyze the live ranges of the buffers and use a coloring algorithm to reduce the register requirement further. In this example, coloring algorithm can not reduce the number of registers further. So there is a total of 21 registers allocated. In Rau et al's paper [72], they used 28 registers which is partly due their assumption that a register can not be released for reuse until all the consumers have *finished* their whole executions, while we assume that a register can be released if all the consumers have read the value, not necessarily finished [69].

The repeating pattern of the schedule is shown in the following table, in which $c_{i+2}$ means instruction $c$ in its original iteration $i+2$:

| | | | | |
|---|---|---|---|---|
| pattern cycle 1: | | | $c_{i+2}$ | $a_{i+9}$ |
| pattern cycle 2: | $f_i$ | $e_{i+1}$ | $d_{i+2}$ | $b_{i+9}$ |

(4.49)

The average buffer queue length is 4.2 buffers. The number of functional units needed to support this scheme is 3 if the "Store" instruction $f$ does not use one. The sequential execution of the loop will take 20 clock cycles to finish one iteration, while the current parallel schedule finishes one iteration by overlapping the iterations in 2 clock cycles. Therefore the speedup is 10.

If we would have taken period $P = 1$, then we could have needed 40 registers and 5 functional units to support this optimal speed.

## 4.11 Related Work

The early work by Aiken and Nicolau [2, 3, 4] did not consider the register allocation problem. In a recent paper, Nicolau et al [62] considered the register allocation problem by renaming for the compaction-by-percolation based algorithms. Ebcioglu et al have proposed the technique of *enhanced software pipelining* with resource constraints [29, 30, 28, 61]. However, they did not consider the minimum register allocation problem as discussed in this paper.

In Lam's work on software pipelining [56], an interesting scheme called *modulo variable expansion* is proposed to allow a scalar loop variable be expanded to use more than one location so that the unnecessary precedence constraints due to scalar variable in different iterations can be removed. However modulo variable expansion is only performed after the schedule has been fixed. The work described in this paper can be considered as an extension to modulo variable expansion in the sense that it is incorporated in a unified framework of time-optimal scheduling, and minimizes the amount of storage for scalar variable expansion and array variable shrinkage.

Callahan, Carr and Kennedy have studied register allocation for subscripted variables. In their method, array references which are live across several iterations are recognized and a source-to-source transformation called *scalar replacement* is performed such that they can be handled by coloring-based register allocators. However, their work is aimed at sequential loop execution and does not consider loop scheduling such as software pipelining. We have shown in Section 4.8 that our OSBA formulation (4.7) includes Callahan et al's result as a special case.

In a recent paper by Rau et al [72], a method of register allocation for software pipelining was presented. In this method, register allocation is performed after the so-called *modular scheduling phase*. Successive iterations are initiated at a fixed *initiation interval*. The register allocation problem is formulated as a bin-packing problem of *vector lifetimes* on a cylinder surface. A heuristic algorithm has been proposed for the register allocation and has been demonstrated to be quite effective by experimental results. However, the paper did not attempt to describe a complete concurrent scheduling-allocation strategy for software pipelining. Other related work can be found in [33, 27].

# Chapter 5

# Cycle Balancing Scheme

On dataflow architectures, there still exists the challenge of how to maximally exploit fine-grain parallelism to speed up loop execution while not incurring excessive storage space overhead. This chapter considers a broader class of scheduling and the storage allocation schemes to support dynamic scheduling on dataflow architectures. The minimum storage requirement to support the maximum computation rate on a dataflow architecture is analyzed and a storage minimization method called *Cycle Balancing Scheme* (CBS) is introduced in this chapter. The Cycle Balancing problem is formulated as an integer programming problem. A polynomial time algorithm of the linear relaxation problem is presented which gives a fractional approximate solution of the Cycle Balancing problem. We also prove that CBS has the *Totally Dual Integral* (TDI) property, which allows the Cycle Balancing problem being solve as a linear programming problem if the right-hand-sides are rounded to their integer ceilings.

## 5.1 Introduction

Under the dataflow model, a computation is described by a dataflow graph. Unlike von Neumann computers, dataflow computers have no program counter or other form of centralized control mechanism. Consequently, the order of instruction execution is restricted only by data dependencies within the dataflow programs. Most dataflow architectures assume that the scheduling of the actors is done by a dynamic scheduler which maintains a pool of enabled actors. In this chapter we propose a balancing technique for the dataflow graph of a loop so that the maximum computation rate can be achieved and only the minimum amount of storage is required.

One of the long standing issues in loop execution on dataflow machines is how to manage the fine-grain parallelism and the storage requirement supporting such parallelism. Under the static dataflow architecture model, the storage for a loop is completely determined at compile-time — because each arc in the dataflow graph is allocated one storage unit. A main restriction of this architecture and storage allocation scheme is that it may not be able to fully exploit the parallelism to achieve the maximum computation rate of the loop. *Dataflow software pipelining* has been proposed to organize the code such that several iterations may be proceeding concurrently [38, 40, 44]. The number of concurrent iterations is restricted by the amount of storage allowed for one copy of the loop body. Under the pure dynamic dataflow model, such a restriction has been eliminated by *unraveling* the loop body dynamically at runtime, and the execution can initiate as many iterations as possible given that enough memory is available, limited only by data dependences [9]. Although dynamic dataflow architectures provide opportunity to fully exploit fine-grain parallelism in the loop, managing the amount of storage has been a challenge [8, 11, 20, 39, 42, 40, 65, 66].

In this chapter, we have developed a framework to determine, at compile-time, the minimum storage requirement to fully exploit the fine-grain parallelism in a loop. The framework is developed under a *FIFO* dataflow model where each arc in the dataflow graph is organized as a FIFO queue of certain size. We recall the result in Theorem 2.6.1 that the maximum computation rate of a loop is bounded by the *critical cycles*

in the data dependence graph or dataflow graph. Based on this observation, we will allocate buffers to the arcs in the dataflow graph such that no cycles are allocated more than what are needed. This will guarantee that the optimal computation rate is not slowed and that no extra storage is allocated. In this chapter, the buffer storage will be allocated to the arcs of the dataflow graph, instead of to the nodes in the previous chapter. The main results of this chapter are:

- The minimum storage requirement to support the maximum computation rate is analyzed and a storage minimization scheme called *Cycle Balancing Scheme* (CBS) is introduced. The basic intuition is that, since the maximum computation rate is dominated by critical cycles in the loop, we should not allocate extra storage beyond a certain bound limited by the *balancing ratio* of the critical cycles, defined in Section 2.6.

- The CBS is formulated as an integer programming problem. Since integer programming problems are hard to solve in general, we concentrate on the linear relaxation problem of the CBS. A polynomial time algorithm of the linear relaxation of the CBS is presented which gives a fractional approximate solution of the Cycle Balancing problem. It reduces the problem to a network flow problem called *minimum circulation flow* problem. We also prove that the CBS has the *Totally Dual Integral* (TDI) property, which allows the Cycle Balancing problem to be solved as a linear programming problem if the right-hand-sides are rounded to their integer ceilings.

The subsequent sections are organized as follows: In Section 5.2 we provide a brief description of dataflow architectures. Then we give an example to motivate our cycle balancing problem in Section 5.3. Then, in Section 5.4 we formulate the cycle balancing problem as an integer programming problem. In Section 5.5 we give a polynomial time algorithm to solve the linear relaxation problem of the CBS. In Section 5.6 we prove that the CBS has the totally dual integrality property.

## 5.2 Dataflow Architectures

Dataflow architectures [23, 24, 25, 7, 26] execute operations represented in a dataflow graph . The original proposed dataflow architectures do not have a memory model to address the issue of how to store the result values or the intermediate values. Their description of computation is pure functional. However storage (such as register) allocation problems must be solved on dataflow architectures to achieve cost effective performance.

Now we introduce some notation for a dataflow graph. Let us first consider an operation which is not a conditional test, say an addition. Such an operation is represented as a node in a dataflow graph. If a node $n_i$ needs the value computed by node $n_j$, then there is an arc from $n_j$ to $n_i$. The arc $(n_j, n_i)$ is called an input arc of node $n_i$, and it is called an output arc of node $n_j$. The arcs in a dataflow graph are used to transmit data between operations. Therefore they represent the same dependence relation as data dependence arcs in a DDG. Data are represented as *tokens* on the arcs in dataflow graphs. A node in the dataflow graph is called enabled if each of its input arcs contains at least one token. An enabled node can be executed or fired. The result of the execution or firing is that exactly one token from each of its input arcs is removed and exactly one token is added to each of its output arcs. Figure 5.1 shows the action of firing of a node in a dataflow graph, where the black dots indicate tokens.

A conditional test operation is treated very differently from an ordinary operation. The test itself is represented by a node. The input arc or arcs to the test node will provide data to the test. However the result of the test will output a special boolean valued token on each of its output arcs. The special boolean token can only take two values, true or false. A computation involving conditional test is represented by the so called conditional schema [22] in Figure 5.2, in which two kinds of special nodes, i.e. the switch nodes and the merge node, are used to select the branch that is taken.

A switch node has two input arcs, one is a data arc and the other is the (control) arc from the conditional test node. The switch node will consume one token from its

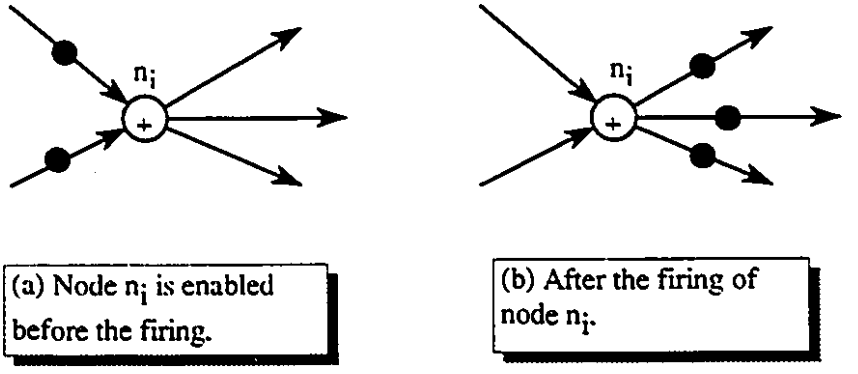



Figure 5.1: Firing of a node in dataflow graph.

data input arc, and consume a token from its input control arc from the conditional test. The switch node has two output arcs which are called true and false output ports. If the boolean token on the input control arc carries a true value, then the output token will be routed to the true port, and the value of the output token equals the value of the input data token on the input data arc. If the value of boolean token on the input control arc is false, then the output token that equals the input data value is put on the false port. in any case only one of the two output arcs from the switch node will obtain a token.

A merge node is used to join the two branches. A merge node has three input arcs, two of them are data input arcs connected to the true port and the false port of the node, respectively. The third input arc is the control arc from the conditional test node. The semantics of firing a merge node is that it will remove one token from either the the true port or the false port, but not both, according to whether the value of the boolean token on the input control arc is true or false. Of course it will also remove the token from the input control arc. Then the merge node will output a token on each of the output arcs which has a value equal to that of the input data token on the chosen port.

Using merge nodes and switch nodes, a loop can be represented in a dataflow graph by the so called iterative schema [22]. Figure 5.3 shows an iterative schema

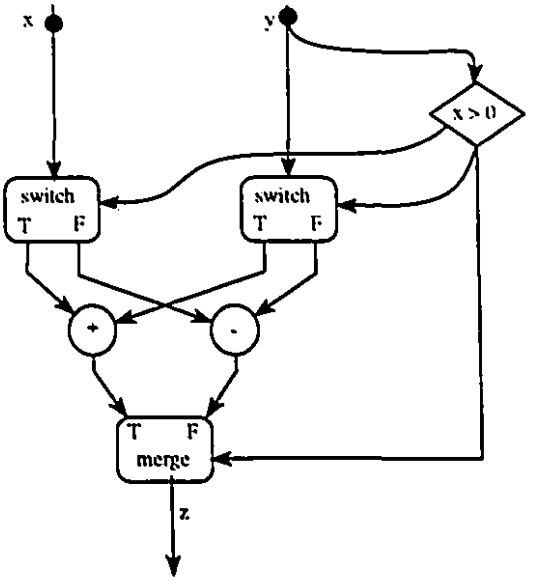



Figure 5.2: A conditional schema in a dataflow graph representing "if x > 0 then z = x+y else z = x-y".

that computes the following loop:

```
for i = 1 to U do
sum = sum + a[i];
enddo;
```
(5.1)

However if the loop upper bound is very large, since all the testings will take the true branch which will come back to the beginning of the loop, we can simplify the loop schema so that the conditional test, the switch nodes and the merge node are omitted. In Figure 5.4 we show the simplified version of the loop schema in Figure 5.3.

In the simplified version of the loop schema, we can see that it is very much like a data dependence graph of the loop. Actually the only difference is that in a dataflow graph the dependence distances of the arc are indicated by the number of tokens on the arcs. Subsequently, we will use simplified version of loop schema to represent a

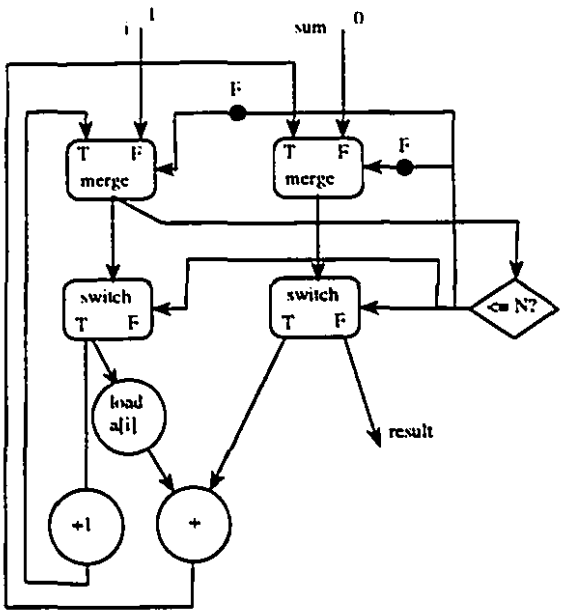



Figure 5.3: A iterative schema in a dataflow graph representing the loop in (5.1).

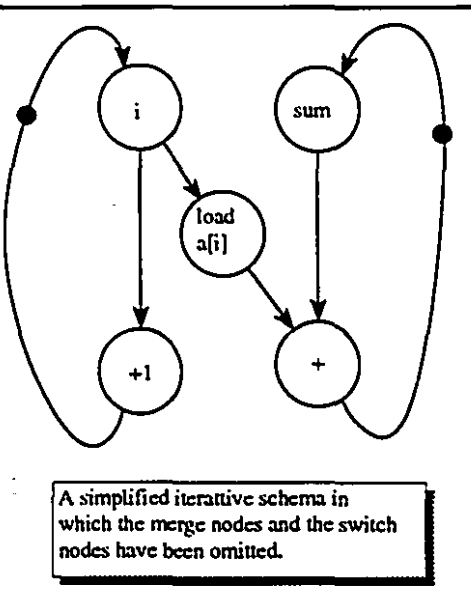



Figure 5.4: A simplified version of loop schema.

loop in later sections. Since the simplified version of dataflow schemata for loops has no difference to data dependence graphs in term of representing the data dependences, the definitions about data dependence graphs can all be applied to these simplified dataflow graphs.

**Argument-Flow Dataflow Architecture [22, 25, 9]** : An argument-flow dataflow computer architecture is one that computes a dataflow graph according to the semantics of the dataflow graph. The data tokens are assumed to be sent explicitly by the producer nodes to the consumer nodes along the arcs.

**Argument-Fetching Dataflow Architecture [26]** : An argument-fetching dataflow computer architecture is one that also computes a dataflow graph according to the semantics of the dataflow graph. However a producer node will store its result data token in the memory system, and a consumer node will have the concept of the address of its input token, and go to that address to explicitly fetch that data from the memory system.

With these definitions we can see that a producer node in an argument-flow architecture has to duplicate its result to its multiple consumers, while as in an argument-fetching architecture model, the producer stores only one copy of its result.

## 5.3 Example and Motivation

In this section, we use an example to show the challenge of minimizing the storage requirement while keeping the maximum computation rate of loop execution on dataflow machines. Related work will be discussed in Section 5.7. Let's consider the loop $L_2$ containing *loop-carried dependencies*, shown in (5.2).

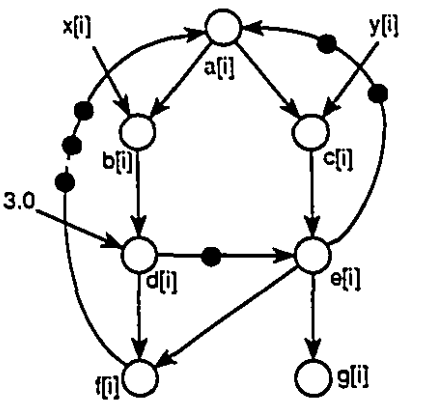
| | |
|---|---|
| $L_2$ : | for i = 1 to n do |
| $a$ : | $a[i] = f[i-3] + e[i-2]$ |
| $b$ : | $b[i] = a[i] + x[i]$ |
| $c$ : | $c[i] = a[i] + y[i]$ |
| $d$ : | $d[i] = b[i] + 3.0$ |
| $e$ : | $e[i] = c[i] + d[i-1]$ |
| $f$ : | $f[i] = d[i] + e[i]$ |
| $g$ : | $g[i] = e[i] + e[i]$ |
| | enddo; |

(5.2)

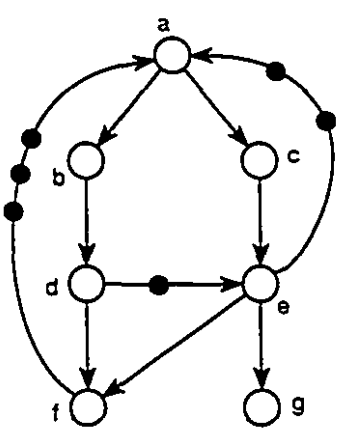
Figure 5.5 shows the dataflow graph of the loop $L_2$. Figure 5.5 (a) contains external input arcs which are omitted from Figure 5.5 (b). Note that a complete translation of $L_2$ into a dataflow graph also contains loop control actors, such as switch and merge actors [24]. We assume that the loop is executed a very large number of iterations. Therefore, it is reasonable to assume that the switch and merge actors for loop control will always take a fixed branch path except for the start and termination of the loop. For simplicity, we omit them from Figure 5.5.

The arcs in the dataflow graph represent the data dependencies. When there are a black dot or dots (called tokens) on an arc $(h, k)$, then that means the arc $(h, k)$ is a loop-carried dependence, and the number of tokens on the arc $(h, k)$ represents the iteration distance of the dependence. For example, there are three tokens on arc $(f, a)$, which means that the result produced by $f$ at the current iteration $i$ will be used by $a$ three iterations later at iteration $i + 3$.

Under the static dataflow model, it is assumed that each arc in the dataflow graph can hold at most one token. So one storage location is allocated for each of the arcs. In a static dataflow architecture, a loop-carried dependence of distance $> 1$ can be represented in one of the two ways. Either we can unroll the loop a number of times so that all the loop-carried dependences are of distance one, or we can use a chain of $m_{hk}$ arcs to join nodes $h$ and $k$ such that there is one token on each of the arcs




(a) Dataflow graph for loop L.




(b) Dataflow graph without outside input arcs.

Figure 5.5: Dataflow graph for loop $L_2$

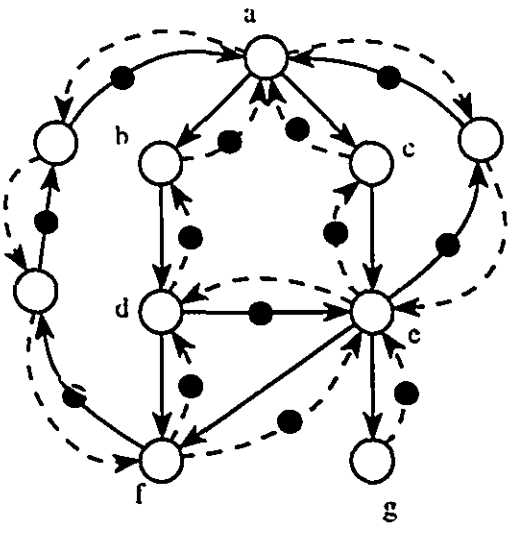



Figure 5.6: Static dataflow graph and its storage allocation.

on the chain. Therefore static dataflow architecture can indeed handle loops with loop-carried dependences of distance more than one. However the limitation of one token per arc in static dataflow architecture will limit its ability to fully exploit the parallelism in the loop and also limit its maximum computation rate to be at most $\frac{1}{2}$. For our example loop $L_2$, the static dataflow architecture uses 13 memory spaces, which can only run at a maximum rate of $\frac{1}{2}$ because the feedback arcs create new cycles of length 2 — for instance cycle $a \rightarrow b \rightarrow a$, as shown in Figure 5.6.

*Dataflow software pipelining* was originally proposed to exploit fine-grain parallelism in loops on static dataflow computers [40, 38]. As a result, there may be several iterations executed concurrently with one copy of the dataflow graph for $L_2$. The code is mapped such that successive waves of element values of the input arrays x, y and z (corresponding to inputs to successive iterations) will be fetched and fed into the dataflow graph of the loop body. So the computation may proceed in a pipelined fashion. However, the loops considered in [40, 38] have no loop-carried dependences. Therefore the main limitation of static dataflow model is that it may not be able to

fully exploit the parallelism in the loop if loop-carried dependences exist. For example, in the example loop $L_2$ in (5.2), its maximum computation rate is $\frac{2}{3}$ limited by critical cycle $a \rightarrow c \rightarrow c \rightarrow a$. However the static architecture model can not achieve that speed.

We will use a model more general than the static dataflow model, but simpler than the dynamic dataflow model, in the organization of the memory structures. We assume that an arc can hold an unbounded number of tokens in a First-In-First-Out (FIFO) queue. Later we will give a more formal definition of our model. Our problem is to find bounds on how many tokens each arc can hold, so that maximum computation rate as defined in Theorem 2.6.1 can be supported.

Under the dynamic dataflow architecture, such limitation is eliminated via *loop unraveling* [7], which virtually provides unbounded amount of storage for each arc. The storage minimization problem, however, still exists for dynamic dataflow machines. To execute a loop on a dynamic dataflow machine, multiple instances of one operation corresponding to different iterations can be initiated concurrently, limited only by the data dependences of the loop. This is accomplished by the *loop unraveling* scheme, where (in more "modern" implementations such as the Monsoon dataflow machine [66]) each iteration is allocated its own *activation frame* containing all memory spaces required to hold its operands. Therefore each frame requires as much memory as the total number of arcs of the dataflow graph. This allows an iteration to begin its initiation as soon as the data values for the iteration have arrived. With many iterations simultaneously active in the machine, dynamic dataflow model may provide an opportunity to exploit far more parallelism than the static dataflow model. As pointed out in [20], however, exploiting more parallelism will invariably increase the resource requirement of a program. The challenge is not to allow the loop to consume more storage than necessary to fully exploit the parallelism in a loop. Consider our example loop $L_2$, there could be three iterations running at the same time. Hence three frames each requiring 10 memory spaces need to be allocated, totaling 30 memory spaces. We will show that our method will require substantially less memory spaces (actually 16 buffers) than that.

Let us ask the following question: what is the minimum amount of storage that the loop $L_2$ needs to run at the *maximum computation rate* under an idealized dynamic dataflow architecture? The idea is to add storage control arcs and check all the cycles so that each cycle is allocated enough storage in order to maintain a balancing ratio (defined in Section 2.6) greater than or equal to the ratio of the critical cycles. We now consider the example loop $L_2$ in Figure 5.5. There are three loop-carried dependency arcs, which form 5 cycles:

$$C_1 = \{(a,b),(b,d),(d,f),(f,a)\}$$
$$C_2 = \{(a,c),(c,e),(e,a)\}$$
$$C_3 = \{(a,b),(b,d),(d,e),(e,a)\}$$
$$C_4 = \{(a,c),(c,e),(e,f),(f,a)\}$$
$$C_5 = \{(a,b),(b,d),(d,e),(e,f),(f,a)\}$$

Their balancing ratios are:

$$R(C_1) = \frac{3}{4},\quad R(C_2) = \frac{2}{3},\quad R(C_3) = \frac{3}{4},\quad R(C_4) = \frac{3}{4},\quad R(C_5) = \frac{4}{5}.$$

Therefore $C_2$ is the critical cycle, and the maximum computation rate is $\frac{2}{3}$. In Figure 5.7 we add some storage control arcs which are indicated by dotted lines. The two tokens on $(d,a)$ mean that we have allocated two buffers, organized as a FIFO queue, to the chain from $a$ to $d$. Similarly the two tokens on arc $(f,d)$ indicated that two buffers have been allocated to arc $(d,f)$, etc. Therefore a total of 16 buffers are allocated, which allows the loop $L_2$ to run at the maximum computation rate of $\frac{2}{3}$. That is because all the cycles in Figure 5.7 have their balancing ratios at least $\frac{2}{3}$.

## 5.4 Cycle Balancing Scheme (CBS)

From Theorem 2.6.1 in Chapter 2 we know that the maximum computation rate of a loop is dominated by its critical cycles. Therefore, we will allocate just enough

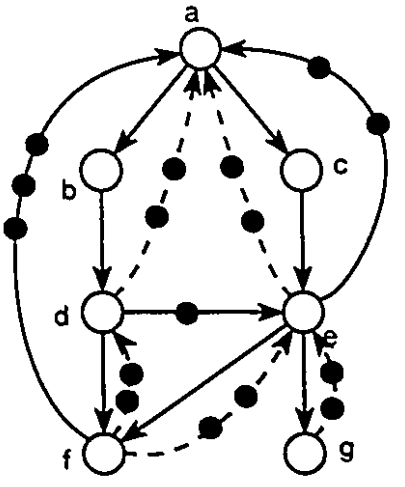



Figure 5.7: Storage Allocation by our CBS uses 16 buffers.

storage to each cycle so that its balancing ratio at least as big as the balancing ratio of the critical cycles. In this section, we propose that a compiler should be able to determine the storage allocation such that all cycles have the same balancing ratio as that of a critical cycle. We call this procedure *cycle balancing* for dataflow graphs [41].

In this section we will show how to formulate the cycle balancing problem as an integer programming problem. There are two steps in the formulation process. The first step is called *chain replacement*, i.e. replacing each chain in the dataflow graph with a single arc. This has the effect of sharing the storage for all the actors along a chain. The second step is to derive an integer programming formulation which will optimize the memory allocation. We will show in Section 5.5 that the relaxation linear programming problem can be solved in polynomial time. We will also show in Section 5.6 that the integer programming problem has the *totally dual integral* (TDI) property, which will also allow us to solve the integer programming itself by solving a linear programming problem.

### 5.4.1 Chain Replacement

Let's first state what is a chain.

**Definition 5.4.1** *Given a dataflow graph $G = (N, A; m, d)$, if a node $n$ has only one input arc and only one output arc, then it is called simple. A path is called a chain if all the nodes lying internally in the path (i.e. not including the two end nodes) are simple.*

Chains are the most simple structures in a dataflow graph. The obvious optimization of storage allocation is that to consider a chain as a single arc if one choose to share the storage among the arcs along a chain. However we should remember that more general sharing of the storage among arbitrary actors gives rise to an NP-complete optimization problem (Theorems 3.2.1 and 3.3.1). Therefore in this chapter we restrict ourself to chain replacement only.

Our method will allocate storage to a dataflow graph on a chain by chain basis. For instance, consider a chain $Q$ of length $x$. Our algorithm may assign a total of $y$ buffers to the chain $Q$. Conceptually, the $y$ buffers will be shared by the tokens traveled along $Q$. For simplicity in later formulation, we can replace each chain by an arc. The implementation issue for chain replacement should be straight forward.

**Definition 5.4.2** *Given a dataflow graph $G = (N, A; m, d)$, and a chain $Q$ in it, let $h$ be the starting node of chain $Q$, and $k$ the end node of $Q$. A chain replacement of $Q$ is the replacement of $Q$ by a new arc joining $h$ and $k$. The length of the new arc is the sum of the lengths of arcs along the chain $Q$. If all the chains have been replaced by arcs, then the resulting graph is called the skeleton, and is denoted by $SG$.*

### 5.4.2 Integer Programming Formulation

After we have done the chain replacement for a given dataflow graph $G$ we obtain a "skeleton" of $G$, indicated by $SG$. Of course if no chain replacement has happened, $G = SG$.

To limit the number of memory spaces allocated on the arcs in the skeleton $SG$, we introduce a new *storage control arc* $(k, h)$ corresponding to each arc $(h, k)$ in $SG$, as defined in Definition 5.4.3. A storage control arc has the effect of limiting the number of tokens that can reside on an arc at any moment. But storage control arcs are only used for the purpose of calculating the amount of storage that should be allocated to each individual arc in the original dataflow graph. In the actual execution of the dataflow graph, storage control arcs do not exist and therefore add no extra data dependences.

**Definition 5.4.3** *Given a skeleton of a dataflow graph SG=(N, A; m, d), for each arc* $(h, k) \in A$, *if* $m_{hk} = 0$ *and* $(k, h) \notin A$, *then we add a storage control arc* $(k, h)$ *with* $x_{kh}$ *initial tokens, which is a variable to be determined later, and which has the effect of limiting the number of tokens residing on arc* $(h, k)$. *The set of storage control arcs will be denoted by* $A^-$ *as they are in the opposite directions of their corresponding dependence arcs. Each arc in* $A^-$ *has a length of 1, which reflects the timing assumption that each node will use one time step to take a token from its input arc and is ready to take the next token after this step* [1]. *The number of initial tokens on each of these new control arcs is defined to be zero. The resulting graph is called Augmented Dataflow Graph, or ASG for short.*

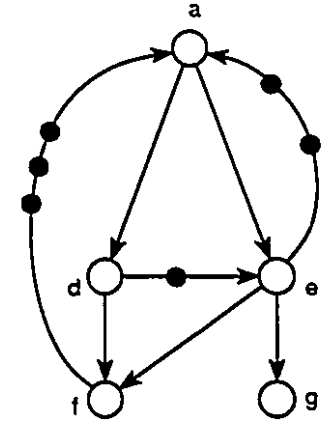
See Figure 5.8 for an example of augmented dataflow graph.

Let us notice that an augmented dataflow graph itself is still a dataflow graph if all the $x_{kh}$'s have been fixed as constants.

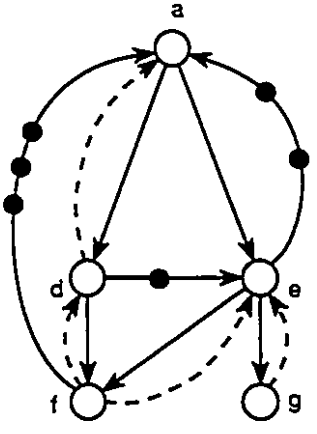
**Lemma 5.4.1** *The simplified dataflow graphs, containing no merge and switch nodes, have the property that during their execution the sum of tokens on any cycle does not change.*

**Proof:** To see why the lemma is true let us fix a cycle $C$ and let $u$ be an actor on $C$. When an actor is fired it consumes one token from its input arc on $C$ and produces

[1] Of course this parameter can be adapted to hardware configurations.




(a) Skeleton of dataflow graph of L after chain replacement.




(b) Augmented skeleton ASG with storage control arcs indicated by dotted lines.

Figure 5.8: An example of dataflow graph and its augmented dataflow graph.

one token on its output arc on $C$. Therefore no token is added to $C$ or lost from $C$, the token is only *moved* from one side of $u$ to the other side of $u$. So the total number of tokens on $C$ does not change. □

Each storage control arc $(k, h)$ we just added in, together with its original dependence arc $(h, k)$, forms a cycle with a sum of $x_{kh}$ tokens on it. Therefore by the above lemma, the number of tokens that are $(h, k)$ can hold in the $ASG$ is at most $x_{kh}$. Hence the total memory required for the execution of the datallow graph is at most:

$$\sum_{(k,h)\in A^-} x_{kh} + \sum_{(h,k)\in A} m_{hk}.$$

in which the second sum is a constant. Thus our objective is to minimize the first term in the above expression. The introduction of the storage control arcs also introduces many new cycles in the $ASG$ that do not appear in $SG$. The balancing ratios of the new cycles might be smaller than that of the original critical cycles in $SG$ if the $x_{hk}$'s are not properly chosen. In order to support the maximum computation rate we must keep enough tokens — equivalently allocate enough memory spaces — in all cycles such that the balancing ratios of the new cycles is not smaller than that of the original critical cycles. In order to give the mathematical formulation of the minimization problem, we first introduce some notations.

**Definition 5.4.4** *Given an* $ASG = (N, A; m, d)$, *let* $C$ *be a directed cycle in the* $ASG$. *We use* $C^+$ *to indicate the arcs in* $C$ *which are not the storage control arcs, i.e.*

$$C^+ = C \bigcap A,$$

*and* $C^-$ *to indicate the arc in* $C$ *which are storage control arcs, i.e.*

$$C^- = C \bigcap A^-.$$

With these notations, we give the following formulation of the minimum memory allocation problem which can support the maximum computation rate.

$$\min \sum_{e=(h,k)\in A^-} x_{hk} \tag{5.3a}$$

subject to

$$\frac{\sum_{r\in C^-} x_r + \sum_{r\in C^+} m_r}{D(C)} \geq \frac{M(C^*)}{D(C^*)},\ \forall C \in \mathcal{C}(ASG) \tag{5.3b}$$

$$x_r \geq 0,\ \ x_r \ \text{integer}, \qquad \forall e \in A^- \tag{5.3c}$$

where $\mathcal{C}(\mathcal{ASG})$ is the set of all cycles in $ASG$ and $C^*$ is a critical cycle in the original dataflow graph $G$ or $SG$.

There exists one constraint in (5.3b) for each cycle $C$ in the $ASG$. It ensures that the values of the $x_r$'s are big enough so that the balancing ratio of cycle $C$ (defined in Section 2.6) is not smaller than that of the critical cycle in the original dataflow graph. By moving the variables to the left hand side and the constant terms to the right hand side, and by defining

$$b_C = D(C)\frac{M(C^*)}{D(C^*)} - \sum_{e\in C} m_e,\ \forall C \in \mathcal{C}(ASG) \tag{5.4}$$

then the formulation can be rewritten into the following form, which is named *Cycle Balancing* problem:

---

**Cycle Balancing Problem (CB):**

$$\min \sum_{e\in A^-} x_e \tag{5.5a}$$

subject to

$$\sum_{e\in C^-} x_e \geq b_C,\ \forall C \in \mathcal{C}(ASG) \tag{5.5b}$$

$$x_e \geq 0, \qquad \forall e \in A^- \tag{5.5c}$$

$$x_e \ \text{integer}, \ \forall e \in A^- \tag{5.5d}$$

---

The solution of the CB problem will provide a storage allocation for all arcs in the skeleton of the dataflow graph, which will be enough to support a maximum rate

loop schedule. However the integer solution constraint gives us some difficulty in solving it as will be explained in Section 5.6. So one approach to obtain a solution is to solve the linear relaxation problem to obtain a fractional solution and up-round the solution to obtain a near optimal integer solution. The linear relaxation of the CB problem is to ignore the integer constraints (5.5d). We write the linear relaxation down as follows for future reference:

---

**Fractional Cycle Balancing Problem (FCB):**

$$\min \sum_{e \in A^-} x_e \tag{5.6a}$$

subject to

$$\sum_{e \in C^-} x_e \geq b_C, \ \forall C \in \mathcal{C}(ASG) \tag{5.6b}$$

$$x_e \geq 0, \quad \forall e \in A^- \tag{5.6c}$$

---

We should notice that there can be an exponential number of cycles in an $ASG$, which means that there could be an exponential number of constraints in (5.5b) and (5.6b) of the formulation of CB problem and FCB problem, respectively. Therefore it is not trivial to solve FCB in polynomial time in terms of size of the given $ASG$. Both the ellipsoid method [55] and the Karmarkar method [54] for solving linear programming have computation complexity in terms of both the number of variables and the number of constraints in the formulation. Since the number of constraints in our formulation is exponential, these algorithms can not be applied to our problem to obtain polynomial algorithms. In the method given in the next section we will explore the properties of the dual problem of FCB to obtain a polynomial time algorithm in terms of the size of $ASG$. Here we write down the formulation of the dual problem of FCB as follows:

---

**Dual of FCB (D-FCB):**

$$\max \sum_{C \in \mathcal{C}(ASG)} b_C z_C \tag{5.7a}$$

subject to

$$\sum_{C \ni e} z_C \leq 1, \qquad \forall e \in A^- \tag{5.7b}$$

$$z_C \geq 0, \qquad \forall C \in \mathcal{C}(ASG) \tag{5.7c}$$

---

The dual problem could have an exponential number of variables, since each variable in the dual problem corresponds to a constraint in the primal problem FCB. However we present a method which will produce at most $|A^-|$ positive $z_C$ values, and all the other $z_C$'s are zero. We will use this property to obtain a polynomial time algorithm in the next section.

## 5.5 Polynomial Time Solution of FCB

In this section, we show that the primal problem of the linear programming problem FCB (5.6) can be solved in polynomial time. We will actually show that its dual problem D-FCB (5.7) is equivalent to the so-called *circulation problem* [35]. Since the circulation problem can be solved in polynomial time [32, 58], we only need to show that the optimal solution of the circulation problem can be translated into an optimal solution of D-FCB (5.7) *in polynomial time*. When the dual problem D-FCB (5.7) can be solved in polynomial time, we can use this dual optimal solution to obtain an optimal solution of the primal problem FCB (5.6) also in *polynomial time*.

The general circulation problem is very similar to the minimum cost flow problem. The distinction is that in minimum cost flow problem, there is a source node and a sink node. The objective is to send a fixed amount of flow from the source to the sink so that a given objective function is minimized. In circulation problem, there is no source node or sink node, and flow circulates in the graph. Now let us consider the circulation problem formulation that fits our need. The circulation problem is, given a directed graph, to find a circular flow so that a given objective function is optimized (maximized or minimized). The mathematical formulation of the maximum

cost circulation problem is given in (5.9), which is defined on the graph $ASG$ with cost coefficients defined as:

$$p_e = l_e \frac{M(C^*)}{D(C^*)} - m_e, \ \forall e \in A \cup A^-, \tag{5.8}$$

where $C^*$ is a critical cycle in the original data flow graph and $l_e$ is defined in Section 2.5.

---

**Maximum Cost Circulation Problem (MCCP):**

$$\max \sum_{e \in A \bigcup A^-} p_e f_e \tag{5.9a}$$

subject to

$$\sum_{e \in \delta^+(h)} f_e - \sum_{e \in \delta^-(h)} f_e = 0, \ \forall h \in N \tag{5.9b}$$

$$f_e \leq 1, \qquad \forall e \in A^- \tag{5.9c}$$

$$f_e \geq 0, \qquad \forall e \in A \bigcup A^- \tag{5.9d}$$

---

The following theorem explains why this version of the circulation problem fits our need.

**Theorem 5.5.1** *Given an optimal solution of MCCP (5.9), we can construct an optimal solution of D-FCB (5.7), and vice versa.*

**Proof:** Let $f = \{f_e\}_{e \in A \bigcup A^-}$ be a feasible solution of MCCP (5.9). Let us define $f^1 = f$, i.e.

$$f^1_e = f_e, \ \forall e \in A \bigcup A^-$$

Let $S(f^1)$ be the support set of $f^1$, that is, the subset of arcs with positive flow values:

$$S(f^1) = \{e; \ e \in A \bigcup A^- \text{ such that } f^1_e > 0\}. \tag{5.10}$$

If the support set $S(f^1)$ is not empty, then it must contain a cycle by the nature of the circular flow it represents. Let $C_1$ be any given cycle in $S(f^1)$. Define $z_{C_1}$ to be the minimum flow value among all the arcs in $C_1$, that is,

$$z_{C_1} = \min\{f_e^1;\ e \in C_1\}. \tag{5.11}$$

Now we define a new circular flow $f^2$ by subtracting $z_{C_1}$ from the current flow $f^1$:

$$f_e^2 = \begin{cases} f_e^1 - z_{C_1}, & \text{if } e \in C_1, \\ f_e^1, & \text{if } e \notin C_1. \end{cases} \tag{5.12}$$

It is easy to check that $f^2 = \{f_e^2\}$ is another feasible circular flow of MCCP (5.9). But the number of arcs in the support set $S(f^2)$ of $f^2$ is strictly less than that in the support set $S(f^1)$ of $f^1$, because at least one arc in $C_1$ must have a zero flow in $f^2$. We can repeat the above procedure to choose another cycle $C_2$ in $S(f^2)$ and produce another circular flow $\{f^3\}$. In general, from the circular flow $f^h$, we choose one cycle $C_h$ in its support set $S(f^h)$ if it is not empty, and define $z_{C_h}$ to be the minimum flow value among all the arcs in $C_h$:

$$z_{C_h} = \min\{f_e^1;\ e \in C_h\}.$$

Then we define the next flow $f^{h+1}$ to be:

$$f_e^{h+1} = \begin{cases} f_e^h - z_{C_h}, & \text{if } e \in C_h, \\ f_e^h, & \text{if } e \notin C_h. \end{cases}$$

This procedure stops when the most recently produced circulation flow $f^{h+1}$ is zero everywhere, that is, its support set $S(f^{h+1})$ is empty.

By that time, we have defined positive variables:

$$z_{C_1},\ z_{C_2}, \cdots,\ z_{C_h}$$

for $h$ cycles

$$C_1, C_2, \cdots, C_h.$$

We define:

$$z_e = 0, \text{ for all the other directed cycles } C \text{ in } \mathcal{C}(ASG).$$

Let us note that for such defined $z_C$'s, there are only $h$ of them are strictly positive, where $h$ is less than or equal to the number of arcs in the ASG. All the others are zero valued. The number $h$ is smaller than or equal to the number of arcs in $A \bigcup A^-$ because each step of producing a new flow will take out at least one arc from the old support sets.

Next we show that such defined $\{z_C\}_{C \in \mathcal{C}(ASG)}$ is a feasible solution of D-FCB (5.7). First let us notice that for such defined $\{z_C\}_{C \in \mathcal{C}(ASG)}$, the following property holds:

$$\sum_{C \ni e} z_C = f_e, \ \forall e \in A \bigcup A^-. \tag{5.13}$$

We can prove (5.13) by induction on the number of cycles for which the support set $S(f)$ can be decomposed into. If $S(f)$ itself is a single cycle, then the amount of flow on all the arcs in this cycle must be the same because the flow is a circular one. So in (5.11), $z_{C_1}$ is equal to the amount of flow on each arc. Since there is only one cycle in the support set, $f^2$ in (5.12) is a zero flow, and all the other $z_C$'s are defined to be zero. Hence, for the arcs in the support set (5.13) is true, and for arcs $e$ not in the support set, i.e. $f_e = 0$, and any cycle $C$ passing through $e$ is defined as zero, therefore (5.13) is also true. Now consider the general case that the support set $S(f)$ can be decomposed into $h$ cycles. Then the support set $S(f^2)$ of $f^2$ can be decomposed into $h-1$ cycles. The induction hypothesis assumes that

$$\sum_{C \ni e \text{ and } C \neq C_1} z_C = f^2_e, \ \forall e \in A \bigcup A^-.$$

By the definition of $f^2$ in (5.12), the following is true:

$$f^1_e = f^2_e + z_{C_1}, \text{ if } e \in C_1,$$

$$f^1_e = f^2_e, \text{ if } e \notin C_1.$$

Therefore we have the following:

$$\sum_{C \ni e, \text{ and } C \neq C_1} z_C + z_{C_1} = f_e^1, \ \forall e \in C_1,$$

and

$$\sum_{C \ni e} z_C = f_e^2 = f_e^1, \ \forall e \notin C_1.$$

Hence (5.13) is true for all the $e$'s.

For any arc $e \in A^-$, the constraint for $e$ in problem D-FCB (5.7b) demands that the sum of the $z_C$'s for the cycles $C$ containing $e$ is bounded above by 1. This is true by the property in (5.13) since $f_e \leq 1$ for $e \in A^-$.

Now the objective value of such defined feasible $\{z_C\}$ is:

$$\begin{aligned}
\sum_{C \in \mathcal{C}} b_C z_C &= \sum_{C \in \mathcal{C}} \left\{ D(C) \frac{M(C^-)}{D(C^-)} - \sum_{e \in C} m_e \right\} z_C \\
&= \sum_{C \in \mathcal{C}} \left\{ \sum_{e \in C} l_e \frac{M(C^-)}{D(C^-)} - \sum_{e \in C} m_e \right\} z_C \\
&= \sum_{C \in \mathcal{C}} \sum_{e \in C} p_e z_C \\
&= \sum_{e \in A \bigcup A^-} \sum_{C \ni e} p_e z_C \\
&= \sum_{e \in A \bigcup A^-} p_e \sum_{C \ni e} z_C \\
&= \sum_{e \in A \bigcup A^-} p_e f_e
\end{aligned}$$

So the objective value of problem D-FCB (5.7) is the same as that of problem MCCP (5.9).

To show the reverse, let $\{z_C\}$ be a feasible solution of D-FCB (5.7), then it is easy to check that the flow defined by the following formula is a feasible solution to MCCP (5.9):

$$f_e = \sum_{C \ni e} z_C, \ \forall e \in A \bigcup A^-.$$

Of course, if one of the feasible solution is optimal, the other is also optimal since they will produce the same objective value. This proves that the MCCP (5.9) and the D-FCB (5.7) are equivalent. □

Therefore D-FCB (5.7) and MCCP (5.9) are equivalent in the sense that the solution of one problem also gives the solution of the other. We can see from our formulation that MCCP (5.9) has $2|N|$ constraints and $|A \bigcup A^-|$ variables. Both of these two numbers are polynomial to the original size of the *ASG*. In fact, the circulation problem MCCP (5.9) can be solved by polynomial time algorithms [36, 32, 58] which are much better than the general methods of the simplex algorithm, the ellipsoid algorithm [55] or Karmarkar's algorithm [54].

**Theorem 5.5.2 (Lawler, [58])** *MCCP (5.9) can be solved in* $O(|A \bigcup A^-|^2 \log |N|) = O(|A|^2 \log |N|)$ *time, where* $|A \bigcup A^-|$ *is the number of arcs of the graph ASG and* $|N|$ *the number of nodes in the graph.*

After we obtain the optimal dual solution of D-FCB (5.7), we can transform it to a primal optimal solution of FCB (5.6) in polynomial time by either the tableau method or the complementary slackness method in [17]. Hence we have proved the following theorem.

**Theorem 5.5.3** *The linear relaxation FCB (5.6) can be solved in polynomial time with a complexity* $O(|A|^2 \log |N|)$.

**Proof:** The solution of the MCCP (5.9) has a complexity of $O(|A|^2 \log |N|)$. Then all the transformations to obtain the solution of the D-FCB (5.7) and FCB (5.6) are not using more than $O(|A|^2 \log |N|)$ time. Therefore the total complexity of solving the FCB (5.6) is $O(|A|^2 \log |N|)$. □

The solution of FCB (5.6) can be rounded up to give an integer solution for CB (5.5). Let us look at our example to see how this is done. In our example of Figure

5.8, the optimal solution of the circulation problem MCCP (5.9) is the following:

$$f_{12} = f_{24} = 2,$$
$$f_{21} = f_{31} = f_{42} = f_{43} = f_{35} = f_{53} = 1,$$
and all other $f_{ij} = 0$.

The support set of this circulation flow is:

$$S(f) = \{(1,2),(2,1),(2,4),(4,2),(4,3),(3,5),(5,3),(3,1)\}.$$

The support set can be decomposed into four cycles as follows:

$$C_1 = \{(1,2),(2,1)\}$$
$$C_2 = \{(2,4),(4,2)\}$$
$$C_3 = \{(3,5),(5,3)\}$$
$$C_4 = \{(1,2),(2,4),(4,3),(3,1)\}$$

By the proof of Theorem 5.5.1 we can construct the optimal solution of the Dual of FCB (5.7) as:

$$z_{C_1} = z_{C_2} = z_{C_3} = z_{C_4} = 1.$$

This solution can be transformed to a solution of the primal problem FCB (5.6) as:

$$x_{21} = b_{C_1} = 2$$
$$x_{42} = b_{C_2} = \frac{4}{3}$$
$$x_{53} = b_{C_3} = \frac{4}{3}$$
$$x_{43} = b_{C_4} = \frac{4}{3}$$

When round-up to an integer solution, it is

$$x_{21} = x_{42} = x_{53} = x_{43} = 2$$

which happens to be optimal for the integer programming CB (5.5).

Although we have shown that FCB (5.6) can be solved in polynomial time, the solution of the integer version CB (5.5) in polynomial time remains an open problem. In the next section, we explore a special property, totally dual integrality, of the constraints of CB (5.5), and propose another approach to solve it by linear programming.

## 5.6 Totally Dual Integrality

In Section 5.5 we have shown that the linear relaxation FCB (5.6) of the CB (5.5) can be solved in polynomial time even though there could be an exponential number of constraints in the formulation. The solution obtained from (5.6) was not always an integer solution. Since we prefer to have an precise integer optimal solution instead of rounding up the fractional optimal solution to integers, a natural question to ask is how to obtain an exact integer optimal solution. One possibility is to show that the constraint matrix in the formulation is totally unimodular, because Theorem 2.10.1 guarantees that if the constraint matrix in the formulation is totally unimodular, then the linear relaxation problem will always have an integer optimum solution when the right hand sides of the constraints are integral. Hence by showing the totally unimodular property of the constraint matrix, one only needs to solve the linear relaxation problem which will guarantee to obtain an integer optimum solution.

Unfortunately, we show in this section that the constraint matrix in CB (5.5) is not *totally unimodular* (TUM). Therefore it is not immediately clear that the optimum solution of the linear relaxation will always be integral.

However we will show that the integer programming problem CB (5.5) has the *Totally Dual Integral* (TDI) property, which is a weaker property than the TUM in the following sense:

- A system of linear inequalities that has the TDI property does not necessarily have the TUM property,

- A system of linear inequalities that has the TUM property also has the TDI property.

Both properties are important for integer programming problems because they can guarantee that their linear relaxation problems can produce integral optimum solutions.

### 5.6.1 CB Problem Does Not Have TUM Property

We recall that in the formulation of CB (5.5), there is a constraint for each cycle $C$ in the augmented dataflow graph $ASG$ and there is a variable for each arc in $ASG$. Consider the dataflow graph $G$ in Figure 5.9 (a). Its augmented graph is in Figure 5.9 (b). We choose three cycles and three arcs in Figure 5.9 (b) so that the 3 by 3 square submatrix of the constraint matrix of CB (5.5) generated by these cycles and arcs has a determinant 2, which is not 0, 1 or $-1$ as required by the TUM property.

The three cycles in Figure 5.9 (b) are:

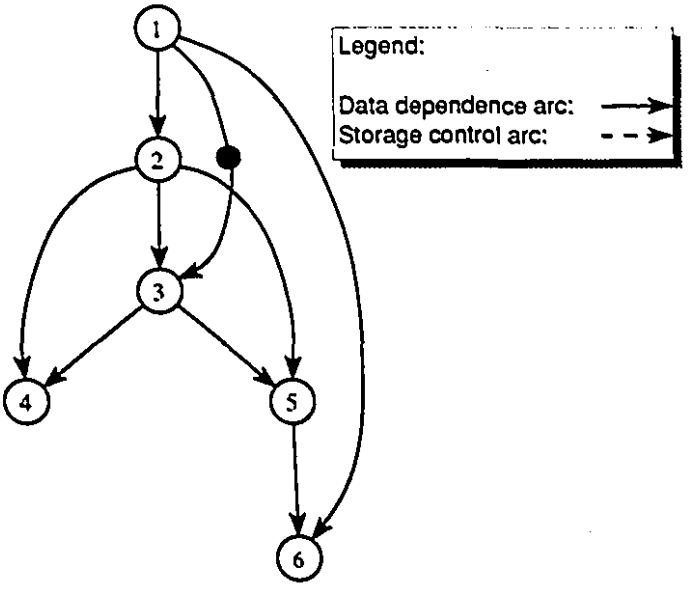
$$C_1 = \{e_{12}, e_{24}, e_{43}^-, e_{35}, e_{56}, e_{61}^-\},$$
$$C_2 = \{e_{13}, e_{32}^-, e_{25}, e_{56}, e_{61}\},$$
$$C_3 = \{e_{24}, e_{43}^-, e_{32}^-\},$$

and the three storage control arcs are:

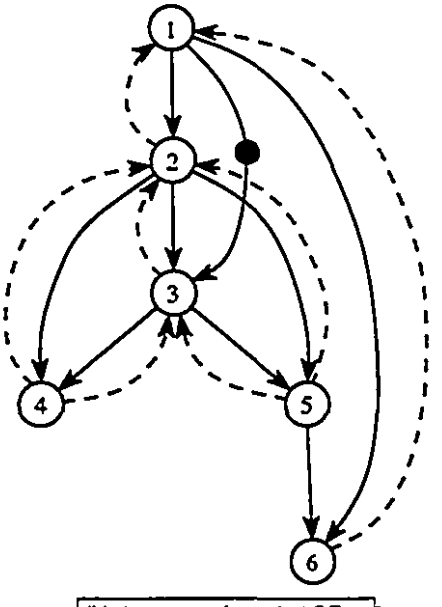
$$e_{61}^-,\ e_{32}^-,\ e_{43}^-.$$

The submatrix containing cycles $C_1$, $C_2$ and $C_3$ and edges $e_{61}^-$, $e_{32}^-$, and $e_{43}^-$ is the following:

| | $e_{61}^-$ | $e_{32}^-$ | $e_{43}^-$ |
|---|---|---|---|
| $C_1$ | 1 | 0 | 1 |
| $C_2$ | 1 | 1 | 0 |
| $C_3$ | 0 | 1 | 1 |




(a) Dataflow graph SG of a loop.




(b) Augmented graph ASG.

Figure 5.9: An example of dataflow graph for which CB is not TUM.

It is easy to see that the determinant of the above 3 by 3 submatrix is 2. So by definition of total unimodularity, the constraint matrix of CB (5.5) is not totally unimodular.

### 5.6.2 CB Problem Has the TDI Property

We first give the definition of total dual integrality. In the following, we will use $A$ to denote the constraint matrix, $b$ the right hand side vector, $c$ the cost coefficient vector of the objective function in the formulation of CB (5.5).

**Definition 5.6.1 ([74])** *Suppose that $A$ is a rational matrix, $b$ is a rational vector. Consider the pair of primal and dual linear programming problems:*

$$\min\{cx \mid Ax \geq b; x \geq 0\} = max\{yb \mid yA \leq c; y \geq 0\}.$$

*Then the linear system $\{Ax \geq b, x \geq 0\}$ is said to have the Totally Dual Integral (TDI) property if and only if the above dual max problem has an integer optimum solution $y$ for each integral vector $c$ with finite maximum.*

The TDI property is weaker than the TUM property but it is still very important for solving integer programming problems due to the following theorem.

**Theorem 5.6.1 (Edmonds and Giles (1977) [31])** *Let $\{Ax \geq b; x \geq 0\}$ be a TDI system. If the right hand side $b$ is an integral vector, then the primal linear programming problem, $\min\{cx \mid Ax \geq b; x \geq 0\}$, has an integer optimal solution for each $c$ such that the solution is finite.*

We want to show that the cycle balancing problem has the TDI property, and therefore it has an integer optimum solution if we replace the right hand side $b_C$ with its integer ceiling $\lceil b_C \rceil$.

In the formulation of CB (5.5), the objective function $c$ has all its entries equal to 1. In order to show its TDI property, we have to consider all integral vectors $c$'s. The more general formulation is shown in (5.14) with the objective function being generalized to an arbitrary integral vector $c$.

---

**CB with generalized objective function (G-CB):**

$$\min \sum_{e=(n_i,n_j)\in A^-} c_e x_e \tag{5.14a}$$

subject to

$$\sum_{e\in C^-} x_e \geq b_C, \quad \forall C \in \mathcal{C} \tag{5.14b}$$

$$x_e \geq 0, \quad \forall e \in A^- \tag{5.14c}$$

---

In order to show that G-CB (5.14) is a TDI system, we have to show that its dual has an integer optimal solution for each $c$ such that the solution is finite, according to Theorem 5.6.1. The dual of G-CB (5.14) is shown below:

---

**Dual of G-CB:**

$$\max \sum_{C\in\mathcal{C}} b_C z_C \tag{5.15a}$$

subject to

$$\sum_{C^-\ni e} z_C \leq c_e, \ \forall e \in A^- \tag{5.15b}$$

$$z_C \geq 0, \quad \forall C \in \mathcal{C} \tag{5.15c}$$

---

Using a similar technique as in Section 5.5 we show that a more general circulation problem GMCCP (5.16) is equivalent to the Dual of GCB (5.15).

---

**Maximum Cost Circulation Problem with General Capacity $c$ (GMCCP):**

$$\max \sum_{e \in A} p_e f_e \tag{5.16a}$$

subject to

$$\sum_{e \in \delta^+(n)} f_e - \sum_{e \in \delta^-(n)} f_e = 0, \ \forall n \in N \tag{5.16b}$$

$$f_e \leq c_e, \qquad \forall e \in A^- \tag{5.16c}$$

$$f_e \geq 0, \qquad \forall e \in A \tag{5.16d}$$

---

**Lemma 5.6.1** *Given an optimal solution of GMCCP (5.16), we can construct an optimal solution of D-GCB (5.15), and vice versa.*

**Proof:** The proof is almost identical to the proof of Theorem 5.5.1 since the only difference between GMCCP and MCCP is the the capacities of the arcs have been generalized from all 1 to an integer vector c. Therefore the detailed proof is omitted. □

It is well known [35, 58] that the circulation problem has an integer optimal solution if all the capacities on arcs are integers, which is indeed the case in GMCCP (5.16).

**Lemma 5.6.2 ([58], page 160)** *In GMCCP (5.16), if the right hand side $c_e$'s are integers and there exists a finite optimal solution, then there exists an integral optimal solution (whether or not the coefficient in the objective function are integers).*

Therefore we can solve the GMCCP (5.16) with integer vector c to obtain an integer optimal solution. The technique used in the proof of Lemma 5.6.1 and Theorem 5.5.1 to transform the optimal solution of circulation problem to an optimal solution of the generalized Dual of G-CB (5.15) will preserve the integrality of the variables because only additions or subtractions are used there. Hence the generalized Dual of G-CB (5.15) has an integral optimal solution for each possible integer vector c. Thus we have actually proved the following theorem:

**Theorem 5.6.2** *The CB (5.5) is a TDI system.*

Let us note that if we replace the rational numbers $b_C$ on the right hand sides of our primal problem by their integral ceilings (the smallest integer greater or equal to $b_C$), we do not change the original problem since the variables are supposed to be integers.

As a consequence of the TDIness shown above, we can round up the right hand sides of CB (5.5) to their ceilings so that the right hand sides are all integers. This step does not change the solution set for the integer programming problem since the left hand sides should also be integers. Thus we obtain the up-rounded version of CB (5.5) shown below:

---

**CB problem with RHS rounded up to ceilings:**

$$min \sum_{e \in A^-} x_e \tag{5.17a}$$

subject to

$$\sum_{e \in C^-} x_e \geq \lceil b_C \rceil, \ \forall C \in \mathcal{C}(ASG) \tag{5.17b}$$

$$x_e \geq 0, \qquad \forall e \in A^- \tag{5.17c}$$

---

We have therefore obtained the following theorem.

**Theorem 5.6.3** *If we round up the right hand sides $b_C$ to $\lceil b_C \rceil$ of the formulation of CB, we obtain the formulation (5.17) which is equivalent to CB (5.5). The linear programming formulation (5.17) has an integer optimal solution.*

Although the formulation in Theorem 5.6.3 has an integer optimal solution by solving the linear programming problem, its number of constraints could be exponential. I have not found a way to resolve the polynomial solvability of the problem yet. However in practice, if the number of cycles is not very big, and if we make an effort to eliminate some of the redundant constraints, the linear programming problem can be solved very fast by the simplex method.

## 5.7 Related Work

In this section, we compare our method with other related work.

### 5.7.1 Loop Storage Optimization for Dataflow Machines

Dataflow software pipelining was originally proposed to exploit fine-grain parallelism in loops on static dataflow computers [38, 40]. Previous work on dataflow software pipelining is targeted to the static dataflow model, hence has the main restriction that the number of concurrent iterations is bounded by what is allowed by one copy of the loop body [38, 40]. Furthermore, even under this restriction, the prior technique for storage allocation, such as the *balancing* technique described in [40], can only handle acyclic dataflow graphs, i.e. loops without loop-carried dependencies.

Loop unraveled under pure dynamic dataflow model can initiate as many iteration as possible, limited only by data dependencies [9]. This is accomplished by the *loop unraveling* scheme, where (in more "modern" implementations such as the Monsoon dataflow machine [66]) each iteration is allocated its own *activation frame* containing all memory spaces required to hold its operands. A main challenge is to minimize the storage used by dynamically *unraveled* concurrent iterations. By far the most successful scheme to control the storage requirement is the *loop bounding* scheme by Culler [20]. One limitation of this scheme is that a fixed number of storage frames (one per iteration) are allocated to a loop, and this amount of storage may not be optimal. Recently, a method of compile time loop scheduling under dynamic loop unraveling has been presented [11].

The method developed in this chapter has addressed the limitations of the loop storage management of both the static and dynamic dataflow models. It provides a basis to allocate statically the minimum amount of storage required for a loop to run at its maximal computation rate.

### 5.7.2 Retiming Synchronous Circuits

The work in this chapter is also related to retiming scheme for hardware circuits. *Retiming* [60, 59] is a circuit transformation method in which registers are added at some points and removed from others in such a way that the functional behavior of the circuit as a whole is preserved.

Although there are some similarities in the problem formulations, our computation model is different from what is used in retiming: ours is asynchronous in nature, while the retiming model is synchronous. Therefore the objectives and formulations are different. One obvious difference is that in retiming, the number of registers on any cycle does not change before and after the retiming [59], while our cycle balancing scheme has no such restriction.

# Chapter 6

# Conclusions

In this chapter, we give a summary of what we have done. We also try to address some future research problems in this direction.

## 6.1 Summary

This dissertation is on the study of the optimal allocation of fast on chip memory, like registers and buffers, for loops on parallel architectures like VLIW and superscalar machines. Previous work in this area separates the register allocation problem from the instruction scheduling problem. The intuition is that separating the scheduling problem and the register allocation problem often results in inefficient code. We propose the idea of combining the scheduling and register allocation together in a single phase. We define a two-step approach to solve the problem for the periodic schedules. The first step is to generate an optimal schedule which uses minimum number of buffers. The second step is to use coloring technique to allow the buffers sharing the same physical registers.

Buffers are allocated to both scalar and array variables appeared in the left-hand-sides of instructions. Buffers can allow the produced values being retained in registers for several iterations so that instructions in later iterations can be scheduled before the previous iterations finish.

We have provided an efficient algorithm for solving the problem. The algorithm is implemented and used to test our scheme for loops selected from typical benchmarks. The testing results include the statistics about the average number of floating point units used, average buffer length, number of registers used.

For a more general class of scheduling techniques, and for different applications, mostly in run-time scheduling applications, we propose a Cycle Balancing technique to optimally allocate buffers so that they can support optimal rate scheduling at run-time. We give polynomial time solution for the linear version of the problem. We also show that the system has the Totally Dual Integral (TDI) property that allows the problem being solved as a linear programming problem if the right-hand-sides of the system are rounded to the integers. Although this does not give an immediate efficient solution, it can help us to solve problems in practical applications efficiently if the number of dependence cycles involved is not too large.

## 6.2 Future Directions

One future direction of continuing this dissertation's work is to combine it with a real parallelizing compiler in which conditionals are dealt with. When this can be done, more testings of our scheme can be applied to more benchmarks and real applications to obtain more accurate statistics. In turn, this knowledge will help to design more efficient and more cost/performance effective VLIW or superscalar architectures in terms of functional units design, register file design and supporting cache design.

Another future direction is to extend our current scheme to larger program structures, like nested loops, procedures or functions, and threads in a multi-threaded architecture. The problem mainly concerns the elimination of unnecessary loads and stores of the values residing in the registers. In [43], instruction scheduling problems for nested loops have addressed. How the register allocation can be incorporated into that scheme is still open.

If the bound of the number of functional units and the bound of the number of registers are both small, the amount of parallelism in the program may exceed what can be supported by the hardware. In this sense our scheme can not be applied directly without modification. Investigation on how to modify our scheme to adapt to low parallel hardware architecture is definitely important. This might involve the introduction of various "slowing down" techniques as spilling codes if registers are not enough, for instance.

# Appendix A

# A Modified OSBA Problem

In this appendix, we give a modified formulation of OSBA problem (4.7) in Section 4.3, by assuming that the destination register of an instruction $s_i$ is reserved at time $t_i + d_i - 1$, i.e. when it is at the output stage of the pipeline.

The formulation and the procedures to deduce it are almost identical to the one in Section 4.3. The solution of the formulation is also very similar as noted after we complete the formulation.

As in Section 4.3, consider an arc (i,j) in the DDG. Now since we commit a buffer to node $i$ time instance $t_i + d_i - 1$ instead of $t_i$, the time span of the result value will become $t_j + Pm_{ij} - (t_i + d_i - 1)$ instead of $t_j + Pm_{ij} - t_i$.

Hence the lower bound on the number $b_i$ of buffers for node $i$ becomes:

$$b_i \geq \frac{t_j + Pm_{ij} - (t_i + d_i - 1)}{P}, \ \forall (i,j) \in \delta^+(i).$$

With these modifications, we formulate our modified optimal schedule and buffer allocation problem into an integer programming problem as follows:

$$\min \sum_{i \in N} b_i$$

subject to

$$b_i \geq \frac{t_j + Pm_{ij} - t_i - d_i + 1}{P}, \ \forall (i,j) \in E \tag{A.1}$$

$$t_j \geq t_i + d_i - Pm_{ij}, \ \forall (i,j) \in E$$

$$t_i, b_i \ \text{integer}, \ \forall i \in N.$$

In the following we rewrite the above formulation (A.1) so that all the variables appear on the left hand sides of the inequalities. We name it as *Modified Optimal Schedule and Buffer Allocation (MOSBA) Problem.*

---

**Modified Optimal Schedule and Buffer Allocation (MOSBA) Problem:**

$$\min \sum_{i \in N} b_i$$

subject to

$$Pb_i + t_i - t_j \geq Pm_{ij} - d_i + 1, \ \forall (i,j) \in E \tag{A.2}$$

$$t_j - t_i \geq d_i - Pm_{ij}, \ \forall (i,j) \in E$$

$$t_i, b_i \ \text{integer}, \ \forall i \in N.$$

---

Notice that the MOSBA formulation (A.2) and the OSBA formulation (4.7) only differ from the right hand sides of the constraints. Therefore the algorithms to solve them can be the same.

# Bibliography


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.

[2] A. Aiken. Compaction-based parallelization. (PhD thesis). Technical Report 88–922, Cornell University, 1988.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. Also in *SIGPLAN Notices*, 23(7), July 1988.

[4] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[5] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, January 1983.

[6] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 233–246, Montréal, Québec, June 17–22, 1984. ACM SIGPLAN. Also in *SIGPLAN Notices*, 19(6), June 1984.

[7] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[8] Arvind and D. E. Culler. Managing resources in a parallel machine. In J. V. Woods, editor, *Fifth Generation Computer Architecture*, pages 103–121. Elsevier Science Publishers, 1986.

[9] Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2):42–49, February 1982.

[10] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[11] Micah Beck, Keshav K. Pingali, and Alex Nicolau. Static scheduling for dynamic dataflow machines. Technical Report TR 90-1076, Department of Computer Science, Cornell University, Ithaca, NY, January 1990.

[12] R. Bellman, A.O. Esogbue, and I. Nabeshima. *Mathematical Aspects of Scheduling and Applications.* Pergamon Press, Oxford, 1982.

[13] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 20–22, 1990. ACM SIGPLAN. Also in *SIGPLAN Notices*, 25(6), June 1990.

[14] P. Camion. Characterizations of totally unimodular matrices. *Proc. Amer. Math, Soc.*, 16:1068–1073, 1965.

[15] G. J. Chaitin. Register allocation & spilling via graph coloring. *ACM SIGPLAN Symp. on Compiler Construction*, pages 98–105, 1982.

[16] G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47–57, January 1981.

[17] V. Chvatal. *Linear Porgramming.* W.H. Freeman and Company., 1983.

[18] E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.

[19] R. W. Conway. *Theory of Scedaling.* Addison-Wesley, Reading, Mass., 1967.

[20] D. E. Culler. Managing parallelism and resources in scientific dataflow programs. Ph.D thesis, Technical Report TR-446, MIT Laboratory for Computer Science, 1989.

[21] Ron Cytron. Computation of output dependences as a data flow problem. Technical report, IBM, 1988.

[22] J. B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springler-Verlag, 1974.

[23] J. B. Dennis. First version of a data flow procedure language. Technical Memo MIT/LCS/TM-61, MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1975.

[24] J. B. Dennis. Data flow for supercomputers. In *Proceedings of the 1984 Comp-Con*, March 1984.

[25] J. B. Dennis. Evolution of the static dataflow architecture. In *Advanced Topics in Dataflow Computing.* Prentice-Hall, 1991.

[26] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of Supercomputing '88*, pages 368–373, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[27] E. Duesterwald, R. Gupta, and M.L. Soffa. Register pipelining: An integrated approacn to register allocation for scalar and subscripted variables. Technical report, Department of Computer Science, University of Pittsburgh, 1991.

[28] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelization loops with unpredictable branches on a VLIW architecture. Technical report, IBM, 1990.

[29] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, December 1987.

[30] K. Ebcioğlu and A. Nicolau. A global resource-constrained parallelization technique. In *Proceedings of the ACM SIGARCH International Conference on Supercomputing*, June 1989.

[31] J. Edmonds and R. Giles. A min-max relation for submodular functions on graphs. In *Studies in Integer Porgramming, Annals of Discrete Mathematics*, volume 1. P.L. Hammer, et al., eds, 1977.

[32] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[33] Christine Eisenbeis, William Jalby, Daniel Windheiser, and Francois Bodin. A strategy for array management in local memory. In *Third Workshop on Programming Languages and Compilers for Parallel Computing*. University of California, Irvine, 1990. To be published by Pitman/MIT Press.

[34] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. *Proceedings of the ACM Symposium on Compiler Construction*, pages 37–47, June 1984.

[35] L. R. Ford and D. R. Fulkerson. *Flow in Networks*. Princeton University Press, Princeton, NJ, 1962.

[36] D.R. Fulkerson. An out-of-kilter method for minimal cost flow problems. *J. SIAM*, 9:18–27, 1961.

[37] G. R. Gao. A pipelined code mapping scheme for solving tridiagonal linear system equations. In *Proceedings of IFIP Highly Parallel Computer Conference*, Nice, France, March 1986.

[38] G. R. Gao. A pipelined code mapping scheme for static dataflow computers. Technical Report TR-371, MIT Laboratory for Computer Science, 1986.

[39] G. R. Gao. Aspects of balancing techniques for pipelined data flow code generation. *Journal of Parallel and Distributed Computing*, 6:39-61, 1989.

[40] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, Massachusetts, December 1990.

[41] G. R. Gao, H. H. J. Hum, and Y. B. Wong. An efficient scheme for fine-grain software pipelining. In *Proceedings of the CONPAR '90-VAPP IV Conference*, pages 709–720, Zurich, Switzerland, September 1990.

[42] G.R. Gao. A flexible architecture model for hybrid dataflow and control-flow evaluation. In *Proceedings of the 16th International Workshop: Dataflow — A Status Report*, Israel, May 1989. in conjunction with the ACM Annual Symposium on Computer Architecture. To be published by Prentice-Hall.

[43] G.R. Gao, Q. Ning, and V. Van Dongen. Software pipelining for nested loops. Technical Report ACAPS Technical Memo 53, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1993.

[44] G.R. Gao, Y.B. Wong, and Q. Ning. A petri net model for loop scheduling. In *the Proceedings of ACM SIGPLAN'91, Toronto, Canada.* June 1991.

[45] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[46] M.R. Garey, D.S. Johnson, Miller. G.L., and C.H. Papadimitriou. Unpublished result. in Computers and Intractability: A guide to the Theory of NP-completeness, New York, 1979.

[47] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. Also in *SIGPLAN Notices*, 21(7), July 1986.

[48] M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase-2 user's manual. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champagn, July 1991.

[49] L. Hendren, G.R. Gao, E. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *Lecture Notes in Computer Science 641*, pages 176–191, October 1992.

[50] J. Hennessy and T. Gross. Postpass code optimization of pipelined constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[51] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[52] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[53] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, Boston, Massachusetts, April 3–6, 1989. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. Also in *Computer Architecture News*, 17(2), April 1989; *Operating Systems Review*, 23, April 1989; *SIGPLAN Notices*, 24, May 1989.

[54] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[55] L. G. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Doklady*, 20:191–194, 1979.

[56] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. Also in *SIGPLAN Notices*, 23(7), July 1988.

[57] Monica S. Lam. Instruction scheduling for superscalar architectures. *Annual Review of Computer Science*, 4:173–201, 1990.

[58] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing, Ft Worth, TX, 1976.

[59] C. E. Leiserson and J. B. Saxe. Optimizing synchronous circuitry by retiming (preliminary version). *Algorithmica*, 6(1):5–35, 1991.

[60] C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. *J. VLSI and Computer Systems*, 1(1):41–68, 1983.

[61] T. Nakatani and K. Ebcioglu. Using a lookahaed window in a compaction-based parallelizing compiler. In *Proceedings of the 23rd Annual Workshop on Microprogramming and Microarchitectures*, pages 57–68, 1990.

[62] A. Nicolau, R. Potasman, and H. Wang. Register allocation, renaming and their impact on fine-grained parallelism. In U. Banerjee et al., editor, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 589, pages 359–373, Santa Clara, California, 1992. Springer-Verlag.

[63] Q. Ning and G. Gao. Minimizing loop storage allocation for an argument-fetching dataflow architecture model. In D. Etiemble and J.-C. Syre, editors, *Proceedings of PARLE '92 – Parallel Architectures and Languages Europe*, pages 585–600, Paris, France, June 15–18, 1992. Springer-Verlag, Lecture Notes in Computer Science 605.

[64] Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. In *Proceedings of 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*, pages 29–42, Charleston, South Carolina, January 10–13 1993.

[65] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, MIT, 1988.

[66] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 18(2), June 1990.

[67] Pierre Peladeau. On the length of the cyclic frustrum in a sdsp-pn. Technical Report ACAPS Technical Note 31, McGill University, Montreal, 1991.

[68] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing, Penn State, St. Charles, IL*, August 1989.

[69] B. R. Rau, 1993. personal communication.

[70] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[71] B. R. Rau, D. Yen, W. Yen, and R. A. Towle. The Cydro 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[72] B.R. Rau, M. Lee, P.P. Tirumalai, and M.S. Schlansker. Register allocation for modulo scheduled loops: Strategies, algorithms and heuristics. In *Proceedings*

*of SIGPLAN '92 Conf. on Programming Languag. Design and Implementation, San Francisco, CA*, 1992.

[73] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, 1968.

[74] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1986.

[75] R. Sethi. Complete register allocation problems. *SIAM J. Comput.*, 4(3):226-248, 1975.

[76] R. F. Touzeau. A FORTRAN compiler for the FPS-164 scientific computer. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, Montréal, Québec, June 17–22, 1984. ACM SIGPLAN. Also in *SIGPLAN Notices*, 19(6), June 1984.

[77] V. Van Dongen, G. Gao, and Q. Ning. A polynomial time method for optimal software pipelining. In *Proceedings of CONPAR '92*, Lecture Notes in Computer Science 634, Paris, France, September 1992.

[78] H.S. Warren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM J. Res. Develop.*, 34(1), January 1990.

[79] Michael Wolfe and Uptal Banerjee. Data dependence and its application to parallel processing. *International Journal on Parallel Processing*, 16(2):137–178, April 1987.

[80] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.