1989

Systolic Array Implementations With Reduced Compute Time.

Abdulkader Omar Barbir
Louisiana State University and Agricultural & Mechanical College

Order Number 9025290

Systolic array implementations with reduced compute time

Barbir, Abdulkader Omar, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1989

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

SYSTOLIC ARRAY IMPLEMENTATIONS WITH REDUCED COMPUTE TIME

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agriculture and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by

Abdulkader Omar Barbir
B.S.E.E. Louisiana State University 1982
M.S.E.E. Louisiana State University 1983

December 1989

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Abdulkader O. Barbir

Major Field: Electrical Engineering

Title of Dissertation: Systolic Array Implementations With Reduced Compute Time

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

Dec 1, 1989

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Dr. J. L. Aravena, my advisor, for suggesting the topic of this research. He has worked closely with me. His support and encouragement have proven to be an inexhaustible source of ideas. I would like to thank him for providing a firm mooring while still allowing me the freedom to explore. His vast knowledge and incisive questions have been vital to this piece of work.

I am thankful to Dr. A. El-Amawy for being a good friend and for his generous support without which I would have been unable to come to Louisiana State University. A special note of appreciation is extended to Dr. S. Kak for his invaluable help and advice. I would also like to thank the rest of my Committee members, Dr. B. Jones , Dr. J. Trahan and Dr. W. Waggenspack for taking the time to read this dissertation. A special note of thanks is extended to Dr. W. Porter for his constructive ideas and to Dr. A. Marshak for his encouragement.

Finally, I wish to express my sincere appreciation to my parents and my family for their love, support, encouragement and above all patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The goal of the research is the establishment of a formal methodology to develop computational structures more suitable for the changing nature of real-time signal processing and control applications. A major effort is devoted to the following question: *Given a systolic array designed to execute a particular algorithm, what other algorithms can be executed on the same array?* One approach for answering this question is based on a general model of array operations using graph-theoretic techniques. As a result, a systematic procedure is introduced that models array operations as a function of the compute cycle.

As a consequence of the analysis, the dissertation develops the concept of *fast algorithm realizations*. This concept characterizes specific realizations that can be evaluated in a reduced number of cycles. It restricts the operations to remain in the same class but with reduced execution time. The concept takes advantage of the data dependencies of the algorithm at hand. This feature allows the modification of existing structures by reordering the input data. Applications of the principle allows optimum time band and triangular matrix product on arrays designed for dense matrices.

A second approach for analyzing the families of algorithms implementable in an array, is based on the concept of array *time constrained operation*. The principle uses the number of compute cycle as an additional degree of freedom to expand the class of transformations

generated by a single array. A mathematical approach, based on concepts from multilinear algebra, is introduced to model the recursive transformations implemented in linear arrays at each compute cycle. The proposed representation is general enough to encompass a large class of signal processing and control applications. A complete analytical model of the linear maps implementable by the array at each compute cycle is developed.

The proposed methodology results in arrays that are more adaptable to the changing nature of operations. Lessons learned from analyzing existing arrays are used to design smart arrays for special algorithm realizations. Applications of the methodology include the design of flexible time structures and the ability to decompose a full size array into sub-arrays implementing smaller size problems.

CHAPTER 1

INTRODUCTION

1.1. The Systolic Arrays

The increasing demand for highly efficient computing structures for real time applications has prompted researchers to consider alternatives to the relatively slow general purpose computer. Recent advances in VLSI technology have made it feasible to build low cost specialized architectures for compute intensive applications. The concept of systolic arrays introduced by H. T. Kung and colleagues [1]-[4] has stimulated a great deal of interest in the scientific community as an attractive option for obtaining efficient and fast structures for a class of compute bound algorithms.

Systolic arrays have been proposed for a wide range of applications [5]-[39]. Examples include arrays designed for arithmetic functions such as computing the greatest common divisor, arrays designed for matrix computation, such as dense / band matrix-matrix and matrix-vector multiplications. One area, in particular that has benefited from the speed improvement offered by systolic arrays is digital signal processing and control applications. The recurrences describing these applications have repeated structures that make them good candidates for VLSI implementation.

Typically, an array consists of a set of locally connected processing elements. Local interconnection among the elements leads to cheaper implementation and higher densities on wafers. Processing elements in the array are assumed to perform simple operations such as add / multiply and may be of the same type or a mixture of few different types. All processing elements in a systolic array operate in a lock-step fashion synchronized by a global clock. Clock skew in large two-dimensional arrays might cause a slow down in the system speed [40]. However, it has been shown that 1-D linear arrays are not very sensitive to the problem and can be synchronized by the same global clock for relatively large systems [40].

In a systolic array, information flows between processors in a pipelined fashion. Communication with the host computer is performed by boundary elements. Pipelining the information flow in a systolic array solves the ***Von Neumann*** bottleneck problem [41], where the operation rate is limited by the bandwidth of the processor-memory communication link. By pipelining the information flow in the array, once a data element is retrieved from the host memory, it will pass through the entire array, contributing to all the partial results that are dependent on it, before it is returned back to memory. This feature of multiple usage of the same data entry gives rise to high computational throughput. In a typical application, systolic array would be attached as a peripheral device to the host computer that supply the array with input data and receives the processed data from it.

Systolic arrays can have different configurations depending upon the problem being solved. Systolic structures can be one-dimensional (*linear*) or two-dimensional (*planar*). Planar arrays can assume different shapes such as mesh, hexagonal, binary

trees and triangular. Recently Porter and Aravena proposed non-planar structures [11] that offer advantages over their 2-D planar counterparts, for the same type of applications. The improvement is in the form of uniform input data rate to the array, better processing elements utilization and increased throughput.

Due to the popularity of the array concept in the scientific community, several methodologies have been proposed for the automatic mapping of algorithms into VLSI array structures. These methodologies can be divided into three basic categories. The first approach uses the concept of data streams to identify the operations that can be performed in parallel at the algorithm level. Proper operators are then used to obtain the systolic structure. Methods that fall under this category include the work reported by Cohen *et al* [42] , Weiser and Davis [43].

The second approach modifies existing designs to obtain new architectures. The work of Leisersons *et al* [44] is a good representative of this class. The final approach is based on transformations performed at the algorithm model level. In addition, systematic procedures are introduced that describe the steps necessary for performing the transformations. The modified algorithm model is then expressed in a format directly mappable into VLSI arrays [45]-[68]. In this area the work of Barada [59] deserves special attention and is described in chapter 2 , section 2.1, when the notion of a systolic algorithm is introduced.

1.2. Limitations of Systolic Arrays

The available methodologies for array design consider the generic representation of an algorithm. This approach leads to structures that are incapable of adapting to the changing nature of the application. A natural conclusion is that no single array will be suitable over the entire range of the application. Sooner or later a problem will arise that requires modification in the course of the solution.

As a solution to the above problem, several techniques have been proposed in the literature that search for the possibility of increasing the flexibility of the array structures. The work of Snyder [69] suggested the use of the CHIP architecture which is a reconfigurable layout of processors. However, no systematic procedure was discussed for automatically switching the array from one mode of operation to another. Furthermore, the class of implementable algorithms is limited by the available interconnection network.

Annaratone *et al* [70] proposed the programmable systolic chip as an alternative to the problem of designing several arrays for different problems. This approach is confined to problems that can be performed on 1-D arrays, and the class of implementable algorithms is restricted by the available interconnection network. In addition, programming the various cells slows down the speed of computations.

Chuang and He [71] proposed a versatile systolic array for matrix computations. Their work was a generalization of the work proposed by Nash *et al* [72]. The architecture is based on an array system derived by implementing the Faddeeva algorithm. It is composed of a triangular array cascaded with a mesh planar array, which was referred to as the

trapezoidal array. Problems that can be implemented on the array include: **LU** decomposition, inner product, matrix-multiplication and convolution. However, the array suffers from unnecessary delays in extracting the output data elements. The array is sub-optimal and may require several passes before computing the solution.

Several authors [73]-[76] have suggested methods for the automatic reconfiguration of the array, however, their emphasis was focused on the design for testability and fault tolerance. Other methods for increasing the versatility of arrays structures includes the design of size independent and fixed depth arrays. The literature is full of techniques that are proposed for partitioning large size algorithms into problems of smaller dimension, suitable for the available size of the array. The interested reader is referred to the references in [77]-[83] for a comprehensive overview of the topic .

Current thrusts in the design of systolic arrays consider the generic representation of an algorithm [84-85]. The resultant structures execute the indicated operation in a fixed number of compute cycles. No allowance is made for special entries in the data elements that might be omitted and could lead to more efficient implementation [85]. As an example, an array designed to perform dense matrix multiplication will take the same time to compute the product of triangular matrices even though half of the computations are multiplications by zero. In addition, none of the techniques is suitable for applications that may vary in time, such as *time varying* filters.

Real time applications, such as on line signal processing and control, are significant examples of compute intensive application. They have received ample attention in the array community. Specialized architectures for 1-D and 2-D convolutions [17], fast Fourier

transform (FFT), and other transforms [31-34], Kalman filters [35], digital filters and controllers [36], have been published in various professional journals. However, the severe timing constraints of these high performance applications have received very little attention. For these applications, the compute time may be significant and should be deducted from the time available to reach a decision. Failure to do so may render the system useless. Critical real time applications may present an interesting example of the *paralysis by analysis syndrome* [86]. An excessive use of time in computing a response may preclude the possibility of implementing the response.

The efficient use of systolic arrays in real time applications, requires the capability of adjusting the arrays to different realizations. Alternatively, it will be desirable to have arrays capable of recognizing, and using to advantage special realizations of a particular algorithm, that can be executed in a reduced number of cycles. Examples include structures that are capable of omitting multiplications by zero or one.

Signal processing and control problems have characteristics that are not shared by standard numerical algorithms. In particular, different algorithms may be acceptable to the designer. The usual tradeoffs during the design stage are a clear example of the principle. For example, a linear filtering problem may be constrained to banded matrices and still yield acceptable results (FIR filters). A specialized architecture for this implementation displays clear advantages over a general matrix multiplication array [43]. This additional degree of freedom can be used to advantage to improve on the arrays computational speed for specific realizations of an algorithm.

1.3. An Overview of Our Approach

The foundation of our approach can be traced to the work done by Aravena [87-88] that introduced the concept of array constrained design with applications to digital filters. The work is motivated by the observation that the compute time and the computing device determine the class of algorithms that can be used in a given application. The research views the array operation as a transformator parameterized by the compute cycle number. The array generates a collection of transformations on the processed data elements as time progresses. The class of algorithms implementable on a single array is determined during the course of evolution of the partial results in the array.

The first step for analyzing array operations as a function of the compute cycle is to establish a proper model of the phenomena of data flow in an array. In this dissertation, an array is viewed as a directed weighted computing graph, with nodes corresponding to the array processing cells, and the arcs as the medium of transporting the processed data. Based on this model, a general graph theoretic approach is proposed that models the flow of data and the interconnection network in the array. The procedure is flexible enough to encompass a large class of planar, non-planar, as well as switchable arrays.

To express the evolution of partial results in an array as a function of the compute cycle, the *precedence timing diagram* (PTD) is used. The PTD is a modification of the familiar precedence diagram. It is a valuable tool that displays the array operations at each compute cycle. The PTD can be mapped into what is called a *timing graph* (TG). The TG provides a visual aid to the user of the evolution of computations in the array as a function of the compute cycle.

A mathematical approach is then introduced that models the recursive transformations implemented in linear arrays at each compute cycle. In addition, a constructive graph theoretic approach is discussed that analyzes the array TG, that leads to the determination of the collection of algorithms implementable on a single array. Novel features of the methodology include the ability of taking advantage of the nature of data dependencies of the algorithm at hand. Based on this observation, data reordering is proposed to speed up execution of algorithms characterized by random sparsity in their matrices on a given array.

1.4. Outline of the Dissertation

The dissertation is organized as follows: chapter 2 describes the basic characteristics in the mathematical description of a systolic algorithm. It provides techniques for recognizing such an algorithm, and its properties. Chapter 3 develops a formal definition and model of a systolic array based on a graph theoretic approach. The array is viewed as a directed weighted computing graph, with nodes corresponding to the array processing elements, and the arcs as the medium of transporting the processed data. In addition, the chapter introduces to the reader the systolic precedence timing diagram (PTD), which is a valuable tool that expresses the state of an array at a given time. A procedure is developed that maps the PTD into a timing graph (TG). The timing graph representation of an array operation offers a visual aid to the user of the evolution of partial results in the array as a function of time. Proper analysis of the TG allows the determination of all possible fast realizations executable on the array in a reduced number of cycles.

Chapter 4 introduces a new general graph theoretic approach that can be used to analyze the timing graph representation of the array. The significance of the methodology to the subject of modification of array operations can be summarized as follows:

1- It can be used to determine the collection of all possible fast forms implementable on a single array.

2- It can be used to determine the best results of array constrained design.

3- It uses to advantage the nature of algorithm data dependence vectors to allow reordering of array operations to achieve fast realization.

4- It allows the design of what are called smart structures (flexible time varying arrays), as a means of increasing the class of implementable algorithms on a single array.

Chapter 5 develops a mathematical model of the evolution of partial results in arrays implementing a class of linear algorithms that can be expressed in the form of matrix-matrix or matrix-vector operations. The concept of array constrained design is formalized. Necessary and sufficient conditions for the existence of fast realizations for bilinear algorithms are established. The methodology is also extended to arrays implementing the triple matrix product algorithm.

Chapter 6 concludes with a brief summary of the results of the research and some suggestions for future work in related topics.

CHAPTER 2

SYSTOLIC ALGORITHMS

Conventional *high level language* (HLL) description of an algorithm is, in general, not useful for the direct design of systolic architectures. This description must be modified to make efficient use of concurrences and to satisfy technological constraints of the implementation. An algorithm that has been transformed to a form directly mappable into a systolic array will be called a *systolic algorithm.*

This chapter describes the basic characteristics in the mathematical description of a *systolic algorithm.* The objective is to provide techniques to recognize a systolic algorithm and use its basic properties. The chapter is divided into four sections. Section 2.1 outlines a general high level description of algorithms suitable for VLSI implementation. Section 2.2 discusses data dependencies and pipelining. Section 2.3 gives several examples of systolic description of some of the most common algorithms. Section 2.4 discusses handling of input data to an array.

2.1. Algorithm Description

An algorithm can be viewed as a structured set of computations which operate on a set of input variables (and / or a set of variables generated as a result of other operations) to produce a set of output variables. A large class of algorithms that are suitable for systolic array implementation can be represented in a HLL as a set of nested loops characterized by a set of assignment statements and possibly some conditional branch statements. The following HLL description of an algorithm has been considered in [59] :

$$
\begin{array}{l}
\text{for } i_1 = l_1 \text{ to } u_1 \text{ do} \\
\text{for } i_2 = l_2 \text{ to } u_2 \text{ do} \\
\quad \ldots \\
\text{for } i_m = l_m \text{ to } u_m \text{ do} \\
\quad \ldots \\
\text{if (condition (j))} \\
\quad S_1{}^1 \\
\quad \ldots \\
\quad S_1{}^{k_1} \\
\text{elseif (condition (j))} \\
\quad S_2{}^1 \\
\quad \ldots \\
\quad S_2{}^{k_2} \\
\quad \ldots \\
\text{elseif (condition (j))} \\
\quad S_q{}^1 \\
\quad \ldots \\
\quad S_q{}^{k_q} \\
\quad \ldots
\end{array}
\qquad (1)
$$

where,

- m is the depth (number of nested loops) of (l).

- $\mathbf{j} = (i_1 , \ldots , i_m)$ is an integer m-tuple. The range of all loops specifies its domain, J^m.

- l_i and u_i , $i = 1 , \ldots , m$ are integer valued linear expressions of outer indices, representing the initial and final values of the indices of each loop. $S_j{}^k$, $k = 1 , \ldots , k_j$, $j = 1, \ldots , q$, are assignment statement of the form

$$a_k (y_j (\mathbf{j})) = F_{jk} [a_1 (y_1 (\mathbf{j})) , \ldots , a_1 (y_m (\mathbf{j})) , \ldots , a_k (y_n (\mathbf{j})) , \ldots , (a_k (y_p (\mathbf{j})) , \ldots , b_1 (c_1 (\mathbf{j})) , \ldots , b_1 (c_h (\mathbf{j})) , \ldots , b_x (c_r (\mathbf{j})) , \ldots , b_x (c_u (\mathbf{j}))] ,$$

where,

- $y_s (\mathbf{j})$, $c_e (\mathbf{j})$ are integer valued linear expressions of $\mathbf{j} \in J^m$, $a_q (y_s (\mathbf{j}))$ and $b_r (c_e (\mathbf{j}))$ are variables described by the indicated integer expressions, and

- F_{jk} is an algebraic (*memoryless*) function that involves operations such as multiplication, division, and addition.

On both sides of the assignment statements there are some input and some output variables specified by the algorithm requirements. A variable is defined as *input* if it appears on the right hand side of the assignment statement, and is defined as *output* if it appears on the left hand side of the assignment statement. A variable can be both an input variable and an output variable, meaning that it may appear on both sides of the assignment statements. To differentiate between the states of the variables, a variable is defined as *used* if it appears on the right hand side of the assignment statement, and is *generated* if it appears on the left hand side of the assignment statement.

As an example, variable $a_q(y_s(\mathbf{j}))$ in assignment statement S_j^k appears on both sides of the assignment statement described by two different linear expressions of $\mathbf{j}$. Thus, in the right hand side of the assignment statement it is an input variable *used*, and on the left hand side of the statement it is an output variable *generated*. On the other hand, variable b_r described by $(c_e(\mathbf{j}))$ appears on the right hand side of the assignment statement only. In this case, this variable is only an input variable (*used*).

A sequential ordered execution of the loops defines a lexicographical ordering of the index space, J^m in loop (I). This is an induced ordering imposed by the way the loops are arranged. This ordering is not unique and may be changed by rearranging the order of the loops.

2.2. Data dependency and Pipelining

Between variables generated at different index points in loop (I), there are some dependencies that describe the algorithm's internal structure and dictate its communication

requirements. These dependencies are described in what is known as *data dependence* vectors.

A formal definition of data dependence vectors can be found in [59]. For the sake of clarity, a brief description is provided here. For this, let X and Y be two variables whose indices are described by two integer functions f_1 and f_2 respectively. Furthermore, let the variable $X(f_1(\mathbf{j}))$ be generated in statement S_i^k and variable $Y(f_2(\mathbf{j}))$ be generated in statement S_j^k. Let $\mathbf{j}_1$, $\mathbf{j}_2$ ε J^m , be two index tuples that belong to the algorithm domain. In this case, variable $Y(f_2(\mathbf{j}_2))$ is said to be dependent on variable $X(f_1(\mathbf{j}_1))$ if and only if the following conditions hold:

1- $\mathbf{j}_1$ is less than $\mathbf{j}_2$ in the lexicographical sense, and

2- $f_1(\mathbf{j}_1) = f_2(\mathbf{j}_2)$.

The data dependence vector is then defined as the difference between the two tuples so that

$$\mathbf{d} = \mathbf{j}_2 - \mathbf{j}_1 .$$

There are several types of data dependence vectors. A data dependence vector is defined as *fixed* if all its components are constant, otherwise, it is defined as *variable*. A data dependence vector is defined as *flexible* if it allows a computation performed at index point $\mathbf{j}_i$ to precede a computation performed at index point $\mathbf{j}_j$, for all $\mathbf{j}_i$, $\mathbf{j}_j$ ε J^m , where $\mathbf{j}_i > \mathbf{j}_j$ in the lexicographical order of the index space. Otherwise, the vector will be defined as *inflexible*.

In some instances, the algorithm data dependence vectors need to be modified to avoid broadcasting of data variables. For an algorithm with dimension m (depth of the loops in loop (I)), broadcasting typically exists if the dimension of the tuple describing a used or generated variable is less than m. Variable *pipelining* is a technique that is used to eliminate all possible data broadcasts which may exist in the original algorithm [59]. This step is very essential for systolic arrays, since locality of communication among the PEs is always required [1-4]. To pipeline the variable, artificial statements are introduced to complete the dimension of the variable describing tuple. According to [59], an algorithm can be pipelined by implementing the following steps:

1- Find the dimension m of the algorithm.

2- Find the dimension, n , of the tuple describing the variable that need to be pipelined.

3- Equate each of the tuple coordinates to a constant to obtain a system of equations.

4- If the system of equations describes a straight line in the m-D space, the new tuple describing the variable will be defined by that line.

5- If the system of equations does not define a line, break it into sub-systems such that each describes a line. The tuple is replaced by the set of new tuples described by the set of equations defining the lines.

6- Repeat steps *2 - 5* for all variables with $n < m$.

For some algorithms, the components of the tuples describing their variables are a function of the algorithm size. These algorithms will result in variable data dependence vectors. In [59] it was shown that a data dependence with l variables can be replaced by a set of $2l+1$ fixed dependence vectors. This leads to the regularization of data flow in the algorithm.

An algorithm described in HLL will be suitable for systolic array implementation if its variables are pipelined and has all of its data dependencies transformed into a set of fixed data dependence vectors. In this case, the algorithm is referred to as a *systolic algorithm.* The previous observations are summarized below.

Lemma 2.1: An algorithm written in HLL is systolic if

i) The dimension of the tuples describing each generated or used variable in the assignment statements is equal to the algorithm dimension.

ii) The data dependence vectors are fixed.

2.3. Examples of Systolic Algorithms

This section presents several common algorithms and their corresponding systolic representation.

2.3.1. Matrix-Vector Multiplication

Consider the operation $\mathbf{y} = \mathbf{Ax}$, $\mathbf{x}$, $\mathbf{y}$ are $nx1$ vectors, and $\mathbf{A}$ is an nxn matrix. The i th component of $\mathbf{y}$ can be found by using the recurrence

$$y_i^k = y_i^{k-1} + a_{ik}\, x_k \quad , \; k = 1 , \; \ldots \; , \; n .$$

In this recurrence, the *used* variable x_k has the index i missing. To pipeline the variable, an artificial new statement $x_k^i = x_k^{i-1}$ is introduced into the recurrence to eliminate the need of broadcasting x_k. The new form of the pipelined recurrence will be

```
for i = 1 to n do
  for k = 1 to n do
      begin
  S1: x(i,k) = x(i-1,k)
  S2: y(i,k) = y(i,k-1) + a(i,k) x(i,k)
      end.
```

In the HLL program, there are two distinct pairs of $<$*generated*, *used*$>$ variables. One involving x , and the other involving y. Let $\mathbf{d}_x$ be the dependence vector corresponding to the pair $< x(i,k) \; , \; x(i\text{-}1, k) >$ and $\mathbf{d}_y$ be the dependence vector corresponding to the pair $< y(i,k) \; , y(i, k\text{-}1) >$. The data dependence vectors can be found by subtracting the coordinates of the $<$*generated*, *used*$>$ pair describing tuples, in the same order as they appear in the HLL nested loops. Therefore, the two data dependence vectors are given by

$$\mathbf{d}_x = (i - (i-1), k - k) = (1, 0) \quad ,$$

$$\mathbf{d}_y = (i-i, k-(k-1)) = (0, 1).$$

The components of $\mathbf{d}_x$ and $\mathbf{d}_y$ are constant. The order of execution of the index points in HLL can be changed without affecting the overall final values. Hence, the resulting data dependence vectors of the algorithm are *fixed* and *flexible*. The resulting statements defines a systolic algorithm for matrix-vector multiplication.

2.3.2. Matrix-Matrix Multiplication

Consider the problem of multiplying two dense matrices $\mathbf{A} = [\ a_{ij}\ ]$, $\mathbf{X} = [\ x_{ij}\ ]$, to get a matrix $\mathbf{Y} = [\ y_{ij}\ ]$, where $\mathbf{A}$, $\mathbf{X}$ and $\mathbf{Y}$ are $n \times n$ matrices. A typical element of $\mathbf{Y}$, y_{ij} can be found by using the following recurrence

$$y_{ij}^{k} = y_{ij}^{k-1} + a_{ik}\ x_{kj} \quad .$$

In this recurrence, the two variables a_{ik} and x_{jk} have missing indices. The index j is missing in a_{ik} and the index i is missing from x_{jk}. To pipeline the algorithm, two new statements have to be introduced to compensate for the missing indices in the recurrence. The resulting pipelined version of the algorithm can be expressed in HLL as

```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do begin
      S1: a (i,j,k) = a(i,j-1,k)
      S2: x (i,j,k) = x (i-1,j,k)
      S3: y (i,j,k) = y (i,j,k-1) + a (i,j,k) x(i,j,k)
      end.
```

In these assignment statements, there are three distinct pairs of $< generated, used >$ variables. Mainly, $< a(i,j,k), a(i,j\text{-}1,k) >$, $< x(i,j,k), x(i\text{-}1,j,k) >$ and $< y(i,j,k), y(i,j,k\text{-}1) >$. The data dependence vectors are found by subtracting the coordinates of the $< generated, used >$ pair describing tuples, in the same order as they appear in the loops. The data dependence vectors are given below

$$\mathbf{d}_a = (i-i, j-(j-1), k-k) = (0,1,0),$$

$$\mathbf{d}_x = (i-(i-1), j-j, k-k) = (1,0,0),$$

$$\mathbf{d}_y = (i-i, j-j, k-(k-1)) = (0,0,1).$$

The components of $\mathbf{d}_a$, $\mathbf{d}_x$ and $\mathbf{d}_y$ are constant. The order of the execution of the index points can be changed without affecting overall values. Hence, the resulting data dependence vectors of the algorithm are *fixed* and *flexible*. The statements define a systolic algorithm for the matrix-matrix multiplication.

2.3.3. LU Decomposition

Consider the problem of factoring an nxn matrix, as the product of a lower triangular matrix $\mathbf{L} = [l_{ij}]$ and an upper triangular matrix $\mathbf{U} = [u_{ij}]$. The entries of the two matrices lij and u_{ij} can be found using the following recurrence based on Dolittle's method

$$a_{ij}^{k} = a_{ij}^{k-1} - l_{ik}\, u_{kj},$$

$$l_{ik} = \begin{cases} 0 & , \quad i < k \\ 1 & , \quad i = k \\ a_{ik}^{k}\, u_{kk}^{-1} & , \quad i > k \end{cases}$$

$$u_{kj} = \begin{cases} 1 & , \quad i > k \\ a_{ij}^{k} & , \quad i \leq k \, . \end{cases}$$

The pipelined version of the LU decomposition algorithm is given by

```
for k = 1 to n-1 do
 for i = k to n do
  for j = k to n do
   begin
    if ( i = j = k ) then
      S1: u(k,i,j) = 1 / a(k,i,j)
    elseif ( i = k ) then
      S2: u(k,i,j) = a(k,i,j)
    elseif ( j = k ) then
      S3: l ( k,i,j) = a( k,i,j) * u(k,i,j)
    else
      begin
     S4: u(k,i,j) = u(k,i-1,j)
     S5: l(k,i,j) = l(k,i,j-1)
     S6: a(k,i,j) = a(k-1,i,j) - l(k,i,j-1) u(k,i-1,j)
      end
   end
```

There are three $< generated , used >$ pairs, for the variables a , l and u. The data dependence vectors of the representation are fixed and given by

$$\mathbf{d}_l = (i-i, j-j, k-(k-1)) = (0,0,1) ,$$

$$\mathbf{d}_u = (i-i, j-(j-1), k-k) = (0,1,0) ,$$

$$\mathbf{d}_a = (i-(i-1), j-j, k-k) = (1,0,0) .$$

The resulting data dependence vectors are *fixed* and *inflexible.*

2.3.4. QR Decomposition Algorithm

Given an nxn matrix $\mathbf{A}$, this algorithm finds an nxn orthogonal matrix $\mathbf{Q}$, such that $\mathbf{QA} = \mathbf{R}$, an upper-triangular matrix. The matrix $\mathbf{Q}$ is formed as the product of plane rotations. Each plane rotation $\mathbf{Q}_{ij}$, with $i < j$, is an orthogonal matrix with all its elements identical to those of the unit matrix except that $q_{ii} = q_{jj} = c$ and $q_{ij} = -q_{ji} = s$, where

$$d = a_{ii}^2 + a_{ji}^2 ,$$
$$c = a_{ii} / \sqrt{d} , \quad \text{and}$$
$$s = a_{ji} / \sqrt{d} .$$

$\mathbf{Q}_{ij}$ will eliminate a_{ji} below the diagonal. It will only affect rows i, and j. The modified rows will be given by

$$a_{ik}' = c\, a_{ik} + s\, a_{jk}$$
$$a_{jk}' = c\, a_{ik} - s\, a_{jk}$$

The pipelined HLL description of the algorithm is given by

```
for i = 1 to n-1 do
  for j = i+1 to n do
    for k = i to n do
      begin
        if ( i=k ) then
          c (i,j,k) = a (i,j-1,k) / [ ( a (i,j-1,k) )^2 + ( a (i,j,k) )^2 ]^0.5
          s (i,j,k) = a (i,j,k) / [ ( a (i,j-1,k) )^2 + ( a ( i,j,k ) )^2 ]^0.5
          a (i,j,k) = [ ( a (i,j-1,k) )^2 + ( a (i,j,k) )^2 ]^0.5
          a (i+1,j,k) = 0
        end
        else
        begin
          c (i,j,k) = c ( i,j,k-1 )
          s (i,j,k) = s ( i,j,k-1 )
          a (i,j,k) = c (i,j,k-1) a (i,j-1,k) + s (i,j,k-1) a (i,j,k)
          a (i+1,j,k) = -s (i,j,k-1) a (i,j-1,k) +c (i,j,k-1) a (i,j,k)
        end.
```

The QR decomposition algorithm results in five $<$ *generated* , *used* $>$ pairs. The two variables c and s produce the same data dependence vector. The variable a results in three distinct pairs. The data dependence vectors of the QR systolic algorithm are

$$\mathbf{d}_c = \mathbf{d}_s = (i-i, j-j, k-(k-1)) = (0, 0, 1) \quad ,$$

$$\mathbf{d}_a = (i-i, j-(j-1), k-k) = (0, 1, 0) \quad ,$$

$$\mathbf{d}_a = (i-(i-1), j-j, k-k) = (1, 0, 0) \quad ,$$

$$\mathbf{d}_a = (i-(i-1), j-(j-1), k-k) = (1, 1, 0) \ .$$

All the dependence vectors of the QR factorization algorithm are *fixed* and *inflexible*.

2.4. Handling of Input Data

At each index point **j** in loop (1), a subset from the set of all variables will be involved. The elements can be partitioned into two different groups. The first group corresponds to the *used* (input) data elements, namely those elements appearing on the right hand side of the assignment statements. The second group corresponds to the *generated* output data elements, basically, those elements that appear on the left hand side of the involved assignment statements.

In some cases, a variable will only appear on the right hand side of the assignment statement, hence it is always *used*. A variable that is always used and is never generated will be called a *free* variable. A dummy data dependence vector can be introduced to describe the contribution of a free variable at each point $\mathbf{j} \in J^m$. This will be useful later in the modeling of data flow in an array.

To introduce a dummy data dependence vector to the HLL representation of the algorithm, simply re-index the nested loops in (1). The steps necessary to obtain a dummy data dependence vector are:

1- Find the number of free variables in the algorithm, let this number be u.

2- Increase the depth of the nested loops in the algorithm to $m' = m + u$.

3- For each variable with a tuple dimension $n < m'$ do

i) Equate each of the tuple coordinates to a constant to obtain a system of equations.

ii) If the system of equations describes a straight line in the m'-D space, the new tuple describing the variable will be defined by that line.

iii) If the system of equations does not define a line, break it into sub-systems such that each describes a line. The tuple is replaced by the set of new tuples described by the set of equations defining the lines.

To illustrate the principle, consider again the matrix-vector algorithm. The variable a_{ik} is a free variable. A dummy dependence vector is introduced that results in this HLL form of the algorithm :

```
for i = 1 to n do
   for k = 1 to n do
      for j = k to k do
         S0: a ( i,k,j ) = a ( i,k,j-1 )
         S1: x ( i,k,j ) = x ( i-1,k,j )
         S2: y ( i,k,j ) = y ( i,k-1,j ) + a ( i,k,j ) x ( i,k,j )
          end.
```

The resulting data dependence vectors are given by:

$$\mathbf{d}_a = (i-i,\ k-k,\ j-(j-1)) = (0,0,1) ,$$

$$\mathbf{d}_y = (i-i,\ k-(k-1),\ j-j) = (0,1,0) ,$$

$$\mathbf{d}_x = (i-(i-1),\ k-k,\ j-j) = (1,0,0) .$$

The addition of dummy data dependence vectors to a systolic algorithm ensures the pipelining of all the variables in the HLL language representation of the algorithm. As a result, data elements of each variable will be pipelined along lines specified by the tuple(s) describing each variable. Arrays constructed by this methodology will be characterized by a uniform data flow.

CHAPTER 3

MATHEMATICAL REPRESENTATION OF ARRAY OPERATIONS AS A FUNCTION OF THE COMPUTE CYCLE

Chapter 2 gives a high level language (software description) of a systolic algorithm. This chapter assumes that the algorithm is implemented in a given systolic array. The most significant result of this chapter is the modeling of the computations in an array as functions of the compute cycle. The technique is intended for analysis up to the normal stop; i.e. , for compute cycle $k \leq T$, where T is the total number of cycles needed by the array to produce the results of the computations.

The chapter is organized as follows: Section 3.1 introduces a hardware model of an array. Section 3.2 discusses the notion of space-time transformation implemented by an array on the input data elements. Section 3.3 derives the concept of data paths in the array. Sections 3.4 and 3.5 formalize the modeling of array operations as a computing graph. Section 3.6 introduces a procedure that models the flow of data in an array, while in sections 3.7 and 3.8 the precedence timing diagram and the timing graph are discussed. Section 3.9 models the operations of several examples of switchable as well as non-switchable arrays.

3.1. Array Hardware Model

A systolic array consists of a set of processing elements (PE) operating in a lock-step fashion, and an interconnection network that binds the various PEs together. This section formalizes a model of the array connectivity network based on a graph theoretic approach. This model is independent of the type of the algorithm being evaluated or the functions performed by the PEs.

3.1.1. Modeling Array Connectivity

In order to view a general model of the array connectivity network, it is convenient to depict the array as a directed graph, whose nodes correspond to the array processing elements, and the arcs correspond to the network links. With this representation, techniques from graph theory can be used to analyze the properties of the array.

Definition 3.1: A systolic array hardware model is defined as a *directed graph.* The nodes of the *graph* represent the array processing elements, and the arcs correspond to the links between the processing cells.

Let N be the total number of processing elements in the array. To construct the *graph* representation of the array, the following steps are needed:

Procedure ***Graph***

1- Enumerate the PEs in the array with unique integer numbers that are greater than zero.

2- In the graph, connect node i to node j if and only if their corresponding PEs are connected by a link.

Once a graphic description of an array is determined, its interconnection network can be described by the directed graph's *adjacency* matrix. In graph theory, the *adjacency* matrix of an N - node graph, is defined as long as the graph contains no parallel edges [89].

Definition 3.2: The connectivity network **P** of a systolic array is defined as the *adjacency* matrix of the resulting graph representation of the array. $\mathbf{P} = [\ p_{ij}\ ]$ is an NxN matrix whose element

$$p_{ij} = 1, \quad \text{if there is a directed arc from node } i \text{ to node } j,$$
$$= 0, \quad \text{otherwise}.$$

For non-switchable arrays, the interconnection matrix **P** is fixed. However, for switchable arrays it will be a function of the compute cycle.

3.1.2. Processing Element Model

A typical processing element (PE) in an array consists of the following: a set of input links, a set of output links, and a set of memory registers. In addition, each PE is characterized by a set of *memoryless* functions that define its operation. The set of algebraic functions associated to each processing element defines the arithmetic and logic expressions that the element is capable of performing.

A processing element in the array acts on data that is either stored in its memory registers or received on its input links from neighboring PEs. The results of the operations are delivered by the PE after one unit time delay. The results may be either stored inside the PE or transmitted on a set of output links to neighboring PEs. The element is assumed to have a set of temporary buffer registers to hold partial computations.

For the analysis, it is convenient to assume that all the data available to the processing element on the input links are stored in internal memory registers. Furthermore, it is useful to think of all the PE memory registers as a stack of registers. At the start of the compute cycle, all data elements are stored in proper locations on the stack. The PE performs an algebraic function that may involve all or a subset of the registers. Data that must be transmitted by the PE is then selected from appropriate locations on the stack and sent on the output links.

A typical processing element can be represented as in Figure 3.1.

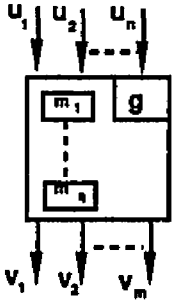

Figure 3.1. PE Model

Referring to Figure 3.1 , g is the identification number of the processing element, n is the number of input links, q is the number of storage registers, and m is the number of output links. For convenience, data elements on the links are represented as vectors ; thus $r\ (g\ ,k)$ is the q -vector of data available in the PE memory registers at the start of the compute cycle , $u\ (g\ ,k)$ is the n -vector of data available on the PE input links at the start of the compute cycle and $v\ (g\ ,k)$ is the m - vector of data that must be transmitted on the output links.

The above notations allows the establishment of a hardware model of a typical processing element irrespective of the algorithm being evaluated. Hence, the processing element hardware model is given by

$$v\ (g\ ,k)\ =\mathrm{F}_{g}(u\ (g\ ,k),r\ (g\ ,k),\ k\ )\ ,$$

where, F_{g} is a memoryless function and k is the compute cycle number.

However, if the array is executing a particular algorithm then the processing element will map a unique *used* element belonging to one variable that is available on an input link, to a unique *generated* element belonging to same variable and transmits it on a particular output link, at a certain compute cycle. This operation will be referred to as the PE mapping function.

Definition 3.3: The PE mapping function is defined by the map $\psi\colon k \rightarrow A$ given by

$$\psi(z\ (k)\ )=\ a\ (j_{k}\ )\ ,$$

where, z represents data elements belonging to algorithm A that are available on the PE input links and memory registers at a unique compute cycle k. The PE will transmit $a\ (j_{k}\ )$ on an output link to a neighboring PE.

3.2. Description of Array Operations

The HLL description of a systolic algorithm introduced in chapter 2, is static, meaning that it does not provide any information about the execution time of the computations in the loops. An ordered execution of a systolic algorithm generates the following 5-tuple

$$A\ (\ J^m\ ,JI\ ,JO\ ,FI\ ,D\ )$$

where,

- J^m is the algorithm domain.
- JI is the set of the algorithm *used* data elements.
- JO is the set of the algorithm *generated* data elements.
- $FI = \{\ f_1\ ,\ f_2\ ,\ \ldots\ ,\ f_r\ \}$; is a set of memoryless functions needed to map the algorithm input data to the algorithm output data; f_j is a memoryless function such as multiply , add, .. etc; r = total number of computation functions needed to map the input data to the output data.
- $\mathbf{D} = \{\ d_1\ ,d_2\ ,\ \ldots\ ,\ d_v\ \}$ is the set of data dependence vectors of the systolic algorithm.

From the description of the algorithm model, the sets defining the tuple can be generated; however, the converse is not necessarily true.

An ordered execution of the HLL description of a systolic algorithm produces a set of *used* elements and a set of *generated* elements at each index point of the algorithm space. A systolic array implementing the algorithm must produce the same sets, with the exception, that the sets are generated at different locations in space and are distributed in time. The array imposes an execution ordering on the algorithm variables characterized by a *time* parameter. This ordering is a *function preserving* transformation, meaning that the following must hold:

i) If $a(j) \in JI, j \in J^m$ is an algorithm *used* element then, it must be an input element to an appropriate PE in the array at a unique compute cycle k.

ii) If $a(j) \in JO, j \in J^m$ is an algorithm *generated* element then, it must be an output element from an appropriate PE in the array at a unique compute cycle k.

iii) To any operation in the original algorithm such as addition, multiplication, etc., an identical operation must be performed by the array processing elements.

iv) Data dependencies of the systolic algorithm are preserved in the array.

3.2.1. Example to Illustrate Algorithm Generated Tuples

Consider the HLL program covered in chapter 2 , for the matrix-vector multiplication algorithm, for the case when $m = n = 2$. The algorithm index space J^2 is given by

$$J^2 = \{ (1,1,1), (1,2,2), (2,1,1), (2,2,2) \}.$$

At index point (1 , 1 , 1) , data element $x(1,1,1)$ is generated in assignment statement *S1* , and is used in statement *S2* . On the other hand, y (1,1,1) is generated in statement *S2* , a (*1,1,1*) is generated in *S0* and used in *S2*. At index point (1 , 2 , 2) , data element $x(1,2,2)$ is generated in statement *S1* , and used in *S2* . Output data element $y(1,2,2)$ is generated in statement *S2* . While, data element $a(1,2,2)$ is used in statement *S2*. Similarly, at index point (2 , 1 , 1), $x(2,1,1)$, is generated in *S1* , y (*2,1,1*) is generated in *S2* , and $a(2,1,1)$ is *used* in statement *S2*. At index point (2 , 2 , 2) , $x(2,2,2)$ is generated in *S1*, y (*2,2,2*) is generated in *S2* , and $a(2,2,2)$ is used in *S2*.

From the example, the set of input data elements to the algorithm is given by

$$\begin{aligned} II = \{ & (a(1,1,0), x(0,1,1), a(1,1,1), x(1,1,1), y(1,0,1)), \\ & (a(1,2,0), x(0,2,2), a(1,2,2), x(1,2,2), y(1,1,2)), \\ & (a(1,1,0), x(1,1,1), a(2,1,1), x(2,1,1), y(2,0,1)), \\ & (a(1,2,1), x(1,2,2), a(2,2,2), x(2,2,2), y(2,1,2)) \}. \end{aligned}$$

Similarly, the set of output data elements of the algorithm is given by

$$JO = \{\ a(1,1,1), x(1,1,1), y(1,1,1), a(1,2,2),$$
$$x(1,2,2), y(1,2,2), a(2,1,1), x(2,1,1),$$
$$y(2,1,1), a(2,2,2), x(2,2,2), y(2,2,2)\ \}.$$

At each index point in assignment statement $S2$, there is a multiplication and an addition involved. Therefore, the set of memoryless functions is composed of

$$FI = \{ (*, +) \}.$$

where, '$*$' indicates multiplication and '$+$' the addition operation.

3.3. Array Data Paths

Let $C = \{ 1, 2, ..., r \}$ be a linear chain of connected PEs in the array. In this chain PE (i) is connected to PE ($i+1$) by a link. The boundary PEs are connected to external sources; i.e., PE (1) receives data from an input source and PE (r) transmits data elements to a receiving source. A linear chain where every data element belong to the same variable defines a *data path* for that variable.

The collection of all linear chains in the array defines the data paths for the variables. In the array, data elements belonging to the same algorithm variable, will follow non-intersecting paths. To distinguish between the paths of elements belonging to different variables in the array, assign a weight w, which is a unique integer number greater than zero, to the path(s) of the elements of each variable in the array. To assign weights to the data paths, the following must be performed:

Procedure *Data Path*

1- Identify the collection of linear chains in the array that transports the data elements of variable $a(.)$.

2- In the chain, identify the input and the output links in each PE that carries the elements of the same variable.

3- Assign the same weight w for those links.

3.4. Modeling Array Operations as a Function of the Compute Cycle

This section formalizes a general model of the array operations as a function of the compute cycle, irrespective of the algorithm being evaluated. In order to view a general model of the array operations as a function of the compute cycle, it is convenient to think of the array as a computing graph [89], with nodes that perform computations on data elements that are transported on the arcs. The arcs of the graph correspond to the array connectivity network.

A computing graph is one that is capable of transporting and processing data elements. The nodes (vertices) in the graph are computational nodes that perform algebraic functions on the data present on the input arcs. The arcs are the means by which data is transmitted between the nodes. The graph starts with a certain state at period k, which is characterized by data associated with all the nodes. In the next period, processed data elements on node i are conveyed to node j on arc (i,j). The graph enters a new state

typified by the new data elements in all the vertices. The process may be repeated again. A *directed computing* graph is a computing graph with arcs that assume a specific direction between the nodes at a particular period. A *weighted directed computing graph* is a directed computing graph with weights assigned to the arcs. This leads to the following definition of a systolic array operation.

Definition 3.4: A systolic array operation as a function of the compute cycle, is defined as a *directed weighted computing graph* (DWCG). The nodes of the DWCG correspond to the processing elements and the weighted directed arcs correspond to the array network links.

The weight of an arc in the DWCG , is a unique integer greater than zero, that is equal to the weight assigned to the data path(s) of the elements of an algorithm variable, in the array. To construct the DWCG representation of an array, the following steps must performed:

Procedure ***DWCG***

1- Enumerate the PEs in the array with unique integer numbers that are greater than zero.

2- Draw the graph of the array with each node representing a processing element. In this graph connect node i to node j if and only if their corresponding PEs are connected by a link. The direction of the arc must be consistent with the direction of data flow from PE i to PE j.

3- Assign weights to the arcs in accordance with the weights assigned to the link's data paths.

During the array operations, its connectivity is defined by the DWCG *weighted adjacency* matrix. The definition of the connectivity matrix is given below.

Definition 3.5: The connectivity matrix **P** of a systolic array during its operation, is defined as the *weighted adjacency* matrix (WAM) of its resulting DWCG, where, **P** is an NxN matrix whose element p_{ij} is given by

$$p_{ij} = u,\ u > 0,\ \text{is the weight of the arc from node } i \text{ to node } j$$
$$= 0,\ \text{otherwise}.$$

For non-switchable arrays, the interconnection matrix **P** is fixed; However, for switchable arrays it will be a function of the compute cycle. In this case, several matrices will be needed to describe the array connectivity at different compute cycles.

3.5. Timing Functions

In the DWCG , a node is a conceptual representation of a systolic processing element. A node can have a set of internal memory registers, but in no event can it be a generator. This means that the node must have inputs in order to generate outputs. The degree of the node is defined as the number of weighted arcs incident at the node. Boundary nodes in the graph are those nodes that receive data from external sources, or transmit data to the outside world. The completion of the degree of the boundary nodes in the graph, requires

the assumption that external data elements are supplied to those nodes by user induced input nodes. Similarly, boundary output nodes are assumed to feed data to user induced receiving nodes. Hence, the DWCG representation of an array can be viewed as a sub-graph of a more general graph that includes the user induced nodes. The complete graph will be referred to as the *array graph* (AG).

In the AG description, it is convenient to think that the user induced input nodes supply their data elements to the array, in the form of a *timing function*. Let $\pi^{k}_{w}(n)$ be the timing function of input boundary node n, that supplies data elements corresponding to a data path of weight w. The timing function is defined by the following map:

Definition 3.6: Let $a(.)$ be a variable of algorithm A, belonging to a data path of weight w. Furthermore, let T be the total number of compute cycles for the array to output the results of the computations. The map π^{k}_{w} given by

$$\pi^{k}_{w} : k \rightarrow A \text{ , such that}$$

$$\pi^{k}_{w}(n) = a(j_k) \text{ , } a(.) \in A, \; k \leq T \text{ ,}$$

defines the timing function of the user induced input node n.

To distinguish between the user induced nodes and the rest of the nodes, in the AG description of the array, separate input (output) weighted tables must be constructed to describe the connectivity of those nodes to the rest of the nodes in the DWCG.

3.6. Modeling of Data Flow in the Array

Introduced in this section is a systematic procedure that can be programmed to model the flow of data in a systolic array. The procedure models the flow of data elements, with no regard to the flow of control signals. The control signals are assumed to be generated by a proper unit. In order to incorporate the change of operation modes in the array PEs , one must impose limits on the compute cycle, and the tuples describing the input data elements. The steps necessary for modeling the flow of data in an array are summarized below :

Procedure Data Flow

1- Let N be the total number of PEs in the array. Enumerate the PEs with unique integer numbers that are greater than zero. This numbering process may be arbitrary, but once a configuration has been decided, it is assumed to be fixed thereafter.

2- Construct the DWCG representation of the array.

3- Complete the graph representation of the array by constructing the AG.

4- Derive the weighted lookup tables for the user induced input and output nodes.

5- Derive the timing functions for each user induced input node.

6- Derive the node (PE) mapping functions , for all nodes in the DWCG

representation of the array. These are the memoryless functions performed by the node at each compute cycle, that relate the inputs to the outputs from the node.

7- Derive the connectivity matrix **P** of the DWCG representation of the array.

8- To model the flow of data in the array, execute the following :

For $k = 0$ to T do in parallel

begin

a) For $n = 1$ to N do in parallel

begin

i) From the weighted lookup I/O tables, check if node n has arc(s) of weight w receiving input from a use induced node. If *yes* , then evaluate the proper timing function for that node.

ii) Use column i of **P** of node n as a lookup table to identify the node originating the input arc to node n incident weighted arc.

iii) Evaluate the originating nodes mapping functions to identify the data elements that node n receives at compute cycle k.

iv) Evaluate node n mapping functions.

end

end.

Remark: The procedure introduced is flexible enough to model the flow of data in dynamically switchable arrays. In these arrays, the following may be a function of the compute cycle: i) the connectivity matrix, ii) the node mapping functions, and , iii) the input timing functions. Procedure *data flow* can be used to model switchable array operations with some minor modifications. The changes required are summarized below:

1- Define a lookup function *M(k)* that identifies the node that changes its mode of operations at compute cycle *k*. *M(k)* must indicate the mode of operation of node n at a particular compute cycle.

2- Derive the different connectivity matrices $P(k)$ for the total allowable number of array operations.

3- Derive the input timing functions for the user induced input nodes, in this case $\pi_w^k(n)$ could be a function of the compute cycle.

4- In procedure data flow, before evaluating any of the steps (*8* i) to *8* iv)) , use the proper tables for that particular compute cycle.

3.7. Precedence Timing Diagram

The operations of the PEs in the array can be described by the *precedence timing diagram* (PTD). The PTD is an adaptation of the familiar precedence diagram [90]. The PTD is a timing diagram that consists of a total of T time levels, where T is the allowable number of compute cycles. Each time level in the PTD consists of a number of nodes. A node in the PTD represents the state of its corresponding PE at that compute cycle. The state of a PE includes all the data elements stored in its stack of registers. Only active nodes will be represented at each level in the PTD , i.e. those nodes in the DWCG that change state for that specific cycle.

In the PTD each node is assigned a number that is equal to the number of the node it relates to in the DWCG. Furthermore, since each node in the PTD represents the state of a node in the DWCG, all the information needed to define the state of that node must be included in the representation. The PTD must also provide information about the flow of data elements in the DWCG. To incorporate that, the arcs of the nodes are assigned weights. The weight of an arc incident to a node is equal to the node number it originated from. Similarly, the weight of an arc originating from a node is equal to the number of the destination node. The weights are found by using the weighted connectivity matrix of the AG as a lookup table.

3.7.1. A Procedure for Constructing the PTD

Given the array DWCG , which includes its timing functions, the nodes mapping functions, the connectivity matrix $\mathbf{P}$, and the allowable number of compute cycles T ,

the PTD of the array can be constructed. The following are general rules that a PTD should adhere to

i) Each node in the PTD represents the state of a DWCG node for a particular compute cycle.

ii) Each node in the PTD is assigned an identification number that is equal to the node number to which in the DWCG it relates. Each node must consist of input and output arcs in accordance with the prospective DWCG node. All data available to the nodes are stored in internal registers.

iii) Input and output arcs to each node are assigned weights that are function of the compute cycle. The weight of the node's incident arc at cycle k, is equal to the number of the node from which it originated. Similarly, the weight of the node's output arc at cycle k, is equal to the number of the node on which it is incident.

Using the concepts developed in procedure *data flow*, the necessary steps for constructing the PTD are listed below:

Procedure *PTD*

1) Use the I/O lookup tables from procedure data flow to identify the boundary nodes in the DWCG representation of the array.

2) Use procedure *data flow* to identify the timing functions of the input nodes.

3) For $k = 0$ to T , do in parallel

begin

i) Execute step 8 a) of procedure *data flow* for each node in the DWCG representation of the array.

ii) Place a DWCG node in the PTD at time level k if and only if it changes state during that compute cycle.

iii) Assign the arcs of each node the proper weights.

iv) For every node added to the PTD , each internal register must indicate the data element it holds.

end.

4) The construction of the PTD stops at time level T. This indicates that all nodes in the DWCG becomes inactive.

3.8. Timing Graph Representation of Array Operations

The previous section introduced the concept of *precedence timing diagram* . In this section, a method for mapping the PTD into an acyclic *timing graph* (TG) is derived. The approach represents the evolution of the array computations as a connected graph. The representation will prove to be beneficial for the analysis of the array operations that is pursued in the next chapter.

The *timing graph* represents the states of the DWCG of an array as a function of the compute cycle. The graph displays the evolution of partial results in the array as time progresses. The TG will consist of a total of T time levels. Each time level depicts the change in the state of a DWCG node, and the direction of data movement from (into) the node for that compute cycle. Given the PTD of an array, the TG can be easily constructed.

The mapping of the PTD to the TG should retain the structure of the array operations, its connectivity and the integrity of the flow of data elements between the PEs. In the TG , a time axis must be defined that connects the state of each node in the PTD at different time levels. This axis will be referred to as the node direction vector and for node n is denoted by $\mathbf{d}_n$. Furthermore, all data elements a (.) that belong to the same algorithm variable , follows lines that are parallel to a direction vector denoted by $\mathbf{V}_a$. Let q be the total number of data paths in the DWCG, then there must be q direction vectors in the TG. The direction vectors can be assigned by the user at his own convenience. The mapping of the PTD into TG consists of the following steps:

1) Construct the set $W = \{W_i\}$, $i = 1, \ldots, T$, here , W_i represents the the nodes in the PTD at time level i .

2) From the I / O lookup tables in procedure data flow, identify the input / output nodes in each set W_i .

3) Define the direction vector $\mathbf{V}_a$ for each of the direction functions describing the

flow of data elements between the PTD nodes. All elements belonging to the same data path, move on lines that are parallel to that direction vector. Furthermore, let $V = \{ V_1, \ldots, V_q \}$ be the set of all direction vectors.

4) For each node $j \in N$ (number of nodes in the DWCG), define a direction vector $\mathbf{d}_j$. Here, $\mathbf{d}_j$ will connect node j at time levels i and $i+1$ if $j \in W_i$ and W_{i+1}. This will be the time axis for the node in the graph. All time axes for the nodes must be parallel to each other.

In what follows, a procedure is presented that constructs the TG from the PTD.

Procedure *TG*

1) For the boundary input / output nodes, assign a unique direction vector V_w for each weighted input link.

2) Place the nodes of W_0 at time level 0 in the TG, with the input edges to each node on lines that are parallel to the corresponding weighted direction vector V_w.

3) For any node n in W_0, derive the node direction vector $\mathbf{d}_n$.

4) For $k = 1$ to T do in parallel

   i) If node $i \in W_k$ has a boundary input / output edge, connect the weighted

edge to the node on a line that is parallel to $\mathbf{V}_w$.

ii) If node $j \in W_{k-1}$ and W_k, place node j in time level k on the line defined by the direction vector $\mathbf{d}_n$, else, define a direction vector that is parallel to $\mathbf{d}_n$ for the node.

iii) Choose a specific direction vector $\mathbf{V}_a$ from the set V.

iv) Connect node $i \in W_{k-1}$ to node $j \in W_k$ on a line that is parallel to $\mathbf{V}_a$, if the nodes have arcs holding elements of variable a (.). If this is not possible, then

a) For $kk = 0, k$ do

b) Renumber the nodes in each time level kk such that if node $i \in W_{kk}$ then it is connected to node $j \in W_{kk+1}$ on the line that is parallel to $\mathbf{V}_a$.

5) For $l = 1, q$ do

a) For $k = 1, T\text{-}1$ do in parallel

b) Connect node $i \in W_k$ to node $j \in W_{k+1}$ on line that is parallel to $\mathbf{V}_l$.

6) Connect the nodes at time level T to the user induced output nodes on edges

that are parallel to the vector V_w , where w is the weight of the output arc connecting the output boundary nodes in the DWCG to the user induced output nodes.

3.9. Examples to Illustrate the Introduced Procedures

This section applies the introduced procedures to model the operations of several arrays as a function of the compute cycle. Examples include switchable as well as non-swithchable arrays.

3.9.1. Matrix-Vector Array

The example considers an array designed to perform matrix-vector multiplication of the form $y = Ax$, where $A \in R^{nxn}$, x and $y \in R^n$. The array for $n = 3$ is depicted in Figure 3.2. The array total execution time is $(3n - 1)$ cycles in general.

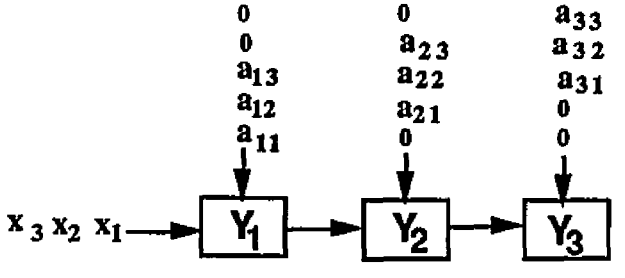

Figure 3.2. Matrix-Vector Array, $n = 3$.

From procedure *data path* , there are three data paths in the array corresponding to the elements of the variables a , x , and y . Assign the weight $w = 1$ for x_{ik} data elements, the weight $w = 2$ for the a_{ij} elements, and the weight $w = 3$ for the y_{ik} elements. The resulting data paths are given in Figure 3.3.

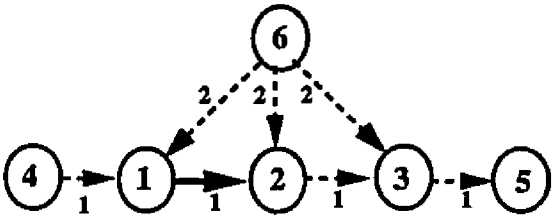

Figure 3.3. AG Matrix-Vector Array.

node / node	1	2	3
4	1	0	0
6	2	2	2

Table 3.1. Input Nodes for Matrix-Vector Array.

node / node	5
3	1

Table 3.2. Output Nodes for Matrix-Vector Array.

From procedure DWCG , there are three PEs in the array, that are assigned the numbers { *1, 2, 3* } . The resulting DWCG and AG of the array are shown in Figure 3.3. From step 4, of procedure *data flow*, the weighted I/O lookup tables are given in tables 3.1 and 3.2 respectively. Utilizing step 5 of the procedure , the boundary nodes in the DWCG are nodes { *1 , 2 , 3* }. The input timing functions for the nodes are listed below

$$\pi_1^k (1) = x (1, k+1) , 0 < j < 4,$$

$$\pi_2^k (1) = a (0, k+1) , 0 < j < 4,$$

$$\pi_2^k (2) = a (2, k) , 0 < j < 4 ,$$

$$\pi_2^k (3) = a (3, k-1) , 0 < j < 4 .$$

From step 6, of the procedure, the resulting node mapping functions are given by

$$v_1^k(n) = u_1^k(n)$$
$$m_3^{k+1}(n) = u_1^k(n) * u_2^k(n) + m_3^k(n).$$

From step 7 of the procedure, the resulting connectivity matrix of the array is given in table 3.3.

node / node	1	2	3
1	0	1	0
2	0	0	1
3	0	0	0

Table 3.3.

Connectivity Matrix for Matrix-Vector Array.

Utilizing procedure *PTD* , the array total execution time is *5* cycles , thus the final value of T is *4* . The range of allowable number of compute cycles is $0 \leq k \leq 4$. From step 3 of the procedure , at $k = 0$ the mapping functions for node $n = 1$ are only defined, thus, it is the only node in the graph that changes its state. There are two data elements on the node that are given by

$$\pi_2^0\ (1) = a\ (1,1)\ = a_{11}\ ,\ \pi_1^0\ (1) = x(0,1)\ = x_1\ .$$

From the node mapping functions the content of the internal register of the node is given by

$$m_3^0\ \ (1)\ = y_1^1\ .$$

At compute cycle 1, nodes 1 and 2 change their states. For node 1 one can write

$$\pi_1^1\ (1) = x_2\ \ ,\ \pi_2^1\ (1) = a_{12}\ .$$

The content of the internal register of node 1 is given by

$$m_3^1\ \ (1)\ = y_1^2\ ,$$

and for node 2,

$$\pi_2^1\ (2\ ) = a_{21}\ \ ,u_1^1\ (2\ )\ x_1\ .$$

The content of the internal register of node 2 is given by

$$m_3^1\ \ (2)\ = y_2^1\ \ .$$

In a similar manner, the rest of the time levels in the PTD can be constructed The final result is depicted in Figure 3.4.

Utilizing step 1 of procedure TG, the nodes $\{1, 2, 3\}$ are boundary input nodes, that receive data elements on weighted input link $w = 2$ carrying $a(.)$. Let the direction vector $\mathbf{V}_a$ be the direction vector of this data path.

In addition, node 1 is a boundary node that receives x_{ik} elements on input link of weight $w = 1$, that is assigned the direction vector $\mathbf{V}_x$. From step 2, reading time level zero in the PTD produces the set $W_0 = \{1\}$, hence, node 1 is placed at level $k = 0$ in the TG. The input data element a_{11} is placed on an edge that is parallel to $\mathbf{V}_a$ and the input data element x_1 is placed on an edge that is parallel to $\mathbf{V}_x$. From step 3, let $\mathbf{d}_1$ be the node direction vector.

The PTD at time level 1 has two nodes, hence, $W_1 = \{1, 2\}$. From step 4, choose $\mathbf{V}_x$ as the direction vector. Node 1 is placed in time level 1 in the TG, on the line defined by $\mathbf{d}_1$. Node 2 at level 1 receives x_1 from node 1 at level 0, from step 4, the input edge holding x_1 must be parallel to $\mathbf{V}_x$. To meet this condition, node 2 must be placed on the right of node 1 in time level 1.

Utilizing step 5 of the procedure, every node at each level receives an input data element belonging to the variable $a(.)$, thus, all the edges must be parallel to $\mathbf{V}_a$. From step 6, node 3 at time level 4 has an edge that is incident on a user induced output node. This edge must be parallel to $\mathbf{V}_x$, the resulting TG is depicted in Figure 3.5.

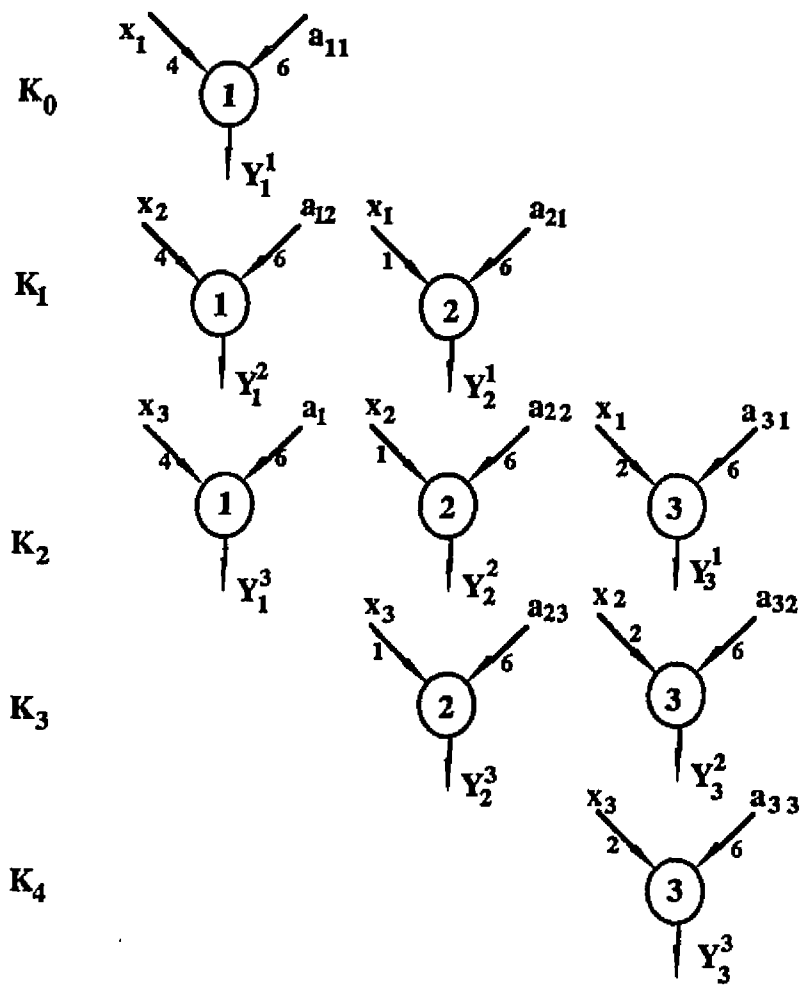

Figure 3.4. PTD Matrix-Vector Array.

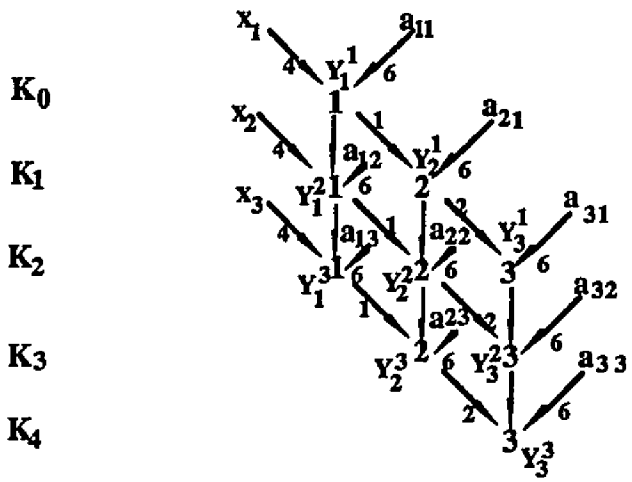

Figure 3.5. TG Matrix-Vector Array.

3.9.2. Matrix Multiplication on Planar Mesh Array

Consider a planar mesh array designed for matrix-matrix multiplication of the form $\mathbf{Y} = \mathbf{AB}$, where $\mathbf{Y}, \mathbf{A}, \mathbf{B}$ are nxn matrices. The array total execution time is $(3n-1)$ in general. The array is depicted in Figure 3.6 for $n=2$.

From procedure *data path*, there are two data paths in the array. The first path consists of the a_{ij} data elements, while the second path consists of the b_{ij} data elements. The data elements corresponding to y_{ij} are accumulated inside the processing elements. Let the weight $w=1$ be assigned to the data paths corresponding to a_{ij} elements and the weight $w=2$ be assigned to the data path corresponding to b_{ij} elements. The weight $w=3$ is assigned to the path of y_{ij} elements. The resulting data paths and their weights are given in Figure 3.7.

From step 1 of procedure DWCG, there are 4 PEs that are assigned the numbers shown in Figure 3.7. Steps 2 and 3, result in the DWCG and AG representation of the array, depicted in Figure 3.7.

From procedure *data flow*, the input nodes in the DWCG are nodes (1, 2, 3), while the output nodes are (2, 3, 4), the weighted lookup tables of the I/O nodes are given in Tables 3.4 and 3.5 respectively.

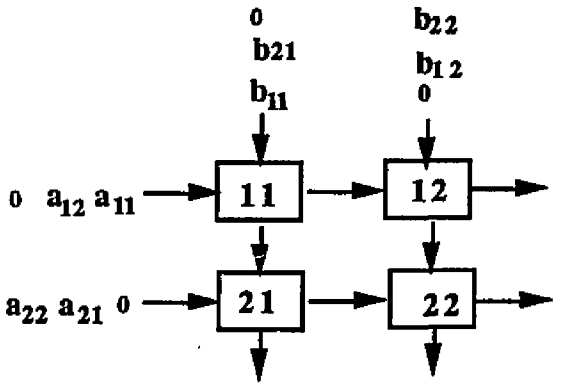

Figure 3.6. Mesh Array, $n = 2$.

node / node	1	2	3
5	1	0	1
8	2	2	0

Table 3.4. Mesh Array Input Nodes.

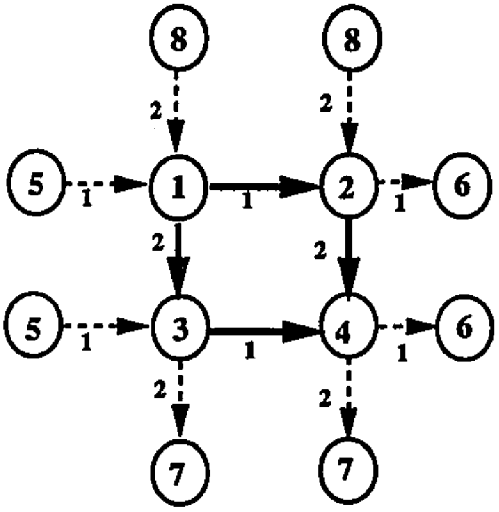

Figure 3.7. AG for Mesh Array.

node / node	6	7
2	1	0
3	2	0
4	1	2

Table 3.5. Mesh Array Output Nodes.

Step 5 of procedure *data flow* results in the input timing function for node $n = 1$, on link of weight $w = 1$ given by

$$\begin{aligned} \pi_1^k(n) &= a(i,j), \\ &= a(n, n+k), 0 < i \leq 1, 1 \leq j < 2, \\ &= nil, \text{ otherwise}, \end{aligned}$$

and for node 3,

$$\begin{aligned} \pi_1^k(n) &= a(i,j), \\ &= a(n-1, n), 1 < i \leq 2, 1 \leq j \leq 2, \\ &= nil, \text{ otherwise}. \end{aligned}$$

Similarly, the input timing function of node 1 of weight $w = 2$ is given by

$$\begin{aligned} \pi_2^k(n) &= b(i,j), \\ &= b(n+k, n), \ 1 \leq i \leq 2, \ 0 < j < 1, \\ &= nil, \text{ otherwise}, \end{aligned}$$

and for node $n = 2$,

$$\begin{aligned} \pi_2^k(n) &= b(i,j) \\ &= b(k, n), \ 1 \leq i \leq 2, \ 1 < j < 2. \\ &= nil, \text{ otherwise}. \end{aligned}$$

Applying step 6 of the procedure, the corresponding node mapping functions are listed below

$$m_3^k(n) = u_1^k(n) * u_2^k(n) + m_3^{k-1}(n),$$

$$v_1 = u_1,$$

$$v_2 = u_2.$$

Step 7 produces the connectivity matrix depicted in table 3.6.

node / node	1	2	3	4
1	0	1	2	0
2	0	0	0	2
3	0	0	0	1
4	0	0	0	0

Table 3.6.

Connectivity Matrix for Mesh Array.

To model the state of node 4 at compute cycle k one must applying step 8 i). From the connectivity matrix **P** node 4 has two weighted links incident on it, namely, $w = 1$ and $w = 2$. Using entry P_{34} as a lookup table, one finds

$$u_1^k\ (4\ ) = v_1^k(3\ )\ ,$$

$$u_2^k\ (4\ ) = v_2^k(2\ )\ .$$

The content of the memory register is

$$m_3^k\ (4\ ) = u_1^k(4\ )\, u_2^k\ (4\ ) + m_1^{k-1}\ (4\ )$$

The outputs from the node are

$$v_1^k(4\ ) = u_1^k\ (4\ ),$$

$$v_2^k(4\ ) = u_2^k\ (4\ ).$$

The array total execution time is 5 cycles. The the final value of T is 4, and the range of the allowable number of compute cycles is $0 \le k \le 4$. From step 3 of procedure PTD, at $k = 0$, node 1 changes its state. There are two data elements on the node that are given by

$$\pi_1^0\ (1\ ) = a_{11}\ ,\ \pi_2^0\ (1\ ) = b_{11}\ .$$

From the PE mapping functions, the internal register content of the node is given by

$$m_3^0\ (1\ ) = y_{11}^1\ .$$

At cycle $k = 1$, nodes 1, 2 and 3 change their state. For node 1 this is given by

$$\pi_1^1(1) = a_{12}, \quad \pi_2^1(1) = b_{21}.$$

The content of the internal register of node 1 is

$$m_3^1(1) = y_{11}^2.$$

for node 2,

$$u_1^1(2) = a_{11}, \quad \pi_2^1(2) = b_{21}, \quad m_3^1(2) = y_{12}^1,$$

and for node 3,

$$u_2^1(3) = b_{11}, \quad \pi_1^1(3) = a_{21}, \quad m_3^1(3) = y_{21}^1.$$

Continued evaluation of the procedure produces the rest of the time levels in the PTD. The final result is depicted in Figure 3.8.

Utilizing step 1 of procedure TG, nodes { 1, 2, 3 } are boundary input nodes. They receive elements from the data paths corresponding to the variables a (.) and b (.). The vectors $\mathbf{V}_a$ and $\mathbf{V}_b$ are assigned for those paths. From step 2, the PTD at time level 0 results in $W_0 = \{ 1 \}$. Hence, node 1 is placed in the TG at time level 0.

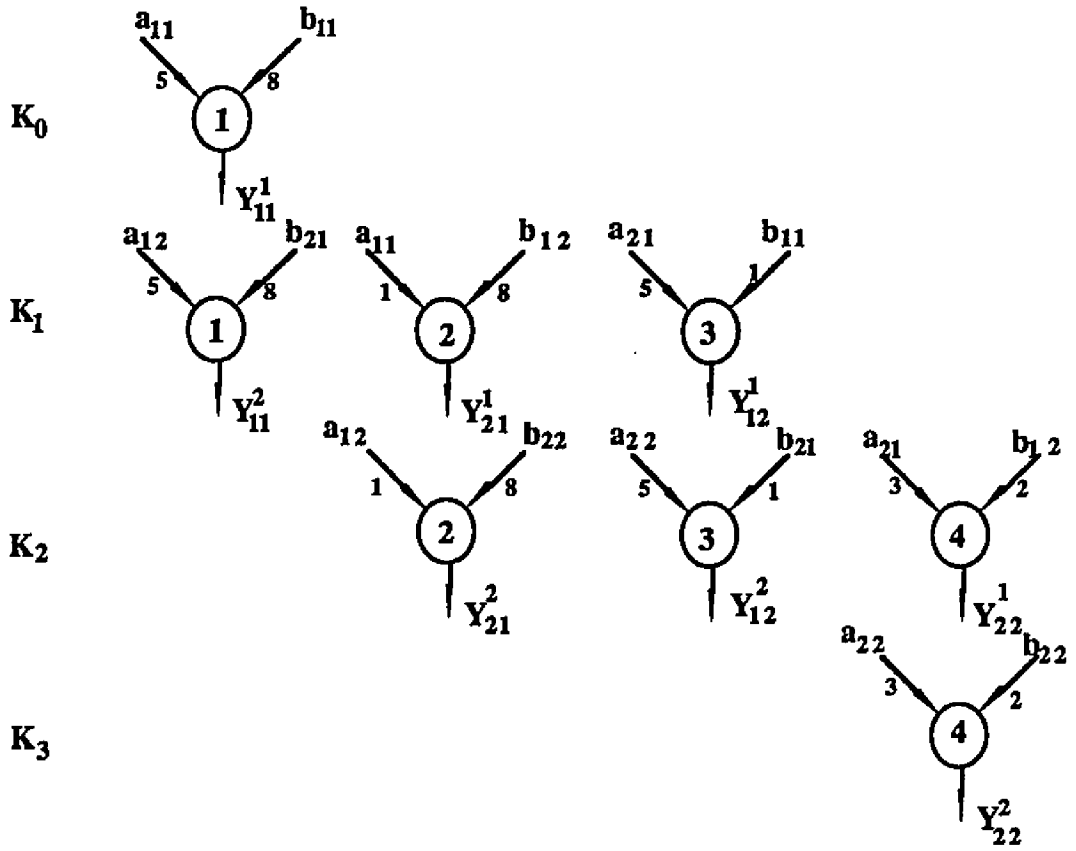

Figure 3.8. PTD for Mesh Array.

From step 3 , let $\mathbf{d}_1$ be assigned as the node vector. In step 4 , choose $\mathbf{V}_a$ as the direction vector. Reading level 1 in the PTD , $W_1 = \{ 1 , 2 , 3 \}$, hence level 1 , in the TG will consist of three vertices. Continued evaluation of the procedure produces the TG depicted in Figure 3.9.

3.9.3. Matrix Multiplication on Cylindrical Array

This example considers the problem of multiplying two matrices of the form $\mathbf{Y = AB}$, where, $\mathbf{Y}$, $\mathbf{A}$ and $\mathbf{B}$ are nxn matrices , on a cylindrical array. The array total execution time is 5 cycles ($(3n-1)$ in general). The array for $n = 2$ is depicted in Figure 3.10.

Procedure *data path* results in two data paths for the elements of the variables a_{ij} and b_{ij} , with weights $w = 1$ and $w = 2$ respectively. The elements of variable y_{ij} are assigned the weight $w = 3$. The resulting paths are displayed in Figure 3.11. From procedure DWCG , the resulting AG of the array is depicted in Figure 3.11.

Procedure *data flow* results in nodes mapping functions that are the same as the mesh array node mapping functions. The I/O lookup tables are given in Tables 3.7 and 3.8 respectively.

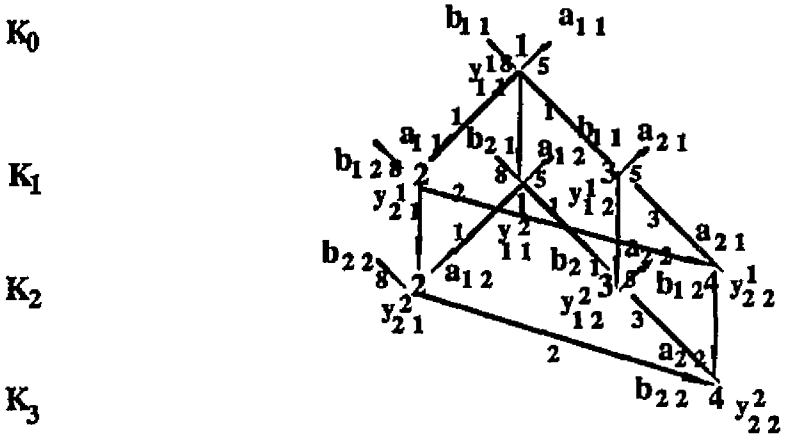

Figure 3.9. TG for Mesh Array.

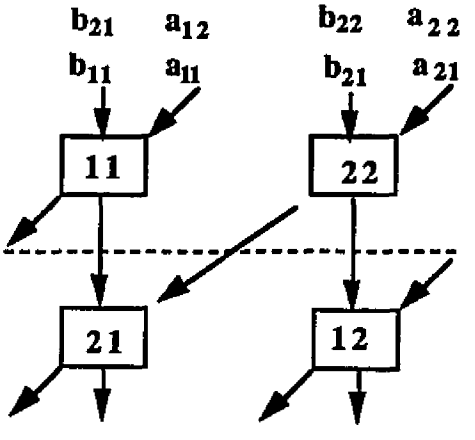

Figure 3.10. Cylindrical Array, $n = 2$.

From step 5, the input timing functions on nodes $n = 1$ and $n = 2$, of weight $w = 1$, are given by

$$\pi_1^k(n) = b(i,j)$$
$$= (n, k+1),\ 1 \le i < 2,\ 1 \le j \le 2.$$
$$= nil,\ \text{otherwise}.$$

The input timing functions on nodes $(1,2)$ of weight $w = 2$ are

$$\pi_2^k(n) = a(i,j)$$
$$= (n, k+1),\ 1 \le i \le 2,\ 1 \le j < 2.$$
$$= nil,\ \text{otherwise}.$$

Step 7 results in the connectivity matrix given in Table 3.9.

node / node	1	2
5	2	2
6	1	1

Table 3.7 Cylindrical Array Input Nodes.

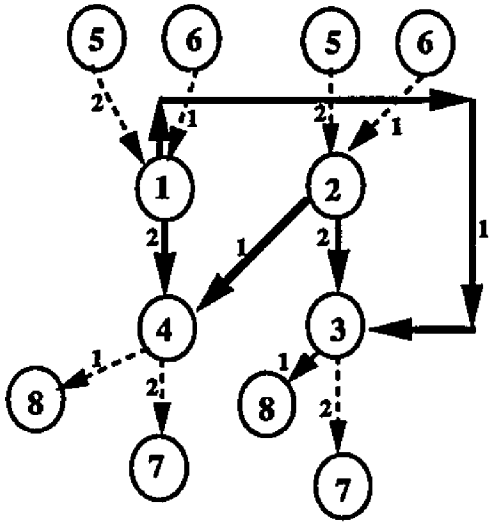

Figure 3.11. AG of Cylindrical Array .

node / node	7	8
3	2	2
4	2	1

Table 3.8 Cylindrical Array Output Nodes .

node / node	1	2	3	4
1	0	0	1	2
2	0	0	2	1
3	0	0	0	0
4	0	0	0	0

Table 3.9.

Cylindrical Array Connectivity Matrix.

To find the state of node 3 at compute cycle k, then applying step 8 i) of procedure data flow, using column 3 of $\mathbf{P}$, there are two links incident on the node. Using entry p_{13} as a lookup table, one finds that the input on link of weight $w = 1$ originates from node 1. Thus,

$$u_1^k\ (3) = v_1^k\ (1)\ ,$$

$$u_2^k\ (3) = v_2^k\ (1)\ .$$

The content of the memory register is

$$m_3^k(3) = u_1^k\ (3)\, u_2^k\ (3) + m_3^{k-1}(3)\ .$$

The output from the node is given by

$$v_1^k\ (3) = u_1^k\ (4),$$

$$v_2^k\ (3) = u_2^k\ (4).$$

From step 3 of procedure *PTD*, at $k = 0$, nodes 1 and 2 change their state. There are two elements on each node, hence

$$\pi_1^0\ (1) = b_{11}\ ,\ \pi_2^0\ (1) = a_{11}\ ,\ \pi_1^0\ (2) = b_{12}\ \text{and}\ \pi_2^0\ (2) = a_{21}\ .$$

The content of the memory registers of the nodes are

$$m_3^0\ (1)\ = y_{11}^1\ \text{and}\ m_3^0\ (2)\ = y_{22}^1\ .$$

At compute cycle *1*, all nodes in the array change their state. Hence,

$$\pi_1^1\ (1) = b_{12}\ ,\ \pi_2^1\ (1) = a_{21}\ ,\ \pi_1^1\ (2) = b_{22}\ \text{and}\ \pi_2^1\ (2) = a_{22}\ .$$

From the mapping functions,

$$u_1^1\ (3) = a_{11}\ ,\ u_2^1\ (3) = b_{12}\ ,\ u_1^1\ (4) = a_{21}\ \text{and}\ u_2^1\ (4) = b_{11}\ .$$

The content of the memory register of each node is given by

$$m_3^1\ (1) = y_{11}^2\ ,\ m_3^1\ (2) = y_{22}^2\ ,\ m_3^1\ (3) = y_{21}^1\ \text{and}\ m_3^1\ (4) = y_{12}^1\ .$$

At time level 2, nodes 3 and 4 change their state. The data on the nodes are given by

$$u_1^2\ (3\ ) = a_{12}\ ,\ u_2^2\ (3\ ) = b_{22}\ ,\ u_1^2\ (4\ ) = a_{22}\ ,\ u_2^2\ (4\ ) = b_{21}\ .$$

The content of the memory registers are

$$m_3^2\ (3\ ) = y_{21}^2 \quad \text{and} \quad m_3^2\ (4\ ) = y_{12}^2\ .$$

The PTD of the array is depicted in Figure 3.12. Procedure TG results in the array timing graph depicted in Figure 3.13.

3.9.4. LU Decomposition Array

Consider an array designed to decompose an *nxn* dense matrix of the form $\mathbf{A} = [\ a_{ij}\ ]$, into an upper triangular matrix $\mathbf{U} = [\ u_{ij}\ ]$ and a lower triangular matrix $\mathbf{L} = [\ l_{ij}\ ]$ such that

$$\mathbf{A = LU}\ .$$

The array for performing the **LU** decomposition for t $n = 3$ is depicted in Figure 3.14. There are two types of processing elements in the array. The array total execution time is (*3n-1*) cycles in general.

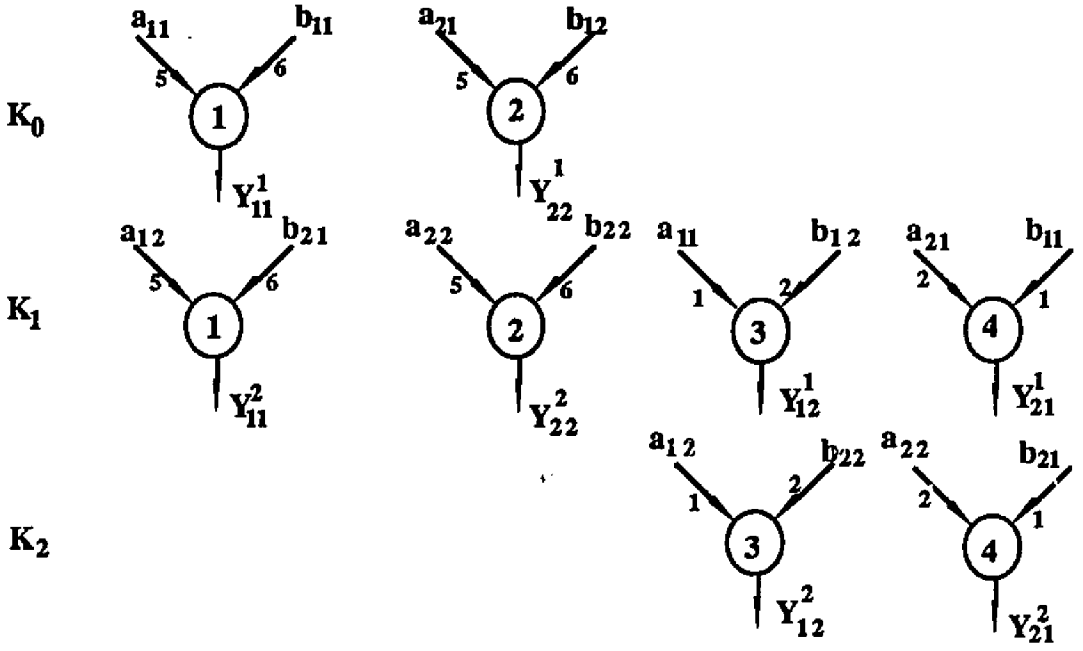

Figure 3.12. PTD Cylindrical Array.

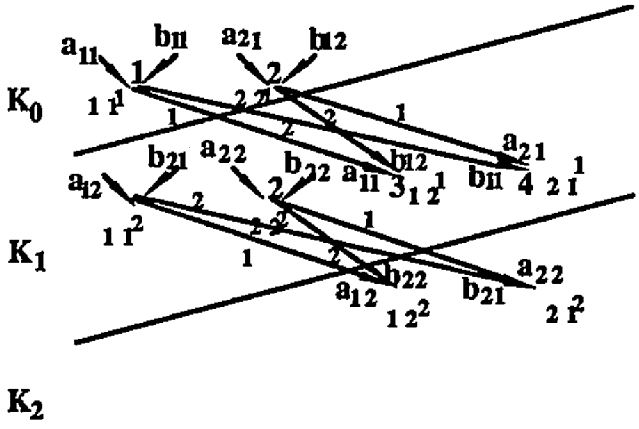

Figure 3.13. TG Cylindrical Array.

Applying procedure *data path*, assign the weight $w = 1$ for the a_{ij} data path, the weight $w = 2$ for the l_{ij} data path, and the weight $w = 3$ to the u_{ij} data path.

Utilizing procedure DWCG, from step 1, there are 6 processing elements in the array that are enumerated as in Figure 3.15. The array AG is displayed in Figure 3.15. From procedure *data flow*, nodes $(1, 2, 3)$ are input nodes. They receive data from the user induced node number 8, on input arc of weight $w = 1$. On the other hand, nodes $(3, 5, 6)$ are output nodes. They transmit data elements to the user induced output node 7. The resulting I/O tables are given in Tables 3.10 and 3.11.

node / node	1	2	3
8	1	1	1

Table 3.10. LU Array Input Nodes.

node / node	7
3	2
5	2
6	2

Table 3.11. LU Array Output Nodes.

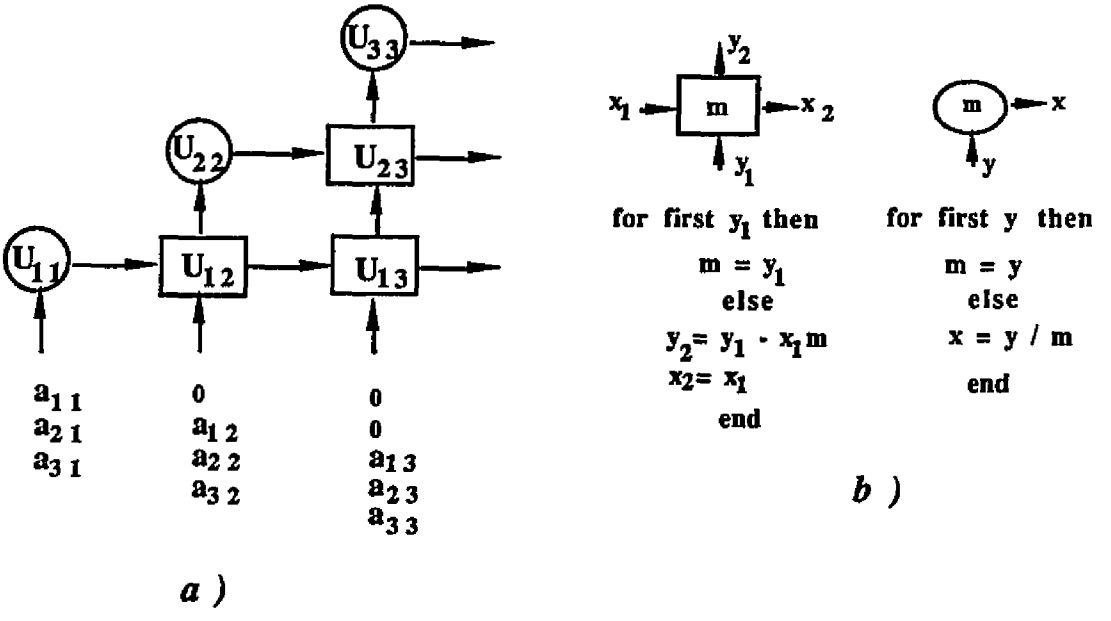

Figure 3.14. a) LU Array, b) PE Models.

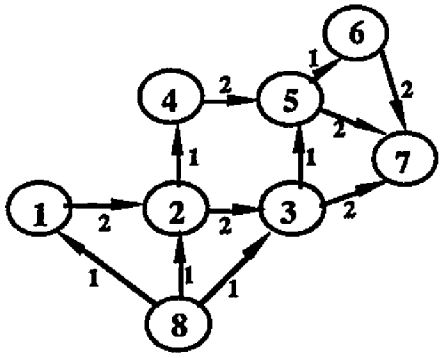

Figure 3.15. AG for LU Array.

Step 5 results in the input timing function for node 1, on link of weight $w = 1$, given below

$$\pi_1^k(1) = a(1+k, 1), 0 < i \leq 3,$$
$$= 0, \text{ otherwise.}$$

For node $n = 2$, the timing function is

$$\pi_1^k(2) = a(k, 2), \ 0 < i <= 3,$$
$$= 0, \text{ otherwise,}$$

and for node $n = 3$, the timing function is

$$\pi_1^k(3) = a(k\text{-}1, 3), \ 0 < i \leq 3,$$
$$= 0, \text{ otherwise.}$$

The PEs have two modes of operation. Hence, a lookup table must be defined to distinguish between the modes of operations of the PEs. Let $M(k)$, with $m_j = i$ if PE j is in mode i of operation, be the lookup table. For nodes $n = 1, 2, 3$, this is given by

$$m_n = 1, \text{ if } k = n - 1,$$
$$m_1 = 2, \text{ otherwise,}$$

and for nodes $n = 4, 5, 6$,

$$m_n = 1 \quad , \text{ if } k = n - 2 ,$$

$$m_1 = 2 \quad , \text{ otherwise .}$$

From step 6, the mapping functions for nodes { 1 , 4 , 6 }, for mode 1 of operation are

$$m_3^k (n) = u_1^k ,$$

and for mode 2 of operations

$$v_2^k (n) = u_4 \quad ,$$

For nodes { 2 , 3 , 5 } , the mapping functions for mode 1 of operations are

$$m_3^k (n) = u_1^k (n) ,$$
$$v_1^k (n) = nil ,$$
$$v_2^k (n) = u_4^k ,$$

and for mode 2 of operations,

$$v_1^k (n) = u_4^k ,$$

$$v_2^k (n) = u_1^k .$$

From step 7, the resulting connectivity matrix of the array is given in Table 3.10. As an example, to simulate the operation of node 2 , at compute cycle 1 , then, from the input nodes lookup table, using mode 1 of operation , one finds

$$u_1^1\ (2\ ) = \pi_1^1 = a_{12}\ .$$

Column 2 of **P** indicates that u_2^1 is originating from node 1. Evaluating node 1 mapping function at $k=0$ results in $u_2^1 = l_{21}$. The content of the memory register is given by

$$m_3^2\ (2\ ) = u_{12}\ .$$

From the definition of mode 1 of operations, the outputs are given by

$$v_1^1\ (2\ ) = v_2^1\ (2\ ) = nil\ .$$

Utilizing step 3 of procedure PTD, at $k=0$, node 1 changes its state, hence,

$$\pi_1^0\ (1\ ) = a_{11}\ .$$

From the node mapping functions, the memory register content is given by

$$m_3^0\ (1\ ) = a_{11}\ .$$

At compute cycle 1 nodes $(1\ ,2\ )$ change their states. Hence,

$$\pi_1^1\ (1) = a_{21} \quad \text{and} \quad \pi_1^1\ (2) = a_{12}\ ,$$

and from the mapping function

$$u_2^1\ (1) = l_{21} \quad \text{and} \quad m_3^1\ (2) = u_{12}\ .$$

The PTD of the array is depicted in Figure 3.16. Procedure TG results in the timing graph depicted in Figure 3.17.

node / node	1	2	3	4	5	6
1	0	2	0	0	0	0
2	0	0	2	1	0	0
3	0	0	0	0	1	0
4	0	0	0	0	2	0
5	0	0	0	0	0	1
6	0	0	0	0	0	0

Table 3.12

Connectivity Matrix for LU array.

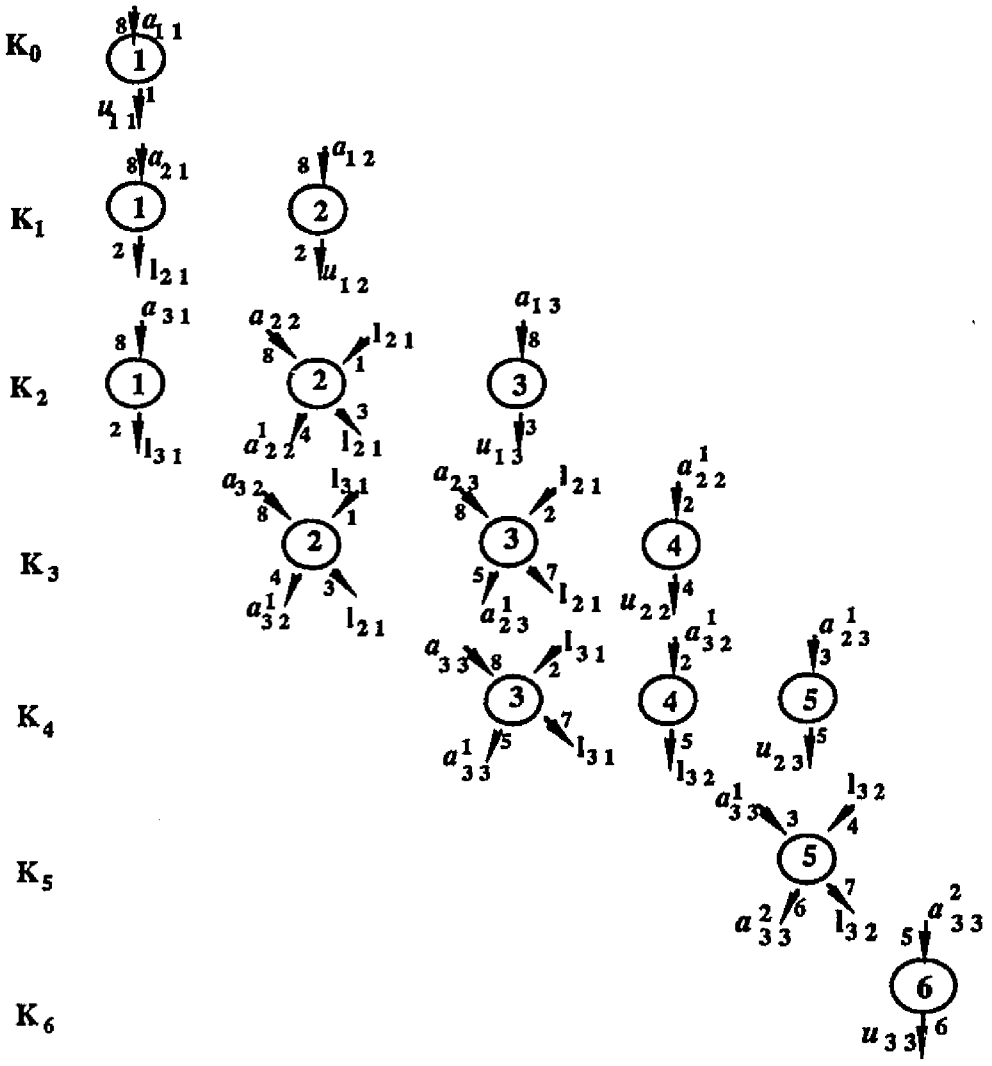

Figure 3.16. PTD for LU Array.

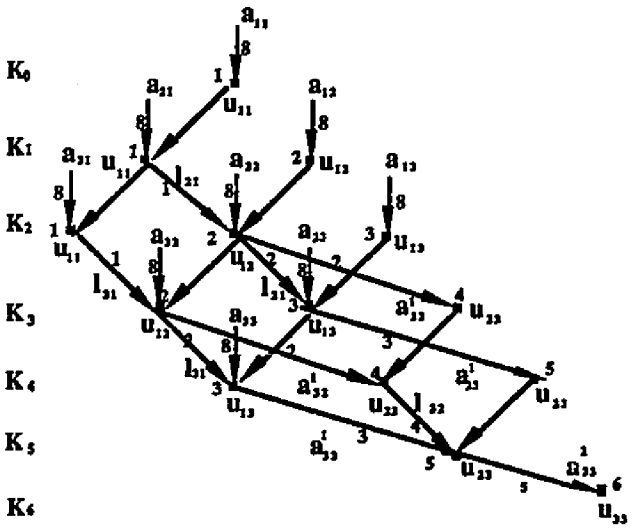

Figure 3.17. TG for LU Array.

CHAPTER 4

FAST ALGORITHM REALIZATIONS

This chapter introduces the concept of *fast algorithm realizations*. These realizations are forms computable in a reduced number of cycles. The objective is to increase the efficiency of an array, and eventually enhance its flexibility. The concept is formally identified in the following:

Definition 4.1: Given a systolic array designed to evaluate an algorithm A in T compute cycles, a *fast realization* for A is a special realization (specified by a particular set of parameters) that can be evaluated in $k < T$ cycles.

To find a fast realization one must determine the collection of operations that, for a given set of parameters, can be omitted without affecting the final numerical value. This objective is accomplished by analyzing the *timing graph* (TG) representation of the array that was introduced in the previous chapter. The restriction of operating an array for k cycles requires that the total number of levels in the TG be reduced to k.

Section 4.1 discusses the technique of *timing graph* compression used to reduce the number of time levels in the TG to k. Section 4.2 develops the basic tools needed to manipulate the timing graph. Sections 4.3 and 4.4 determine the fast forms for matrix arrays, with examples of some of the most popular arrays. Sections 4.5 and 4.6 introduce the concept of timing graph cut and paste. Section 4.7 develops fast forms for non-linear arrays. Section 4.8 applies the developed techniques to modify the operations of some generic arrays, such that they can handle band matrix multiplications, triangular and randomly sparse matrices.

Relevant features of the approach proposed here include:

- It can be used to determine the collection of all possible fast forms implementable on a single array.

- It uses to advantage the nature of algorithm data dependence vectors to allow rearrangement of operations in the array to achieve fast realizations.

- It can be used to enlarge the class of algorithms implementable in a given array.

4.1. Timing Graph Compression

The TG representation provides important information about the evolution of partial results in the array. Each time level in the graph displays the nodes that change their state at that instant of time. Reading the input arcs of the boundary nodes in the TG produces the timing functions of the boundary nodes.

The timing graph of an array consists of a total of T time levels. To obtain exact results from an array in $k < T$ cycles, the number of time levels in the TG must be reduced by $q = T - k$ levels. The concept of TG compression can be used in this case as the vehicle for determining fast forms of linear arrays. It is possible to distinguish between the following basic compression strategies.

Bottom-up compression of the TG corresponds to the identification of the collection of data elements that do not contribute to the partial results in the later cycles of the array operation. This is equivalent to determining the collection of data elements that are irrelevant to the final results for the time interval $[k+1, T]$ in the TG. In this case, the array can be stopped at cycle k. The results available at that time are considered as the final results.

Stopping the array before its normal execution time may result in improper draining of the array, where some of the data elements remain in the PEs. Bottom-up compression of the TG does not require any change in the timing functions of the input boundary nodes. However, adjustments might be needed in the way the output data elements are extracted from the array.

Top-down compression of the TG corresponds to the determination of the collection of entries that can be skipped from the input data to the array to meet the deadline of k cycles of operation. This can be accomplished by compressing the TG from the top by a total of k levels. Hence, time levels 0 to $k-1$ in the TG will contain the elements that are irrelevant to the partial results. The omission of data in early levels in the TG must not violate the available bandwidth of the array. Top-down compression of the TG does not cause improper draining of the array. However, it requires the modification of the input timing functions that supply data elements to the boundary PEs . In addition, adjustments might be needed in interpreting the output results from the array.

Top-bottom compression of the TG is a combination of the above two methods. This might be required in some instances to meet the deadline of k cycles. In this case, the collection of data elements that must be skipped in the early stages as well as in the later stages of the array operation needs to be identified. Top-bottom compression of the TG requires adjustments in the input timing functions as well as modifications in the way output data from the array are extracted.

The compression of the TG may require a modification in the way the data elements are fed to the array. This will lead to a change in the timing functions of the user induced input nodes. In the next section, proper tools are introduced that allow the determination of the form of the modified input timing functions for those nodes.

4.2. Tools for Modifying the Timing Graph

In the TG , elements that belong to a data path travel on lines that are parallel to the vector $\mathbf{V_a}$ defining the path direction. Consider a node g , represented in the TG at time levels i , $i+1$, ... , j . The collection of elements available on the node's input arc q , that carry elements belonging to the same data path (specified by the direction vector $\mathbf{V_a}$) for those time levels , defines a timing sequence, denoted by X ($\mathbf{V_a}$, g , j-i) .

Consider for example, the TG of the matrix-vector array, depicted in Figure 3.5. Reading the elements available on node 1 , on lines that are parallel to the x data path direction vector , for time levels { 0 , 1 , 2 , 3 , 4 } , the timing sequence { x_1 , x_2 , x_3 , 0 , 0 } results. The sequence indicates that at time 0 the input node contains data element x_1 , at time 1 it contains data element x_2 , and so forth.

Timing sequences on all lines must be of length T , where T is the total number of time levels in the TG . If one sequence is shorter than the other, zero entries must be appended to the sequence in the proper time slots , such that the resulting sequence is of length T . In what follows , the necessary steps for deriving the timing sequence on a line parallel to a particular direction vector are presented .

Procedure *T*

This procedure constructs the timing sequence of node g of data elements a (.) belonging to the data path specified by the direction vector $\mathbf{V_a}$. In this procedure , N

is the number of nodes in the DWCG, T is the total number of time levels in the TG , and $L\ (n\ )$ is a vector that holds the time of the initial element in the sequence. The procedure generates sequences of length T for all the nodes.

1- Construct the array TG.

2- For $n = 1$ to N do

Begin

Initialize *count* to zero.

For $i = 0$ to T do

Begin

If node n has an input arc q parallel to V_a then include the element $a(.)$ available on that arc to $X(.)$.

$count = count + 1$.

$L\ (n\ ) = i$, if $count = 1$.

End.

2- For $n = 1$ to N do

Begin

$j = L\ (n\ )$, $jj = j + count$.

i) For $kk = 0$ to $j\text{-}1$ do

Begin

Insert zeros to $X(.)$.

End.

ii) For kk = *count* to T do

Begin

Append zeros to $X(.)$.

End.

End .

For the purpose of this study it is fundamental to notice that timing sequences can be manipulated. It is possible to slide the sequence *up* or *down* on the time axis. For example, consider the sequence $\{ x_1, x_2, x_3, 0, 0 \}$ derived earlier, for the TG of the matrix-vector array. By sliding the sequence up by one cycle on the time axis, the sequence $\{ x_1, x_2, x_3, 0, 0, 0 \}$ results. The interpretation is that at time slot -1 the data element x_1 is available, at time slot 0 the data element x_2 is available, and so forth. However, to keep the sequence of the same length as the original the x_1 data element must be deleted. Hence, the new sequence is given by $\{x_2, x_3, 0, 0, 0 \}$. The notation $U(X(.))$, will be used to indicate sliding the sequence up, on the time axis by one cycle. Sliding a sequence *up* on the time axis can be used in a cascaded fashion. To slide the sequence up n times, one can write

$$U^n (X(.)) = U (U^{n-1} (X(.))) .$$

Sliding up a sequence of length T by one cycle is equivalent to deleting the first element in the sequence and appending a zero to its last entry. To track those elements that are deleted from a sequence as a result of sliding it up, one must construct a set that stores those entries. Let, $\hat{X}(.)$ be the new sequence resulting from sliding up the sequence

$X(.)$ n times. Then the collection of elements x_i that are deleted from $X(.)$ to generate $\grave{X}(.)$ form the set $\mu(X(.), n)$. Hence,

$$\mu(X(.), n) = \{ x_i : x_i \in X(.) \wedge x_i \notin U^n(X(.)) \}.$$

Timing sequences can also be slided *down*. Sliding a sequence down by one cycle is equivalent to inserting a new element in the first time slot, and deleting the last element in the sequence. The notation $D(X(.))$ will be used to indicate sliding a sequence down on the time axis by one cycle. Sliding a sequence *down* can be used in a cascaded fashion. To slide a sequence down n times, one can write

$$D^n(X(.)) = D(D^{n-1}(X(.))).$$

Let, $\grave{X}(.)$ be the new sequence resulting from sliding down the sequence $X(.)$ n times. Then the collection of elements x_i that are deleted from $X(.)$ to generate $\grave{X}(.)$ form the set $\beta(X(.), n)$. Hence,

$$\beta(X(.), n) = \{ x_i : x_i \in X(.) \wedge x_i \notin D^n(X(.)) \}.$$

The introduced tools can be used effectively to derive the modified timing functions of the user induced input nodes, when the TG of the array is compressed.

4.3. Fast Realizations of Matrix Arrays

In this section, the collection of the fast realizations allowable on arrays implementing matrix operations of the form $\mathbf{Y} = \mathbf{AX}$ or $\mathbf{Y} = \mathbf{XA}$, is determined. Here, it is assumed that the designer has the ability to choose the structure of the linear transformator specified by $\mathbf{A}$.

The first stage of the analysis proceeds with the assumption that the designer does not have any control over the choice of the data entries $x_{ij} \in \mathbf{X}$. If those entries can also be controlled, a broader selection of fast forms can be determined. Utilizing the concept of TG compression of an array, a complete description of the fast forms of $\mathbf{A}$ that admits exact results on an array in k cycles, is now possible. In what follows, procedures are introduced that determine the nature of the matrix $\mathbf{A}$ that allow exact computations on *linear* arrays in a reduced number of cycles. The procedures are applicable to matrix-matrix and matrix-vector arrays.

Procedure *ABU*

The procedure is based on the concept of bottom-up compression of an array TG. It produces the fast form of $\mathbf{A}$ for k cycles of operations.

1- Construct the TG of the array.

2- Identify the direction vector $\mathbf{V}_a$ that defines the movement of the a_{ij} data paths.

3- For $i = 0$ to k do

Begin

Construct the set $M(i) = \{ n_j \}$, where n_j are the nodes at time level i in the TG.

End.

4- Determine the set of all vectors that are parallel to $\mathbf{V_a}$ in the TG . Let r be the number of vectors in the set.

5- Using the array TG , execute the following :

i) For $j = 1$ to r do

Begin

ii) For $i = 0$ to T do

Begin

Evaluate $X(\mathbf{V_a}, g, w)$ for all nodes $n_i \in M(i)$, at each time level i .

End .

End.

6- To determine the collection of the elements $a_{ij} \in A$ that need to be zero, one must execute the following :

i) For $i = T$ down to $k+1$ do

Begin

ii) For $j = 1$ to r do

Begin

For each node $n_i \in M(i)$, evaluate its set

$\mu = U^k(X(.))$.

End.

End.

Procedure ***ATD***

This procedure is based on the concept of top-down compression of the TG . It finds the fast form of A that admits exact results in k cycles. The procedure modifies the data paths corresponding to the x_{ij} entries for matrix-matrix arrays. However, this will not be the case for matrix-vector arrays, because the a_{ij} entries are free elements. The necessary steps for the procedure are :

1- Construct the TG of the array.

2- Identify the direction vector V_a of the a_{ij} data paths.

3- For $i = 0$ to k do

Begin

Construct the set $M(i) = \{ n_j \}$, where n_j are the nodes at time level i in the TG.

End.

4- Determine the set of all vectors that are parallel to V_a in the TG . Let r be

the number of vectors in the set.

5- Using the TG of the array execute the following:

i) Initialize *count* to zero.

ii) For $i = 0$ to k do

Begin

Check if any node $n_i \varepsilon M(i)$ at time level i is not a boundary node, if *yes* go to step 6, else increment *count* by 1.

End.

6- If *count* $\geq k$, go to step 7, else the procedure is not applicable for the current value of k, stop.

7- Using the array TG execute the following :

i) For $j = 1$ to r do

Begin

ii) For $i = 0$ to T do

Begin

Evaluate $X(V_a, n, w)$, for all nodes $n_i \varepsilon M(i)$.

End.

End.

8- To determine the collection of a_{ij} elements that need to be skipped from

the input data , execute the following :

i) For $i = 0$ to k do

Begin

ii) For $j = 1$ to r do

Begin

For each node $n_j \in M(i)$, evaluate the set $\mu = U^k(X(.))$.

End.

End.

9- If the a_{ij} entries are not free elements, then the new input timing functions for the x_{ij} entries are given by :

For $i = 0$ to k do

Begin

for each node n shifted down , evaluate

$\pi_x'(.) = D^k((\pi_x(.))$.

End .

Procedure *ATB*

This procedure is based on a combination of procedures *ABU* and *ATD* to produce the required fast forms of **A** . The necessary steps are listed below:

1- Construct the TG of the array.

2- Identify the direction vector $\mathbf{V_a}$ of the a_{ij} data paths.

3- For $i = 0$ to k do

Begin

Construct the set $M(i) = \{ n_j \}$, where n_j are the nodes at time level i in the TG.

End.

4- Determine the set of all vectors that are parallel to $\mathbf{V_a}$ in the TG . Let r be the number of vectors in the set.

5- Using the array TG execute the following :

i) Evaluate steps 4 and 5 , of procedure *ATD*, save the value of *count*.

ii) For $i = 0$ to *count* do

Begin

a) Evaluate step 7 of procedure *ATD* .

b) For $kk = i$ to k do

Begin

Evaluate step 6 of procedure *ABU* .

End.

c) The new input timing functions for $n_m \in M(i)$, is given by $\pi_x'(.) = D^j((\pi_x(.))$, where $j = count - i$.

End.

The collection of all matrices that meet the deadline of k cycles, are determined by applying procedure ATB for all the combinations of $q = T - k$.

4.4. Examples to Illustrate Procedures for Determining Fast Realizations

To illustrate the use of the procedures introduced in the previous sections, two examples are considered. The arrays are the matrix-vector array and the cylindrical array for matrix-matrix multiplications.

4.4.1. Matrix-Vector Array

Consider the *3x3* matrix-vector array, depicted in Figure 3.2. The number of compute cycles for the array is *5* cycles, that ranges from $[0, 4]$. The TG of the array is given in Figure 3.5. Assume, that exact results must be delivered in *4* cycles, then it is required to determine the fast form of **A** that makes this possible.

Applying procedure ABU, from step 1 the TG of the array is depicted in Figure 3.5. From the TG at time level k_4 node 3 is operational. From step 2 of the procedure node 3 receives the a_{ij} data entries on vectors that are parallel to V_a of weight $w = 6$. From steps 3, 4 and 5 the following timing sequence results.

$$X(.) = \{0, 0, a_{31}, a_{32}, a_{33}\}.$$

Evaluating step 6 generates the following set

$$\mu(X(.), 1) = \{ a_{33} \}.$$

Hence, the fast form of A is given by

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 0 \end{bmatrix}.$$

Procedure *ATD* does not apply in this case, because at time level *1* , node *2* is a boundary node.

4.4.2. Matrix-Matrix Multiplication on the Cylindrical array

In this example, the collection of fast forms implementable on a cylindrical array designed to perform matrix-matrix multiplication is determined. The array execution time is *3* cycles that ranges from [*0* , *2*]. The TG of the array is given in Figure 3.13. Assume, that the array has to output the results in $k = 2$ cycles. Procedures *ABU* and *ATD* result in the following collection of fast forms for A

$$\begin{bmatrix} a_{11} & 0 \\ a_{21} & 0 \end{bmatrix}, \begin{bmatrix} 0 & a_{12} \\ 0 & a_{22} \end{bmatrix}.$$

4.5. Timing Graph Cut and Paste

The procedures covered in the previous sections, considered limiting matrix-vector arrays operations for a certain number of compute cycles. The analysis was based on the concept of compressing the TG to meet the deadline of k cycles. The results were based on those entries in the data elements , that could be skipped in the early (or) later stages of operations. To increase the class of the constrained forms of the matrix A , a combination of the two methods was also used.

In what follows, new procedures are introduced that take advantage of the nature of the data dependence vectors of the algorithm at hand. In particular, the method solves the problems that arise when the data entries that are irrelevant to the final results are located in non boundary nodes in the AG, at the constrained cycle. This problem was encountered in procedure *ATD* , where the procedure was deemed inapplicable when that condition was detected.

Even though these procedures are established to the particular algorithm, they indicate a possible alternative for a more general problem. The generalization of the results will be given in a subsequent section when the problem of random sparsity in matrices is considered.

To illustrate the idea , consider the TG representation of the matrix-vector array that was analyzed earlier. Assume, that the array operations are restricted to 4 cycles, with the requirement that the entries $\{a_{11}, a_{12}, a_{13}, a_{31}, a_{32}, a_{33}\}$ must not be zero. In

this case, none of the compression techniques will produce the fast form of A that admits exact realization in the required number of cycles.

Careful analysis of the TG indicates that if the operations performed by nodes 2 and 3 are interchanged, exact results can still be achieved in 4 cycles. Hence, the scope of fast realizations implementable in the array is increased. Here, interchanging the operations of the nodes is possible because the algorithm at hand is characterized by fixed-flexible data dependence vectors. Hence, the fast form of A is given by

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} \\ a_{21} & 0 & a_{23} \\ a_{31} & 0 & a_{33} \end{bmatrix}$$

The way the input data is fed to the array must be modified, in particular columns 2 and 3 of A need to be interchanged. This will modify the timing functions that generate the a_{ij} data elements. For the example under study, the elements corresponding to the a_{ij} data paths are characterized by a dummy data dependence vector . Hence, in this case, it is possible to interchange the nodes' operations without imposing any restrictions on the data paths corresponding to the x_{ij} elements. However, this is not true for matrix-matrix arrays.

Interchanging the nodes' operations in the TG of a matrix-vector array can be pursued in two different ways. The first approach interchanges the nodes on lines that are parallel to the node direction vectors. The second approach interchanges the operations on lines

that are parallel to the direction vector of the x_{ij} data elements. The combination of the two methods expands the collection of all fast realizations implementable on the array in k cycles. Interchanging the nodes' operations must not violate the available bandwidth of the array. This condition is satisfied by restricting the internal nodes that can be interchanged with the boundary input nodes. Next, two procedures are introduced that interchange the nodes' operations based on the two options.

Procedure ***MV***

This procedure interchanges two nodes' operations in the TG of a matrix-vector array, on lines that are parallel to the nodes' direction vector. Here, $\mathbf{d}_n$ is the node direction vector and $\mathbf{V}_x$ is the direction vector of the x data paths. The necessary steps are :

1- Construct the TG of the array.

2- For $i = 0$ to k do

Begin

Construct the set $M(i) = \{ n_j \}$, where n_j are the nodes at time level i in the TG.

$b(i)$ holds the number of nodes at time level i.

End.

3- Using the TG of the array execute the following :

For $k = 0$ to T do

Begin

For $kk = 1$ to $b(k)$ do

Begin

$count = 0$.

For $j = k$ to T do

Begin

If $n_j \in M(j)$ then increment *count.*

End.

Evaluate $X(V_a, kk, count)$ and $X(V_x, kk, count)$.

End.

End.

4- To interchange the operations performed by nodes n_i and n_j, then

i) Let the first occurrence of node n_i be at time level i and the first occurrence of node n_j be at time level j. Node n_j can replace node n_i if and only if $j > i$.

ii) The modified timing sequences on node n_j are given by :
$U^{j-i}(X(V_a, n_j, ..))$ and $U^{j-i}(X(V_x, n_j, ..))$.

iii) The modified timing sequences on node n_i are given by :
$D^{j-i}(X(V_a, n_i, ..))$ and $D^{j-i}(X(V_x, n_i, ..))$.

Procedure ***MVI***

This procedure interchanges two nodes' operations in a matrix-vector array on lines that are parallel to the x_{ij} elements direction vector. The steps are given next :

1- Construct the TG of the array.

2- For $i = 0$ to k do

Begin

Construct the set $M(i) = \{ n_j \}$, where n_j are the nodes at time level i in the TG.

$b(i)$ holds the number of nodes at time level i.

End.

3- Let V_1 and V_2 be the two direction vectors, parallel to V_x, specifying the nodes to be interchanged. Using the array TG perform the following:

i) Using V_1 ,

For $k = 0$ to T do

Begin

Form the set $M_1(k) = \{ n_j \}$, n_j is a node on V_1 .

Save i , which is the number of the time level of the first node occurrence.

End .

ii) Using V_2 ,

For $k = 0$ to T do

Begin

Form the set $M_2(k) = \{ n_m \}$, n_m is a node on V_2 .

Save o , which is the number of the time level of the first node occurrence.

End .

ii) $count\ 1 = 0$, $count\ 2 = 0$

For $k = 0$ to T do

Begin

If $n_i \in M_1(k)$ is a boundary node, then $count\ 1 = count\ 1 + 1$

If $n_m \in M_2(k)$ is a boundary node, then $count\ 2 = count\ 2 + 1$

End .

4- To interchange the modes of $M_1(k)$ and $M_2(k)$ then

i) If $o > i$ and $count\ 2 \leq count\ 1$ then

ii) Interchange the nodes of $M_2(k)$ with the nodes of $M_1(k)$.

5- The new input timing function for the set $M_1(k)$ is $\pi'(.) = U^{o-i}(\pi(.))$.

6- The new input timing function for the set $M_2(k)$ is $\pi'(.) = D^{o-i}(\pi(.))$.

4.5.1. Examples to Illustrate Timing Graph Cut and Paste

Consider the TG of the matrix-vector array that was analyzed in a previous section. The array total execution time is *5* cycles. Assume, that *4* cycles are available for the array to output its final results. Procedure *MV* leads to the following collection of matrices that admits exact results in *4* cycles.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix},$$

$$\begin{bmatrix} 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 0 \end{bmatrix}.$$

Procedure *MV1* produces the following collection of fast forms.

$$\begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 0 \end{bmatrix}, \begin{bmatrix} a_{11} & 0 & a_{13} \\ a_{21} & 0 & a_{23} \\ a_{31} & 0 & a_{33} \end{bmatrix}, \begin{bmatrix} 0 & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix}.$$

4.6. Fast Realizations of Non-Linear Arrays

This section considers arrays implementing systolic algorithms characterized by fixed-nonflexible data dependence vectors. Such algorithms are sensitive to the order in which the computations are performed. Hence, data re-arrangement is not possible. One class of algorithms that is of particular interest contains *unary* algorithms that include: LU decomposition, QR triangularization and transitive closure. Arrays implementing such algorithms will be referred to as *non-linear* arrays.

The approach pursued for determining the collection of fast realizations executable on linear arrays is not appropriate for arrays implementing such algorithms. These algorithms are sensitive to the order in which computations are performed because their data dependence vectors are non-flexible. However, the TG can still be used to determine the collection of fast forms allowable on a single non-linear array. Here, a different course of action must be pursued. The approach is based on decomposing the array operations into sub-arrays implementing problems of smaller sizes. The concept of TG decoupling will be used to identify the collection of fast realizations executable on a single non-linear array.

4.6.1. Timing Graph Decoupling

Timing graph decoupling is defined as the process of determining those entries in the input data to non-linear arrays that can be omitted or forced to a particular pre-set value, that lead to decomposing the array operations into sub-arrays implementing smaller size problems. Array decomposition speeds up the response time of the generic array in two different ways. First, it skips those entries from the input data that are irrelevant to the

final results. Second, it allows the trading of a delay element in the input data with an entry from the input elements to be processed.

Due to the nature of the algorithms at hand, it is important to note that if an entry is omitted or forced to a particular value, such as zero, care must be taken to insure the numerical correctness of all elements that are dependent on it . As an example, suppose the array to be analyzed is implementing *QR* triangularization of a matrix. In this case, none of the entries on the diagonal of the matrix, can assume the *zero* value because the *QR* algorithm would fail .

Analyzing the TG of such arrays is not as simple as the analysis of the TG of linear arrays. To determine the collection of entries that can be omitted from the array, one must ensure that the weight of each node that lies on the direction vector carrying those entries is defined. Once this condition is satisfied, the TG can be properly analyzed and the collection of all fast forms can easily be determined. In the next section , a procedure is introduced that marks all the nodes of the array that can skip their input data. Once the collection of those elements are determined, another procedure is presented that decomposes the array into sub-arrays.

Procedure *D*

This procedure determines the collection of entries in a nonlinear array that can be assigned a particular value , such as zero, and skipped from the array input . The necessary steps are listed below :

1- For $i = 0$ to k do

Begin

Construct the set $M(i) = \{ n_j \}$, where n_j is a boundary node at level i in the TG.

$b(i)$ holds the number of nodes at time level i.

End.

2- Let $\mathbf{V_a}$ be the direction vector that carries the input elements to the boundary nodes .

3- For $kk = 0$ to k do

Begin

a) For $j = 1$ to $b(kk)$ do

i) For each $n_j \in M(kk)$ do

Begin

Evaluate the node mapping functions at cycle kk for the modified numerical value of the element. If the weight of the node is not defined, mark the node as ineligible for omission. Go to step 3 a) .

End.

ii) Let m be the total number of nodes that lie on a line parallel to $\mathbf{V_a}$ and starting at node n_j .

For $i = 1$ to m

Begin

Evaluate the mapping functions of node n_i.

If the weight of the node is not defined, mark the node as ineligible for skipping.

End.

End.

Procedure ***NSA***

This procedure determines the structure of the modified sub-arrays and their resulting input timing functions. The procedure depends on the array connectivity matrix **P** that was introduced in chapter 3. The necessary steps are summarized below:

1- For each boundary node marked as eligible for skipping in the TG identify the vector $\mathbf{V}_a$ that carries the input data elements.

2- If all nodes belonging to $\mathbf{V}_a$ are eligible for omission, then find i and j the first and last time levels of the nodes.

i) For $k = i + 1$ to j do

Begin

a) Move the boundary node and all the nodes that lie on $\mathbf{V}_a$ to time level $k-1$ if and only if all the nodes along the lines stay

properly connected as specified by the connectivity matrix **P**.

b) The modified input timing function of the boundary node is $\pi'(.) = U^{k-1}(\pi(.))$.

End.

3- Repeat step 2 , for all boundary nodes that are eligible for omission.

The collection of fast realizations executable on non linear arrays can be enhanced if the delay entries in the input can be traded with entries from the elements to be processed. This objective is achieved by the recursive decoupling of the TG. The technique identifies the collection of fast realizations allowable on a single non-linear array.

Recursive decoupling of the TG is defined as the process of decomposing a non-linear array operation into sub-arrays and the determination of the collection of fast forms of the sub-arrays. Next, a procedure is presented that determines the collection of fast realizations implementable on a given non-linear array. The necessary steps are:

Procedure *TGD*

1- Invoke procedure *D* for the original full size array.

2- Invoke procedure *NSA* for the original full size array.

3- Re-label the nodes in the TG into separate sets, such that each set

corresponds to the nodes of one sub array. Let l be the number of the sets.

4- For $ll = 1$ to l do

Begin

i) For $i = 0$ to k do

Begin

Construct the set $M(i) = \{ n_j \}$, where n_j is a boundary node at level i in the TG , of the set ll .

$b(i)$ holds the number of nodes at time level i .

End.

ii) For each boundary node, identify the vector $\mathbf{V}_a$ that carries the input data elements.

iii) For $kk = 0$ to k do

Begin

For $j = 1$ to $b(kk)$ do

Begin

For each $n_j \in M(kk)$ do

Begin

Evaluate the node mapping functions for cycle kk for the modified numerical value of the element. If the weight of the node is not defined, mark the node as ineligible for skipping.

End.

End.

End.

iv) Let m be the total number of nodes that lie on a line parallel to V_a and starting at node n_j .

For $i = 1$ to m

Begin

Evaluate the mapping functions for node n_i .

If the weight of the node is not defined , mark the node as ineligible for skipping.

End.

v) If all nodes belonging to V_a are eligible for skipping , then find i and j , the first and last time levels of the nodes.

For $k = i + 1$ to j do

Begin

a) Move the boundary node and all the nodes that lie on V_a to time level $k-1$ if and only if all the nodes along the line stay properly connected as specified by the connectivity matrix **P**.

b) The modified input timing function for the boundary node is $\pi'(.) = U^{k-1}(\pi(.))$.

End.

c) Repeat step *2* for all boundary nodes that are eligible for skipping.

End.

4.7. Examples to Illustrate Non-Linear Array Decomposition

In this section two examples of non-linear arrays, namely, the *LU* decomposition array and the *QR* factorization array , are considered. The developed concepts are applied to decompose these arrays into sub-arrays implementing problems of smaller dimension. In addition, the collection of all fast forms executable on the arrays are determined by the recursive decoupling of the TG.

4.7.1. LU Decomposition Array

In this example, the collection of data entries that decompose the *LU* array into sub-arrays for the case $n = 4$ are determined. The array is depicted in Figure 4.1 and the partial TG is given in Figure 4.2. Procedure *D* produces the nodes that can skip a zero entry in their input data. The collection of nodes that permit the omission of their input data entries are shown in the dotted boxes in the figure. Procedure *NSA* produces the modified array depicted in Figure 4.3. The array is decomposed into two sub-arrays. The first sub-array consists of the PEs accumulating $\{ u_{11}, u_{12}, u_{22} \}$. The second sub-array consists of the PEs accumulating $\{ u_{33}, u_{34}, u_{44} \}$. The modified array produces results

in $2n$ cycles , while the generic array requires $3n$ cycles. The fast form of **A** is given by

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}.$$

Recursive decoupling produces the following collection of fast forms for the first sub-array :

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ 0 & 0 \\ 0 & 0 \end{bmatrix},$$

$$\begin{bmatrix} a_{11} & 0 \\ a_{21} & a_{22} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} a_{11} & 0 \\ 0 & a_{22} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

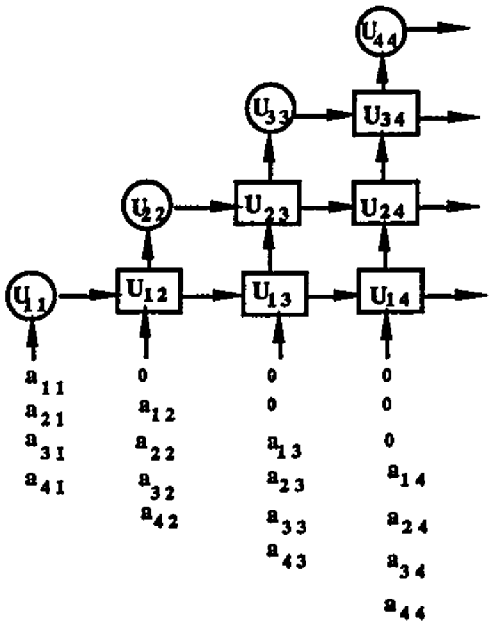

Figure 4.1. LU Array , case $n = 4$.

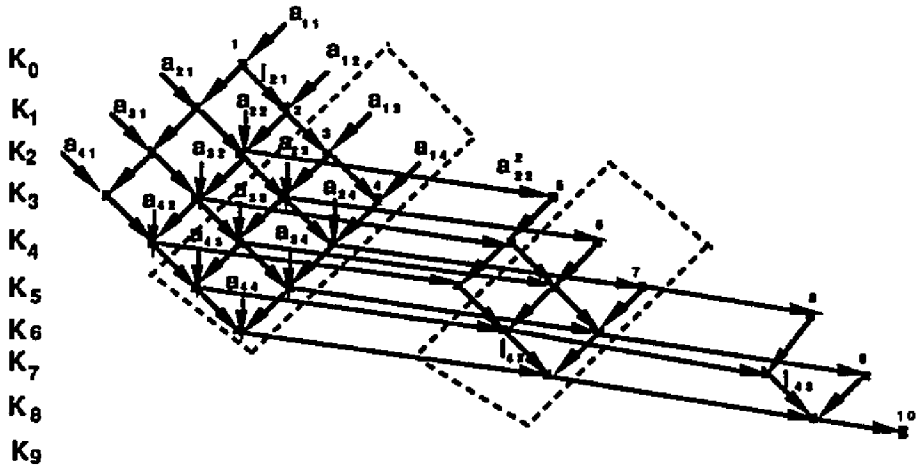

Figure 4.2. Partial TG for LU Array .

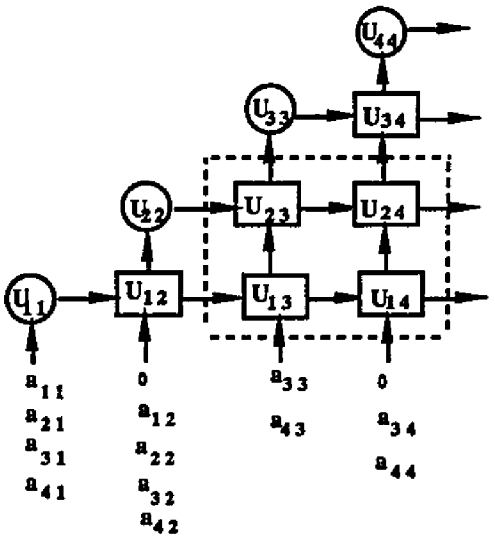

Figure 4.3. Fast LU Array .

Similarly, the collection of fast forms for the second sub-array are given by

$$\begin{bmatrix} a_{33} & a_{34} \\ 0 & a_{44} \end{bmatrix}, \quad \begin{bmatrix} a_{33} & 0 \\ 0 & a_{44} \end{bmatrix}.$$

4.7.2. QR Triangularization Array

Consider the QR array for $n = 4$ depicted in Figure 4.4. The partial TG of the array is given in Figure 4.5. Procedure D results in the collection of nodes that can skip a zero entry. The nodes are identified in Figure 4.5 in dotted boxes. The decomposed array is depicted in Figure 4.6 with the modified input timing functions. The array consists of two sub-arrays. The first sub-array is composed of the PEs accumulating $\{q_{11}, q_{12}, q_{22}\}$. The second sub-array consists of PEs accumulating $\{q_{33}, q_{34}, q_{44}\}$. The fast form of A is given by

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{bmatrix}.$$

Recursive decoupling produces the following collection of fast forms for the first sub-array

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \end{bmatrix},$$

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \end{bmatrix}.$$

For the second sub-array the following collection of fast forms are obtained

$$\begin{bmatrix} a_{33} & 0 \\ a_{43} & a_{44} \end{bmatrix}, \quad \begin{bmatrix} a_{33} & 0 \\ 0 & a_{44} \end{bmatrix}.$$

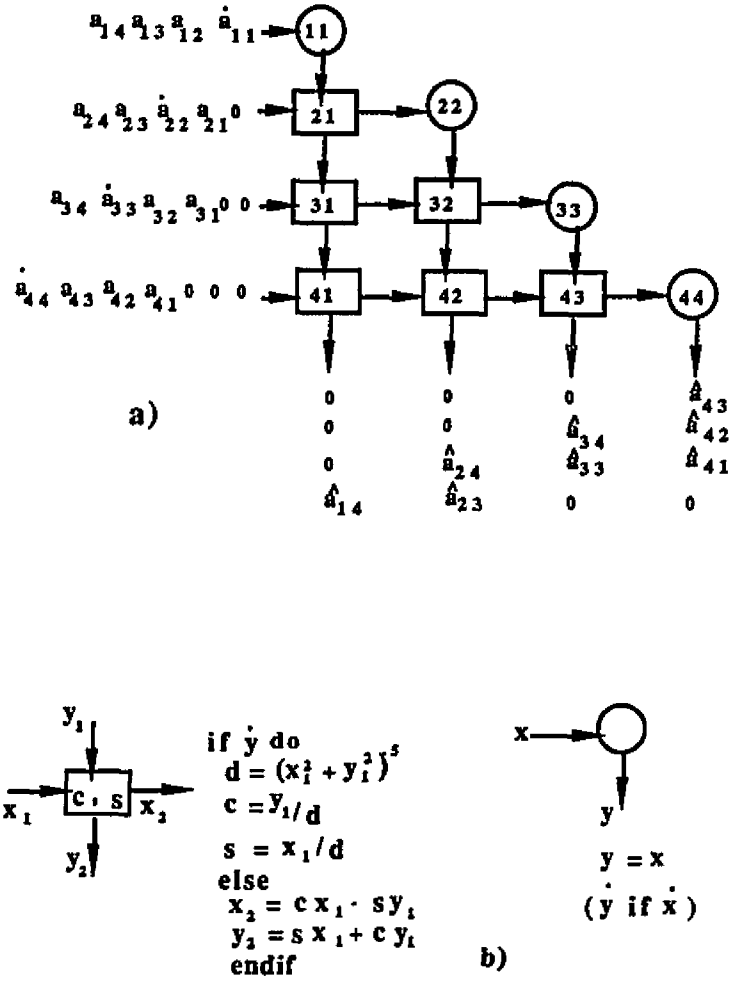

Figure 4.4. a) QR Array , b) PE mode Definitions .

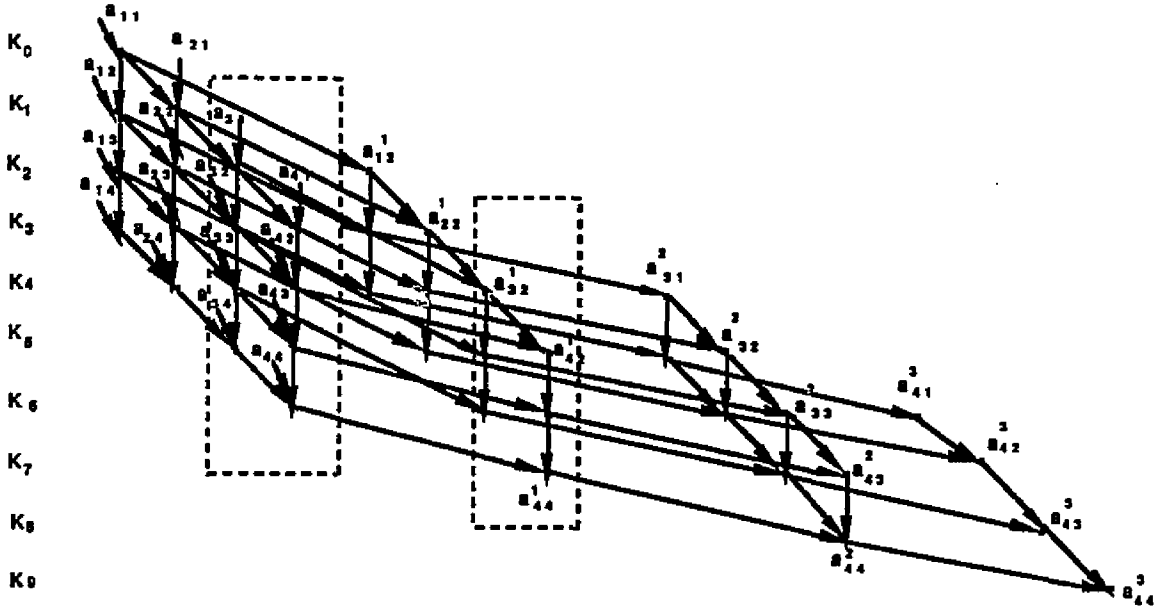

Figure 4.5. Partial TG for QR Array .

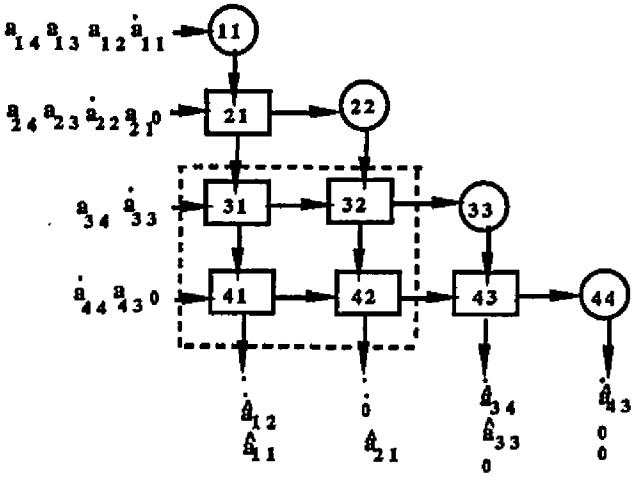

Figure 4.6. Fast QR Array .

4.8. Applications: Design of Smart Structures

The techniques presented in this chapter can be used to design more flexible arrays. The approach expands on the concept of using the compute cycle number as an additional degree of freedom to design a new breed of arrays that are better adapted to the changing nature of the application. Here, the main thrust of the work focuses on the expansion of the class of algorithms executable on a single array.

The concepts introduced in the previous sections are applied to implement different forms of band matrix multiplications on the generic mesh array in a reduced number of cycles. The modified array can evaluate band matrices with execution time that is equal to the time of a specialized architecture. The problem of multiplying randomly sparse matrices more efficiently on a generic array is also considered.

4.8.1. Band Matrix Multiplications on the Mesh Array

The mesh array for matrix multiplication can be modified to implement band matrices. Here, the objective is to trade a delay element from the input data with a zero entry from the band matrix. The collection of all band matrices executable on the array is determined by employing procedure *ATD* . The execution time of the modified array is a function of the bandwidth of the matrices to be multiplied.

For example, consider the mesh array for the case $n = 3$ depicted in Figure 4.7. The partial TG of the array is given in Figure 4.8. Compressing the TG by one level from the top and one level from the bottom leads to the following band matrix multiplication :

$$A = \begin{bmatrix} 0 & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & 0 \end{bmatrix}.$$

Compressing the TG by two levels from the top and one level from the bottom, results in the following structure of band matrix multiplication :

$$A = \begin{bmatrix} 0 & 0 & a_{13} \\ 0 & a_{22} & a_{23} \\ a_{31} & a_{32} & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & b_{13} \\ 0 & b_{22} & b_{23} \\ b_{31} & b_{32} & 0 \end{bmatrix}.$$

Compressing the TG by one level from the top and two levels from the bottom, leads to this form of band matrix multiplications :

$$A = \begin{bmatrix} 0 & a_{12} & a_{13} \\ a_{21} & a_{22} & 0 \\ a_{31} & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & b_{12} & b_{13} \\ b_{21} & b_{22} & 0 \\ b_{31} & 0 & 0 \end{bmatrix}.$$

The modified array execution time is equal to the time of an array specially designed to solve the problem [43]. The specialized structure is a hexagonal array with PEs of fan-in and fan-out of *3*, while the mesh array PEs are characterized by fan-in and fan-out of *2*. To illustrate the concept, the modified partial TG for the third case is given in Figure 4.9. The resulting array is depicted in Figure 4.10. The execution time of the modified array is *5* cycles, while the generic array requires *9* cycles.

4.8.2. Multiplication of Triangular Matrices on the Mesh Array

The generic mesh array can be modified to implement triangular matrix multiplications more efficiently. This objective is easily met by using the TG to identify the time level k at which all the entries in the triangular matrices have contributed to their partial results. Time constraining the array operations for k cycles generates the modified array. The steps necessary for evaluating triangular matrices on the mesh array are given in the next procedure.

Procedure *Triangular*

1- Construct the TG of the array.

2- Apply steps 2, 3, 4 and 5 of procedure *ABU*, with the constraint on the nonzero entries in the triangular matrices.

3- Determine the time level k in the TG that the nonzero entries contributes to the partial results. Stop the array operations at that time level.

The mesh array depicted in Figure 4.7 can be modified to implement triangular matrix multiplication. The new form of the array operations is determined from the partial TG shown in Figure 4.11. In the figure, the TG consists of a total of six levels. Hence, stopping the array at compute cycle $k = 6$ generates the proper results of the multiplication. The generic array requires 9 cycles to perform the same computations.

The previous examples modified the mesh array to implement band (triangular) matrix multiplication. However, some arrays will not admit such modifications. For example, the orbital array [11] does not allow for any zero entries in the matrices that can produce results in a reduced number of cycles. The next lemma provides necessary conditions to determine if an array admits faster realizations .

Lemma 4.1: A linear array designed to implement matrix-matrix multiplication will not admit fast realizations if the resulting TG of the array has nodes corresponding to all the DWCG nodes at all time levels.

Proof : If the TG has nodes corresponding to all PEs in the array then, all data elements are involved at each cycle. Constraining the operations at any time level produces the *zero* matrix as the solution for achieving exact results in a reduced number of cycles.

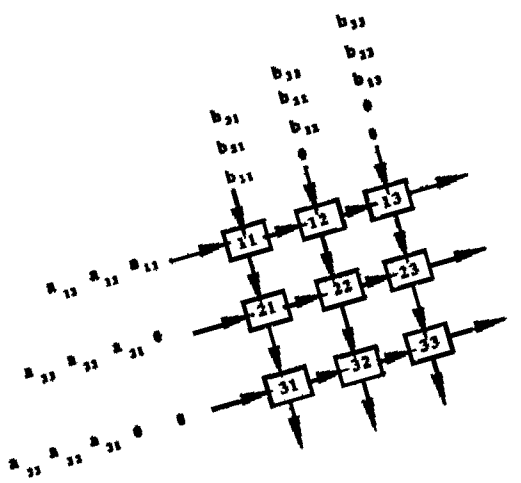

Figure 4.7. 3x3 Mesh Array .

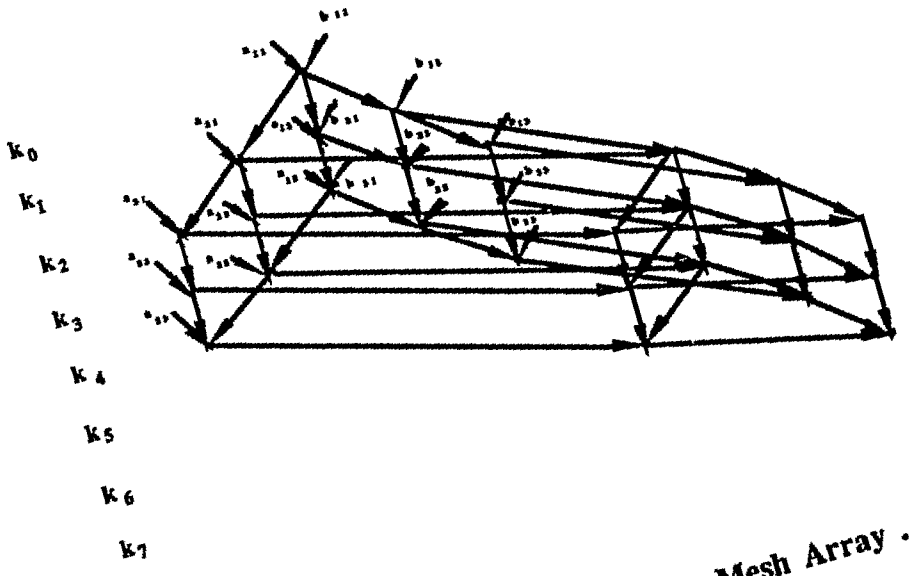

Figure 4.8. Partial TG for the 3x3 Mesh Array .

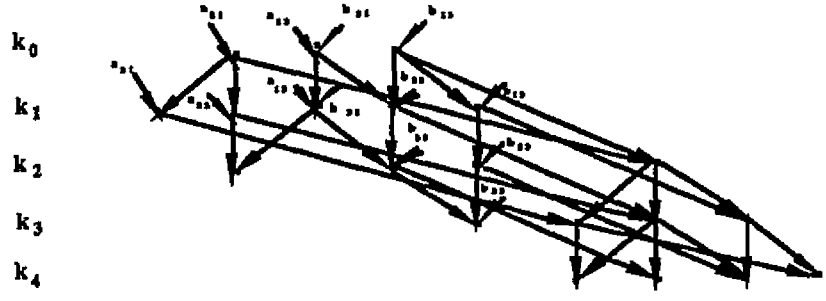

Figure 4.9. Partial TG for Band Matrix Multiplication.

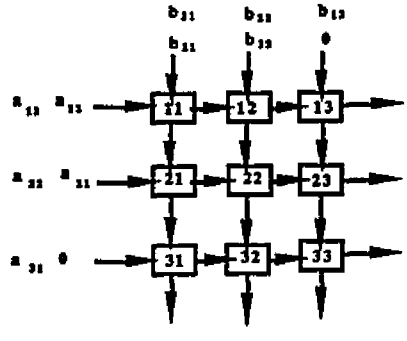

Figure 4.10. Band Matrix Mesh Array .

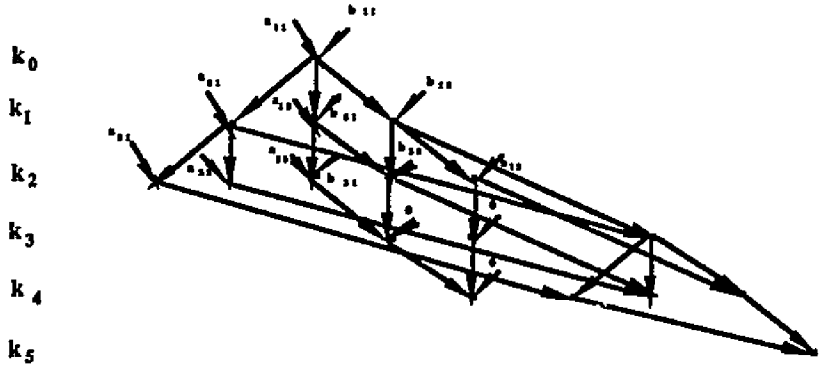

Figure 4.11. Partial TG for Triangular Matrix Multiplication .

4.8.3. Random Sparsity in Matrices

This section considers the problem of multiplying two randomly sparse matrices on a generic array. This problem has received considerable attention [91-92]; several solutions have been proposed, however, all the methods assume a special form of sparsity in the matrices.

In this dissertation a method based on fast realizations is proposed. First , the collection of fast forms executable on the given array is determined. Each form has an associated sparsity pattern. Second, row / column permutations are used to attempt the embedding of the given patterns into one associated to a fast form. The technique assumes that reordering is feasible and can be accomplished off-line. The questions related to on-line shuffling will not be discussed here.

Each row/column permutation is characterized by an orthogonal permutation matrix. If the product to be computed is

$$\mathbf{Y} = \mathbf{AB} ,$$

and all row/column changes in the output are feasible, the collection of possible outputs is given by

$$\mathbf{PYP}_1 = (\mathbf{PAQ})(\mathbf{Q}^* \mathbf{B P}_1) .$$

In this expression , $\mathbf{P}$, $\mathbf{P}_1$ and $\mathbf{Q}$ denote permutation matrices. The next result follows immediately.

Lemma 4.2: Two randomly sparse matrices can be multiplied faster in a generic array if and only if there exist permutation matrices , $\mathbf{P}$, $\mathbf{P}_1$ and $\mathbf{Q}$, that map the sparsities in $\mathbf{A}$ and $\mathbf{B}$ to forms compatible with fast realizations.

Before establishing necessary conditions for the existence of such permutation matrices for the mesh array, the need arise to define the image of a matrix. This is given next.

Definition 4.2 : The image of an nxn matrix $\mathbf{A}$ is an nxn matrix $\mathbf{A}'$, whose element a'_{ij} is given by

$$a'_{ij} = 1 \quad \text{if} \quad a_{ij} \neq 0 \text{ , and}$$
$$= 0 \text{ , otherwise.}$$

The fast forms of the mesh array consist of a collection of band matrices, with a minimum of n entries on the opposite diagonal of the matrix. Let $\mathbf{A}$ be a fast form of the mesh array and $\mathbf{A}'$ be its corresponding image. Furthermore, let $\mathbf{B}$ be a randomly sparse matrix whose image is $\mathbf{B}'$. Let the n - vectors $\mathbf{r}_a$ and $\mathbf{r}_b$ be the vectors formed by finding the summation of entries in each row of $\mathbf{A}'$ and $\mathbf{B}'$ respectively. Similarly, the n - vectors $\mathbf{c}_a$ and $\mathbf{c}_b$ are the vectors formed by finding the summation of the entires in each column of $\mathbf{A}'$ and $\mathbf{B}'$ respectively. Necessary conditions for the existence of the permutation matrices that map $\mathbf{B}'$ to $\mathbf{A}'$ are in lemma 4.3.

Lemma 4.3 : The following are necessary conditions for the existence of permutation matrices that embed a randomly sparse matrix into its fast form

1- There must be a match between the entries of the vectors $\mathbf{r}_a$ and $\mathbf{r}_b$. This means that one vector must be a permutation of the other.

2- There must be a match between the entries of the vectors $\mathbf{c}_a$ and $\mathbf{c}_b$. This means that one vector must be a permutation of the other.

3- The absolute value of the determinant of $\mathbf{A}'$ and $\mathbf{B}'$ must be equal.

Proof : Let $\mathbf{P}$ and $\mathbf{Q}$ be two permutation matrices, then one can write $\mathbf{PA'Q} = \mathbf{B'}$. Each permutation matrix is orthogonal, hence $\mathbf{Q}\,\mathbf{Q}^{*} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Conditions 1 and 2 follows immediately from the identity $\mathbf{PA'} = \mathbf{B'Q}^{*}$. In addition, the absolute value of the determinant of a permutation matrix is equal to one, hence condition three results.

The above conditions must be satisfied before an attempt can be made to find the permutation matrices $\mathbf{P}$, $\mathbf{P}_1$ and $\mathbf{Q}$.

The chapter has formalized the concept of fast realizations. Systematic procedures have been introduced to analyze the operations of a large class of arrays. The concept enlarges the class of algorithms executable on a single array. The modified arrays are more adaptable to the changing nature of the applications.

CHAPTER 5

FAST LINEAR ALGORITHM IMPLEMENTAIONS

The previous chapter used the concept of *Timing Graph* to analyze the operation of an array as a function of the compute cycle. In particular , several fast realizations have been found for linear arrays . Our interest in matrix-matrix arrays results from the observation that many signal processing and parameter estimation algorithms utilize those operations. For example , a linear filtering problem can be expressed in the form of a matrix product. The filter can be constrained to different forms of band matrices and still yields acceptable results (FIR filters) .

The concept of fast realizations restrict the operations to remain in the same class. A matrix-matrix multiplication remains as such; Only selected entries are omitted to speed-up the execution time. This chapter takes the concept of fast realizations one step further. The basic constraint is the number of compute cycles. As long as that constraint is satisfied there is no condition on the actual mathematical representation. For example, it has been shown that an orbital array with complete preloading does not have fast realizations of matrix products. It is clear, however, that the operation of the array can be stopped earlier. The resulting computation is a linear transformation but it is not a matrix product.

A mathematical model of a transformation is presented. The proposed representation is general enough to encompass a large class of signal processing and control applications. At the same time, it provides sufficient structure for a systematic treatment. The array implementation is considered and the operations are parameterized by the number of the compute cycle. A model of the evolution of partial transformations in such arrays, as time progresses, is developed.

The chapter is organized as follows : Section 5.1 introduces the class of algorithms. Section 5.2 develops the necessary and sufficient conditions for the existence of fast realizations of bilinear algorithms. Section 5.3 reviews the results of parallel implementation of linear algorithms. Section 5.4 analyzes the operations of time constrained arrays. Examples include structures for matrix-matrix computations and the triple matrix product. Section 5.5 gives examples of fast forms for the triple matrix product.

5.1. The Class of Algorithms

This section considers a class of compute bound algorithms that can be modeled by the map

$$\mathbf{y} = \Lambda(\mathbf{p}, \mathbf{x}). \qquad (5.1)$$

In this expression, $\mathbf{x} \in E^n$ represents external data to be processed. The vector $\mathbf{p} \in E^m$ corresponds to the parameters defining the map. The proposed map is general enough to encompass a large class of algorithms arising from signal processing applications. The

family of algorithms has sufficient structure for a systematic mathematical treatment. This class of algorithms admits implementations in regular arrays.

Clearly, algorithms based on matrix-vector and matrix-matrix multiplications are a special case of the class introduced. For example, if in the above equation , Λ corresponds to matrix-multiplication then, the vectors $\mathbf{p}$ and $\mathbf{x}$ would contain the entries of the matrices to be multiplied.

Assume that $\Lambda(\mathbf{p}, \mathbf{x})$ is a bilinear transformation, i.e. , the maps $\Lambda(., \mathbf{x})$ and $\Lambda(\mathbf{p}, .)$ are both linear. It is well known [93-96] that in this case there exists a unique linear transformation $\mathbf{L}$, defined on the tensor product space $E^m \otimes E^n$, satisfying

$$\mathbf{y} = \mathbf{L}(\mathbf{p} \otimes \mathbf{x}) . \qquad (5.2)$$

The notation $\mathbf{p} \otimes \mathbf{x}$ indicates the tensor product of the corresponding vectors. Using standard properties of the tensor products one can write

$$\mathbf{p} \otimes \mathbf{x} = (\mathbf{p} \otimes \mathbf{I})(1 \otimes \mathbf{x}) ,$$

where $\mathbf{I}$ is the nxn identity matrix. Replacing the expression for $\mathbf{y}$ in Equation (5.2) gives

$$\mathbf{y} = [\mathbf{L}(\mathbf{p} \otimes \mathbf{I})]\mathbf{x} . \qquad (5.3)$$

The matrix $\mathbf{L}$ is of dimension $nxnm$, while $\mathbf{p} \otimes \mathbf{I}$ is of dimension $mnxn$. Hence, the expression in the square bracket is an nxn matrix. Thus, the following is established.

Lemma 5.1: The collection of bilinear algorithms of the form

$$\mathbf{y} = \Lambda (\mathbf{p}, \mathbf{x}), \quad \mathbf{x} , \mathbf{y} \text{ in } E^n, \quad \mathbf{p} \text{ in } E^m ,$$

corresponds to the class of matrix-vector multiplications of the form

$$\mathbf{y} = \mathbf{A} (\mathbf{p}) \mathbf{x} .$$

Each entry of the matrix $\mathbf{A} (\mathbf{p})$ is a linear function of the vector $\mathbf{p}$.

To allow for a systematic treatment of the representation, one must determine the general expression of the entries in the matrix $\mathbf{A}$. From Equation (5.3),

$$\mathbf{A} (\mathbf{p}) = \mathbf{L} (\mathbf{p} \otimes \mathbf{I}) . \qquad (5.4)$$

The $mnxn$ matrix $\mathbf{p} \otimes \mathbf{I}$ has the representation

$$\mathbf{p} \otimes \mathbf{I}_n = Diag (\mathbf{p} , \mathbf{p} , \ldots , \mathbf{p}) .$$

If the $nxnm$ matrix $\mathbf{L}$ is written in a partitioned form using $1xm$ row matrices $\mathbf{l}_{ij}$, then the indicated product in equation (5.4) enables one to show the following result.

Lemma 5.2: For the bilinear algorithms described in Lemma *5.1*, the matrix $\mathbf{A}(\mathbf{p})$ has entries

$$a_{ij}(\mathbf{p}) = < \mathbf{l}_{ij}, \mathbf{p} > .$$

Remark: The result in this lemma is particularly useful for the analysis of possible fast forms. It is apparent that the constraints required by the fast realization define a linear set of equations on the parameter vector $\mathbf{p}$.

5.2. Fast Realizations of Bilinear Algorithms

The previous section has established that the collection of bilinear algorithms defined by the map $\Lambda(.,.)$ corresponds to a class of matrix-vector multiplication. In this regard, it is possible to assume that a matrix-vector array is the vehicle used to implement the computations. The array can be thought of as an architecture characterized by factors such as : the allowable number of compute cycles (T), the possible functions executable in each processing element (FI), the array connectivity matrix ($\mathbf{P}$) and the available bandwidth (IO) capabilities.

In view of lemmas *5.1* and *5.2*, for the case of bilinear algorithms, the determination of fast realizations is reduced to the problem of finding fast forms for the matrix-vector

array executing the multiplication . The conditions for finding fast forms for bilinear algorithms are summarized in the next theorem.

Theorem 5.1 : Assume that the matrix vector product $\mathbf{y} = \mathbf{Ax}$ can be executed faster in a given array if

$$a_{ij} = 0 \quad \text{for} \quad (i,j) \quad \varepsilon \; S \; ,$$

where S defines the sparsity pattern associated to the fast form. The bilinear algorithm $\mathbf{y} = \Lambda (\mathbf{p} , \mathbf{x})$ will have a fast realization in such an architecture if and only if there is a nontrivial solution to the set of equations expressed by

$$< \mathbf{l}_{ij} \; , \mathbf{p} \; > = 0 \quad , \quad (i,j) \; \varepsilon \; S \; .$$

Proof : The proof follows directly from Lemmas *5.1* and *5.2* .

In this case any nontrivial solution for $\mathbf{p}$ leads to a fast bilinear realization executable on the architecture in the same number of compute cycles. The vectors $\mathbf{l}_{ij}$ are determined as in Lemma *5.2* .

5.2.1. Example to Illustrate Fast Bilinear Algorithms

In this example, Lemmas *5.1* , *5.2* and Theorem *5.1* are applied to derive a fast form of a typical bilinear algorithm. Given the results in Lemma *5.1* , one can assume a matrix-vector array as the vehicle to execute the computations. One possible array that

performs the operation $\mathbf{y} = A\mathbf{x}$ for $n = 3$, was introduced in chapter 3, Figure 3.2. The execution time of the array is 5 cycles.

Assume that the bilinear transformation is of the form

$$\mathbf{y} = \mathbf{c}\,\mathbf{x}^{*}\,\mathbf{b},$$

where, $\mathbf{c}, \mathbf{x}, \mathbf{b} \in E^{m}$ and '$*$' means the vector transpose. The transformation can be expressed in this form

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} c_1 b_1 & c_1 b_2 & c_1 b_3 \\ c_2 b_1 & c_2 b_2 & c_2 b_3 \\ c_3 b_1 & c_3 b_2 & c_3 b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\dot{\mathbf{y}} = \begin{bmatrix} p_1 & p_4 & p_7 \\ p_2 & p_5 & p_8 \\ p_3 & p_6 & p_9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

If 4 cycles are available to output the results, then a fast realization for the array is possible provided that $a_{33} = 0$. Depending on the number of parameters one may have a subspace of solutions. However, if each entry in $\mathbf{L}(\mathbf{p})$ can be selected independently then, $\mathbf{p} \in E^{9}$, and $\mathbf{p}$ is given by

$$\mathbf{p} = (p_1 , p_2 , p_3 , p_4 , p_5 , p_6 , p_7 , p_8 , p_9)^{*} .$$

Using the properties of the tensor product the matrix $\mathbf{L} \in E^{3x27}$ is given by

$$\mathbf{L} = \begin{bmatrix} 100 & 000 & 000 & 010 & 000 & 000 & 001 & 000 & 000 \\ 000 & 100 & 000 & 000 & 010 & 000 & 000 & 001 & 000 \\ 000 & 000 & 100 & 000 & 000 & 010 & 000 & 000 & 001 \end{bmatrix} .$$

From equation *5.4* , $(\mathbf{p} \otimes \mathbf{I}) \in E^{27x3}$ and from lemma *5.2* , the following can be established

$$a_{33} = < \mathbf{l}_{33} , \mathbf{p} > = 0 .$$

The row vector $\mathbf{l}_{33}$ has the following form

$$\mathbf{l}_{33} = (0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1).$$

Then , if $p_9 = c_3 b_3 = 0$ (either $c_3 = 0$ or $b_3 = 0$) a fast realization of the bilinear algorithm can be implemented exactly on the matrix-vector array in *4* cycles .

As a second example, assume that a *3x3* cylindrical array is implementing the operation . Here, the computations of different vectors could be interleaved. Assume that

4 cycles must produce the final results. The fast form of the array is achieved provided that $a_{13} = a_{23} = a_{33} = 0$. Here, $\mathbf{l}_{13}$, $\mathbf{l}_{23}$ and $\mathbf{l}_{33}$ are given by

$$\mathbf{l}_{13} = (0,0,0, 0, 0, 0, 1, 0, 0),$$

$$\mathbf{l}_{23} = (0,0,0, 0, 0, 0, 0, 1, 0),$$

$$\mathbf{l}_{33} = (0,0,0 , 0, 0, 0,0, 0, 1).$$

From Theorem 5.1 , solving the system of equations results in

$$< \mathbf{l}_{13} , \mathbf{p} > = c_1 b_3 = 0 ,$$

$$< \mathbf{l}_{23} , \mathbf{p} > = c_2 b_3 = 0 , \text{and}$$

$$< \mathbf{l}_{33} , \mathbf{p} > = c_3 b_3 = 0 .$$

Thus, if $b_3 = 0$ a fast realization of the bilinear algorithm does exist in 4 cycles .

5.3. Parallel Implementation of Linear Algorithms

Many signal extraction , parameter estimation and automatic control algorithms utilize matrix-vector operations. The most efficient arrays perform matrix-vector operations in $O(n)$ time units . For very critical real time applications , and large value of n , even

this response time may not be sufficient . This section presents a procedure to enhance concurrency and increase compute speed.

Recently, in [39] it has been shown that every linear algorithm can be decomposed as a finite sum of a special class of algorithms. This special class of linear algorithms permits computations in $O(\sqrt{n})$ time. The approach considers the matrix-vector operation

$$\mathbf{y} = \mathbf{A}\ \mathbf{x} \quad , \qquad (5.5)$$

where, $\mathbf{A} \in E^{mn}$ and $\mathbf{y}$, $\mathbf{x}$ are n - vectors with $n = p\,q$.

Let $\mathbf{X} \in E^{pxq}$ be a matrix formed by mapping the vector $\mathbf{x}$ on a two dimensional array. This can be done by using segments of $\mathbf{x}$ as columns of $\mathbf{X}$. Similarly, let $\mathbf{Y} \in E^{pxq}$ be a matrix formed from the vector $\mathbf{y}$. In [39], it is shown that equation *5.5* can be represented as a parallel implementation of triple matrix products in the form

$$\mathbf{Y} = \Sigma\ \mathbf{L}_i \mathbf{X} \mathbf{R}_i \quad , i = 1, \ldots, l \quad . \qquad (5.6)$$

with $l \leq \min(p^2, q^2)$, $\mathbf{R}_i \in E^{qxq}$ and $\mathbf{L}_i \in E^{pxp}$. In equation 5.6 , the matrices $\mathbf{L}_i$ and $\mathbf{R}_i$ are determined by the the linear transformator $\mathbf{A}$. In many important cases the number of terms l is small and all triple products can be executed concurrently.

If pq multipliers are available, and all the data elements can be loaded simultaneously to the array, equation *5.5* can be evaluated in $T_1 = pq$ cycles. On the other hand, the triple matrix product can be executed in parallel in a total of $T_2 = q + 2p - 1$ cycles. However, if only one array is used then the total execution time is $T_2 = l\ (q + 2p - 1)$ cycles. Even the sequential execution of the triple matrix product may be faster than the matrix-vector case. If $p = q = \sqrt{n}$, then T_2 is less than T_1 provided that $l < \sqrt{n}/3$. The representation will be *time efficient* if $l\ (q + 2p - 1) << p\,q$.

The speed of the sequential execution of the triple matrix product representation of the matrix-vector multiplication is a function of the parameter l. A smart choice of l, for a given p and q, leads to a faster execution. In [39], an algorithm is presented that finds the minimum number of left matrices $\{ \mathbf{L}_i \}$.

5.4. Time Constrained Implementations

This section develops a recursive model of the evolution of linear transformations in linear arrays, as a function of the compute cycle. The objective here is the determination of the form of the transformation implemented by the array at each cycle.

The initial development will consider the simple case of linear transformations modeled as matrix-matrix products. The analysis will then be expanded to model the evolution of transformations represented by triple matrix products. Since these later maps can be used for a parallel representation of general linear transformations, it will be possible to implement time constrained operations for the general case.

5.4.1. The Basic Cases

In this section , our attention is focused on arrays implementing linear transformations that can be described as the product of two matrices. Utilizing the fact that matrices can also be viewed as elements in their corresponding vector space [96], it is convenient to represent a matrix as an n^2 vector by arranging the data columnwise. Using the properties of the tensor products [96], the linear transformation expressed in equation *5.6* can be described as

$$\begin{aligned} \mathbf{y}^{\circ} &= (\mathbf{I} \otimes \mathbf{A})\,\mathbf{x}^{\circ}, \\ &= \mathring{\mathbf{A}}\,\mathbf{x}^{\circ}, \end{aligned} \qquad (5.7)$$

where $\mathring{\mathbf{A}}$ is an $n^2 x n^2$ matrix , '$\otimes$' indicates the tensor product , $\mathbf{I}$ is the nxn identity matrix and $\mathbf{x}^{\circ}, \mathbf{y}^{\circ} \varepsilon E^{n^2}$ are the vector representation of the matrices $\mathbf{X}$, $\mathbf{Y}$ by arranging the data elements columnwise.

A similar equation can be derived for the linear transformation of the form $\mathbf{Y} = \mathbf{X}\,\mathbf{A}$, which is given by

$$\begin{aligned} \mathbf{y}^{\circ} &= (\mathbf{A} \otimes \mathbf{I})\,\mathbf{x}, \\ &= \mathring{\mathbf{A}}\,\mathbf{x}^{\circ}. \end{aligned} \qquad (5.8)$$

Given the time sequential operations of the systolic array, it is possible to obtain a recursive representation of the evolution of partial transformations in the array, parameterized by k . Given the linearity of the transformations it is possible to write

$$\mathbf{y}^{\circ}_{k} = \mathring{\mathbf{A}}_{k}\, \mathbf{x}^{\circ}$$

Here, $\mathring{\mathbf{A}}_{k}$ represents the transformation on the input data after k cycles. The final transformation performed by the array on the input data elements is given by $\mathring{\mathbf{A}}_{T} = \mathring{\mathbf{A}}$, where T is the total number of allowable compute cycles. The evolution of entries in the selection matrix $\mathring{\mathbf{A}}_{k}$, $k = 0, 1, \ldots, T$, is a function of the architecture evaluating the linear transformation. In the following, the partial transformations implemented by the mesh and the cylindrical arrays are considered.

5.4.1.1. Mesh Array Partial Transformations

The example presents the partial transformations of a mesh array evaluating the product $\mathbf{Y} = \mathbf{AX}$, where $\mathbf{A}$ represents a transformation to be performed on the elements of the data matrix $\mathbf{X}$. For simplicity the case $n = 2$ is considered. The mesh array is depicted in Figure 3.6 . The PTD of the array is given in Figure 3.8. Reading the arcs of the nodes holding the a_{ij} entries at levels { $0, 1, 2, 3$ }, results in the partial transformations listed in Figure 5.1 (a , b, c , d) . The transformation implemented at compute cycle $k = 3$ corresponds to the matrix product operation .

5.4.1.2. Cylindrical Array Transformations

This example derives the partial transformations of a cylindrical array designed to implement the linear transformation $\mathbf{Y} = \mathbf{A}\,\mathbf{X}$, for $n = 2$. The array is depicted in Figure 3.9 and the resulting PTD is given in Figure 3.10. Reading the arcs of the nodes holding the a_{ij} entries at levels { $0, 1, 2$ } produces the total possible transformations

executed by the array . This is illustrated in Figure 5.2 (a , b , c) . The transformation generated at cycle $k = 2$ corresponds to the matrix product .

A similar analysis leads to the collection of partial transformations depicted in Figure 5.3 (a , b , c) , for the case when the array is implementing the linear transformation $\mathbf{Y} = \mathbf{XA}$. The transformation at $k = 2$ is equal to the indicated matrix product.

Remark : A simple examination of the partial transformations implemented by the arrays in the previous examples for $k < T$, where T is the number of compute cycles, indicates that none of the transformations can be expressed as a tensor product. Hence , if the operations are constrained to k cycles , where $k < T$, a transformation different from the matrix product is performed by the array . If exact results are required in k cycles, then the transformation $\mathring{\mathbf{A}}_k$ describes the structure of the matrix $\mathbf{A}$ that makes this possible.

5.4.2. Triple Matrix Product

In this section, the triple matrix product identity of the form $\mathbf{Y} = \mathbf{LXR}$ is considered. Here, the two matrices $\mathbf{L}$ and $\mathbf{R}$ ϵ E^{nxn} represent a linear transformation to be performed on the nxn data matrix $\mathbf{X}$.

$$A_0 = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_1 = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & 0 & 0 & 0 \\ 0 & 0 & a_{11} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

a) *b)*

$$A_2 = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & a_{21} & 0 \end{bmatrix}, \quad A_3 = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & a_{21} & a_{22} \end{bmatrix}.$$

c) *d)*

Figure 5.1. Mesh Array Y = AX , Transformations

a) k = 0 , b) k = 1 ,

c) k = 2 , d) k = 3 .

$$A_0 = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & a_{21} & 0 \end{bmatrix}, \quad A_1 = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & 0 & 0 & 0 \\ 0 & 0 & a_{11} & 0 \\ 0 & 0 & a_{21} & a_{22} \end{bmatrix},$$

a) b)

$$A_2 = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & a_{21} & a_{22} \end{bmatrix}.$$

c)

Figure 5.2. Cylindrical Array Y = AX

Transformations . a) k = 0 ,

b) k = 1 , c) k = 2 .

$$A_0 = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{12} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_1 = \begin{bmatrix} a_{11} & 0 & a_{12} & 0 \\ 0 & a_{11} & 0 & a_{12} \\ a_{21} & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{22} \end{bmatrix},$$

a) b)

$$A_2 = \begin{bmatrix} a_{11} & 0 & a_{12} & 0 \\ 0 & a_{11} & 0 & a_{12} \\ a_{21} & 0 & a_{22} & 0 \\ 0 & a_{21} & 0 & a_{22} \end{bmatrix}.$$

c)

Figure 5.3. Cylindrical Array Y = XA

Transformations . a) k = 0 ,

b) k = 1 , c) k = 2 .

The parallel representation of the matrix-vector identity requires efficient architectures for the computation of a triple matrix product. The most convenient form of implementing the product consists of spliting the operations into two stages. In the first stage, the product

$$\mathbf{Z} = \mathbf{XR} \ ,$$

is performed. The second stage implements the product

$$\mathbf{Y} = \mathbf{LZ} \ .$$

Using the same convention of representing the matrices as column vectors by arranging the data entries columnwise and utilizing the properties of the tensor product, the sequential algorithm for evaluating the triple matrix product (TMP) identity can be expressed as

$$\begin{aligned} \mathbf{z}^{\circ} &= (\mathbf{R} \otimes \mathbf{I}) \mathbf{x}^{\circ} \ , \\ \mathbf{y}^{\circ} &= (\mathbf{I} \otimes \mathbf{L}) (\mathbf{R} \otimes \mathbf{I}) \mathbf{x}^{\circ} \ , \\ \mathbf{y}^{\circ} &= \mathring{\mathbf{A}} \, \mathbf{x}^{\circ} \ , \end{aligned} \qquad (5.9)$$

where $\mathring{\mathbf{A}}$ is an $n^2 x n^2$ matrix , '$\otimes$' indicates the tensor product , $\mathbf{I}$ is the nxn identity matrix and $\mathbf{x}^{\circ}$, $\mathbf{y}^{\circ}$, $\mathbf{z}^{\circ} \in E^{n^2}$ are the vector representation of the matrices $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ by arranging the data elements columnwise. After k cycles the recursive evolution of partial results in the array is expressed by

$$\mathbf{y}^{\circ}_{k} = \mathring{\mathbf{A}}_{k} \, \mathbf{x}^{\circ} \ .$$

The form of $\mathring{A}_k$ is architecture dependent. Special cases will be analyzed in the next section.

5.4.3. Time Constrained Forms of the Triple Matrix Product

The cylindrical array can be modified to implement the sequential TMP algorithm. Minimization of I/O operations is achieved by using PEs with two distinct modes of operations . In the first mode , the PE is capable of multiplying the incoming data and accumulates the partial results. In the second mode , the PE multiplies the incoming data by a stored constant and transmits the partial results . The array for $n=2$ and the PE modes of operation are depicted in Figure 5.4 .

The operation can be pipelined for better PE utilization. This is achieved by switching the first row of PEs in the array to mode 2 as soon as the first data elements from the first row of **Z** are computed. Similarly, row 2 is switched to mode 2 as soon as it finishes contributing to the partial z_{ij} results. In total, it takes n cycles to operate the array in mode 2 . However, to keep the architecture in its simplest form, it is assumed that the array will operate in mode 1 for $2n-1$ cycles to compute the matrix **Z**, and then it will switch globally to mode 2 to compute **Y**. In this case, the total execution time is equal to $4n-2$ cycles .

The evolution of partial transformations in the array for $\mathbf{Z = XR}$ as a function of the compute cycle is depicted in Figure 5.5 (a , b , c), for compute cycles $k = 0, 1, 2$. Similarly, the evolution of partial transformations for $\mathbf{Y = LZ}$ is depicted in Figure 5.6 (a , b , c), for compute cycles $k = 3, 4, 5$. If the array operations are to be

constrained, then by a simple inspection one can conclude that none of the partial results for $k < 5$ can be expressed in the form of a tensor product. Hence, a transformation different from the triple matrix identity is performed by the array.

In a similar fashion, the orbital array can be modified to implement the sequential TMP algorithm. The array for computing the TMP for $n = 2$ is depicted in Figure 5.7. The PEs will have two mode of operations. The PEs will operate in mode 1 for the first n cycles to evaluate the product $\mathbf{Z} = \mathbf{XR}$. The PEs will switch globally to mode 2 of operations for the next n cycles to evaluate the product $\mathbf{Y} = \mathbf{LZ}$. In general, it takes the preloaded array $2n$ cycles to evaluate the sequential TMP algorithm.

The partial transformations for the $\mathbf{Z} = \mathbf{XR}$ case are depicted in Figure 5.8, and the partial transformations for the case $\mathbf{Y} = \mathbf{LZ}$ are depicted in Figure 5.9. Similarly, by a simple inspection one can conclude that none of the partial results for $k < 3$ can be expressed in the form of a tensor product. Hence, a transformation different from the triple matrix identity is performed by the array.

Two common techniques to increase the power of an array are reconfiguration capability and multi-function PEs. Further use of these techniques provides greater flexibility. The ability of the PE to change its function can be used to advantage in generating new forms of transformations. As an example, for the orbital array implementing the sequential TMP algorithm, if it is possible to alternate the PE mode of operation at any time, a large number of new transformations can be generated.

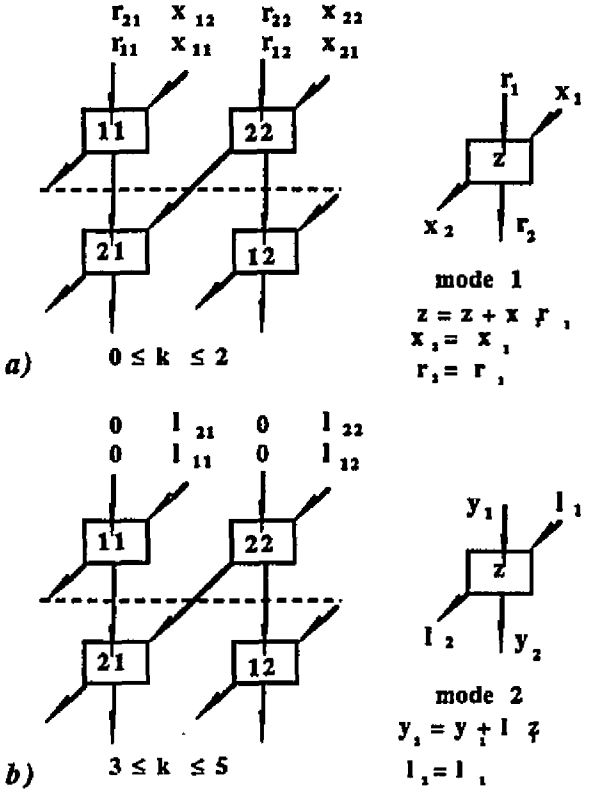

Figure 5.4. Cylindrical Array for TMP.

$$\hat{A}_0 = \begin{bmatrix} r_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & r_{12} & 0 & 0 \end{bmatrix}, \quad \hat{A}_1 = \begin{bmatrix} r_{11} & 0 & r_{21} & 0 \\ 0 & r_{11} & 0 & 0 \\ r_{12} & 0 & 0 & 0 \\ 0 & r_{12} & 0 & r_{22} \end{bmatrix}, \quad \hat{A}_2 = \begin{bmatrix} r_{11} & 0 & r_{21} & 0 \\ 0 & r_{11} & 0 & r_{21} \\ r_{12} & 0 & r_{22} & 0 \\ 0 & r_{12} & 0 & r_{22} \end{bmatrix}.$$

a) b) c)

Figure 5.5. Cylindrical Array Transformations for Z=XR, a) k = 0 , b) k = 1 , c) k = 2 .

$$\hat{A}_3 = \begin{bmatrix} r_{11}l_{11} & 0 & r_{21}l_{11} & 0 \\ 0 & r_{11} & 0 & r_{21} \\ r_{12} & r_{12}l_{12} & r_{22} & \zeta_{22}l_{12} \\ 0 & r_{12} & 0 & r_{22} \end{bmatrix}, \quad \hat{A}_4 = \begin{bmatrix} r_{11}l_{11} & r_{11}l_{12} & r_{21}l_{11} & r_{21}l_{12} \\ r_{11}l_{21} & r_{11} & r_{21}l_{21} & r_{21} \\ r_{12}l_{11} & r_{12}l_{12} & r_{22}l_{11} & \zeta_{22}l_{12} \\ 0 & r_{12}l_{22} & 0 & r_{22}l_{22} \end{bmatrix},$$

a) b)

$$\hat{A}_5 = \begin{bmatrix} r_{11}l_{11} & r_{11}l_{12} & r_{21}l_{11} & r_{21}l_{12} \\ r_{11}l_{21} & r_{11}l_{22} & r_{21}l_{21} & r_{21}l_{22} \\ r_{12}l_{11} & r_{12}l_{12} & r_{22}l_{11} & \zeta_{22}l_{12} \\ r_{12}l_{21} & r_{12}l_{22} & r_{22}l_{21} & r_{22}l_{22} \end{bmatrix}.$$

c)

Figure 5.6. Cylindrical Array Transformations for Y=LZ, a) k = 3 , b) k = 4 , c) k = 5 .

Assuming that the array is operated in such a way that the PEs will alternate their mode of operations every other cycle, the transformation depicted in Figure 5.10 will be generated. This transformation is different from the ones generated previously. Hence, the power of an array can be increased by taking advantage of the ability of the processing elements to change their mode of operations.

5.4.4. Fast Realizations of the Triple Matrix Product Algorithm

The matrix-vector identity can be expressed as the summation of small TMP products. The speed of the implementation depends on the number of products to be performed and the number of arrays that are performing the operation. Let β be the time required to compute one TMP and m be the number of available arrays. Furthermore, let T_p be the time of computing the l TMPs. Then, the lower bound for T_p is β for $m = l$. However, when $l > m$, the required time is $T_p = [\, l/m \,]\, \beta$, where $[\,.\,]$ indicates the largest integer function. The upper bound is $T_p = l\beta$ for $m = l$. Hence, in general $\beta < T_p < l\beta$ for $1 \leq m \leq l$.

If the operations are to be constrained to $k < T_p$ cycles, a simple inspection of the partial transformations presented in the previous section indicates that a transformation different from the TMP is instead executed by the array. However, if the transformation at compute cycle k must be exact, then it is possible to find the fast forms of the TMP. In this regard, the TG can be used to derive the fast forms of the array. The objectives here are the determination of those special forms of the left and right matrices (**L** and **R**) that admits computations in $\beta' < \beta$ for each TMP.

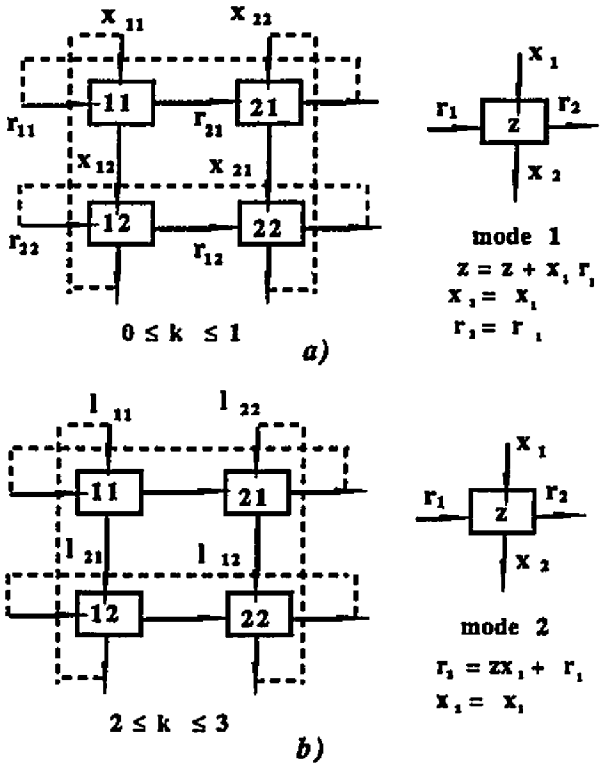

Figure 5.7. Orbital Array for TMP.

$$A_0 = \begin{bmatrix} r_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & r_{21} \\ 0 & 0 & r_{22} & 0 \\ 0 & r_{12} & 0 & 0 \end{bmatrix}, \quad A_1 = \begin{bmatrix} r_{11} & 0 & r_{21} & 0 \\ 0 & r_{11} & 0 & r_{21} \\ r_{12} & 0 & r_{22} & 0 \\ 0 & r_{12} & 0 & r_{22} \end{bmatrix}.$$

a) *b)*

Figure 5.8. Orbital Array Transformations for Z=XR, a) k = 0, b) k = 1 .

$$A_2 = \begin{bmatrix} r_{11}l_{11} & 0 & r_{21}l_{11} & 0 \\ 0 & r_{11}l_{22} & 0 & r_{21}l_{22} \\ r_{12} & r_{12}l_{12} & r_{22} & r_{22}l_{12} \\ r_{12}l_{21} & r_{12} & r_{22}l_{21} & r_{22} \end{bmatrix}, \quad A_3 = \begin{bmatrix} r_{11}l_{11} & r_{11}l_{12} & r_{21}l_{11} & r_{21}l_{12} \\ r_{11}l_{21} & r_{11}l_{22} & r_{21}l_{21} & r_{21}l_{22} \\ r_{12}l_{11} & r_{12}l_{12} & r_{22}l_{11} & r_{22}l_{12} \\ r_{12}l_{21} & r_{12}l_{22} & r_{22}l_{21} & r_{22}l_{22} \end{bmatrix}.$$

a) *b)*

Figure 5.9. Orbital Array Transformations for Y=LZ, a) k = 2 , b) k = 3 .

$$A = \begin{bmatrix} l_{11}l_{12}r_{11} & l_{12}r_{11} & 0 & 0 \\ 0 & 0 & l_{21}r_{21} & l_{21}l_{22}r_{21} \\ 0 & 0 & l_{21}l_{22}r_{12} & l_{22}r_{22} \\ l_{11}r_{12} & l_{11}l_{12}r_{12} & 0 & 0 \end{bmatrix}.$$

Figure 5.10. Orbital Array Final Transformation with Alternating PE Modes of Operations.

Consider the partial TG of the cylindrical array for the TMP depicted in Figure 5.11. The array requires 6 cycles to evaluate the product for $n = 2$. Assume that only 5 cycles are available to perform the computations. Due to the sequential nature of algorithm evaluating the indicated product, the saving in time can result from constraining the **Z** matrix operations and (or) constraining the **Y** matrix operations. The analysis is based on the assumption that the designer does not have control over the data matrix entries.

Utilizing the concept of TG compression, the fast forms obtained by constraining the matrix **R** for compute cycles $0 \leq k \leq 2$ are given by

$$\begin{bmatrix} r_{11} & r_{12} \\ 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 0 & 0 \\ r_{21} & r_{22} \end{bmatrix}.$$

In the TG, time levels $3 \leq k \leq 5$ represent the contribution of the $l_{ij} \in \mathbf{L}$ entries to the $y_{ij} \in \mathbf{Y}$ partial results. The PEs at these time levels are operating in mode 2. To constrain the **Y** matrix operations , two choices are available. First, compressing the TG by one level from the bottom results in the following two conditions:

i) $l_{22} = 0$ or $z_{11} = 0$ and $y_{21}^{1} = 0$, and

ii) $l_{21} = 0$ or $z_{12} = 0$ and $y_{22}^{1} = 0$.

However, $y_{21}^{1} = l_{21}\ z_{11}$ implies that y_{21}^{1} is zero provided that $l_{21} = 0$ or $z_{11} = 0$. Similarly, $y_{22}^{1} = l_{22}\ z_{22}$ and y_{22}^{1} are zero provided that $l_{22} = 0$ or $z_{22} = 0$. In this case, the two conditions can be restated as

i) $l_{22} = 0$ or $z_{11} = 0$ and $l_{21} = 0$ or $z_{11} = 0$, and

ii) $l_{21} = 0$ or $z_{12} = 0$ and $l_{22} = 0$ or $z_{22} = 0$.

The condition $z_{11} = 0$ is satisfied provided that $r_{11} = r_{21} = 0$. The entries z_{12} and z_{22} will be zero if $r_{21} = r_{22} = 0$.

The second option considers compressing the TG by one level from the top. In this regard, the following conditions must hold :

i) $l_{11} = 0$ and $l_{12} = 0$ or $z_{11} = 0$ and $z_{22} = 0$,

ii) $z_{21} = 0$ and $y_{11}^{1} = 0$, and

iii) $z_{12} = 0$ and $y_{12}^{1} = 0$.

The collection of **L** and **R** matrices that results in realizations in 5 cycles is determined by taking the combinations of all the above conditions. Certainly , a wide range of choices is obtained. The same analysis can be carried for different values of the constrained cycle k . The overall savings in time is a function of l , m and β'. Examples of fast forms for bottom up compression of the TG are given in Figure 5.12.

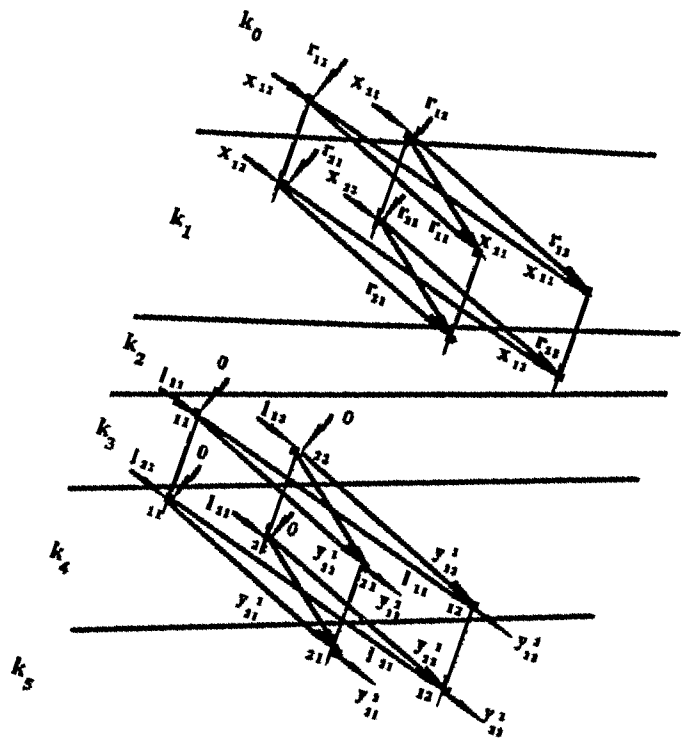

Figure 5.11. Partial TG for TMP Cylindrical Array .

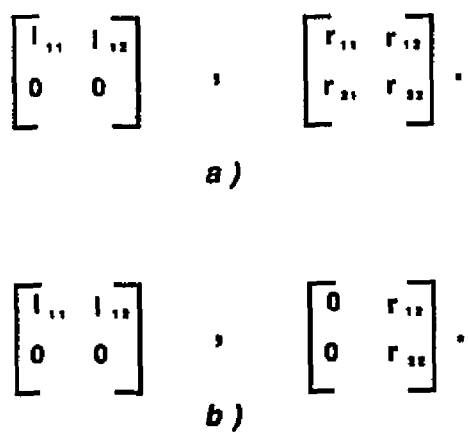

Figure 5.12. Fast Forms for TMP Cylindrical Array ,

a) Constraining matrix L , b) Constraining L and R matrices.

The chapter has derived a recursive model of the transformations generated in a class of linear arrays as functions of the compute cycle. It was shown that, by taking advantage of the compute cycle as an added degree of freedom, different families of transformations can be generated by an array. This technique enhances the power of the array as a transformation machine.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The dissertation has established the basis of a formal methodology that can be used for the development of computational structures more suitable for the changing nature of real time signal processing and control applications. The work views an array as a transformational device as opposed to a computational one. The contributions of the dissertation to the area of more flexible arrays can be summarized into two basic concepts:

1- Algorithm *fast realizations*

This concept determines special realizations of an algorithm that can be executed more efficiently on a generic array. The concept restricts the operations to remain in the same class, but with reduced execution time. The dissertation developed the necessary tools needed to model the evolution of partial results in the array, parametrized by the number of the compute cycle. As a consequence of the analysis, a proper model of the phenomena of data flow in an array was developed. The array is viewed as a directed weighted computing graph, with nodes corresponding to the array processing cells, and the arcs as the medium of transporting the processed data. The model is general enough to encompass a large class of planar, non-planar, as well as switchable arrays. The *precedence timing*

diagram (PTD) was used to express the evolution of partial results in an array as a function of the compute cycle. It is a valuable tool that displays the array operations at each compute cycle. Systematic procedures were introduced to construct the PTD of an array. The PTD was mapped into a *timing graph* (TG). The TG is a valuable visual aid of the evolution of computations in the array as a function of the compute cycle. A constructive graph theoretic approach was introduced that analyzes the array TG. This approach determines the collection of fast realizations implementable on a single array. The methodology takes advantage of the nature of data dependencies of the algorithm at hand. Data reordering is used to achieve faster execution of algorithms characterized by random sparsity in their matrices on a given array.

2- *Time Constrained* Operation

This principle takes the concept of fast realization one step further, where the basic constraint is the compute cycle number. As long as that constraint is satisfied their is no condition on the actual mathematical representation. A mathematical approach based on concepts from multilinear algebra, is introduced that models the recursive transformations implemented in linear arrays at each compute cycle. The proposed representation is general enough to encompass a large class of signal processing and control applications. A complete analytical model of the linear maps implementable by the array at each compute cycle is developed. By taking advantage of the number of the compute cycle as an additional degree of freedom and the ability of the processing element to change its mode of operations, it is determined that an array generates a large number of transformations during it course of operations.

Throughout the dissertation, we have demonstrated our methodology by using different examples of existing arrays. We have analyzed many arrays, planar and nonplanar, for the matrix-vector multiplication, matrix-matrix multiplication, LU decomposition, QR factorization and the triple matrix product. The methodology resulted in the ability of modifying the operations of those arrays to achieve faster execution for special realizations of an algorithm. Our emphasis was focused on analyzing full size arrays.

Lessons learned from analyzing existing arrays were used to design more efficient arrays for special realizations of the algorithm. By using the number of compute cycle as an extra dimension, the class of transformations that can be generated by a single array is enlarged. The principle results in the ability of performing triangular and band matrix product in an optimum time on a class of arrays designed to perform dense matrix multiplication. Application of the methodology includes the design of flexible time varying structures, and the ability to decompose a full size array into sub-arrays implementing smaller size problems.

With regard to future work, the following topics are suggested:

1- Extension of the concepts to fixed-size and fixed-depth arrays.

2- The development of a mathematical model for the class of algorithms generated by alternating the modes of operations of the processing elements as a function of the number of the compute cycle.

3- The development of efficient algorithms that embed a randomly sparse matrix into its fast form.

4- The establishment of a global model that relates the < *generated* , *used* > elements available on the PE input / output links at the constrained cycle back to the high level language description of the algorithm. This will allow a systematic treatment of the fast forms.

5- Establishing the relation between the array data paths and the algorithm data dependence vectors and using it for the automatic reconfiguration of the array such that it can executes several algorithms.

REFERENCES

1- H.T. Kung, " *Why systolic architectures* ? ", Computer, vol. 15, Jan. 1982, pp. 37-46.

2- H.T. Kung, " *Let's design algorithms for VLSI systems* ", Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication, Cal. Tech., Jan. 1979, pp. 65-90.

3- H.T. Kung and C.E. Leiserson, " *Systolic arrays* (*for VLSI*) ", SIAM, 1979, pp. 256-282.

4- M.J. Foster and H.T. Kung, " *The design of special purpose VLSI chips* ", Computer, vol. 13, no. 1, Jan. 1980, pp. 26-40.

5- H.T. Kung, " *Special-purpose devices for signal and image processing : an opportunity in VLSI* ", Proc. SPIE, vol. 241, July 1980, pp. 76-84.

6- H.T. Kung and P.L. Lehman, " *Systolic (VLSI) arrays for relational database operations* ", Proc. ACM-Sigmod 1980 Int'l Conf. Management of Data, May 1980, pp. 105-116.

7- C.E. Leiserson, " *Systolic priority queues* ", Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication, Cal. Tech., Jan. 1979, pp. 199-214.

8- C.S. Yeh, I.S. Reed and T.K. Truong, " *Systolic multipliers for finite fields* ", IEEE Trans. on Computers, vol. C-33, no. 4, April 1984, pp. 351-356.

9- R.P. Brent and H.T. Kung, " *VLSI arrays for linear-time GCD computation* ", in VLSI 83, F. Anceau and E.J. Aas (eds.), North Holland, 1983.

10- W.A. Porter and J.L. Aravena, " *Cylindrical arrays for matrix multiplication* ", Proc. 14th Allerton Conf., Allerton, IL, Oct. 1986, pp. 595-692.

11- J.L. Aravena and W.A. Porter, " *Nonplanar switchable arrays* ", Circuits Systems, Signal Processing, vol. 7, no. 1, 1988.

12- W.A. Porter, " *Nonplanar array processing* ", Proc. 20th Asilomar Conf., Ca, Nov. 1986.

13- J.L. Aravena, " *Triple matrix product architectures for fast signal processing* ", 20th Asilomar Conf., Nov. 1986, pp. 53-57.

14- A. El-Amawy, W.A. Porter, and J.L. Aravena, " Array architecture for iterative matrix calculations ", Proc. IEE, vol. 134, no. 3, May 1987, pp. 149-154.

15- A. El-Amawy, " *Fast LU-decomposition of dense matrices on an optimal array* ", 20th Asilomar Conf., Nov. 1986, pp. 518-522.

16- A. El-Amawy, " *A systolic architecture for fast dense matrix inversion* ", IEEE Trans. on Computers, vol. 38, no. 3, March 1989, pp. 449-455.

17- H.T. Kung and S.W. Song, " *A systolic 3-D convolution chip* ", in Multicomputers and Image Processing: Algorithms and Programs, K. Perston and L. Uhr (eds.), Academic Press, 1982, pp. 373-384.

18- H.T. Kung, L.M. Ruan and D.W.L. Yen, " *A two-level pipelined systolic array for convolutions* ", Proc. CMU Conf. on VLSI Systems and Computation, Oct. 1981, pp. 255-264.

20- P.R. Cappello and K. Steiglitz, " *Digital signal processing applications of systolic algorithms* ", Proc. CMU Conf. on VLSI Systems and Computation, Oct. 1981, pp. 245-254.

21- G.R. Arce and P.J. Warter, " *A median filter architecture for VLSI implementation* ", Proc. 22nd Annu. Allerton Conf. on Comm. Control and Computing, 1984.

22- K. Oflazer, " *Design and implementation of a single chip 1-D median filter* ", IEEE Trans. on ASSP, vol. 31, no. 5, Oct. 1983.

23- J.L. Guibas and F.M. Liang, " *Systolic stacks, queues and counters* ", Proc. Conf. on Advanced Research in VLSI, M.I.T., 1982.

24- F.C. Lin and Wu, " *Systolic arrays for transitive closure algorithms* ", Proc. Int. Symp. VLSI Sys., Nov. 1986.

25- Y. Robert and D. Trystram, " *An orthogonal systolic array for the algebraic path problem* ", Res. report 553, TIM3/IMAG, France, 1985.

26- S.Y. Kung, S. Lo, and P. Lewis, " *Optimal systolic design for the transitive closure and the shortest path problem* ", IEEE Trans. on Computers, vol. C36, no. 5, May 1987, pp. 603-614.

27- S.Y. Kung and S. Lo, " *A spiral systolic architecture/ algorithm for transitive closure problems* ", Proc. IEEE Int. Conf. on Computer Design, 1985.

28- H. Barada and A. El-Amawy, " *A fixed-size systolic system for the transitive closure and shortest path problems* ", Proc. Int. Conf. on Advances in Communication and Control Systems, Oct. 1988, pp. 623-633.

29- S. Y. Kung et al., " *Mapping Digital Signal Processing Algorithms onto VLSI Systolic/Wavefront Arrays* ", Proc. 20th Asilomar Conf. on Signal, System & Computers, pp. 6, 1986.

30- K. Y. Liu, " *A Pipelined Tree Machine Architecture for Computing a Multidimensional Convolution* ", IEEE Trans. CAS, Vol. 29, No. 4, April 1982.

31- E. H. Wold and A. M. Despain, " *A Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementation* ", IEEE Trans. Comp., Vol. c-33, No. 5, May 1984.

32- W. A. Perera, " *Architectures for Multiplierless Fast Fourier Transform Hardware Implementation in VLSI* ", IEEE Trans. ASSP, Vol. 35, No. 12, Dec. 1987, pp. 1750-1760.

33- C. Zhang and D. Yun, " *Multidimensional Systolic Networks for Discrete Fourier Transforms* " , Proc. ICCD, pp. 215-222, 1984.

34- H. S. Hou, " *A Fast Recursive Algorithm for Computing the Discrete Cosine Transform* ", IEEE Trans. ASSP, Vol. 35, pp. 1455-1461, Oct. 1987.

35- H. G. Yeh, " *Systolic Implementation of Kalman filtering* ", IEEE Trans. ASSP, Vol. 35, No. 9, pp. 1514-1517, Sep. 1987.

36- C. L. Nikias, A. P. Chrysafis and A. N. Venetsanpoulos, " *The LU Decomposition Theorem and its Implication to the Realization of 2-D Digital Filters* " , IEEE Trans. ASSP, Vol. 33, No. 3, June 1985.

37- M. J. Chen and K. Yao, " *Linear Systolic Arrays for Least Square Estimation* ", 1988 IEEE Int. Conf. on Systolic Arrays, pp. 83-93.

38- J. L. Aravena and W. A. Porter, " *Triple Matrix Product Architectures for Fast Signal Processing* ", 20th Asilomar Conf. on Signal Syst. and Comp., Non. 1986.

39- J. L. Aravena, " *Triple Matrix Product Algorithms and Architectures for Fast Signal Processing* ", IEEE Trans. Circ. Syst., Vol. CAS-31, No. 1, Jan 1988.

40- A.L. Fisher and H.T. Kung, " *Synchronizing large systolic arrays* ", Proc. of the 10th Annual Int. Symp. on Computer Architecture, 1983, pp. 54-58.

41- J. Backus, " *Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs* ", ACM, vol. 21, Aug. 1978, pp. 613-641.

42- L. Johnson and D. Cohen, " *A mathematical approach to modeling the flow of data and control in computational networks* ", in VLSI Systems and Computations, ed. H.T. Kung et al., Computer Science Press, Oct. 1981, pp. 213-225.

43- V. Weiser and A. Davis, " *A wavefront notational tool for VLSI array design* ", in VLSI Systems and Computations, ed. H.T. Kung et al., Computer Science Press, Oct. 1981, pp. 226-234.

44- C.E. Leiserson, F.M. Rose and J.B. Saxe, " *Optimizing synchronous circuitry by retiming* ", Proc. 3rd Caltech Conf. on VLSI, ed. R. Bryant, Computer Science Press, 1983, pp. 87-116.

45- J.A.B. Fortes, K.S. Fu and B.W. Wah, " *Systematic approaches to the design of algorithmically specified systolic arrays* ", Proc. ICCD, 1985, pp. 300-303.

46- P.R. Cappello and K. Steiglitz, " *Unifying VLSI array designs with geometric transformations* ", Proc. Conf. Parallel Processing, 1983, pp. 448-457.

47- M. Lam and J. Mostow, " *A transformational model of VLSI systolic design* ", Proc. IFIP 6th Int. Symp. Comput. Hardware Description Lang. Appl., Pittsburgh, PA, May 1983.

48- H.T. Kung and W.T. Lin, " *An algebra for VLSI algorithm design* ", Proc. Conf. Elliptic Problem Solvers, Monterey, CA, 1983.

49- J.M. Jover and T. Kailath, " *Design framework for systolic type arrays* ", Proc. ICASSP, 1984, pp. 851-854.

50- D.S. Schwarts and T.P. Barnwell III, " *A graph theoretic technique for the generation of systolic implementations for shift-invariant flow graphs* ", Proc. of ICASSP 84, 1984, pp. 8.3.1-8.3.4.

51- I.V. Ramakrishnan, D.S. Fussell and A. Silberschartz, " *Mapping homogeneous graphs on linear arrays* ", IEEE Trans. on Computers, vol. C-35, no. 3, March 1986, pp. 189-209.

52- L. Guo-Jie and B. Wah, " *The design of optimal systolic arrays* ", IEEE Trans. Computers, vol. 34, no. 1, Jan. 1985, pp. 66-77.

53- H.D. Cheng, W.C. Lin and K.S. Fu, " *Space-time domain expansion approach to VLSI and its application to hierarchical scene matching* ", Proc. 8th Int. Conf. Pattern Recognition, Montreal, Canada, Aug. 1984.

54- D.I. Moldovan, " *On the analysis and synthesis of VLSI algorithms* ", IEEE Trans. on Computers, vol. C-31, no. 11, 1982, pp. 1121-1126.

55- D.I. Moldovan, " *On the design of algorithms for VLSI systolic arrays* ", Proc. of the IEEE, vol. 71, no. 1, Jan. 1983, pp. 605-612.

56- D.I. Moldovan, " *ADVIS: A software package for the design of systolic arrays* ", 84 Int'l Conf. on Computer Design: VLSI in Computers, Oct. 1984, pp. 158-164.

57- J.A. Fortes, " *Algorithm transformations for parallel processing and VLSI architecture design* ", PhD dissertation, University of Southern California, Los Angeles, CA, Dec. 1983.

58- P. Quinton, " *Automatic synthesis of systolic arrays from uniform recurrent equations* ", Proc. 11th Ann. Sym. Comp. Architecture, 1984.

59- H. R Barada, " *A comprehensive methodology for algorithm characterization, regularization and mapping into optimal VLSI arrays* ", PhD dissertation, Louisiana State University, Baton Rouge, Louisiana, Auguest 1989.

60- M. Chen, " *A Design Methodology for Synthesizing Parallel Algorithms and Architectures* ", Journal of Parallel and Distributed Computing, pp. 268-273, 1985.

61- K. K. Chung and O. Wing, " *Mapping Strategy for Automatic Design of Systolic Arrays* ", 1988 IEEE Int. Conf. on Systolic Arrays, pp. 285-294.

62- M. Payer, " *Formal Derivation of Systolic Arrays, A Case Study* ", 1988 IEEE Int. Conf. on Systolic Arrays, pp. 331-340.

63- H.W. Nelis, " *Automatic design and partitioning of systolic/wavefront arrays for VLSI* ", Circuit, Systems and Signal Processing, vol. 7, no. 2, 1988.

64- S. Y. Kung, " *V LSI Array Processors* ", Information and System Science Series, Prentice Hall, 1988.

65- K. Jainandunsing, " *Systematic Design of Fixed Size VLSI Systolic Processor Arrays Applied to the Orthogonal Faddeeva Equations Solver* ", ECCTD, Sep. 1987

66- J. A. B. Fortes and D. I. Moldovan, " *Parallelism detection and Algorithm Transformation Techniques Useful for VLSI Architecture Design* ", J. Parallel Distribution Comp., Vol. 2, No. 3, pp. 277-301, 1985.

67- C. Guerra and R. Melhem, " *Synthesis of Systolic Designs* ", Proc. Int. Conf. Parallel Proc., pp. 765-772, 1986.

68- W. A. Porter and J. L. Aravena " *Array Resolution* ", Circuits Systems and Signal processing, special issue VLSI, 1986.

69- L. Snyder, " Introduction to the configurable high parallel computer ", IEEE Comp., pp.47-55, Jan. 1982.

70- M. Annaratone et al., " *The warp computer: architecture, implementation and performance* " , IEEE Trans. Comp. vol. C-36, pp. 1523-1537, Dec. 1987.

71- G. He and H. Henry, " *A versatile systolic array for matrix computations* ", *12*th Symp. Comp. Archt., pp. 315-321, June 1985.

72- A. Nash *et al* ., " *A general purpose array processor based on the Faddeeva algorithm* ", IEEE Comp., pp. 55-62, Jan. 1984.

73- M. Sami and R. Stefanelli, " *Reconfiguration Architectures for VLSI Processing Arrays* ", Proc. of the IEEE, May 1986, pp. 712-722.

74- J. H. Kim and S. M. Reddy, " *On Easily Testable and Reconfigurable Two Dimensional Systolic Arrays* ", Intl. Conf. on Parallel Process., Aug. 1988.

75- P. P. Sanjay and M. A. Bayoumi, " *A Reconfigurable VLSI Array for Reliability and Yield Enhancement* ", 1988 IEEE Int. Conf. on Systolic Arrays, pp. 631-642.

76- R. Hughey and D. P. Lopresti, " *Architecture of a Programmable Systolic Array* ", 1988 IEEE Int. Conf. on Systolic Arrays, pp. 41-49.

77- D.I. Moldovan and J.A. Fortes, " *Partitioning and mapping algorithms into fixed systolic arrays* ", IEEE Trans. on Computers, C-35, no. 1, Jan 1986, pp. 1-12.

78- D. Heller, " *Partitioning big matrices for small systolic arrays* ", in VLSI and Modern Signal Processing, pp. 185-199, ed. T. Kailath, Prentice Hall, NJ, 1985.

79- K. Jainandunsing, " *On optimal Partitioning Schemes for Systolic/Wavefront Array Processors* ", Proceedings ISCAS, San Jose, 1986.

80- J. J. Navarro, J. M. Llaberia and M. Valero, " *Partitioning: An Essential Step in Mapping Algorithms into Systolic Array Processors* ", IEEE Computer 20(7), pp. 77-89, Jul. 1987.

81- J. H. Moreno and T. Lang, " *On Partitioning the Faddeeva Algorithm* ", 1988 IEEE Int. Conf. on Systolic Arrays, pp. 125-135.

82- J. Nash, S. Hansen, and K. Przytula, " *Systolic Partitioned and Banded linear Algebraic Computations* ", SPIE Real-Time Signal Processing IX, pp. 10-16,1986.

83- J. J. Navarro, J. M. Llaberia, and M. Valero, " *Computing Size Independent Matrix Problem on Systolic Array Processors* ", 13th Int. Symp. on Comp. Archt., pp. 271-278, Jun. 1986.

84- A. O. Barbir and J. L. Aravena, " *Fast systolic algorithm realizations* ", Third Parallel Processing Symp., Fullerton California, March 1989.

85- J. L. Aravena and A. O. Barbir, " *Time constrained operations of systolic arrays* ", 21st Southeastern Symp. on System Theory, Tallahassee, Florida, March 1989.

86- C. D. Johnson and D. W. Thomes, " *The ' Paralysis by Analysis ' Problem in Resource Allocation Control with Application to SDI* " , Proc. 26th CDC, Los Angeles, CA, Dec. 1987, pp. 1670-1697.

87- J. L. Aravena and W. A. Porter, " *Array Constrained Design of Digital Filters* " , (under review ASSP Trans.).

88- J. L. Aravena and W. A. Porter; " *Array Based Design of Digital Filters* ", Proc. ComCon 88, Oct. 1988, pp. 1372-1379.

89- M.N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms* , Wiley, New York, 1981.

90- A. V. Aho, J. E. Hocroft and J. D. Ullman, *The design and analysis of computer algorithms*, Adison Wesly, MA 1974.

91- H. Amano *et al.*, " *$(sm)^2$-II : A new version of the sparse matrix solving machine* ", *12*th Symp. Comp. Archt., pp. 100-107, June 1985.

92- K. Onaga and T. Takechi, " *On design of rotary array communication and wavefront-driven algorithms for solving large scale band limited matrix equations* ", *13*th Annual Inter. Symp. Comp. Architecture, pp.308-315, June 1986.

93- J. L. Aravena and W. A. Porter, " *Multidimensional Aperture Control* ", IEEE Trans. ASSP, Vol. 33, No. 1, Feb. 1985, pp. 185-194.

94- J. L. Aravena and W. A. Porter, " *Algorithms for Fast Multidimensional Signal Extractions* ", J. Math. Anal. and Appl., Vol. 107, No. 1, April 1985, pp. 300-315.

95- R. R. Mohler, *Bilinear Control Processes*. New York: Academic Press, 1973.

96- R. M. Hirschorn, " *Control of bilinear systems* ", SIAM J. Contr., vol. 10, no. 2, May 1972.

VITA

Abdulkader Omar Barbir was born in Beirut, Lebanon on October 30, 1959. He earned his Bachelor of Science and his Master of Science degrees in Electrical Engineering from Louisiana State University, in May of 1982 and December of 1983, respectively. He was a teaching assistant (January 1984 to August 1984) in the department of Nuclear Science and (August 1985 to December 1986 , January 1988 to August 1989) in the department of Electrical and Computer Engineering at Louisiana State University. He was a research assistant (January 1987 to December 1987) in the department of Electrical and Computer Engineering at Virginia Tech and from (January 1986 to December 1986 , January 1988 to August 1989) in the department of Electrical and Computer Engineering at Louisiana State University.

He is currently with the department of Mathematics and Computer Science at Western Carolina University. He is a member of Alpha Theta Kappa , Eta Kappa Nu , IEEE , IEEE Computer Society and ACM.