AN ABSTRACT OF THE THESIS OF

Lap K. Mui for the degree of Master of Science in Electrical and Computer Engineering presented on March 5, 1992.

Title: Scheduling System of Affine Recurrence Equations by Means of Piecewise Affine Timing Functions

Abstract approved: Redacted for Privacy

Sayfe Kiaei

Many systematic methods exist for mapping algorithms to processor arrays. The algorithm is usually specified as a set of recurrence equations, and the processor arrays are synthesized by finding timing and allocation functions which transform index points in the recurrences into points in a space-time domain.

The problem of scheduling (i.e. finding the timing function) of recurrence equations has been studied by a number of researchers. Of particular interest here are *Systems of Affine Recurrence Equations* (*SAREs*). The existing methods are limited to affine (or linear) schedules over the entire domain of computation. For some algorithms, there are points in the computation domain where the dependencies point in opposite directions, and an affine schedule does not exist, although a valid *Piecewise Affine Schedule* (*PAS*) can exist. The objective of this thesis is to examine these schedules and obtain a

systematic method for deriving such schedules for *SAREs*. *PAS* can be found by first partitioning the computation domain and then obtaining a new *SARE* by renaming the variables. By partitioning the computation domain, we can obtain additional parallelism from the dependency graph, and find faster schedules over subspaces of the domain. In this paper, we describe a procedure for partitioning the domain and to generate a new *SARE* by renaming the variables. Some heuristics are introduced for partitioning the domain based on the properties of dependence vectors. After the partitioning and renaming, an existing method (due to Mauras et al.) is applied to find the schedules. Examples of *Toeplitz System* and *Algebraic Path Problem* are used to illustrate the results.

Scheduling System of Affine Recurrence Equations

by Means of

Piecewise Affine Timing Functions

by

Lap K. Mui

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed March 5, 1992
Commencement June 1992

APPROVED:

Redacted for Privacy

Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented March 5, 1992

Typed by Lap K. Mui

Formatted by Lap K. Mui

ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. Sayfe Kiaei, who has given me valuable advice and encouragement.

I am indebted to Dr. Sanjay Rajopadhye, who initiated this project. Without our weekly discussions and his guidance, this work would not be completed.

I would also like to thank my graduate committee member, Dr. Bella Bose, Dr. Jack Kenney, and Dr. Ernest Wolff for their time and advice.

Most of all, I would like to thank my parents and my family for their love and support.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

SCHEDULING SYSTEM OF AFFINE RECURRENCE EQUATIONS BY MEANS OF PIECEWISE AFFINE TIMING FUNCTIONS

Chapter 1

INTRODUCTION

1.1 Overview of the Problem

The design and applications of special purpose parallel architectures has been studied since the late 1950s. Systematic procedures have been developed for mapping a given algorithm onto processor arrays. Typically the initial algorithm can be specified as a set of *Uniform recurrence Equations (URE),* or *Regular Iterative Algorithms (RIA)* (Based on the work by Quinton [13], Rao [18], and Jagadish [3].) These methods are restricted to the class of recurrence called *Affine Recurrence Equations (ARE)* was introduced by Rajopadhye [16] and Delosme and Ipsen [2]. The synthesis of the processor arrays consists of mapping the index-space of the algorithm expressed in terms of recurrence to a space-time domain. The recurrence is usually represented graphically as a *Dependency Graph (DG)* which shows the dependencies between points (nodes) in the index-space. For most scheduling algorithms one needs to work with a *Reduced Dependency Graph (RDG).* In this case, every node in the *DG* is assigned to a processor at a particular time. In most cases the allocation and scheduling functions are assumed to be linear. There is a well developed theory for determining affine schedules and affine allocation functions

to perform this assignment [16].

The problem of finding linear schedules for *SARE* has been studied by a number of researchers. Delosme and Ipsen [2] give a method for obtaining schedules for *SARE*. In their method they determine a certain class of cycles (those that can be iterated arbitrarily many times) in the *RDG*, and then partitioning the computation domain for each variable based on these cycles. A problem in this method is that it may not terminate and a schedule is not guaranteed to be found. Saouter and Quinton [19] have shown that the scheduling problem is undecidable in general. Rajopadhye [16], and Quinton and Van Dongen [14] show that by examining properties of *Convex Polyhedral Domains* (*CPD*), the affine schedules for a single *ARE* can be determined. Yaacoby and Cappello [21] give necessary and sufficient conditions for finding an affine schedule for a restricted class of *SARE* (A *SARE* is a set of *AREs*). In all these approaches the linear part of the schedule is the same for all variables. More recently Mauras et al. [10] described an algorithm for computing affine schedules that are "variable-dependent" in the sense that the slope of the timing hyperplanes may be different for different variables. However, all these methods require a single affine schedule over the entire computation domain of a given variable. For some *SARE*, it is not possible to find an affine schedule, but a valid *Piecewise Affine Schedule* (*PAS*) may exist.

[EXAMPLE 1] Consider the *SARE*:

$$f(i, j) = h(i, i) \text{ for } i = j \qquad (1.1)$$

$$f(i, j) = f(i, j - 1) \text{ for } i < j \qquad (1.2)$$

$$f(i, j) = f(i, j + 1) \text{ for } i > j \tag{1.3}$$

The *SARE* does not have an affine schedule for the variable *f*, but for the following *SARE* which computes the same function, there is an affine schedule for each variable:

$$f1(i, j) = h(i, i) \text{ for } i = j \tag{1.4}$$

$$f2(i, j) = f1(i, j - 1) \text{ for } i = j - 1 \tag{1.5}$$

$$f2(i, j) = f2(i, j - 1) \text{ for } i < j - 1 \tag{1.6}$$

$$f1(i, j) = f1(i, j + 1) \text{ for } i > j \tag{1.7}$$

The schedule for variable *f1* can be:

$$t_{f1}(i, j) = i - j$$

and the schedule for variable *f2* can be:

$$t_{f2}(i, j) = j - i$$

The above procedure where two separate timing functions are obtained is known as piecewise affine scheduling.

[EXAMPLE 2] Consider the *SARE*:

$$f(i, j) = h(i, i) \text{ for } i = j \tag{1.8}$$

$$f(i, j) = f(i, j - 1) \text{ for } i < j \tag{1.9}$$

$$f(i, j) = f(i - 1, j) \text{ for } i > j \tag{1.10}$$

A linear schedule can be found for the variable *f* over the domains of the three equations:

$$t_f(i, j) = i + j; \text{ for } 0 \leq i, j \leq N$$

A piecewise affine schedule can also be found for the variable *f*.

$$t_f(i, j) = j; \text{ for } i < j$$

$$i; \text{ for } i \geq j$$

In this case, the computation time for the first schedule is $2N$, and for the second schedule it is N.

As illustrated by the above examples, there are two benefits to be gained by *PAS*:

(1) Finding schedules when affine schedules do not exist.

(2) Improving affine schedules for certain algorithms.

PAS allows schedules to have different slopes (linear parts) for the subspaces of a domain. The procedures for finding piecewise affine schedules involves partitioning of computation domain of equation and renaming the variables. By partitioning a domain we can obtain more parallelism from a recurrence, and better schedules can be found.

The goal of this thesis is to improve upon the existing synthesis methods for scheduling algorithms by applying piecewise affine (linear) variable dependent schedules. We first use heuristics based on the properties of the dependency vectors to find the partitioning hyperplanes. Then a renaming procedure similar to the method by Delosme [2] is used to generate a new *SARE*. Finally, an existing method by Mauras et al. [10] is applied to find the schedules for different variables.

In chapter 2 we give a formal definition of *SARE* and review the general procedure of automatic mapping of *SARE* into processor arrays. We also describe variable dependent affine timing functions and the necessary and sufficient conditions for the existence of affine timing functions as given by Mauras et al. [10], and a definition of normal form for a *SARE* [15]. Chapter 3 discuss a procedure for obtaining a new *SARE* by renaming

the variables. Next we propose a method for partitioning the domains of computation for each variable based on heuristics. Variable dependent affine timing function is used for computing the affine schedule for each variable in the new *SARE*. In chapter 4, several examples to illustrate the new scheduling method are given.

Chapter 2

SYSTEMS OF AFFINE RECURRENCE EQUATIONS

2.1 Systems of Affine Recurrence Equations

The synthesis procedure is based on expressing the algorithm in terms of a System of Affine Recurrence Equations defined using the notation of Quinton and Van Dongen [15].

[Definition 1] A *System of Affine Recurrence Equations* (*SARE*) is a set of recurrence equations of the form:

$$U(z) = f[V(I(z)),...]; \text{ for } z \in D$$

where

(1) z is a point of Z^n, where Z denotes the set of integers.

(2) U and V are *variable Names* belonging to a finite set S. Each variable is indexed with an integral index z, whose dimension is constant for a given variable. The variable $U(z)$ is the *result* and $V(I(z))$ is the *argument.*

(3) D is the computation domain belonging to a convex polyhedron of Z^n. We assume that D is bounded and is defined by a finite set of linear inequalities. Such domain can also be described by a set of unique vertices $\boldsymbol{M}$ and a set of (extremal) rays $\boldsymbol{K}$. If the domain D has m vertices and k extremal rays, then any point in D can be uniquely expressed as the sum of a *convex combination* of the vertices and a *positive combination* of the rays

as follows:

$$z = \Sigma^{m}_{i=1} a_i \sigma_i + \Sigma^{k}_{i=1} c_i \rho_i, \textit{ where } a_i, c_i > 0 \textit{ and } \Sigma^{m}_{i=1} a_i = 1$$

where $\sigma_i \in \boldsymbol{M}$, $\rho_i \in \boldsymbol{K}$, $a_i, c_i \in R$; R denotes the set of nonnegative real numbers.

(4) $I(z)$ is an affine mapping from Z^n to Z^l called *index (or dependence) mapping*. $I(z)$ has the form

$$I(z) = Az + b$$

where the constants A and b are integral matrices: A with dimension l x n and b is a l x 1 vector.

(5) f is a single-valued function that depends *strictly* on its arguments; we assume that the function f has constant time complexity.

(6) The '...' means that there can be other arguments of the same form as $V(I(z))$.

(7) The domains of two equations having the same variable as result (in the left-hand side of the equation) are disjoint. This hypothesis ensures that a variable is not defined twice.

[Definition 2] A recurrence equation is called a *Uniform Recurrence Equation* (*URE*) if the Index mapping $I(z)$ has the form:

$$I(z) = z + b$$

where $b \in Z^l$. The *URE* is a subclass of *ARE* with the linear part A equal to the identity matrix.

2.2 Dependency Graph

A *DG* provides a graphical representation of a *SARE*. It shows the dependence of the computations that occur in an algorithm.

[Definition 3] A *Dependency Graph* (*DG*) is a *Directed Acyclic Graph* (*DAG*) where the graph $G = (N, A)$, where N is the set of nodes and A is the set of directed arcs. Each node represents operations performed by an algorithm on some data, and the arcs are used to represent data dependencies.

In a *DG*, nodes with no incoming arcs (in-degree zero) are input nodes, and nodes with no outgoing arcs (out-degree zero) are output nodes. If a node p requires data from the node q, then the dependence is shown as p --> q. In this thesis, the direction of the arcs are the same as the direction of data flow. An algorithm is computable if and only if its *DG* contains no loops or cycles.

A *DG* is only a partial representation of an algorithm. It specifies the dependencies among variables. The operations inside each node are usually ignored in the *DG* since they only affect the implementation of the processing element. The *DG* also imposes certain precedence constraints on the order that these operations can be performed.

The *DG* described above is a complete representation of the precedence constraints of *SARE*. A *Reduced Dependency Graph* (*RDG*) is a simplified version of a *DG* which captures the dependence with regard to variable names but does not take into account index values. In the paper by Delosme and Ipsen [2], they proposed a method for

partitioning and renaming of recurrence equations which is based on cycles on a *RDG*.

[Definition 4] A *Reduced Dependency Graph* (*RDG*) is a directed graph with a node for each variable array in the *system of recurrence equations* (*SRE*). An arc from a node U to a node V represents a dependence mapping I_{vu} in the *SRE*. That is:

$$I_{vu}$$

$$U \text{ --------> } V$$

where I_{vu} is a mapping from domain D_{vu} to range R_{vu}. A composition of dependence mapping of the following form may lead to a cycle:

$$I_{vu} \circ I_{uw} \circ I_{wv}$$

2.3 Procedure for Mapping SARE onto Processor Arrays

Once an algorithm is expressed as a *SARE*, the objective is to map its *DG* onto a set of processors. If the target architecture is locally connected (eg. a systolic array), then a well-known technique called pipelining [16] is used to convert the *SARE* into *SURE*. The next step is to find an appropriate timing and allocation functions for the *SARE* and map each index point of the recurrence domain into the space-time domain. In this case, we will restrict our attention to linear scheduling and allocation function.

[Definition 5] An *Affine Timing Function* (*ATF*) is a scalar function of the form:

$$t(z) = \lambda^T_t z + \alpha_t$$

where λ^T_t is a constant schedule vector and α_t is a scalar constant. The timing or scheduling function t is a mapping of all points in the computation domain D to a set of positive integers. It is necessary for t to satisfy the rule of causality which states that if node p requires data from node q, then $t(p) > t(q)$.

[Definition 6] An *Affine Allocation Function* (*AAF*) is a function of the form:

$$a(z) = \lambda^T_a z + \alpha_a$$

where λ^T_a is an $(n - 1)$ x n matrix and α_a is an $(n - 1)$ vector. The allocation function maps each computation node in the *DG* to a particular processor. The allocation function can also be expressed by the projection vector u. It is necessary for the projection vector u to satisfy the conflict-free condition, i.e. $\lambda^T u \neq 0$ [16].

2.4 Affine Variable Dependent Timing Functions

Affine variable dependent timing function allows each variable to have schedule with different linear part.

[Definition 7] An *Affine Variable Dependent Timing Function* (*AFT*) is a timing function for each variable of the form:

$$t_U(z) = \lambda^1_U z_1 + \ldots + \lambda^n_U z_n + \alpha_U$$

where $\lambda^1_U, \ldots, \lambda^n_U$ are integers. In the following, we let $\lambda_U z = [\lambda^1_U \ldots + \lambda^n_U]$.

This corresponds to having different timing functions with different variables. Mauras et

al. [10] show the following necessary and sufficient conditions. t_U define a timing function iff:

(1) for each edge *(U, V, D, I)* of the *RDG*, we have:

- (i) for each vertex σ of D, $\lambda_U \cdot \sigma - \lambda_V \cdot (I(\sigma)) + \alpha_U - \alpha_V > 0$
- (ii) for each ray ρ of D, $\lambda_U \cdot \rho - \lambda_V \cdot A \cdot \rho \geq 0$

(2) for all variable U,

- (iii) for all vertex σ of the domain D_U of U, $\lambda_U \cdot \sigma + \alpha_U \geq 0$
- (iv) for all ray ρ of D_U, $\lambda_U \cdot \rho \geq 0$

2.5 Normal Form of a System of Equations

A normal form [15] provides a more convenient system for the analysis of *SARE* by separating the input, output and computation equations. An equation in the normal form is fully indexed which facilitates the evaluation of dependence vector.

[Definition 8] A system of linear recurrence equations is said to be in *normal form* if:

1. all the variables are either *input*, *output*, or *intermediate variables*.
2. all equations are either *input*, *output*, or *computation equations*.
3. all the computation equations are *fully indexed* and have the same index dimension.

The definition and transformation of a system of equation into normal form is shown by Quinton and Van Dongen [15].

First, we will use an example to discuss the terminology used.

[Example 3] The general equation for 1-dimensional convolution is:

$$Y(n) = \Sigma^{N-1}_{j=1} X(j).W(n - j) \qquad (2.1)$$

where $0 \le n \le 2N - 2$; $0 \le j \le N - 1$. In form of *SARE*, the equations are:

$$y(i, j) = 0; for\ j = 0 \qquad (2.2)$$

$$w(i, j) = W(i); for\ j = 0 \qquad (2.3)$$

$$x(i, j) = X(j); for\ i = 0 \qquad (2.4)$$

$$y(i, j) = y(i, j - 1) + w(i, j) * x(i, j); for\ \{0 \le i \le N - 1, 0 \le j \le i\}\ and\ \{N \le i \le 2N - 2, j - N + 1 \le j \le N - 1\} \qquad (2.5)$$

$$w(i, j) = w(i - 1, j - 1); for\ \{0 \le i \le N - 1, 0 \le j \le i\}\ and\ \{N \le i \le 2N - 2, j - N + 1 \le j \le N - 1\} \qquad (2.6)$$

$$x(i, j) = x(i - 1, j); for\ \{0 \le i \le N - 1, 0 \le j \le i\}\ and\ \{N \le i \le 2N - 2, j - N + 1 \le j \le N - 1\} \qquad (2.7)$$

$$Y(i) = y(i, j); for\ \{0 \le i \le N - 1, j = i\}\ and\ \{N \le i \le 2N - 2, j = N\} \qquad (2.8)$$

A *SARE* represents a finite set of *equation instances* which associated with a particular point z in D. For example: $x(0, 1) = X(1)$ is an equation instance of *(2.4)*. We called *variable instance* of the system any term $X(z)$ that appears in an equation instance. For example: $x(0,1)$ is a variable instance of the variable x. A variable instance that appears only in the left-hand side of an equation is called an *output*. Examples for outputs are the variable instances $Y(0)$, $Y(N)$. Similarly, a variable instance that appears only in

the right-hand side of an equation is called an *input*. Examples for inputs are the variable instances *W(0)*, *X(0)*, and *y(0,0)*. Variable instances that are neither outputs nor inputs are called *intermediate data*. Examples for intermediate data are the variable instances *w(1,1)*, *x(2,2)*. Variables whose instances are all input (output) are called *input* (*output*) *variables*. Variables whose instances are not all input or output are called *intermediate variables*. In the above example W and X are input variables, Y is an output variable, and y, x, and w are intermediate variables.

The original definition [15] requires all instances of intermediate variable to be intermediate data, but the definition of intermediate variable we used is less restrictive. The reason is that the original definition requires a transformation called *variable normalization*, which cannot be done automatically.

[Example 4] Let the *SARE* be:

$$A(i) = A\,(i - 1)\ for\ \ 0 < i \leq N \qquad (2.9)$$

A(i) for $i = 0$ are inputs, and *A(i)* with $i = N$ are outputs, and the remaining values of *A(i)* are intermediate data. To satisfy the original definition, all variables have to be normalized by renaming variable instances which are inputs or outputs respectively. So the equation will be rewritten as:

$$A(i) = A(i - 1)\ for\ 2 \leq i \leq N - 1 \qquad (2.10)$$

$$A(i) = A'(0)\ for\ i = 1 \qquad (2.11)$$

$$A''(i) = A(i - 1)\ for\ i = N \qquad (2.12)$$

All input instances of A are renamed A', and all output instances of A are renamed A''.

The above transformation cannot be automated if the domain is not bounded and we have to check for all equation instances.

[Definition 9] The argument of the equation is said to be *fully indexed*, if its index dimension is the same as the index dimension of the result of the equation. An equation is fully indexed if all its arguments are fully indexed. Examples: Equations *(2.5-2.7)* are fully indexed, their indexes are all 2-dimension, but equations *(2.2-2.3)* and *(2.8)* are not fully indexed.

An equation is an input equation if f is the identity function, and the only argument of the equation is an input variable. Similarly, an equation is an output equation if f is the identity function and U is an output variable. Finally, an equation is a computation equation if its results and arguments are all intermediate variables. Examples: *(2.2-2.4)* are input equations, *(2.8)* is an output equation, and *(2.5-2.7)* are computation equations.

For this thesis we assume the *SARE* has already been normalized.

Chapter 3

PIECEWISE AFFINE TIMING FUNCTIONS

3.1 Overall Procedure

The procedure for deriving the *Piecewise Linear Variable Dependent Timing Function* consists of the following steps:

Step 1 Transform the *SARE* into normal form - This can be done by the method of Quinton and Van Dongen [15]. This method transforms a *SARE* into a new equivalent system which is more convenient for the analysis. There are two purposes for this transformation:

(1) To separate the input, the outputs, and the computation equations. Only the dependencies between variables in *computation equations* have to be considered for decomposing the domains.

(2) For the rest of the analysis, it is necessary that all indices in an equations have the same dimension.

Step 2 Check if an affine schedule can be found for each variable - This can be done by the method of Mauras et al. [10].

Step 3 If it is not possible to find an affine schedule, then partition domains of equations by choosing appropriate hyperplanes (or linear subspaces).

Step 4 Obtain an equivalent system of equations by renaming the variables according to

the result of partitioning. New equations are introduced if needed. This includes updating the result (left-hand side) and arguments (right-hand side) for each equation.

Step 5 Find schedule for each new variable using the method of Mauras et. al. [10].

The main component of this procedure is the method for partitioning of domains of computation and renaming of variables. In section 3.2, we will present an algorithm for obtaining a new *SARE* by renaming variables. Given a hyperplane (or a linear subspace) and a *SARE*, the procedure transforms a *SARE* into an equivalent *SARE* which is guaranteed to terminate. In section 3.3, we will describe the heuristics which allow us to choose the linear subspaces for partitioning.

3.2 Obtaining a New SARE by Renaming Variables

The renaming procedure we use is based on the one proposed by Delosme and Ipsen [2]. We modify some steps so that the procedure is guaranteed to terminate. First we explain the reason for the modification, then the detail of the procedure is presented.

Delosme and Ipsen solved the problem of *partitioning of the domains* by determining a certain class of cycles in the *Reduced Dependency Graph* (*RDG*), and then partitioning the domains of computation for each variable based on these cycles. According to their method, the partitioning and renaming is based on two criteria:

(1) Partition the domain of a variable U into subsets induced by the *intersections* and *differences* of the domains of all the cycles of U. It is called *composition-induced partition.* Each subset must be explicitly renamed.

(2) Partition and rename the domain of U according to the *non-iterative cycles* (A cycle is said to be *iterative* if it can be *iterated arbitrarily many times*).

One of the problems with this method is that the *differences* of the domains are not necessarily convex polyhedra. Another problem is that they do not prove the termination of their algorithm for finding iterative cycles.

[Example 5] Let a *SARE* be:

$$z \in D_U \text{ ---> } U(z) = f[U(I_{UU}(z), V(I_{VU}(z)),...] \qquad (3.1)$$

$$z \in D_V \text{ ---> } V(z) = f[V(I_{VV}(z), U(I_{UV}(z)),...] \qquad (3.2)$$

The *RDG* of this *SARE* is shown on Figure 1. For variable U there are two cycles C_{UU} and C_{VU}. For variable V there are also two cycles C_{VV} and C_{UV}.

For cycle C_{UU}, its domain is D_U and its cyclic dependence mapping $I_{C\text{-}UU}(z) = I_{UU}(z)$.

For cycle C_{VU}, its domain is D_U and its cyclic dependence mapping $I_{C\text{-}VU}(z) = I_{UV} \circ I_{VU}(z)$.

For cycle C_{VV}, its domain is D_V and its cyclic dependence mapping $I_{C\text{-}VV}(z) = I_{VV}(z)$.

For cycle C_{UV}, its domain is D_V and its cyclic dependence mapping $I_{C\text{-}UV}(z) = I_{VU} \circ I_{UV}(z)$.

Let use the above example to show how the method of Delosme and Ipsen works. For variable U, if the domains for the cycles C_{UU} and C_{VU} are different (they are the same in this example), then partitioning for the domain of U may be needed which based on the composition-induced partition. We will find the intersection and difference of the domains for C_{UU} and C_{VU}, and the resulting subsets will be given new variable names.

For the cycle C_{UU}, if $I_{C\text{-}UU}(D_U) \cap D_U \neq \phi$ (ϕ is the empty set), then we said the cycle C_{UU} can be iterated once. If the iteration can be repeated arbitrarily many times, then the

cycle C_{UU} is said to be iterative. Let $D_{C\text{-}UU}{}^{1}$ be the domain resulted from iterating the cycle once, and $D_{C\text{-}UU}{}^{k}$ be the domain resulted from iterating the cycle k times. If $D_{C\text{-}UU}{}^{k} = \phi$ for a value of k, then the subsets resulted from $D_{C\text{-}UU}{}^{l} - D_{C\text{-}UU}{}^{l\text{-}1}$ (where $2 \leq l \leq k$) have to be given new variable names. More details can be found in [2].

In our method we assume the domain D_U is partitioned into two domains D_{U1} and D_{U2} according to the dependency mapping I_{UU}. Two new variables $U1$ and $U2$, and their domains D_{U1} and D_{U2} will be resulted from equation *(3.1)*:

$$z \in D_{U1} \text{ ---> } U1(z) = f[....] \tag{3.3}$$

$$z \in D_{U2} \text{ ---> } U2(z) = f[....] \tag{3.4}$$

The variable U is renamed to $U1$ and $U2$. Any occurrence of the variable U on the right-hand side of any equation must be replaced by one of the new variables $U1$ or $U2$. For each index point z such that $I_{UU}(z)$ lies in the domain D_{U1}, we replace the U term by $U1$; for each index point z such that $I_{UU}(z)$ lies in the domain D_{U2}, we replace the U term by $U2$. *(3.3)* gives us two new equations ($I^{-1}{}_{UU}$ is the inverse mapping of the index mapping I_{UU}, and it is assumed to exist):

$$z \in I^{-1}{}_{UU}(I_{UU}(D_{U1}) \cap D_{U1}) \text{ ---> } U1(z) = f[U1(I_{UU}(z),...] \tag{3.5}$$

$$z \in I^{-1}{}_{UU}(I_{UU}(D_{U1}) \cap D_{U2}) \text{ ---> } U1(z) = f[U2(I_{UU}(z),...] \tag{3.6}$$

We can see that the domain D_{U1} is partitioned into two new domains. If we give new variable names $U3$ and $U4$ to these two new domains. Then *(3.5)* has to be changed to *(3.7)*, and *(3.6)* has to be changed to *(3.8)*:

$$z \in I^{-1}{}_{UU}(I_{UU}(D_{U1}) \cap D_{U1}) \text{ ---> } U3(z) = f[U1(I_{UU}(z),...] \tag{3.7}$$

$$z \in I^{-1}{}_{UU}(I_{UU}(D_{U1}) \cap D_{U2}) \text{ ---> } U4(z) = f[U2(I_{UU}(z),...] \tag{3.8}$$

However, the variable *U1* on the right-hand side does not exist anymore, and we have to replace it by *U3* and *U4*. This renaming process has to go on and on, and it is not guaranteed to terminate. The same process has to be done on all other variables.

The main feature of our renaming procedure is that we **DO NOT** give a new variable name every time a domain is partitioned. Given a hyperplane (or a linear subspace), our procedure only rename a variable once such as from *U* to *U1* and *U2*. As a result our renaming procedure will not give us *(3.7)* and *(3.8)*.

In some cases the inverse of an affine mapping may not exist where the linear part may be singular. Howerver, since its pre-image is a linear space, its intersection with the original domain is a convex polyhedral domain. Later, we will use this fact for finding the new domain of the equation even if its inverse mapping does not exist.

Equations with new domains will be introduced by two types of partitioning: one is based on the heuristics we will present in section 3.3; another is due to the renaming process. The main difference is that the first type of partitioning introduces new variable names, while the second method does not.

We can summarize the renaming as two steps.

(1) Determine the new variable names to be used and their domains.

From the partitioning, we determine the subdomains of a variable *U*. Each subdomain will be given a new name. For each equation in the original *SARE* with *U* as the result, we find the *intersection* of each subdomain and the original domain of the equation. If it is not empty, then a new variable and its domain have to be introduced.

(2) Replace the old variable names with the new variable names in each equation.

Introduce new equations and determine their domains if needed.

We have to rename the result and its arguments of each equation. The domain of computation for each equation is changed accordingly.

Here are the details for the second step that generates a new *SARE*:

(i) For each dependency find the image of the dependence mapping. Consider the equation *(3.1)*: If a hyperplane partitions the domain D_U into two subdomains D_{U1} and D_{U2}, then two new equations with the new variable names *U1* and *U2* as results have to be introduced. Note that the image of D_{U1} under the dependence mapping I_{UU} is $I_{UU}(D_{U1})$.

(ii) Find the intersection of the image $I_{UU}(D_{U1})$ with each subdomain D_{U1} and D_{U2} . If the intersection is not empty, then all variable instances in this portion of the image have to be renamed according to the name of the subdomain. That is, if $D_{U1} \cap I_{UU}(D_{U1})$ is not empty, then an equation has to be introduced with the argument U renamed as $U1$.

(iii) Change the domain of the equation. Let the domain D_{U1} be represented by the inequalities $A_{U1}z \geq b_{U1}$, and the domain $I_{UU}(D_{U1})$ be represented by the inequalities $A_{UU\text{-}U1}p \geq b_{UU\text{-}U1}$. The dependency be represented by $p = I_{UU}(z) = Az + b$ where $p \in I_{UU}(D_{U1})$. The new domain is equal to the set $\{A_{U1}z \geq b_{U1} \wedge A_{UU\text{-}U1}(Az + b) \geq b_{UU\text{-}U1}\}$.

Each dependency may introduce more than one equation which will partition the subdomain. In the region where the image, $I_{UU}(D_{U1})$, intersects D_{U1}, we introduce a new equation and rename the argument as $U1$. In the region where the image, $I_{UU}(D_{U1})$, intersects D_{U2}, we introduce a new equation and rename the argument as $U2$. The domain of the equation is changed accordingly and *(3.5)* and *(3.6)* will become:

$$z \in \{A_{U1}z \geq b_{U1} \wedge A_{UU\text{-}U1}(Az + b) \geq b_{UU\text{-}U1}\}$$

$$---> U1(z) = f[U1(I_{UU}(z), V(I_{VU}(z)),...] \quad (3.9)$$

$$z \in \{A_{U1}z \geq b_{U1} \wedge A_{UU\text{-}U2}(Az + b) \geq b_{UU\text{-}U2}\}$$

$$---> U1(z) = f[U2(I_{UU}(z), V(I_{VU}(z)),...] \quad (3.10)$$

The same principle has to be applied for each equation. The renaming procedure can be illustrated by the following examples. Consider the *SARE* shown in Example 1:

$$f(i, j) = h(i, i); for\ i = j \quad (3.11)$$

$$f(i, j) = f(i, j - 1); for\ i < j \quad (3.12)$$

$$f(i, j) = f(i, j + 1); for\ i > j \quad (3.13)$$

The *DG* is shown on Figure 2. The dependence vectors are opposite to each other for the two regions: $i \geq j$, and $i < j$. Let the linear subspaces be $i \geq j$, and $i < j$, and rename f as $f1$ and $f2$. The partitioned domain for f is shown in Figure 3.

The first linear subspace D_{f1} is $i \geq j$. The intersection of D_{f1} and the domain of *(3.11)* is the line $i = j$, and an equation with domain $i = j$ and $f1$ as the result is needed. The intersection of D_{f1} and the domain of *(3.12)* is the empty set, and no equation with $f1$ as result is needed for *(3.12)*. The intersection of D_{f1} and the domain of *(3.13)* is the subspace $i > j$, and an equation with domain $i > j$ and $f1$ as the result is needed.

The second linear subspace D_{f2} is $i < j$. The intersection of D_{f2} and the domain of *(3.11)* is the empty set, and no equation with $f2$ as result is needed for *(3.11)*. The intersection of D_{f2} and the domain of *(3.12)* is the subspace $i < j$, and an equation with domain $i < j$ and $f2$ as the result is needed. The intersection of D_{f2} and the domain of *(3.13)* is the empty set, and no equation with $f2$ as result is needed for *(3.13)*.

The next step is to generate a new *SARE*. For an equation with domain $i < j$ and $f2$

as result, the image of the dependence mapping $(i, j - 1)$ is $i \leq j$. For the region of the image intersect with the subspace $i \geq y$, we put $f1$ as the argument. For the region of the image intersect with the subspace $i < y$, we put $f2$ as the argument.

After the same procedure for each pair of domain and result of equation, the *SARE* becomes:

$$f1(i, j) = h(i, i); \text{for } i = j \tag{3.14}$$

$$f2(i, j) = f1(i, j - 1); \text{for } i = j - 1 \tag{3.15}$$

$$f2(i, j) = f2(i, j - 1); \text{for } i < j - 1 \tag{3.16}$$

$$f1(i, j) = f1(i, j + 1); \text{for } i > j \tag{3.17}$$

Let the timing function for $f1$ and $f2$ be $t_{f1}(i, j) = \lambda^i_{f1} i + \lambda^j_{f1} j + \alpha_{f1}$ and $t_{f2}(i, j) = \lambda^i_{f2} i + \lambda^j_{f2} j + \alpha_{f2}$ respectively; and h is an input variable. Using the constraints in section 2.4, we have:

$$\alpha_{f1} \geq 0;\ \alpha_{f2} \geq 0$$

$$\lambda^j_{f1} < 0;\ \lambda^j_{f2} > 0$$

Let $\alpha_{f1} = \alpha_{f2} = 0$, $\lambda^i_{f1} = 1$, $\lambda^j_{f1} = -1$, $\lambda^i_{f2} = -1$, and $\lambda^j_{f2} = 1$, the schedule for variable $f1$ is $t_{f1}(i, j) = i - j$ and the schedule for variable $f2$ is $t_{f2}(i, j) = j - i$. The linear part of the timing functions are different for variable $f1$ and $f2$: For $f1$, the linear part is: $[1, -1]$; For $f2$, the linear part is: $[-1, 1]$

[Example 6] Let the *SARE* be:

$$f(i, j) = a_0 \quad \text{for } i = j = 0 \tag{3.18}$$

$$f(i, j) = f(j - 1, j - 1) + 3 \quad \text{for } 0 \leq i \leq N;\ 1 \leq j \leq N \tag{3.19}$$

The equations are already in normal form. The variable a_0 is an input variable, and f is an intermediate variable. *(3.18)* is an input equation and *(3.19)* is a computation equations. The above equations are two-dimension and are fully indexed.

The domain of *(3.19)* has vertices at $\sigma_1 = (0, 1)$, $\sigma_2 = (N, 1)$, $\sigma_3 = (0, N)$, $\sigma_4 = (N, N)$. Let $t_f(i,j) = \lambda^i_f i + \lambda^j_f j + \alpha_f$ and using the conditions (i) and (iii) stated in section 2.4, we have:

Condition (i):

for $\sigma_1 = (0, 1)$ -> $\lambda^j_f 1 + \alpha_f - \alpha_f > 0$

for $\sigma_2 = (N, 1)$ -> $\lambda^i_f N + \lambda^j_f 1 + \alpha_f - \alpha_f > 0$

for $\sigma_3 = (0, N)$ -> $\lambda^j_f N + \alpha_f - \lambda^i_f(N-1) - \lambda^j_f(N-1) - \alpha_f > 0$

for $\sigma_4 = (N, N)$ -> $\lambda^i_f N + \lambda^j_f N + \alpha_f - \lambda^i_f(N-1) - \lambda^j_f(N-1) - \alpha_f > 0$

Condition (iii):

for $\sigma_1 = (0, 1)$ -> $\lambda^j_f 1 + \alpha_f \geq 0$

for $\sigma_2 = (N, 1)$ -> $\lambda^i_f N + \lambda^j_f 1 + \alpha_f \geq 0$

for $\sigma_3 = (0, N)$ -> $\lambda^j_f N + \alpha_f \geq 0$

for $\sigma_4 = (N, N)$ -> $\lambda^i_f N + \lambda^j_f N + \alpha_f \geq 0$

Solving the above constraints, the schedule is $t_f(i,j) = j$ and no partition is needed. However, if we want to localize the data flow, we must partition the domain.

Let the domain be partitioned into two regions:

$$D_1: \ (0 \leq i \leq N; \ 0 < j \leq N) \wedge i - j + 1 \geq 0$$

$$D_2: \ (0 \leq i \leq N; \ 0 < j \leq N) \wedge i - j + 1 < 0$$

Let $f \in D_1$ be renamed $f1$, and $f \in D_2$ be renamed $f2$. The partitioned domain for f is

shown on Figure 3. The new *SARE* are:

$$f2(i, j) = a_0; \quad for\ i = j = 0 \tag{3.20}$$

$$f1(i, j) = f1(j - 1, j - 1) + 3; \quad for\ (0 \le i \le N; 0 < j \le N) \wedge i - j + 1 \ge 0 \tag{3.21}$$

$$f2(i, j) = f1(j - 1, j - 1) + 3; \quad for\ (0 \le i \le N; 0 < j \le N) \wedge i - j + 1 < 0 \tag{3.22}$$

The *SARE* we got from the renaming procedure is equivalent to the original *SARE*. The reasons are:

(1) We can repeat a equation arbitrarily many time. A equation such as:

$$z \in D_U \text{ ---> } U(z) = f[U(I_{UU}(z), V(I_{VU}(z)),...]$$

can always be written as:

$$z \in D_{U1} \text{ ---> } U(z) = f[U(I_{UU}(z), V(I_{VU}(z)),...]$$

$$z \in D_{U2} \text{ ---> } U(z) = f[U(I_{UU}(z), V(I_{VU}(z)),...]$$

where D_{U1} and D_{U2} are disjoint and their union is D_U.

(2) The renaming does not change the dependency structure of the equations.

3.3 Partitioning of Domains

In section 3.2, we described a procedure for renaming a *SARE* given any partitioning hyperplane (or any linear subspace), the next step is finding these hyperplanes. We propose two heuristic methods for choosing the linear subspaces for the partitioning. The first one is based on the dependence vector where the vector $z - I(z)$ for each point z is the *dependence vector*.

For an index mapping $I_{VU}(z)$ if the dependence vector is constant (i.e. uniform

dependence), all instances of the vector have the same direction over the entire domain and no partitioning is needed. In general the dependence vector consists of variable terms, and some instances of vector may have different directions in different regions in the domain. Thus the domain may have to be partitioned based on these variable terms.

The motivation for the first heuristic is as follows: In one-dimensional space, if some instances of dependence vectors are opposite to each other, then schedules with different linear parts are needed. The point where the dependence vectors change direction is ideal for partitioning the domain. This can be extended for higher dimensional space to localize the data flow.

However, if the image of the index mapping of a n-dimensional space is to a (n - 1) dimensional space, then the (n - 1) dimensional space can be chosen to be the hyperplane for time = 0, and a linear schedule can always be found. Let take an example to illustrate this:

[Example 7] A *SARE* is represented by the equation:

$$f(i, j) = h(i, i); \text{ for } 0 \le i, j \le N \qquad (3.23)$$

The *DG* is shown on Figure 5. The dependence vector $z - I(z)$ is equal to $[0, j - i]^T$. It contains a variable term $j - i$. Based on the hyperplane $j - i = 0$: for $i < j$ the dependence vector is $[0, -k]$; for $i = j$ the dependence vector is $[0, 0]$; for $i \ge j$, the dependence vector is $[0, +k]$ (where k is a positive integer). In the three regions, the second component of the dependence vector is respectively, negative, zero, and positive. Thus we use the hyperplane $i = j$ to partition the domain into the following two regions:

$$j - i \geq 0 \text{ and } j - i < 0$$

The choice of the equality is arbitrary.

The original domain can be partitioned into two regions by taking the *intersection* of the original domain and the domains for the two inequalities, we have:

$$D_1: \ (0 \leq i, j \leq N) \wedge (i \geq j)$$

$$D_2: \ (0 \leq i, j \leq N) \wedge (i < j)$$

Based on the original *SARE*, if we try to find a schedule for f using the constraints in section 2.4, we will find that the partitioning is unnecessary. For the original domain we have the vertices $\sigma 1 = (0, 0)$, $\sigma_2 = (0, N)$, $\sigma_3 = (N, 0)$, and $\sigma_4 = (N, N)$. Let the timing function for f and h be $t_f(i, j) = \lambda^i_f i + \lambda^j_f j + \alpha_f$ and $t_h(i, j) = \lambda^i_h i + \lambda^j_h j + \alpha_h$ respectively; Using conditions (i) and (iii) in section 2.4, we have:

From condition (i):

$for\ \sigma_1 = (0, 0) \rightarrow \ \alpha_f - \alpha_h > 0$

$for\ \sigma_2 = (0, N) \rightarrow \ \lambda^i_f N + \alpha_f - \alpha_h > 0$

$for\ \sigma_3 = (N, 0) \rightarrow \ \lambda^i_f N + \alpha_f - \lambda^i_h N - \alpha_h > 0$

$for\ \sigma_4 = (N, N) \rightarrow \ \lambda^i_f N + \lambda^j_f N + \alpha_f - \lambda^i_h N - \lambda^j_h N - \alpha_h > 0$

From condition (iii):

$for\ \sigma_1 = (0, 0) \rightarrow \ \alpha_f \geq 0$

$for\ \sigma_2 = (0, N) \rightarrow \ \lambda^j_f N \geq 0$

$for\ \sigma_3 = (N, 0) \rightarrow \ \lambda^i_f N \geq 0$

$for\ \sigma_4 = (N, N) \rightarrow \ \lambda^i_f N + \lambda^j_f N + \alpha_f \geq 0$

Intuitively, the reason that partitioning is not necessary is that once the values of

variable h along the line $i = j$ are calculated, all of the values for f can be calculated simultaneously.

If h is an input variable, then all instances of h are available at time = 0. Moreover, every instances of f can be calculated at time > 0. Consequently, input equations can be ignored during partitioning.

If we want to localize *(3.23)*, then it will be rewritten as in Example 1:

$$f(i, j) = h(i, i) \text{ for } i = j \qquad (3.24)$$

$$f(i, j) = f(i, j - 1) \text{ for } i < j \qquad (3.25)$$

$$f(i, j) = f(i, j + 1) \text{ for } i > j \qquad (3.26)$$

For the above *SARE*, the partitioning is necessary.

Besides partitioning based on the variable terms in the dependence vectors, we may have to partition based on the domains of equations as originally given in the *SARE*. The reason is that for *(3.24-3.26)* the dependence vector has different directions in the two domains ([0, 1] in domain $i < j$, and [0, -1] in domain $i > j$). Thus schedules with different slopes are required.

To partition based on the domains as originally given in the equations, we have to make sure that it is necessary. It can be done by first checking for valid affine schedules. For the following *SAREs* valid affine schedules can be found:

System 1: $f(i, j) = f(i, j - 1) \text{ for } i < j \qquad (3.31)$

$$f(i, j) = f(i - 1, j - 1) \text{ for } i \geq j \qquad (3.32)$$

System 2: $f(i, j) = f(i, j - 1) \text{ for } i < j \qquad (3.33)$

$$f(i, j) = f(i, j - 1) \text{ for } i \geq j \qquad (3.34)$$

For some *SAREs*, although piecewise affine schedule are not necessary, they may provide more flexible and faster schedules because of the relaxed constraints.

The following is an example where both heuristic methods have to be used.

[Example 8] *Let the SARE* be:

$$f(i, j) = a_0; \quad for\ i = j = 0 \tag{3.31}$$

$$f(i, j) = f(j - 1, j - 1) + f(2N - i, j); \quad for\ N \le i < 2N, 0 < j \le 2N \tag{3.32}$$

$$f(i, j) = f(j - 1, j - 1) + f(2N - i - 1, j); \quad for\ 0 \le i < N, 0 < j \le 2N \tag{3.33}$$

$$f(i, j) = f(j - 1, j - 1); \quad for\ i = N, 0 < j \le 2N \tag{3.34}$$

The *DG* for the *SARE* is shown on Figure 6. Based on the domains given in *(3.32-3.34)* we use the hyperplane $i = N$ to partition the domain. For the index mapping *(j - 1, j - 1)*, the second component of the dependence vector have different values in three regions: in $i - j + 1 < 0$, it is negative; in $i - j + 1 = 0$, it is zero; in $i - j + 1 > 0$, it is positive. Thus we can partition the domain of f by two hyperplanes: $i = N$ and $i - j + 1 = 0$. 4 regions are created and f is renamed as $f1, f2, f3$, and $f4$.

The 4 possible regions are :

	$i - j + 1 \ge 0$	$i - j + 1 < 0$
$i \ge N$	$f1$	$f2$
$i < N$	$f3$	$f4$

The partitioned domain for f is shown on Figure 7. After renaming, the new set of *SARE* are:

$$f3(i, j) = a_0; \quad for\ i = j = 0 \tag{3.35}$$

$$f1\hat{}Xx, j) = f3(j - 1, j - 1) + f3(2N - i, j);\ for\ (i \geq N + 1) \wedge (j \geq 1) \wedge (j + i - 2N \leq 0) \tag{3.36}$$

$$f1(i, j) = f3(j - 1, j - 1) + f4(2N - i, j);\ for\ (i \leq N - 1) \wedge (j \geq N - 1) \wedge (j + i - 2N > 0) \tag{3.37}$$

$$f1(i, j) = f3(j - 1, j - 1) + f4(2N - i, j);\ for\ N < i < 2N, j = 2N \tag{3.38}$$

$$f1(i, j) = f1(j - 1, j - 1) + f4(2N - i, j);\ for\ N < i < 2N, N < j \leq i \tag{3.39}$$

$$f1(i, j) = f3(j - 1, j - 1);\ for\ i = N, 0 < j \leq N \tag{3.40}$$

$$f2(i, j) = f4(j - 1, j - 1);\ for\ i = N, j = N + 1 \tag{3.41}$$

$$f2(i, j) = f2(j - 1, j - 1);\ for\ i = N, N + 1 < j \leq 2N \tag{3.42}$$

$$f2(i, j) = f2(j - 1, j - 1) + f4(2N - i, j);\ for\ N < i < 2N, i < j \leq 2N \tag{3.43}$$

$$f3(i, j) = f3(j - 1, j - 1) + f1(2N - i - 1, j);\ for\ 0 \leq i < N, 0 < j \leq i \tag{3.44}$$

$$f4(i, j) = f1(j - 1, j - 1) + f2(2N - i - 1, j);\ for\ (j \leq 2N) \wedge (i \leq N + 1) \wedge (j + i - 2N > 0) \tag{3.45}$$

$$f4(i, j) = f1(j - 1, j - 1) + f1(2N - i - 1, j);\ for\ (j \geq N + 1) \wedge (i \geq 0) \wedge (j + i - 2N \leq 0) \tag{3.46}$$

$$f4(i, j) = f3(j - 1, j - 1) + f1(2N - i - 1, j);\ for\ 0 \leq i < N, i < j \leq N \tag{3.47}$$

3.4 Finding a Schedule for Each Variable

Once we have partitioned and renamed the SARE, we can use the method of Mauras et al. [10] for finding the schedules. After the partitioning and renaming, we have a *SARE* with the following properties:

(1) The domain for each equation is a convex polyhedron.

(2) The domains of two equations having the same variable as result are disjoint.

(3) No dependence vectors in a domain point in opposite directions.

For each equation, we find the vertices and rays for the domain and construct the *RDG* for the *SARE*. For each variable we use the constraints described in section 2.4 to get a system of inequalities, and a schedule can be found by solving these inequalities.

[Example 9] Let the *SARE*:

$$V(i) = a_0; \quad for\ i = N \tag{3.48}$$

$$V(i) = V(2N - i - 1); \quad for\ 0 \le i < N \tag{3.49}$$

$$V(i) = 1 + V(2N - i); \quad for\ N < i < 2N \tag{3.50}$$

The *DG* for the *SARE* is shown on Figure 8.

Step 1: Transform the *SARE* into normal form.

a_0 is an input variable, and V is an intermediate variable. *(3.48)* is an input equation; *(3.49-3.50)* are computation equations. All equations are one-dimension and are fully indexed.

Step 2: Check if an affine schedule can be found for each variable.

For $0 \le i < N$ the dependence vectors are pointed to the $+i$ direction and for $N < i < 2N$ the dependence vectors are pointed to the $-i$ direction, and a single schedule cannot be found for the two domains.

Step 3: Partition the domains of variable based on the dependence vectors. No further partition is needed inside each domain. The line $i = N$ separates the two domains $0 \le i$

$< N$ and $N < i < 2N$.

Step 4: Obtaining an equivalent system of equations by renaming the variables according to the result of partitioning.

For the domain $0 \leq i < N$, V is renamed as $V1$ and for the domain $N \leq i < 2N$, V is renamed as $V2$. From *(3.48)*, a new equation with domain $i = N$ and result $V2$ is needed. From *(3.49)*, a new equation with domain $0 \leq i < N$ and result $V1$ is needed. From *(3.50)*, a new equation with domain $N < i < 2N$ and result $V2$ is needed.

After renaming the variables for the left-hand side and right-hand side of each equation, the new *SARE* are:

$$V2(i) = a_0; \quad for\ i = N \tag{3.51}$$

$$V1(i) = V2(2N - i - 1); \quad for\ 0 \leq i < N \tag{3.52}$$

$$V2(i) = 1 + V1(2N - i); \quad for\ N < i < 2N \tag{3.53}$$

Step 5: Find schedule for each variable using the constraints in section 2.4.

The vertices for $V1$ are $\sigma_1 = 0$ and $\sigma_2 = N - 1$; the vertices for $V2$ are $\sigma_3 = N + 1$ and $\sigma_4 = 2N - 1$. The *RDG* for the equations have two arcs: $(V1, V2, D_{V1}, I_{V2V1})$ and $(V2, V1, D_{V2}, I_{V1V2})$. Let the timing function for $V1$ and $V2$ be $t_{V1}(i) = \lambda_{V1} i + \alpha_{V1}$ and $t_{V2}(i) = \lambda_{V2} i + \alpha_{V2}$ respectively; from conditions (i) and (iii) in section 2.4, we have:

For variable $V1$;

$\sigma_1 = 0 \rightarrow \alpha_{V1} - \lambda_{V2}(2N - 1) - \alpha_{V2} > 0$

$$\alpha_{V1} \geq 0$$

$\sigma_2 = N - 1 \rightarrow \lambda_{V1}(N - 1) + \alpha_{V1} - \lambda_{V2} N - \alpha_{V2} > 0$

$$\lambda_{V1}(N - 1) + \alpha_{V1} \geq 0$$

For variable *V2*;

$\sigma_3 = N + 1 \rightarrow \alpha_{V2}.(N + 1) + \alpha_{V2} - \lambda_{V1}(N - 1) - \alpha_{V1} > 0$

$\alpha_{V2}(N + 1) + \alpha_{V2} \geq 0$

$\sigma_4 = 2N - 1 \rightarrow \lambda_{V2}(2N - 1) + \alpha_{V2} - \lambda_{V1}(1) - \alpha_{V1} > 0$

$\lambda_{V2}(2N - 1) + \alpha_{V2} \geq 0$

Solving the above set of inequalities, we obtain the following schedules:

$t_{V1}(i) = -2i + 2N$

$t_{V2}(i) = 0 \quad for\ i = N$

$= 2i - 2N + 1 \quad otherwise$

Chapter 4

EXAMPLES

4.1 Toeplitz System

Toeplitz system involves solving a equation of the form:

$$Tx = y$$

where T is the matrix whose elements are $t(i, j) = t_{i-j} = t_k$; *for* $-N \leq k \leq N$.

That is, $T = \begin{bmatrix} t_0 & t_{-1} & t_{-2} & t_{-3} \\ t_1 & t_0 & t_{-1} & t_{-2} \\ t_2 & t_1 & t_0 & t_{-1} \\ t_3 & t_2 & t_1 & t_0 \end{bmatrix}$

To solve for x, there are two steps:

1) Triangular Decomposition of matrix T as $LT = U$, where U is an upper triangular matrix and L is a lower triangular matrix.

2) Back-substitution

A method based on the Schur's algorithm is given for performing the first step which is described in a *SARE* as follows [8]:

$v(1,j) = t_j$; *for* $-N \leq j \leq N$, $i = 1$ *(4.1)*

$u(1,j) = t_j$; *for* $-N \leq j \leq N$, $i = 1$ *(4.2)*

$v(i,j) = v(i - 1, j) - u(i - 1, 1).u(i - 1, j + 1) / v(i - 1, 0);$

$for\ 1 < i \le N + 1, -N \le j \le N$ (4.3)

$u(i,j) = u(i - 1, j + 1) - u(i - 1, -i + 1).v(i - 1, j) / v(i - 1, -i + 2);$

$for\ 1 < i \le N + 1, -N \le j \le N$ (4.4)

v's are elements of matrix *L* and *u's* are elements of matrix *U*. The *DG* for the variable *v* is shown on Figure 9, 10, and the *DG* for the variable *u* is shown on Figure 11, 12. For the calculation of *v(i, j)*,

we first evaluate the dependence vectors:

$I^1_{vv} = (i - 1, j) \text{ --> } vector = [1, 0]^T$

$I^1_{uv} = (i - 1, 1) \text{ --> } vector = [1, j - 1]^T$

$I^2_{vv} = (i - 1, 0) \text{ --> } vector = [1, j]^T$

$I^2_{uv} = (i - 1, j + 1) \text{ --> } vector = [1, -1]^T$

There are two variable terms, and we can partition the domain of *v* by the two hyperplane $j - 1 = 0$ and $j = 0$. There are 3 possible regions:

$j \ge 1$	$j < 1$
$j \ge 0$	$j \ge 1$	$j = 0$
$j < 0$	*xxxx*	$j < 1$

Let $D_{v1} = 1 < i \le N + 1, 1 \le j \le N$

$D_{v2} = 1 < i \le N + 1, j = N$

$D_{v3} = 1 < i \le N + 1, -N \le j < 1$

and rename *v* by *v1*, *v2*, and *v3* accordingly. The partitioned domain of *v* is shown on

Figure 13.

For the calculation of $u(i, j)$, we first evaluate the dependence vectors:

$I^1_{uu} = (i - 1, j + 1) \text{ --> } vector = [1, -1]^T$

$I^2_{uu} = (i - 1, -i + 1) \text{ --> } vector = [1, j + i - 1]^T$

$I^1_{vu} = (i - 1, -i + 1) \text{ --> } vector = [1, j + i - 2]^T$

$I^2_{vu} = (i - 1, j) \text{ --> } vector = [1, 0]^T$

There are two variable terms, and we can partition the domain of u by the two hyperplane $j + i - 1 = 0$ and $j + i - 2 = 0$. There are 3 possible regions:

$j + i - 1 \geq 0$ $\quad$ $j + i - 1 < 0$

$j + i - 2 \geq 0$ $\quad$ $j + i - 2 \geq 0$ $\quad$ $xxxx$

$j + i - 2 < 0$ $\quad$ $j + i - 1 = 0$ $\quad$ $j + i - 1 < 0$

Let $D_{u1} = (1 < i \leq N + 1) \wedge (1 \leq j \leq N) \wedge (j + i - 2 \geq 0)$

$D_{u2} = (1 < i \leq N + 1) \wedge (j = N) \wedge (j + i - 1 = 0)$

$D_{u3} = (1 < i \leq N + 1) \wedge (-N \leq j < 1) \wedge (j + i - 1 < 0)$

and rename u by $u1$, $u2$, and $u3$. The partitioned domain of u is shown on Figure 14.

The original *SARE* can be rewritten as:

$v1(1,j) = t_j; \text{ for } 1 \leq j \leq N, i = 1$ (4.5)

$v2(1,j) = t_j; \text{ for } j = 0, i = 1$ (4.6)

$v3(1,j) = t_j; \text{ for } -N \leq j \leq -1, i = 1$ (4.7)

$u1(1,j) = t_j; \text{ for } 1 \leq j \leq N, i = 1$ (4.8)

$u2(1,j) = t_j$; *for* $j = 0, i = 1$ (4.9)

$u3(1,j) = t_j$; *for* $-N \leq j \leq -1, i = 1$ (4.10)

$v1(i,j) = v1(i - 1, j) - u1(i - 1, 1).u1(i - 1, j + 1) / v2(i - 1, 0)$; *for* D_{v1} (4.11)

$v2(i,j) = v2(i - 1, j) - u1(i - 1, 1).u1(i - 1, j + 1) / v2(i - 1, 0)$; *for* D_{v2} (4.12)

$v3(i,j) = v3(i - 1, j) - u1(i - 1, 1).u1(i - 1, j + 1) / v2(i - 1, 0)$; *for* D_{v3}

$\wedge (j + i - 2 \geq 0)$ (4.13)

$v3(i,j) = v3(i - 1, j) - u1(i - 1, 1).u2(i - 1, j + 1) / v2(i - 1, 0)$; *for* D_{v3}

$\wedge (j + i - 1 = 0)$ (4.14)

$v3(i,j) = v3(i - 1, j) - u1(i - 1, 1).u3(i - 1, j + 1) / v2(i - 1, 0)$; *for* D_{v3}

$\wedge (j + i - 1 < 0)$ (4.15)

$u1(i,j) = u1(i - 1, j + 1) - u3(i - 1, -i + 1).v1(i - 1, j) / v2(i - 1, -i + 2)$;

for $D_{u1} \wedge (j \geq 1 \wedge i = 2)$ (4.16)

$u1(i,j) = u1(i - 1, j + 1) - u3(i - 1, -i + 1).v1(i - 1, j) / v3(i - 1, -i + 2)$;

for $D_{u1} \wedge (j \geq 1 \wedge i > 2)$ (4.17)

$u1(i,j) = u1(i - 1, j + 1) - u3(i - 1, -i + 1).v2(i - 1, j) / v2(i - 1, -i + 2)$;

for $D_{u1} \wedge (j = 0 \wedge i = 2)$ (4.18)

$u1(i,j) = u1(i - 1, j + 1) - u3(i - 1, -i + 1).v2(i - 1, j) / v3(i - 1, -i + 2)$;

for $D_{u1} \wedge (j = 0 \wedge i > 2)$ (4.19)

$u1(i,j) = u1(i - 1, j + 1) - u3(i - 1, -i + 1).v3(i - 1, j) / v3(i - 1, -i + 2)$;

for $D_{u1} \wedge (j < 0)$ (4.20)

$u2(i,j) = u2(i - 1, j + 1) - u3(i - 1, -i + 1).v3(i - 1, j) / v2(i - 1, -i + 2)$;

for $D_{u2} \wedge (i = 2)$ (4.21)

$u2(i,j) = u2(i - 1, j + 1) - u3(i - 1, -i + 1).v3(i - 1, j) / v3(i - 1, -i + 2);$

$for\ D_{u2} \wedge (i > 2)$ (4.22)

$u3(i,j) = u3(i - 1, j + 1) - u3(i - 1, -i + 1).v3(i - 1, j) / v2(i - 1, -i + 2);$

$for\ D_{u3} \wedge (i = 2)$ (4.23)

$u3(i,j) = u3(i - 1, j + 1) - u3(i - 1, -i + 1).v3(i - 1, j) / v3(i - 1, -i + 2);$

$for\ D_{u3} \wedge (i > 2)$ (4.24)

4.2 Algebraic Path Problem

In Rajopadhye [17], an algorithm for solving the algebraic path problem in a **SARE** is given as follows:

$$
\begin{aligned}
f(i, j, k) &= f(i, j, k - 1)^*; for\ i = j = k \\
&= f(i, k, k) \cdot f(i, j, k - 1); for\ i = k \wedge j \neq k \\
&= f(i, k, k - 1) \cdot f(k, j, k); for\ j = k \wedge i \neq k \\
&= f(i, k, k - 1) \ @ f(i, k, k) \cdot f(k, j, k - 1); for\ i \neq k \wedge j \neq k
\end{aligned}
\qquad (4.25)
$$

"*", "·", and "@" are operators which depend on the exact problem to be solved. First evaluate the dependence vector:

$I^1_{ff} = (i, j, k - 1) \text{ --> } vector = [0, 0, 1]^T$

$I^2_{ff} = (i, k, k) \text{ --> } vector = [0, j - k, 0]^T$

$I^3_{ff} = (k, j, k) \text{ --> } vector = [i - k, 0, 0]^T$

$I^4_{ff} = (k, j, k - 1) \text{ --> } vector = [i - k, 0, 1]^T$

There are two variable terms, and we can partition the domain of v by the two

hyperplane $j - k = 0$ and $i - k = 0$. The two hyperplanes are shown on Figure 15. There are 4 possible regions:

	$j \geq k$	$j < k$
$i \geq k$	$f1$	$f2$
$i < k$	$f3$	$f4$

where $f1$ is the name of the variable in the domain $(i \geq k) \wedge (j \geq k)$

$f2$ is the name of the variable in the domain $(i \geq k) \wedge (j < k)$

$f3$ is the name of the variable in the domain $(i < k) \wedge (j \geq k)$

$f4$ is the name of the variable in the domain $(i < k) \wedge (j < k)$

The original *SARE* can be rewritten as:

$$f1(i, j, k) = f1(i, j, k - 1)^*; \text{ for } i = j = k \quad (4.26)$$

$$f1(i, j, k) = f1(i, k, k) \cdot f1(i, j, k - 1); \text{ for } (i = k) \wedge (j > k) \quad (4.27)$$

$$f2(i, j, k) = f1(i, k, k) \cdot f1(i, j, k - 1); \text{ for } (i = k) \wedge (j = k - 1) \quad (4.28)$$

$$f2(i, j, k) = f1(i, k, k) \cdot f2(i, j, k - 1); \text{ for } (i = k) \wedge (j < k) \quad (4.29)$$

$$f1(i, j, k) = f1(i, j, k - 1) \cdot f1(k, j, k); \text{ for } (j = k) \wedge (i > k) \quad (4.30)$$

$$f3(i, j, k) = f1(i, j, k - 1) \cdot f1(k, j, k); \text{ for } (j = k) \wedge (i = k - 1) \quad (4.31)$$

$$f3(i, j, k) = f2(i, j, k - 1) \cdot f1(k, j, k); \text{ for } (j = k) \wedge (i < k - 1) \quad (4.32)$$

$$f1(i, j, k) = f1(i, j, k - 1) \cdot f1(i, k, k) @ f1(k, j, k); \text{ for } (i > k) \wedge (j > k) \quad (4.33)$$

$$f2(i, j, k) = f1(i, j, k - 1) \cdot f1(i, k, k) @ f1(k, j, k); \text{ for } (i > k) \wedge (j = k - 1) \quad (4.34)$$

$$f2(i, j, k) = f2(i, j, k - 1) \cdot f1(i, k, k) @ f2(k, j, k); \text{ for } (i > k) \wedge (j < k - 1) \quad (4.35)$$

$$f3(i, j, k) = f1(i, j, k - 1) \cdot f3(i, k, k) \,@\, f1(k, j, k); \quad for\ (i = k - 1) \wedge (j > k) \tag{4.36}$$

$$f3(i, j, k) = f3(i, j, k - 1) \cdot f3(i, k, k) \,@\, f1(k, j, k); \quad for\ (i < k - 1) \wedge (j > k) \tag{4.37}$$

$$f4(i, j, k) = f1(i, j, k - 1) \cdot f3(i, k, k) \,@\, f1(k, j, k); \quad for\ (i = k - 1) \wedge (j = k - 1) \tag{4.38}$$

$$f4(i, j, k) = f2(i, j, k - 1) \cdot f3(i, k, k) \,@\, f2(k, j, k); \quad for\ (i = k - 1) \wedge (j < k - 1) \tag{4.39}$$

$$f4(i, j, k) = f3(i, j, k - 1) \cdot f3(i, k, k) \,@\, f1(k, j, k); \quad for\ (i < k - 1) \wedge (j = k - 1) \tag{4.40}$$

$$f4(i, j, k) = f4(i, j, k - 1) \cdot f3(i, k, k) \,@\, f2(k, j, k); \quad for\ (i < k - 1) \wedge (j < k - 1) \tag{4.41}$$

CONCLUSIONS

For a system of affine recurrence equations, linear schedules do not always exist. By using piecewise affine schedules, linear subspaces of a domain can have different linear schedules, hence allow a wider class of problem to be solved. In some cases, piecewise affine schedules provide faster schedule by getting additional parallelism from an algorithm.

In this thesis, we present a procedure for finding piecewise affine schedules. We provide some heuristics for finding partitioning hyperplane, and an algorithm for generating a new *SARE* by renaming the variables. A method for finding these schedules is given.

Several open questions remain:

(1) How to choose partitioning hyperplanes so that piecewise affine schedule are guaranteed to be found?

(2) How to improve the renaming procedure for finding optimal linear schedules for each variable?

Further studies are needed to answer these question.

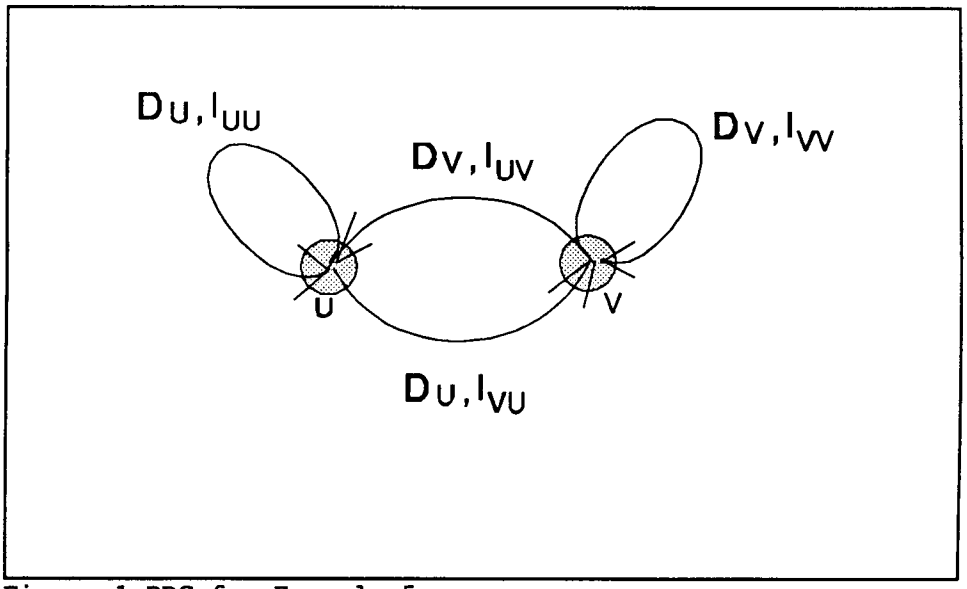

Figure 1 RDG for Example 5

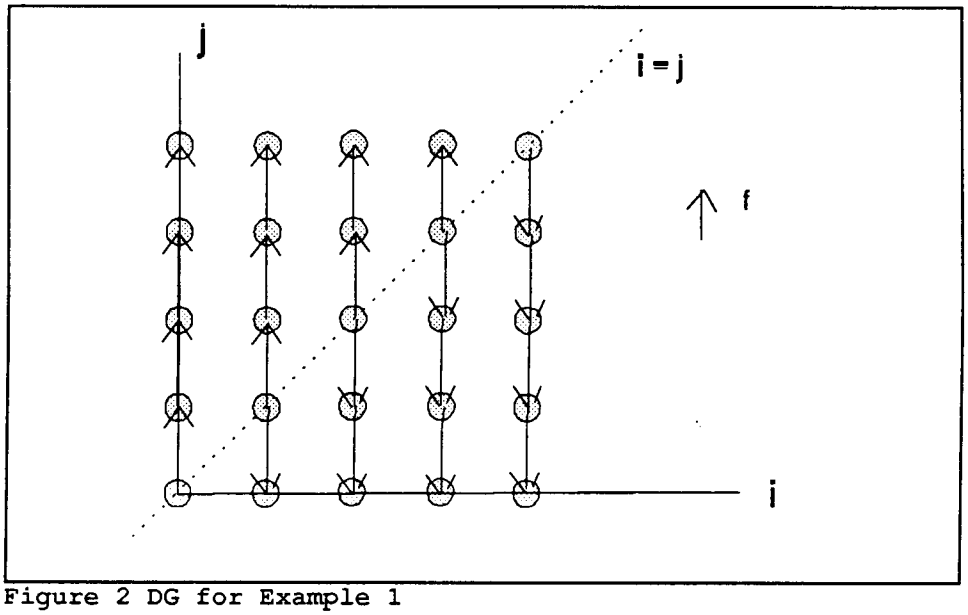

Figure 2 DG for Example 1

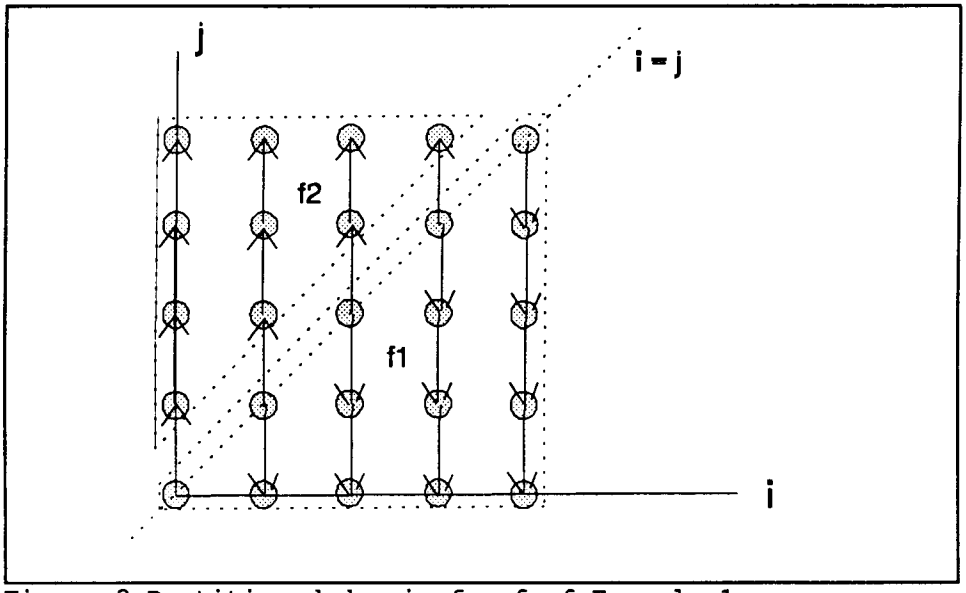

Figure 3 Partitioned domain for f of Example 1

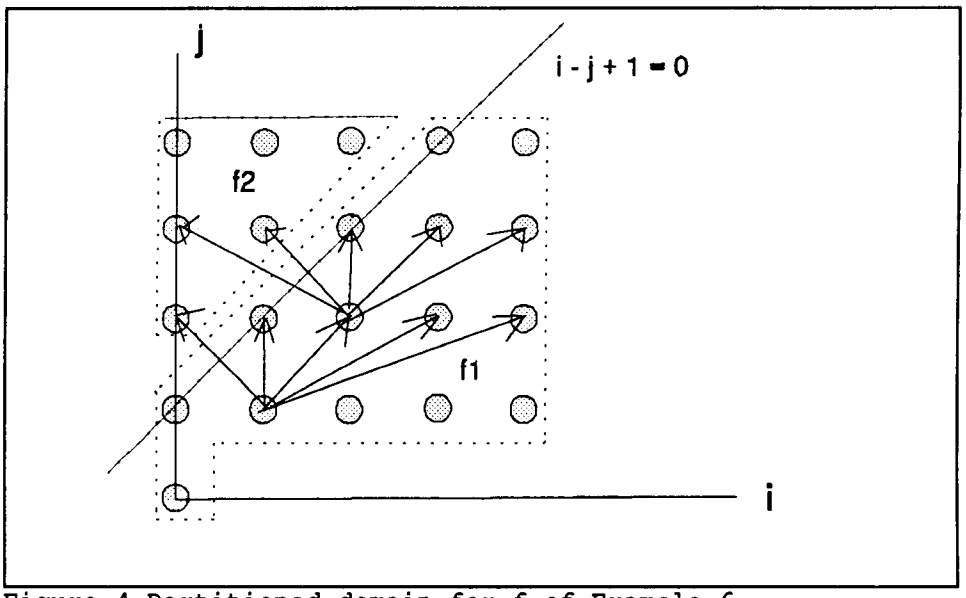

Figure 4 Partitioned domain for f of Example 6

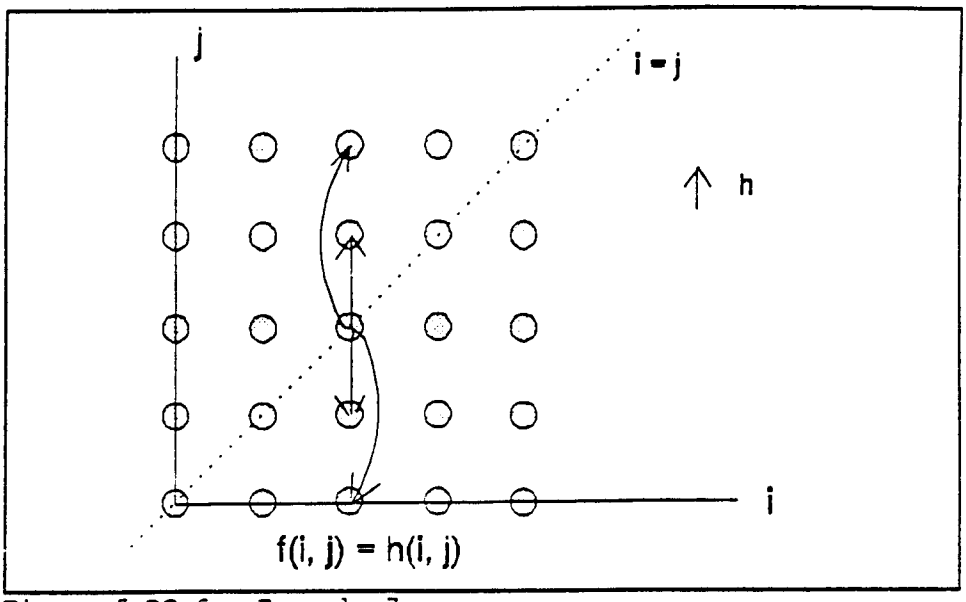

Figure 5 DG for Example 7

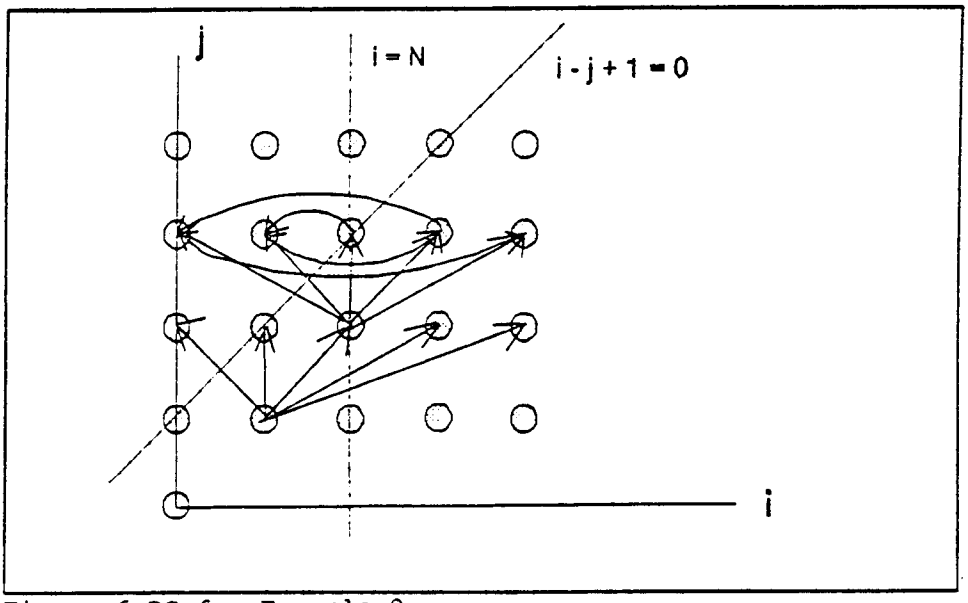

Figure 6 DG for Example 8

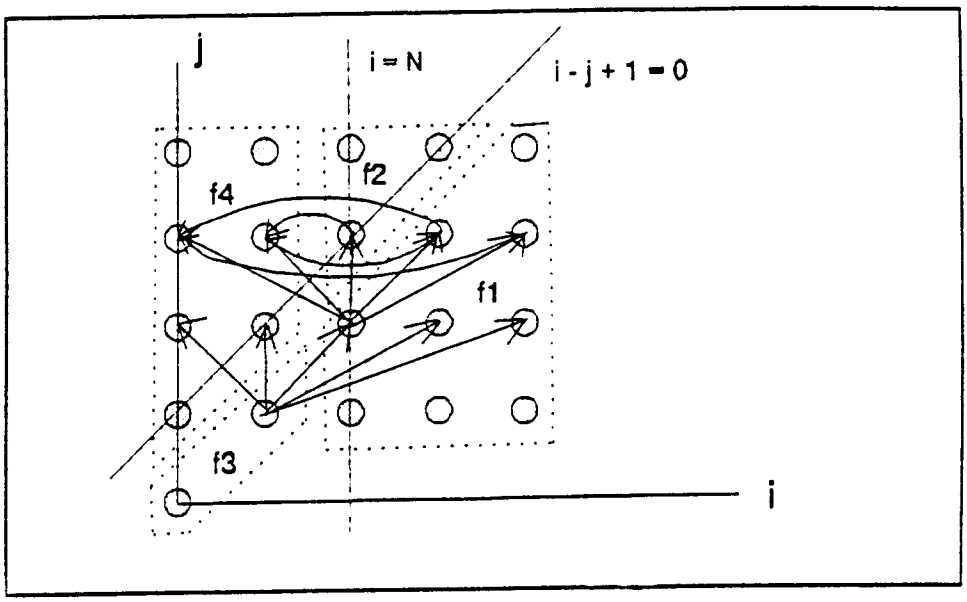

Figure 7 Partitioned domain for f of Example 8

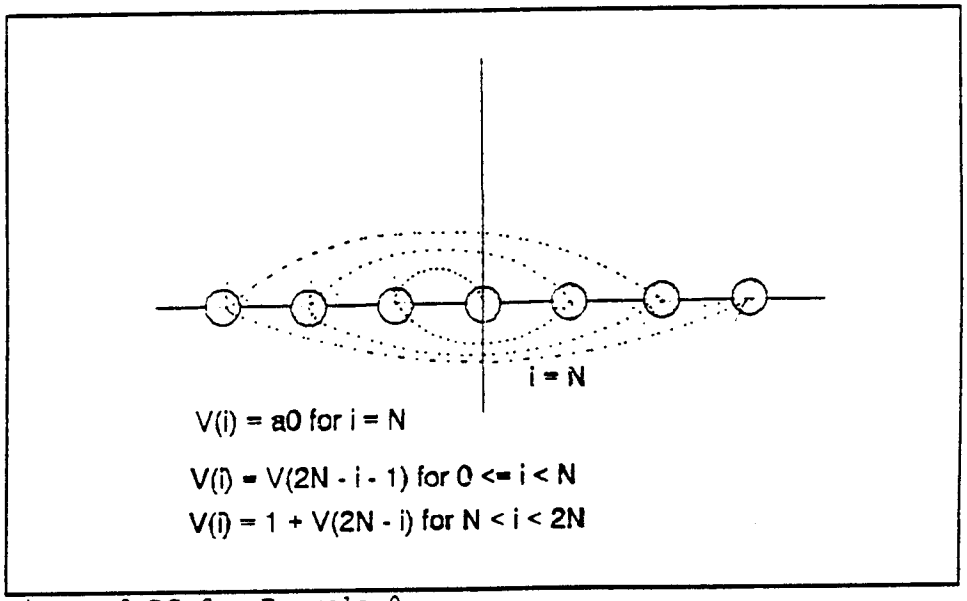

Figure 8 DG for Example 9

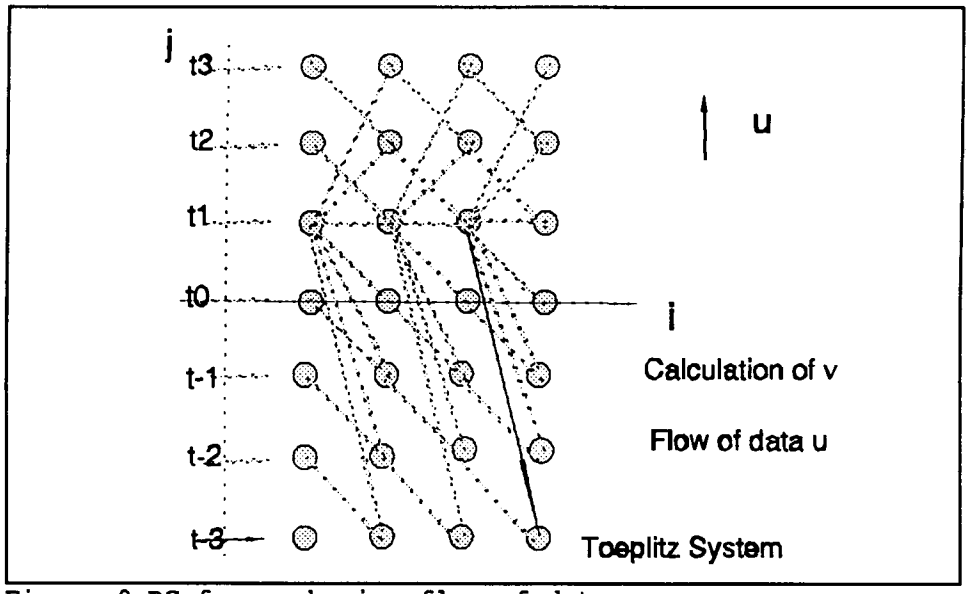

Figure 9 DG for v showing flow of data u

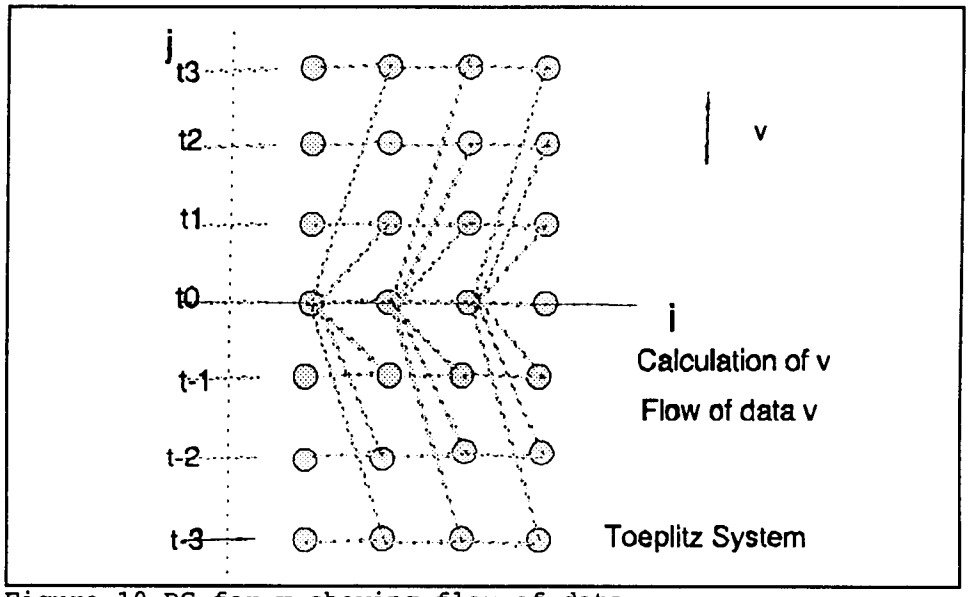

Figure 10 DG for v showing flow of data v

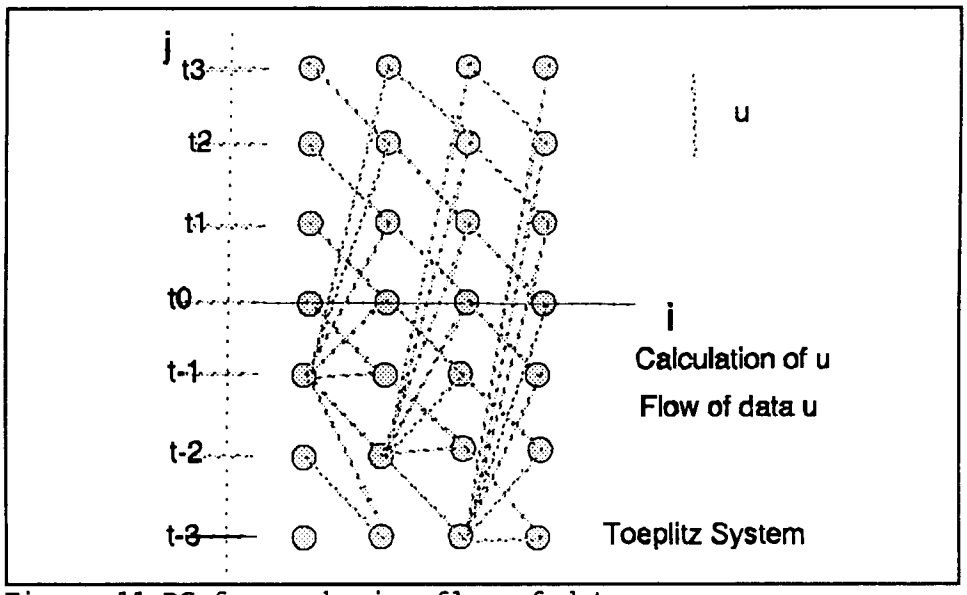

Figure 11 DG for u showing flow of data u

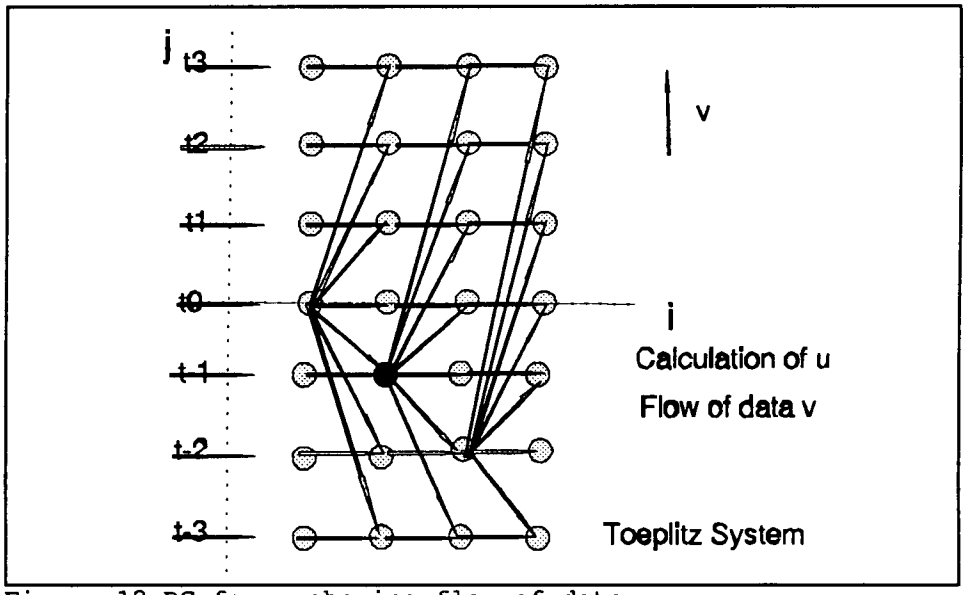

Figure 12 DG for u showing flow of data v

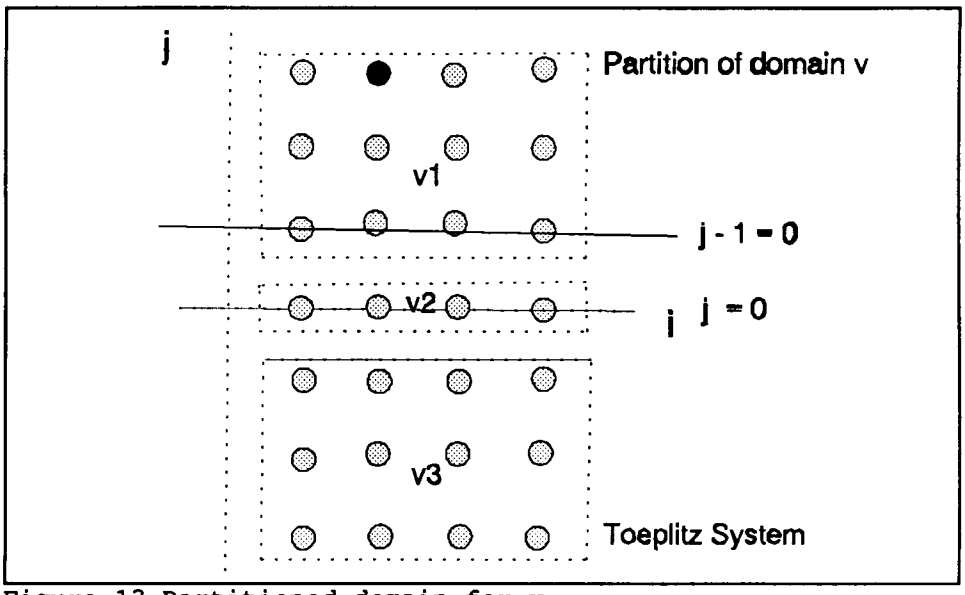

Figure 13 Partitioned domain for v

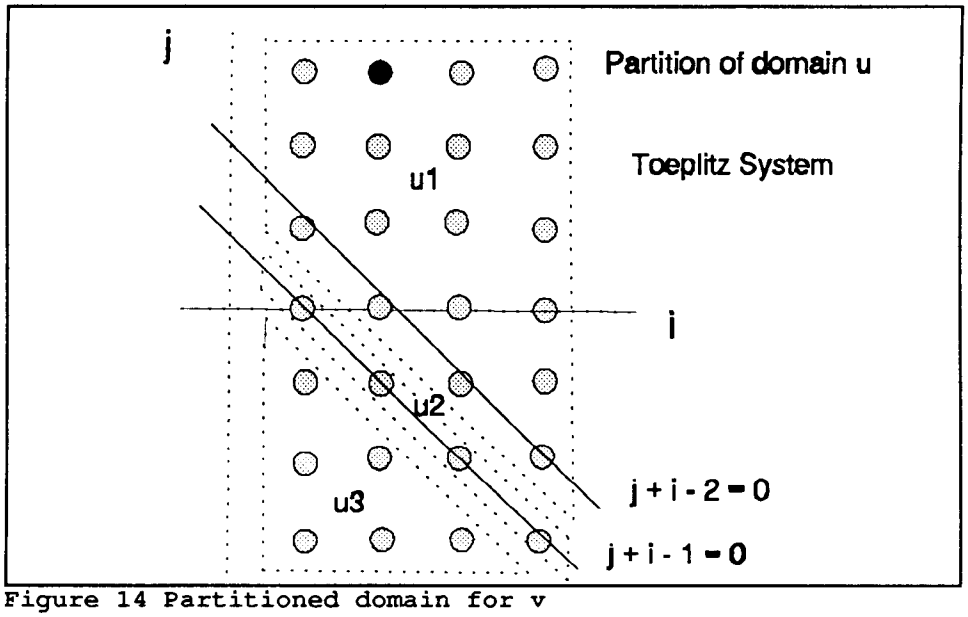

Figure 14 Partitioned domain for v

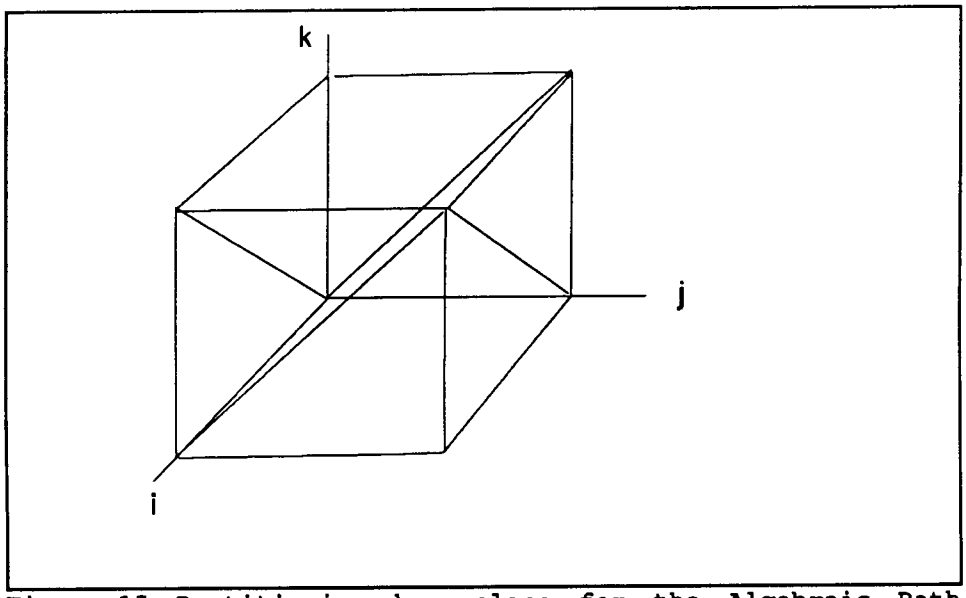

Figure 15 Partitioning hyperplane for the Algebraic Path Problem

BIBLIOGRAPHY

1. D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, 1989.

2. J.M. Delosme and I.C.F. Ipsen, "Systolic array Synthesis: Computability and Time Cones," In M.Cosnard, P. Quinton, Y. Robert, and M. Tchuente, editors, *Parallel Algorithms and Architectures Conference*, pp. 295-312. 1986.

3. H.V. Jagadish, S.K. Rao, and T. Kailath, "Array Architectures for Iterative Algorithms," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1304-1321, September 1987.

4. R.M. Karp, R.E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *J. ACM*, vol. 14, no. 3, pp. 563-590, July 1967.

5. H.T. Kung, "Why Systolic Architectures?" *Computer*, vol. 15, no. 1,pp. 37-46, January 1982.

6. S.Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proceedings of the IEEE*, vol. 72, no. 7, pp. 867-884, July 1984.

7. S.Y. Kung, *VLSI Array Processors*, Prentice-Hall, 1988.

8. S.Y. Kung and Y.H. Hu, "A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-31, no. 1, pp. 66-76, February 1983.

9. S.Y. Kung, H.J. Whitehouse and T. Kailath, *VLSI and Modern Signal Processing*, Prentice-Hall, 1985.

10. Mauras, P. Quinton, S.V. Rajopadhye, and Y. Saouter, "Scheduling Affine

Parameterized Recurrences by means of Variable Dependent Timing Functions," In S.Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pp. 100-110, Princeton, New jersey, IEEE Computer Society, September 1990.

11. W.L. Miranker and A. Winkler, "Spacetime Representations of Computational Structures," *Computing*, vol. 32, pp. 93-114, 1984.

12. D.J. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proceedings of the IEEE*, vol. 71, no. 1, pp. 113-120, January 1983.

13. P. Quinton, *The Systematic Design of Systolic Arrays*, Tech. Rep. 216, Institut National de Recherche en Informatique et en Automatique INRIA, July 1983.

14. P. Quinton and V. Van Dongen, *The Mapping of Linear Recurrence Equations on Regular Arrays*, Manuscript M 264, Philips Research Lab, Brussels, October 1988.

15. P. Quinton and V. Van Dongen, "The Mapping of Linear Recurrence Equations on Regular Arrays," *Journal of VLSI Signal Processing*, vol. 1, pp. 95-113, 1989.

16. S.V. Rajopadhye and R.M. Fujimoto, "Synthesizing systolic arrays from recurrence equations," *Parallel Computing*, vol. 14, pp. 163-189, 1990.

17. S.V. Rajopadhye, "Parallel and Systolic Implementations of the Algebraic Path Problem," *Seminar*, Oregon State University, Comput. Sci. Dept., Corvallis, OR, 1991.

18. S.K. Rao and T. Kailath, "Regular Iterative Algorithms and their implementation on Processor Arrays," *Proceedings of the IEEE*, vol. 76, no. 3, pp. 259-269, March 1988.

19. Y. Saouter and P. Quinton, *Computability of Affine Recurrence Equations*. Tech. Rep. 1203, INISA, Campus de Beauleau, Rennes, France, April 1990.

20. Y. Yaacoby and P.R. Cappello, *Scheduling a System of Affine Recurrence Equations onto a Systolic Array*, Tech. Rep. TRCS87-19, University of California at Santa Barbara, Comput. Sci. Dept., Santa Barbara, CA, February 1988.

21. Y. Yaacoby and P.R. Cappello, "Scheduling a System of Nonsingular Affine Recurrence Equations onto a processor Array," *Journal of VLSI signal Processing*, vol. 1, pp. 115-125, 1989.