AN ABSTRACT OF THE THESIS OF

Louis Wai-Fung Lee for the degree of Master of Science in Electrical and Computer Engineering presented on October 11, 1991.

Title: Fully Efficient Pipelined VLSI Arrays for Solving Toeplitz Matrices

Abstract approved: Redacted for Privacy

Sayfe Kiaei

Fully efficient systolic arrays for the solution of Toeplitz matrices using Schur algorithm [1] have been obtained. By applying clustering mapping method [2], the complexity of the algorithm is O(n) and it requires n/2 processing elements as opposed to n processing elements developed elsewhere [1].

The motivation of this thesis is to obtain efficient pipeline arrays by using the synthesis procedure to implement Toeplitz matrix solution. Furthermore, we will examine pipeline structures for the Toeplitz system factorization and back-substitution by obtaining clustering and Multi-Rate Array structures. These methods reduce the number of processing elements and enhance the computational speed. Comparison and advantage of these methods to other method will be presented.

Fully Efficient Pipelined VLSI Arrays for Solving Toeplitz Matrices

By

Louis Wai-Fung Lee

A THESIS

Submitted to

Oregon State University

In partial fulfillment of
the requirements for the
degree of

Master of Science

Completed October 11, 1991

Commencement June 1992

APPROVED:

Redacted for Privacy

Assistant Professor of Electrical and Computer Engineering in charge of Major

Redacted for Privacy

Head of department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented: October 11, 1991

Typed by Louis Wai-Fung Lee for Louis Wai-Fung Lee

Table of Contents

Chapter 1. Introduction 1

1.1 Demand for High Performance Computing 1
1.2 Overview of the Dissertation 2

Chapter 2. Toeplitz Matrix 4

2.1 Introduction 4
2.2 Solving Toeplitz Matrix 5
2.2.1 Decomposition (Schur Algorithm) 6
2.2.2 Back-substitution Algorithm 10

Chapter 3. Overview of the Synthesis of Array Processors 11

Chapter 4. Systolic Solution of Toeplitz Solver 15

4.1 UREs and DG for the (Schur Algorithm) Algorithm I 15
4.2 UREs and DG for (Back-Substitution) Algorithm II 18
4.3 Timing and Allocation Function of Systolic Solution 23
4.3.1 Timing and Allocation Function of Algorithm I 23
4.3.2 Timing and Allocation Function of Algorithm II 25
4.4 Performance Evaluation of Systolic Solution 28

Chapter 5. Clustering Mapping Method 32

5.1 Toeplitz Solver with Clustering Mapping Method 32
5.1.1 Cluster Array for Algorithm I (Schur Algorithm) 32
5.1.2 Cluster Array for Algorithm II (Back-Substitution) 35
5.2 Performance Evaluation of Cluster Toeplitz Solver 37

Chapter 6. Multi-Rate Array Solution 41

6.1 Advantage of MRA Solution 41
6.2 MRA Solution of Algorithm I (Schur Algorithm) 42
6.3 Performance Evaluation of MRA Solution 45

Chapter 7. Combined Architecture 46

Summary 58

References 64

Appendix A. Simulation Code of Systolic Solution for Schur Algorithm.......... 67

Appendix B. Simulation Code of Clustering Mapping Method for Schur Algorithm.......... 70

Appendix C. Simulation Code of MRA Solution for Schur Algorithm.......... 73

Appendix D. Simulation Code of Systolic Solution for Back-Substitution.......... 75

Appendix E. Simulation Code of Clustering Mapping Method for Back-Substitution.......... 78

List of Figures

Figure 1. Combine UREs to produce a single DG of Schur algorithm 17

Figure 2. Dependency vectors of Schur algorithm 19

Figure 3. Dependency vectors of Back-substitution 19

Figure 4. Dependency Graph of Back-substitution 22

Figure 5. Schedule vector **S** = [2 1] of Schur algorithm is obtained 24

Figure 6. Systolic solution - allocation function of Schur algorithm 26

Figure 7. Timing and systolic allocation of Back-substitution 27

Figure 8. Systolic solution of Schur algorithm 31

Figure 9. Systolic solution of Back-substitution 31

Figure 10. Schur algorithm - merge every 2 processor into one 34

Figure 11. Back-substitution with clustering mapping 37

Figure 12. Schur algorithm - clustering result 40

Figure 13. Back-substitution - clustering result 40

Figure 14. Allocation function of MRA solution 43

Figure 15. MRA simulation realization 44

Figure 16. Parallel shift for manipulating the data **U** which output from Algorihtm I's solution to feed into Algorithm II........ 51

Figure 17. Overview of combined architecture........ 53

Figure 18. Efficiency Vs n of Algorithm I (Schur Algorithm)........ 62

Figure 19. Efficiency Vs n of Algorithm II (Back-Substitution)........ 63

List of Tables

Table 1. Schur's - space-time table of conventional solution.......... 33

Table 2. Schur's - space-time table with clustering mapping.......... 33

Table 3. Back-substitution - space-time of Conventional Mapping.......... 36

Table 4. Back-substitution - space-time of clustering mapping.......... 36

Table 5. Space-time table of MRA solution.......... 44

Table 6. Output from systolic solution of Algorihtm I.......... 48

Table 7. Input of systolic solution of Algorithm I.......... 49

Table 8a. Required order of **U** for first back-substitution step.......... 52

Table 8b. Required order of **U** for second back-substitution step.......... 52

Table 9a Ouput from Cluster Array of Algorithm I.......... 56

Table 9b **U** for Cluster Array of first back-substitution step.......... 57

Table 10. Summary of solutions of Schur algorithm.......... 60

Table 11. Summary of solutions of Back-substitution.......... 61

Fully Efficient Pipelined VLSI Arrays for Solving Toeplitz Matrices

Chapter 1. Introduction

1.1 Demand for High Performance Computing

For many signal processing applications, there is a strong demand for high performance parallel computers. For example, in speech processing the number of required operations may reach 1,500,000 operations/second. To radar processing, requiring up to 5,000,000,000 operations/second is needed [3]. Those performance requirements cannot be obtained by using the conventional sequential structure.

One of the prominent solutions for achieving high-speed computation is the use application specific ICs (ASICs). Systolic arrays have played a significant role in the VLSI implementation of many signal and image processing algorithms [3]. Such arrays consist of a set of pipelined processing elements connected locally in a regular structure providing very high throughput rate. The main advantages of these arrays are [5,6]: (i) a regular flow of data through the array of processors ensuring multiple computations per memory access without increasing I/O requirements; (ii) synchronous - regular timing with local control of data flow; and (iii) local nearest neighbor interconnection to minimize VLSI design complexity and long delays.

Based on the early works of number of researchers such as Cappello [7], Fortes [4], Quinton [13], Rao [14], and others, a unified synthesis theory of automatic derivation of systolic arrays have been developed. The motivation of this thesis is to obtain an efficient pipeline and systolic arrays by using the synthesis procedures. In this case, we will examine pipeline structures for the Toeplitz system factorization which has numerous applications ranging from speech, image, and neurophysics to radar, sonar, geophysics, and astronomical signal processing [15].

For many applications, the efficiency involved in transforming an application problem to systolic arrays is not 100% [2]. Moreover, due to the uniform data propagation in the array, additional delay time for the transmission of data is required which degrades the computational rate and the processor utilization [16,17].

It has been shown that for systolic arrays only few consecutive processors are active at any time unit. In this case, clustering mapping method would be need to merge several neighboring processors onto the same node to obtain a new array which is fully efficient [2]. Furthermore, for many fine-grain operations, due to the locality restriction and uniform data transmission rate of systolic arrays additional delay time is introduced thereby reducing the computation rate. This additional delay time could be avoided by using a Multi-Rate Array (MRA) structure where the variables are propagated at different rates achieving higher speedup and efficiency. We will exam both clustering mapping method and Multi-Rate Array solutions for the design of Toeplitz solver.

1.2 Overview of the Dissertation

The main objective of this dissertation is to show a systematic method to design a systolic array for factorizing and solving Toeplitz matrix which includes the Toeplitz matrix decomposition and the back-substitution. The synthesis is specified by four steps: (1) to specify the computation in terms of set of recurrence equations; (2) to examine the Dependency Graph (DG); (3) to obtain a timing function (or a schedule) specifying the time instant for each computation in the algorithm; and (4) to obtain allocation functions that map each computation onto a specific processor.

This thesis is organized in the following way. In Chapter 2, we briefly review the Schur algorithm to solve the Toeplitz matrix. In Chapter 3, overviews of the synthesis is presented. In Chapter 4, we provide conventional systolic solutions for the Toeplitz Solver and examine its performance. In Chapter 5, clustering mapping method has been applied. In Chapter 6, the MRA solution is presented, and finally, a comparison of the different arrays and their performance is summarized in the conclusion.

Chapter 2. Toeplitz Matrix

2.1 Introduction

The object of solving the Toeplitz system is to find **x** from the set of linear equations

$$\mathbf{T}\mathbf{x} = \mathbf{y} \tag{2.1}$$

where **T** is a (N+1) X (N+1) Toeplitz matrix, **x** and **y** are vectors of length N+1. A symmetric Toeplitz structure **T**, where $t(i, j) = t(|i-j|)$, is shown below

$$\mathbf{T} = \begin{bmatrix} t_0 & t_1 & \cdots & t_N \\ t_1 & t_0 & \cdots & t_{N-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ t_N & t_{N-1} & \cdots & t_0 \end{bmatrix}$$

This system appears in many digital signal processing problems. For an example, the Least Mean-Squares estimation for predicting a sequence $\{y_t\}$ from the observations of $\{x_t\}$ where the estimated y_t is calculated by taking a finite linear combination of the present and past samples of x_t. The standard procedure is to form the "sample covariance" estimate of the second-order statistic,

$$R_k = E\{y_t, y_{t+k}\} \tag{2.2}$$

of the stationary process $\{y_t, t \geq 0\}$.

$$\begin{bmatrix} R_0 & R_1 & \cdots & R_N \\ R_1 & R_0 & \cdots & R_{N-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ R_N & R_{N-1} & \cdots & R_0 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_N \end{bmatrix} = \begin{bmatrix} J_N \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \qquad (2.3)$$

The coefficients **a** can be obtained by solving the Yule-Walker equations above, where J_N is given as the minimum (error) variance solution $E\{e^2(N)\}$. The matrix R_k has is a Toeplitz matrix. Least squares prediction of stationary time series by the autocorrelation method of linear prediction has been studied in various applications such as noise cancelling, spectral estimation, channel equalization, and the linear prediction of speech [18], which involved the solution of the vector **a** by factorization of the Toeplitz matrix R_k.

2.2 Solving Toeplitz Matrix

The major objective in solving the symmetric Toeplitz system is to perform a triangular decomposition of the matrix **T** as

$$\mathbf{T} = \mathbf{U}^T\mathbf{D}^{-1}\mathbf{U} \qquad (2.4)$$

where $\mathbf{D} = \text{diag}[u_1, \ldots, u_{(N+1)}]$, and **U** is an upper triangular matrix. The solution **x** of (2.1) can be solved explicitly with back substitution:

$$\mathbf{x} = \mathbf{T}^{-1}\mathbf{y} = \mathbf{U}^{-1}\mathbf{D}(\mathbf{U}^T)^{-1}\mathbf{y} \qquad (2.5)$$

which can be separated into two back-substitution steps,

$$\mathbf{g} = \mathbf{D}(\mathbf{U}^T)^{-1}\mathbf{y} \qquad (2.6a)$$

and

$$\mathbf{x} = \mathbf{U}^{-1}\mathbf{g}. \qquad (2.6b)$$

To solve the Toeplitz system, two back-substitution steps needs to be performed. In the following sections, detailed algorithms required for decomposition (find **U**) and back-substitution (find **g** and **x**) will be shown. Throughout the thesis, the algorithm for decomposition will be referred to as Algorithm I and algorithm of back-substitution will be called Algorithm II.

2.2.1 Decomposition (Schur Algorithm)

Standard Gauss or Choleski methods for solving nxn Toeplitz system requires $O(n^3)$ arithmetic operations. These Toeplitz solvers are numerically unstable, when applied to arbitrary Toeplitz systems [19]. There are several algorithms (e.g., QR Decomposition) for obtaining the solution in $O(n^2)$ operations [20,21]. More recently, an algorithm has been presented for solving a Toeplitz system of equations using $O(n \log_2 n)$ operations based on Levinson and Durbin [21,23].

Although Levinson algorithm consists of only simple recursive operations; the parallelism is hampered by the presence of inner product operations which are dot product of two vectors of length n, and does not readily lend itself to a systolic array implementation [22]. In each one of the n recursion step, the inner product operation, as shown by other researchers, will require a minimum of $\log_2 n$ computation for additions, and to compute all the n recursions, the total computing time amounts to $O(n \log_2 n)$ on a linear processor array [1]. An improved Levinson algorithm has been developed to resemble the Schur algorithm reducing time complexity to O(n) [1]. Numerical experiments show that the stability and round-off errors of Schur algorithm are competitive with other methods such as the QR factorization and the Levinson algorithm [22,23].

To demonstrate the triangularization procedure using the Schur algorithm we use an example of 4 X 4 matrix. The problem is to find the elements $\{u_{ij}\}$ such that

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 \\ L_{31} & L_{32} & 1 & 0 \\ L_{41} & L_{42} & L_{43} & 1 \end{bmatrix} \mathbf{T} = \begin{bmatrix} u_{10} & u_{11} & u_{12} & u_{13} \\ 0 & u_{21} & u_{22} & u_{23} \\ 0 & 0 & u_{32} & u_{33} \\ 0 & 0 & 0 & u_{43} \end{bmatrix} \qquad (2.7)$$

denoted as

$$\mathbf{LT} = \mathbf{U}$$

The top rows of **L** and **U** are determined by the structure.

$$\mathbf{L}_1 = [1 \quad 0 \quad 0 \quad 0]$$
$$\mathbf{U}_1 = [u_{10} \quad u_{11} \quad u_{12} \quad u_{13}] = [t_0 \quad t_1 \quad t_2 \quad t_3]$$

To find the second row, we start with the following equation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \mathbf{T} = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_1 & t_0 & t_1 & t_2 \end{bmatrix} \qquad (2.8)$$

where t_i's are the elements of **T**. Performing row operations on both sides of this equation:

$$\begin{bmatrix} 1 & K(2) \\ K(2) & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \mathbf{T} = \begin{bmatrix} v(2,0) & 0 & v(2,2) & v(2,3) \\ 0 & u(2,1) & u(2,2) & u(2,3) \end{bmatrix}$$

(2.9)

where the coefficients K(i) (also termed reflection coefficients) are computed as:

$$K(2) = \frac{-t_1}{t_0} \qquad (2.10)$$

This equation can be rewritten as

$$\begin{bmatrix} 1 & K(2) & 0 & 0 \\ K(2) & 1 & 0 & 0 \end{bmatrix} \mathbf{T} = \begin{bmatrix} v(2,0) & 0 & v(2,2) & v(2,3) \\ 0 & u(2,1) & u(2,2) & u(2,3) \end{bmatrix} \tag{2.11}$$

By comparing with the second row of the right hand side (RHS) of Eq. 2.7, it is clear that a zero is created by the row operation and the desired second rows of **L** and **U** are obtained as

$$\mathbf{L}_2 = [L_{21}\ L_{22}\ \ 0\ \ \ 0] = [\,K(2)\ 1\ \ \ 0\ \ \ 0\,]$$
$$\mathbf{U}_2 = [0\ \ u_{21}\ \ u_{22}\ \ u_{23}] = [0\ \ u(2,1)\ \ u(2,2)\ \ u(2,3)]$$

To compute the third row of the matrices **L** and **U**, the same procedure can be repeated. For the next recursion, we first right-shift the second row on both sides of Eq. 2.11, i.e.,

$$[K(2)\ 1\ 0\ 0] \ \text{--->}\ [0\ K(2)\ 1\ 0]$$

$$\begin{bmatrix} 1 & K(2) & 0 & 0 \\ 0 & K(2) & 1 & 0 \end{bmatrix} \mathbf{T} = \begin{bmatrix} v(2,0) & 0 & v(2,2) & v(2,3) \\ u(2,-1) & 0 & u(2,1) & u(2,2) \end{bmatrix} \tag{2.12}$$

Note that by using the Toeplitz structure of the matrix **T**, we have u(2) in Eq. 2.11 right-shifted accordingly and the only new term is u(2,-1), which is equal to v(2,2), since $u(2,-1) = K(2)\ t_1 + t_2 = v(2,2)$.

Through this shift operation, the two zeros created in the previous recursion on the RHS are realigned into the same column. They will remain unaffected by the linear combination of the two rows in the next recursion. With this arrangement, a similar procedure as in the previous recursion can now be repeated:

$$\begin{bmatrix} 1 & K(3) \\ K(3) & 1 \end{bmatrix} \begin{bmatrix} 1 & K(2) & 0 & 0 \\ 0 & K(2) & 1 & 0 \end{bmatrix} \mathbf{T} = \begin{bmatrix} v(3,0) & 0 & 0 & v(3,3) \\ 0 & 0 & u(3,2) & u(3,3) \end{bmatrix} \qquad (2.13)$$

where

$$K(3) = \frac{-v(2,2)}{u(2,1)} \qquad (2.12)$$

Repeating this procedure for u(3) with third row on the RHS of Eq. 2.7, clearly, the third rows of the matrices **L** and **U** are obtained:

$$\mathbf{L}_3 = [L_{31}\ L_{32}\ \ L_{33}\ \ 0\,] = [\,K(3) \quad (K(3)K(2)+K(2)) \ \ 1 \ \ 0\,]$$
$$\mathbf{U}_3 = [0\ \ 0\ \ u_{32}\ \ u_{33}] = [0\ \ 0\ \ u(3,2)\ \ u(3,3)]$$

This completes the second recursion. By induction, the future recursions can be carried out in the same manner until all the rows of the matrices **L** and **U** are computed. Summarizing the above procedure, pseudo-code of the algorithm (Algorithm I) can be made [1] :

```
/* initial conditions */
for (j = 0 to N) {
    v(1, j) = u(1, j) = $t_j$

/*main algorithm */
for (i = 1 to N) {
    K(i+1) = -u(i, 1) / v(i, 0)

    for (j = 0 to N) {
        v(i+1, j) = v(i, j) + K(i+1) u(i, j+1)
        u(i+1, j) = u(i, j+1) + K(i+1) v(i, j)
    }
}
```

where Toeplitz system **T** is an (N+1) X (N+1) matrix, $u_{ij} = u(i, j)$.

2.2.2 Back-substitution Algorithm

Back-substitution algorithm to solve linear system of nxn matrix system:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{B} \tag{2.13}$$

is shown as following expression (Algorithm II) :

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right) \quad , \quad \text{for } i = n, n-1, \ldots, 2, 1 \tag{2.14}$$

where x_i's are elements of 1xn vector **x**, a_{ij}'s are elements of nxn matrix **A**, and b_j's are elements of 1xn vector of **B**.

The objective is apply the synthesis procedures to obtain a set of fully efficient arrays for Algorithm I and II.

Chapter 3. Overview of the Synthesis of Array Processors

Typically, the synthesis process begins with a specification of the algorithm to be solved in terms of a set of recurrence equation. Earlier work on synthesizing systolic arrays was based on the analysis of the data dependencies of such initial specifications under the assumption that the dependencies were expressed as constant vectors in Euclidean space. One such technique, developed by Quinton [13] proposes the notation of **Uniform Recurrence Equations (UREs)** as an adequate initial specification. A general specification proposed by Rajopadhye [8] and Fortes [8] addresses the problem that the Uniform Recurrence Equations (URE) are unnecessarily restrictive as an initial specification of the algorithm. To overcome the limitations of UREs they proposed to permit the dependencies to be arbitrary linear (affine) functions and to adopt **Affine Recurrence Equations (AREs)** as an alternative initial specification.

Notation of recurrence equations has been well known to mathematicians for expressing a large class of computations. In general, the initial set of recurrence equations can be expressed as follows:

Definition 1: A Recurrence Equation over a domain **D**, is defined to be an equation of the form

$$f(p) = g(f(q_1), f(q_2) \ldots f(q_k))$$

where $p \in \mathbf{D}$; $q_i \in \mathbf{D}$ for $i = 1 \ldots k$ and g is a single valued function which is strictly dependent on each of its arguments. $\mathbf{D} \in Z^n$ in Euclidean space.

A system of m Recurrence Equations over a domain **D** is defined to be a family of m mutually recursive equations, where each of the function f_i is defined by an equation of the form

$$f_i(p) = g(f_{i1}(q_{i1}), f_{i2}(q_{i2}) \ldots f_{ik}(q_{ik}))$$

The computation involves the evaluation of a function g at all points in a domain **D**. The recurrence equation specifies how the value of g at point p in **D** depends on the value of f at other points in the domain.

Uniform Recurrence Equation defines a computation where the dependencies can be completely described by a finite number of constant vectors, regardless of the size of the domain [8,9].

Definition 2 A Recurrence Equation of the form $f(p) = g(f(q_1), f(q_2) \ldots f(q_k))$ is called a **Uniform Recurrence Equation (URE)**

$$\text{iff} \quad q_i - w_i = p, \qquad \text{for } i = 1 \ldots k,$$

where w_is are constant n-dimensional vectors.

A large number of interesting problems cannot be naturally expressed as UREs. Therefore, a more general class of recurrence equations termed Affine Recurrence Equations (AREs) had been introduced where the dependencies are affine functions of the point.

Definition 3 A Recurrence Equation as given by Definition 1 is said to be an **Affine Recurrence Equation (ARE)** if for $i = 1 \ldots k$, $q_i = A_i p + b_i$ where A_i is a constant nxn matrix and b_i is a constant n-dimensional vector. Thus the recurrence has the following form.

$$f(p) = g(f(A_1 p+b_1), f(A_2 p+b_2), \ldots f(A_k p+b_k))$$

Given an algorithm specified in term of a set of recurrence equations, its computational structure could be examined by its dependency graph. The **Dependence Graph** (DG) shows the dependence of the computation that occurs in an algorithm. By viewing each dependency relation or computations that occur as an arc between the corresponding variables located in the index-space, localized dependence graph can be viewed as all variables are (directly) dependent upon the variables of the neighboring nodes [9].

Dependence mapping aims at extracting the dependences between the variables of the algorithm and mapping the algorithm onto a systolic array in such a way that the dependences are preserved. The objective is then to find the timing and allocation function. The problem of finding optimal timing functions can be reduced to a linear programming or a sequence of linear programming.

Timing Function: In this case, we restrict ourselves to linear timing functions that is a linear function of the form [13],

$$t(p) = \lambda_T^T p - \alpha_t$$

where λ_T^T is a constant schedule vector along the direction **S** and α_t is a scalar constant. Timing function t is a mapping of all points in **D** to the positive integers such that if p->q then t(p) > t(q). t(p) may naturally be interpreted as the time at which f(p) is computed. Timing function must satisfy the **causality condition**:

$$d_i = \lambda_T^T e_i \geq 1, \text{ for any } \mathbf{e}$$

where **e** is the dependency vectors and d is the propagation delay.

Allocation Function: the allocation function maps each computation onto a finite domain and defines the processor at which each computation is performed. The computation g performed at any point p in **D** defines the granularity of the processors. The allocation function, which maps every point p to an (n-1) dimensional processor space, is defined by

$$a(p) = \lambda_a^T p - \alpha a$$

where λ_a is an (n-1) x n matrix and α_a is an n-1 vector [10]. It is necessary and sufficient for allocation vector λ_a to satisfy the **conflict free condition**:

$$\lambda_T^T u \neq 0 \qquad \text{(where } \lambda_a^T u = 0)$$

In another words,

$$\text{if } t(p) = t(q), \text{ then } a(p) \neq a(q).$$

In the next chapter, we will apply these synthesis methods for the pipeline structure of Toeplitz systems.

Chapter 4. Systolic Solution of Toeplitz Solver

In Chapter 2, we have presented the decomposition algorithm (Algorithm I) and the back-substitution algorithm (Algorithm II). To solve the Toeplitz system, two back-substitution steps has to be performed. In the following sections, a detailed approach for the decomposition (find **U**) and back-substitution (find **g** and **x**) will be provided.

4.1 UREs and DG for the (Schur Decomposition) Algorithm I

Algorithm I for the Schur decomposition of Toeplitz matrix is given as

```
/*main algorithm */
for (i = 1 to N) {
    K(i+1) = -u(i, 1) / v(i, 0)

    for (j = 0 to N) {
        v(i+1, j) = v(i, j) + K(i+1) u(i, j+1)
        u(i+1, j) = u(i, j+1) + K(i+1) v(i, j)
    }
}
```

The main algorithm is the portion which we have to consider because the initial conditions can be obtained directly from **T** as input values. The main algorithm can be split into two separate algorithms, algorithm Ia for finding the **K** and algorithm Ib for computing **v** and **u**, as follow:

```
/* Algorithm Ia */
for (i = 1 to N) {
    K(i+1) = -u(i,1) / v(i,0)
}
```

```
/* Algorithm Ib */
for (i = 1 to N) {
        for (j = 0 to N) {
                v(i+1, j) = v(i, j) + K(i+1) u(i, j+1)
                u(i+1, j) = u(i, j+1) + K(i+1) v(i, j)
        }
}
```

Formulating the AREs of Algorithm I:

AREs Ia: The above Algorithm Ia can be rewritten to produce AREs. At each point $[i,j] \in \mathbf{D}$ $(= \{[i,j] \mid 2 \le i \le N+1\})$, the computation requires the values of **u** and **v**. Of these, the values of **u** and **v** are inputs, and must be obtained from outside the domain.

$$K(i, 0) = -u(i-1, 1) / v(i-1, 0)$$

AREs Ib: Algorithm Ib can be rewritten as **K** can be obtained from global broadcasting along j-axis. We note that the variables i and j determine a domain given by $\mathbf{D} = \{[i,j] \mid 2 \le i \le N+1, 0 \le j \le N\}$.

$$v(i, j) = v(i-1, j) + K(i, 0)\ u(i-1, j+1)$$
$$u(i, j) = u(i-1, j+1) + K(i, 0)\ v(i-1, j)$$

Localization of Algorithm I:

We now can rewritten the AREs

$$K(i, j) = K(i, j-1)$$

In this form, each time that the statement is performed, the indices (i, j) are different, and we have eliminated the global broadcasting. Hence we can propagate each value of variable **K** along j axis by the pair (i, j).

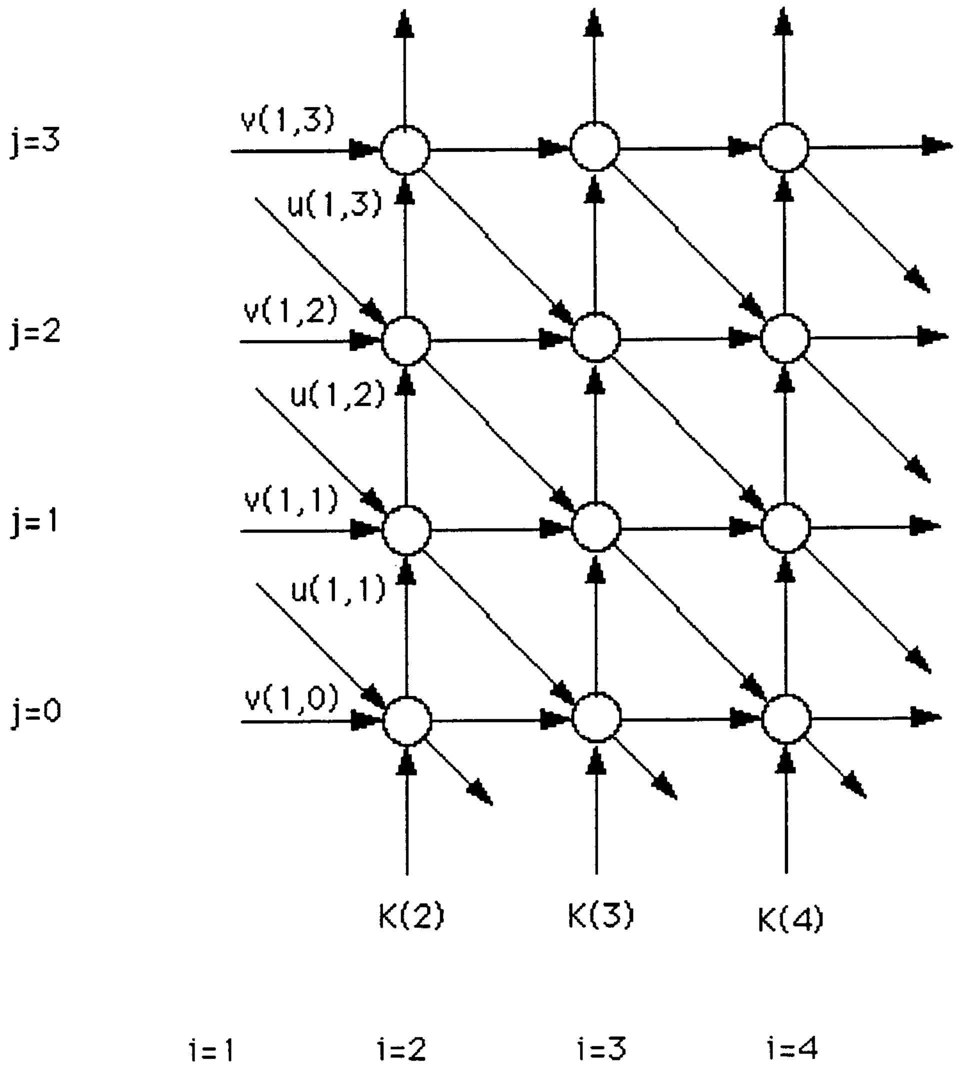

Fig. 1. Combine UREs to produce a single DG of Schur algorithm.

Obtain UREs and DG of Algorithm I:

UREs Ia: At each point $[i,j] \in \mathbf{D}$ $(= \{[i,j] \mid 2 \leq i \leq N+1,\ j = 0\})$

$$K(i, j) = -u(i-1, j+1) / v(i, j) + K(i, j-1)$$
$$v(i, j) = v(i-1, j)$$
$$u(i, j) = u(i-1, j+1)$$

UREs Ib: the variables i and j in $\mathbf{D} = \{[i,j] \mid 2 \leq i \leq N+1,\ 0 \leq j \leq N\}$

$$v(i, j) = v(i-1, j) + K(i, j)\ u(i-1, j+1)$$
$$u(i, j) = u(i-1, j+1) + K(i, j)\ v(i-1, j)$$
$$K(i, j) = K(i, j-1)$$

Note that Eq. 4.1a and Eq. 4.1b have the same dependencies (see Fig. 2) which allow both UREs to be combined into a single DG. Dependency Graph (Fig. 1) combines UREs Ia and UREs Ib and shows the dependency relationships for the Schur algorithm.

4.2 UREs and DG for (Back-Substitution) Algorithm II

The back-substitution algorithm to solve linear system of nxn matrix system is given as Algorithm II

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right) , \quad \text{for } i = n, n-1, \ldots, 2, 1$$

where x_i's are elements of 1xn vector **x**, a_{ij}'s are elements of nxn matrix **A**, and b_j's are elements of 1xn vector of **B**. Algorithm II can be decompose as follows

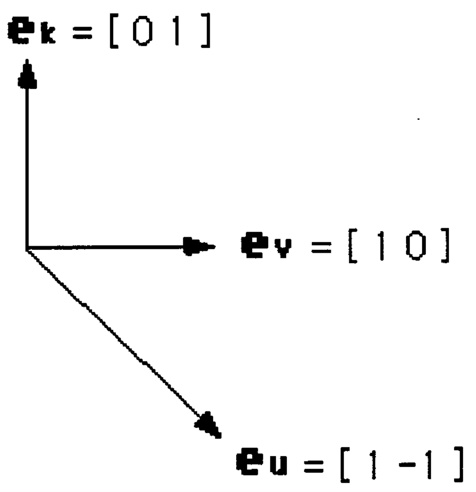

Fig. 2. Dependency vectors of Schur algorithm.

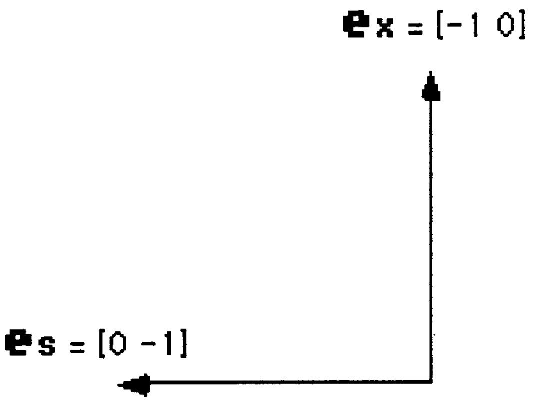

Fig. 3. Dependency vectors of Back-substitution

Algorithm IIa:

$$s_i = \sum_{j=i+1}^{n} a_{ij} x_j \,, \qquad \text{for } i = n, n\text{-}1, \ldots, 2; \text{ and } j > i$$

It is clear that for all operations, j is greater than i. The equation can also be written as pseudo-code form

```
for (i = n downto 2)
    for (j = i+1 to n)
        s(i) = a(i, j) * x(j) + s(i)
```

Algorithm IIb:

$$x_i = \frac{1}{a_{ii}} \left(b_i - s_i \right) \,, \qquad \text{for } i = n, n\text{-}1, \ldots, 2, 1$$

In order to make it consistence with the **a** in Algorithm IIa, We change the index of **a** and also restrict that all the operations can be performed at only i equal to j.

$$x_i = \frac{1}{a_{ij}} \left(b_i - s_i \right) \text{,for } i = j = n, n\text{-}1, \ldots, 2, 1; \text{ and } i = j$$

Formulating the AREs of Algorithm II:

AREs IIa: the Algorithm IIa can be rewritten to produce the following AREs. The variables i and j in **D** = {[i,j] | 1 ≤ i ≤ n-1, 2 ≤ j ≤ n, j > i}, the computation requires the values of **a** and **x**. Of these, the values of **x** is broadcasted globally as i-axis and the values of **a** is presented locally. **s** is summed along j-axis.

$$s(i, j) = a(i, j)\, x(0, j) + s(i, j+1)$$

AREs IIb: the Algorithm IIb can also be rewritten so that at each point $[i,j] \in$ **D** $(= \{[i,j] \mid 1 \le i \le n,\ 1 \le j \le n,\ i = j\})$, the computation requires the values of **a, b** which presented locally and **s** can be obtained from outside the domain.

$$x(0, j) = \frac{1}{a(i, j)} \left(b(0, j) - s(i, j+1) \right)$$

Localization of Algorithm II:

We now have to localize values **x** which is globally broadcasted along i-axis

$$x(i, j) = x(i+1, j)$$

Obtain UREs and DG of Algorithm II:

UREs IIa: the variables i and j in **D** $= \{[i,j] \mid 1 \le i \le n-1,\ 2 \le j \le n,\ j > i\}$

$$\begin{aligned} s(i, j) &= a(i, j)\, x(i, j) + s(i, j+1) \\ x(i, j) &= x(i+1, j) \end{aligned}$$

UREs IIb: at each point $[i,j] \in$ **D** $(= \{[i,j] \mid 1 \le i \le n,\ 1 \le j \le n,\ i = j\})$

$$\begin{aligned} x(i, j) &= \frac{1}{a(i, j)} \left(b(i, j) - s(i, j+1) \right) + x(i+1, j) \\ s(i, j) &= s(i, j+1) \end{aligned}$$

From above UREs show that the summation **s** pass along j-axis. **x** is provided along i-axis, and **a, b** is presented locally. Combining dependencies of UREs IIa and UREs IIb, a complete Dependency Graph can be obtained (see Fig. 3 and 4).

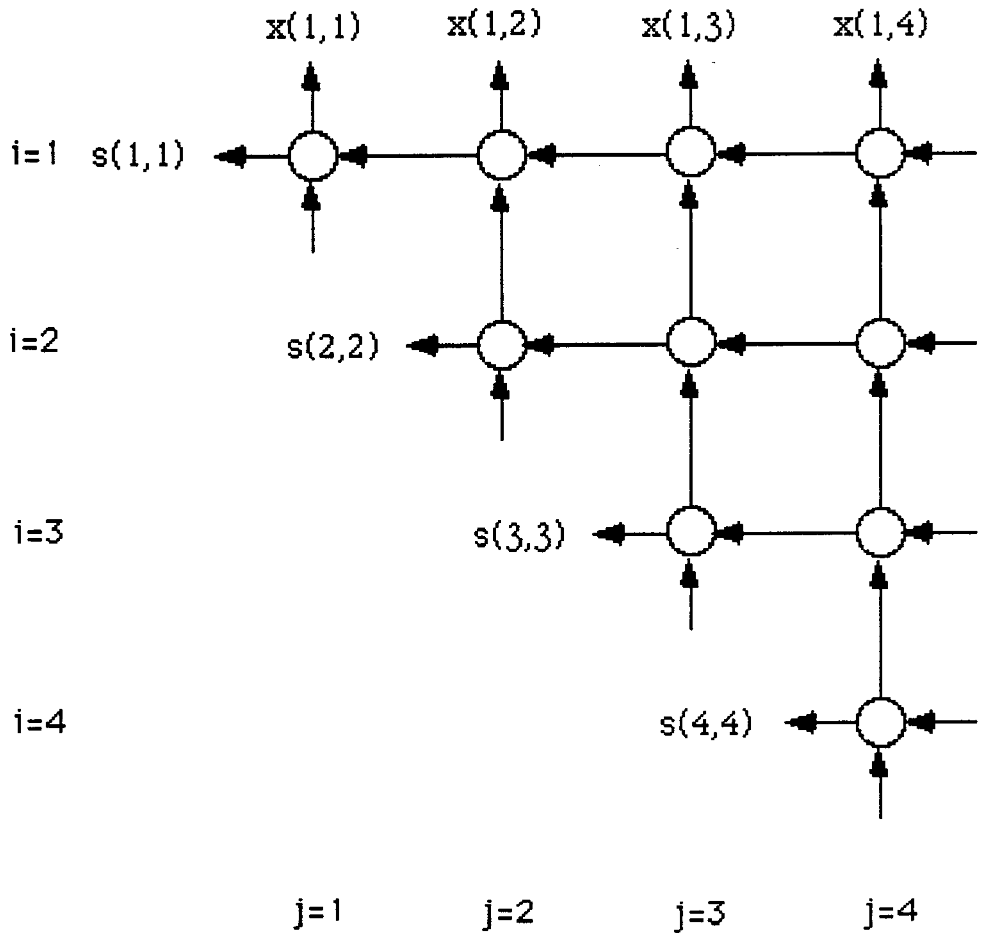

Fig. 4. Dependency Graph of Back-substitution

4.3 Timing and Allocation Function of Systolic Solution

<u>4.3.1 Timing and Allocation Function for Algorithm I</u>

Timing function: A linear timing function is

$$t(p) = \lambda_T^T p - \alpha_t$$

which for dependencies $\{\mathbf{e}_i\}$ must satisfy:

$$\lambda_T^T \mathbf{e}_i \geq 1, \text{ for any } \mathbf{e}; \text{ (causality condition)}$$

$$[\lambda_1 \;\; \lambda_2] \begin{bmatrix} 0 \\ 1 \end{bmatrix} \geq 1$$

$$[\lambda_1 \;\; \lambda_2] \begin{bmatrix} 1 \\ 0 \end{bmatrix} \geq 1$$

$$[\lambda_1 \;\; \lambda_2] \begin{bmatrix} 1 \\ -1 \end{bmatrix} \geq 1$$

where $[\lambda_1 \;\; \lambda_2] = [2 \;\; 1]$, the delays for each dependencies are

$$d_i = \lambda_T^T \mathbf{e}_i$$

$$d_k = 1, d_v = 2, d_u = 1$$

To find the offset value α_t, we can assign $t(p) = 1$. Then the actual timing function is

$$t(p) = [2 \;\; 1]\, p - 3 \qquad (4.1)$$

Allocation function: $a(p) = \lambda_a^T p - \alpha_a$

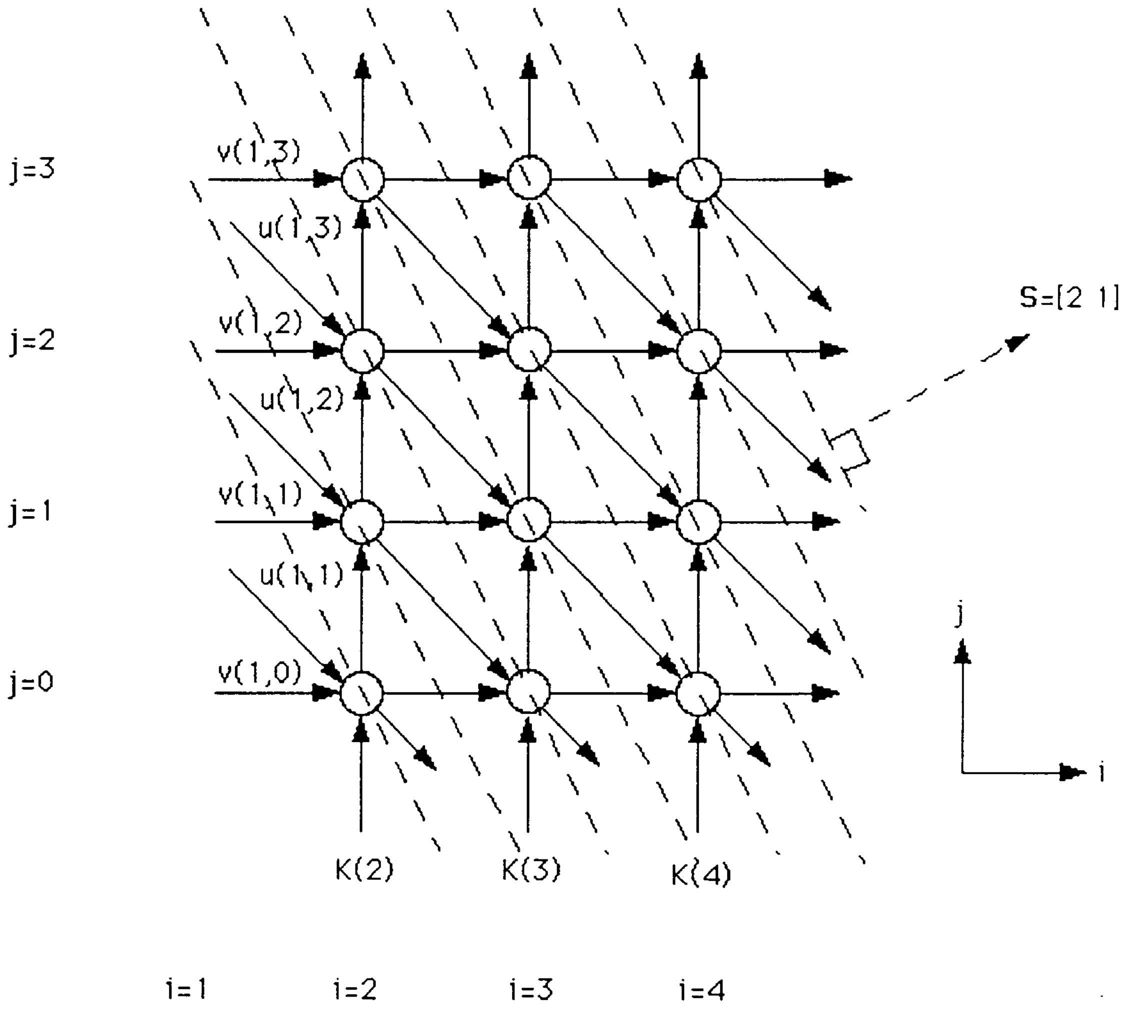

Fig. 5. Schedule vector **S** = [2 1] of Schur algorithm is obtained.

In this mapping scheme (as shown in Fig. 6), offset value α_a can be found by assigning a(p) = 1 where point p is performed at PE 1. The allocation function is:

$$a(p) = [\,0 \quad 1\,]\, p + 1 \qquad (4.2)$$

4.3.2 Timing and Allocation Function of Algorithm II

Timing function: $d_i = \lambda_T^T \mathbf{e}_i \geq 1$, for any **e**; (causality condition). Hence, delays for all the variables (see Fig. 4):

which for dependencies $\{\mathbf{e}_i\}$ must satisfy:

$$\lambda_T^T \mathbf{e}_i \geq 1, \text{ for any } \mathbf{e}; \text{ (causality condition)}$$

$$[\,\lambda_1 \quad \lambda_2\,] \begin{bmatrix} 0 \\ -1 \end{bmatrix} \geq 1$$

$$[\,\lambda_1 \quad \lambda_2\,] \begin{bmatrix} -1 \\ 0 \end{bmatrix} \geq 1$$

where $[\,\lambda_1 \quad \lambda_2\,] = [\,-1 \quad -1\,]$, the delays for each dependencies are

$$d_i = \lambda_T^T \mathbf{e}_i$$

$$d_S = 1 \text{ and } d_X = 1$$

From Fig. 7, the actual timing function will be

$$t(p) = [\,-1 \quad -1\,]\, p + 9 \qquad (4.3)$$

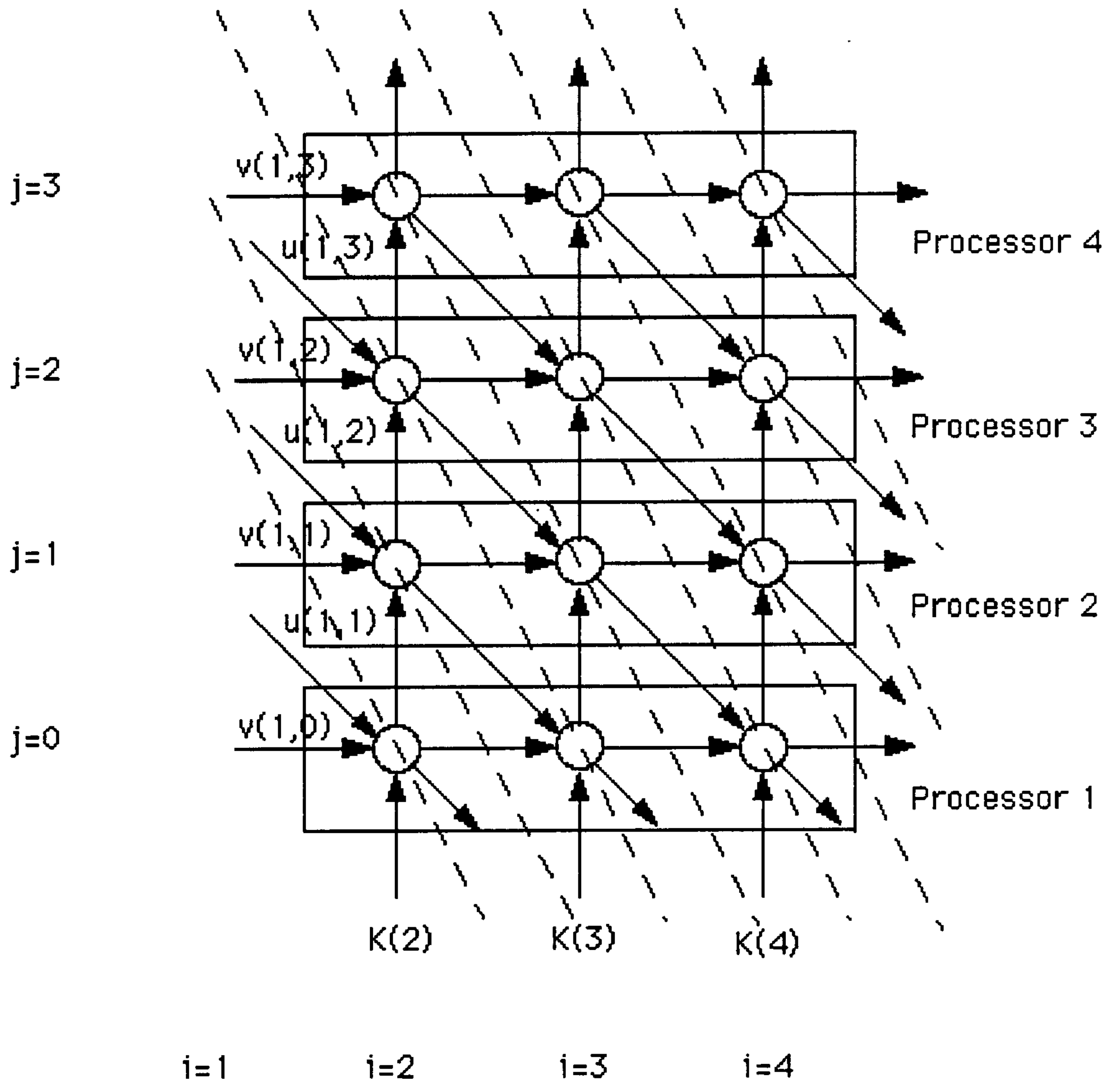

Fig. 6. Systolic solution - allocation function of Schur algorithm.

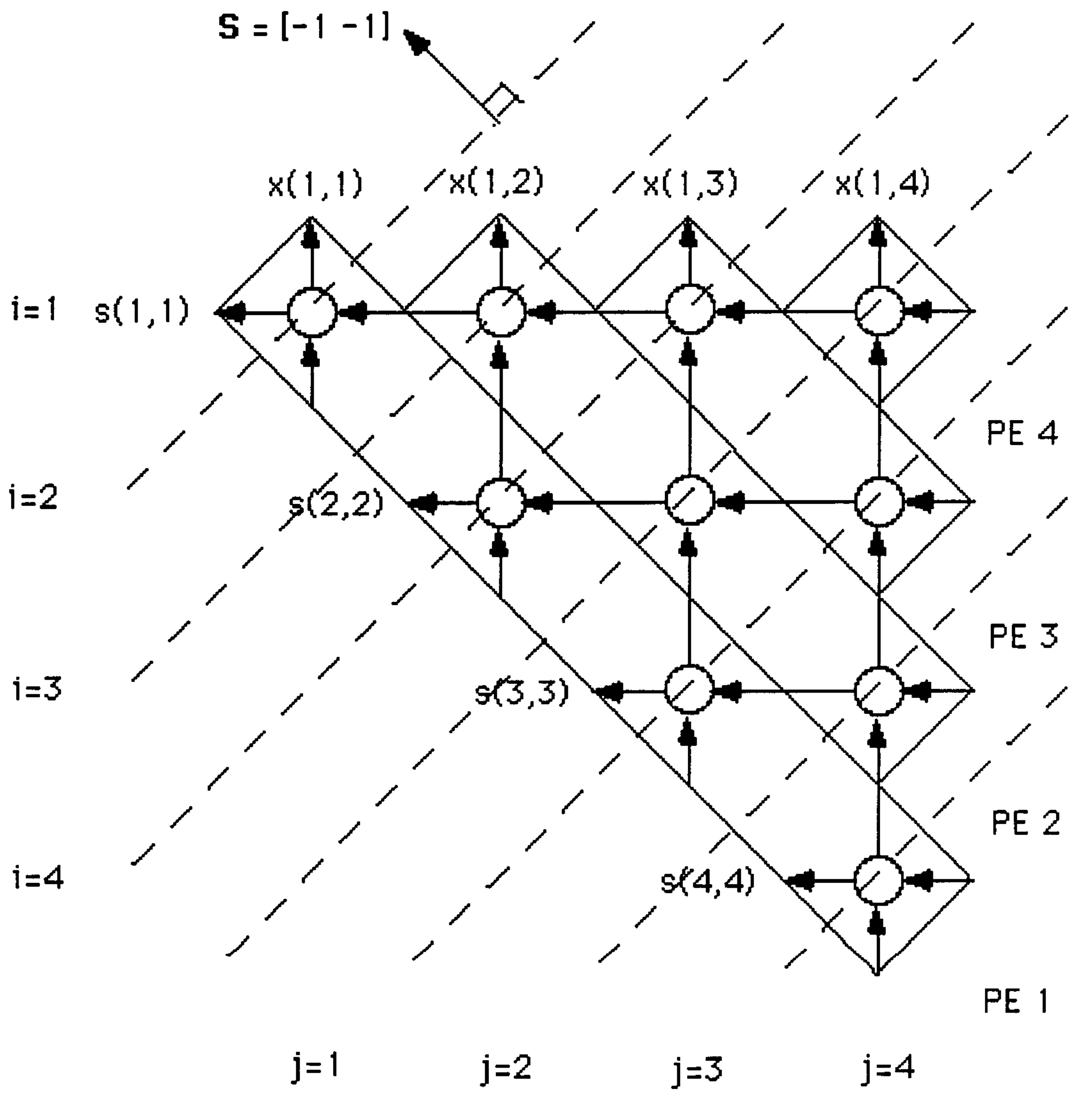

Fig. 7. Timing and systolic allocation of Back-substitution.

Allocation function: the allocation function is

$$a(p) = [\,-1 \;\; 1\,]\, p + 1 \tag{4.4}$$

4.4 Performance Evaluation of Systolic Solution

To evaluate the performance of the Toeplitz array structure, we will examine time complexity, space complexity, speedup as compared to sequential algorithm, pipelining rate (throughput), and the array efficiency defined as follows [3]:

Time Complexity: Time used by the algorithm as a function of algorithm order of O(n).

Speedup: Speedup is defined by the ratio of the execution time T_S on a serial computer or uniprocessor to execution time T_P on the parallel computer:

$$S = \frac{T_S}{T_P} \tag{4.5}$$

Efficiency: Ratio of speedup to number of processors used. Efficiency indicates the utilization rate of the available resources:

$$E = \frac{S}{P} \tag{4.6}$$

Space Complexity: Area used by an algorithm as a function of problem size (ie: number of processing elements).

Pipelining rate: The throughput per time unit.

Evaluation of Algorithm I (Schur Algorithm)

The systolic array solution is simulated by using Systolic Array Simulator developed by [24]. Processors interconnection realization is shown is Fig. 8. For n x n matrix system, the array takes n-1 steps for pre-loading data **u**. Computation will requires 3n-4 steps to obtain **u** and **K** values. Thus, total time required for obtaining all results is the number of steps of pre-loading data plus computations which will be 4n-5, the complexity is O(n).

For the Schur algorithm, $T_S = n \times (n - 1)$ and $T_P = 4n-5$. Giving the following speedup and efficiency

$$S = \frac{n^2-n}{4n-5} \approx O(n)$$

$$E = \frac{n-1}{4n-5}$$

For large number of n, E will be close to $\frac{1}{4}$. Linear speedup has been achieved.

Evaluation of Algorithm II (Back-Substitution)

For nxn matrix system, computation will requires $T_P = 2n-1$ steps to obtain all the results. It has computational complexity of O(n). T_S of back-substitution on serial machine will be $\frac{n(n+1)}{2}$.

$$S = \frac{T_S}{T_P} = \frac{n(n+1)}{2(2n-1)}$$

$$E = \frac{n+1}{2(2n-1)}$$

For large number of n, E will be close to $\frac{1}{4}$.

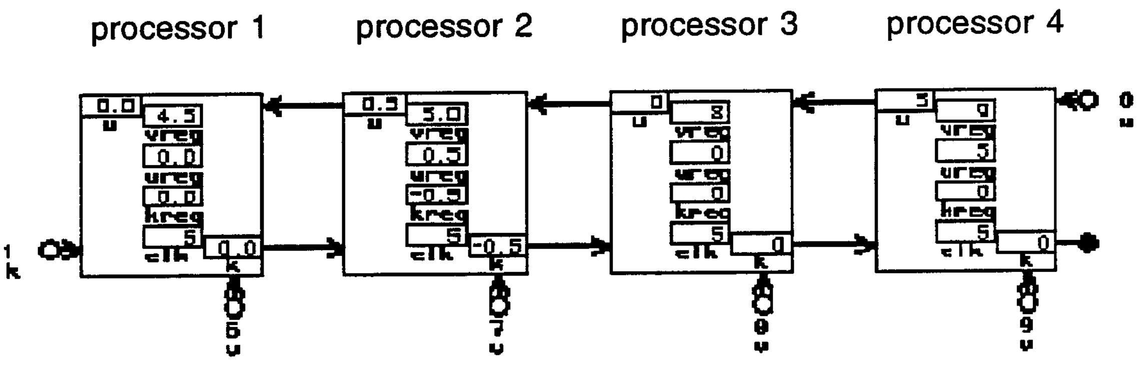

Fig. 8. Systolic solution of Schur algorithm.

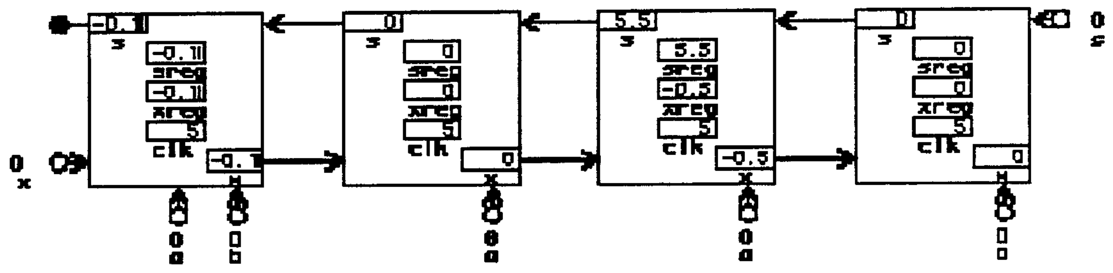

Fig. 9. Systolic solution of Back-substitution.

Chapter 5. Clustering Mapping Method

The systolic arrays derived by the conventional integral linear transformation is not fully efficient, where only one out of several consecutive processors are active at any time unit. For the decomposition array and back-substitution array, every 1 in 2 PEs was active in any given time. The activation period δ of processing elements can be obtained as:

$$\delta = \lambda_T^T u \qquad (5.1)$$

where u is the projection direct orthogonal to the allocation function $\lambda_a^T u = 0$.

5.1 Toeplitz Solver with Clustering Mapping Method

It has been shown that [2] if a processor is active for only one out of every δ clock cycles, there may be δ-1 other neighboring processors such that only one of them is active at any instant. In this case, one can merge these processors onto a single PE that is always active. Such a processor would have the same cost as the original processor, except for a few additional multiplexers and registers. This merging is equivalent to using allocation function that are not integer but rational matrices, then obtaining integral processors labels by using the floor function. New array has the same computation time complexity but the number of PEs is reduced by $1/\delta$.

5.1.1 Cluster Array for Algorithm I (Schur Algorithm)

Fig. 6 and Table 1 show the inefficiency of the conventional solution. From Table 1, one can see that each processor is inactive for every 1 out of 2 clock cycle. By applying eq. 5.1,

CLK	Processor 1			Processor 2			Processor 3			Processor 4		
1			v_{10}			v_{11}			v_{12}		u_{11}	v_{13}
2			v_{10}			v_{11}		u_{11}	v_{12}			v_{13}
3			v_{10}		u_{11}	v_{11}			v_{12}		u_{12}	v_{13}
4	K_2	u_{20}	v_{20}			v_{11}		u_{12}	v_{12}			v_{13}
5	•	•	v_{20}	K_2	u_{21}	v_{21}			v_{12}		u_{13}	v_{13}
6	K_3	u_{30}	v_{30}	•	•	v_{21}	K_2	u_{22}	v_{22}			v_{13}
7	•	•	v_{30}	K_3	u_{31}	v_{31}	•	•	v_{22}	K_2	u_{23}	v_{23}
8	K_4	u_{40}	v_{40}	•	•	v_{31}	K_3	u_{32}	v_{32}	•	•	v_{23}
9			v_{40}	K_4	u_{41}	v_{41}	•	•	v_{32}	K_3	u_{33}	v_{33}
10			v_{40}			v_{41}	K_4	u_{42}	v_{42}	•	•	v_{33}
11			v_{40}			v_{41}			v_{42}	K_4	u_{43}	v_{43}

• processor is inactive due to inefficiency mapping

Table 1. Schur's - space-time table of conventional solution.

CLK	Processor 1				Processor 2			
1			v_{10}	v_{11}		u_{11}	v_{12}	v_{13}
2			v_{10}	v_{11}		u_{11}	v_{12}	v_{13}
3		u_{11}	v_{10}	v_{11}		u_{12}	v_{12}	v_{13}
4	K_2	u_{20}	v_{20}	v_{11}		u_{12}	v_{12}	v_{13}
5	K_2	u_{21}	v_{20}	v_{21}		u_{13}	v_{12}	v_{13}
6	K_3	u_{30}	v_{30}	v_{21}	K_2	u_{22}	v_{22}	v_{13}
7	K_3	u_{31}	v_{30}	v_{31}	K_2	u_{23}	v_{22}	v_{23}
8	K_4	u_{40}	v_{40}	v_{31}	K_3	u_{32}	v_{32}	v_{23}
9	K_4	u_{41}	v_{40}	v_{41}	K_3	u_{33}	v_{32}	v_{33}
10			v_{40}	v_{41}	K_4	u_{42}	v_{42}	v_{33}
11			v_{40}	v_{41}	K_4	u_{43}	v_{42}	v_{43}

Table 2. Schur's - space-time table with clustering mapping.

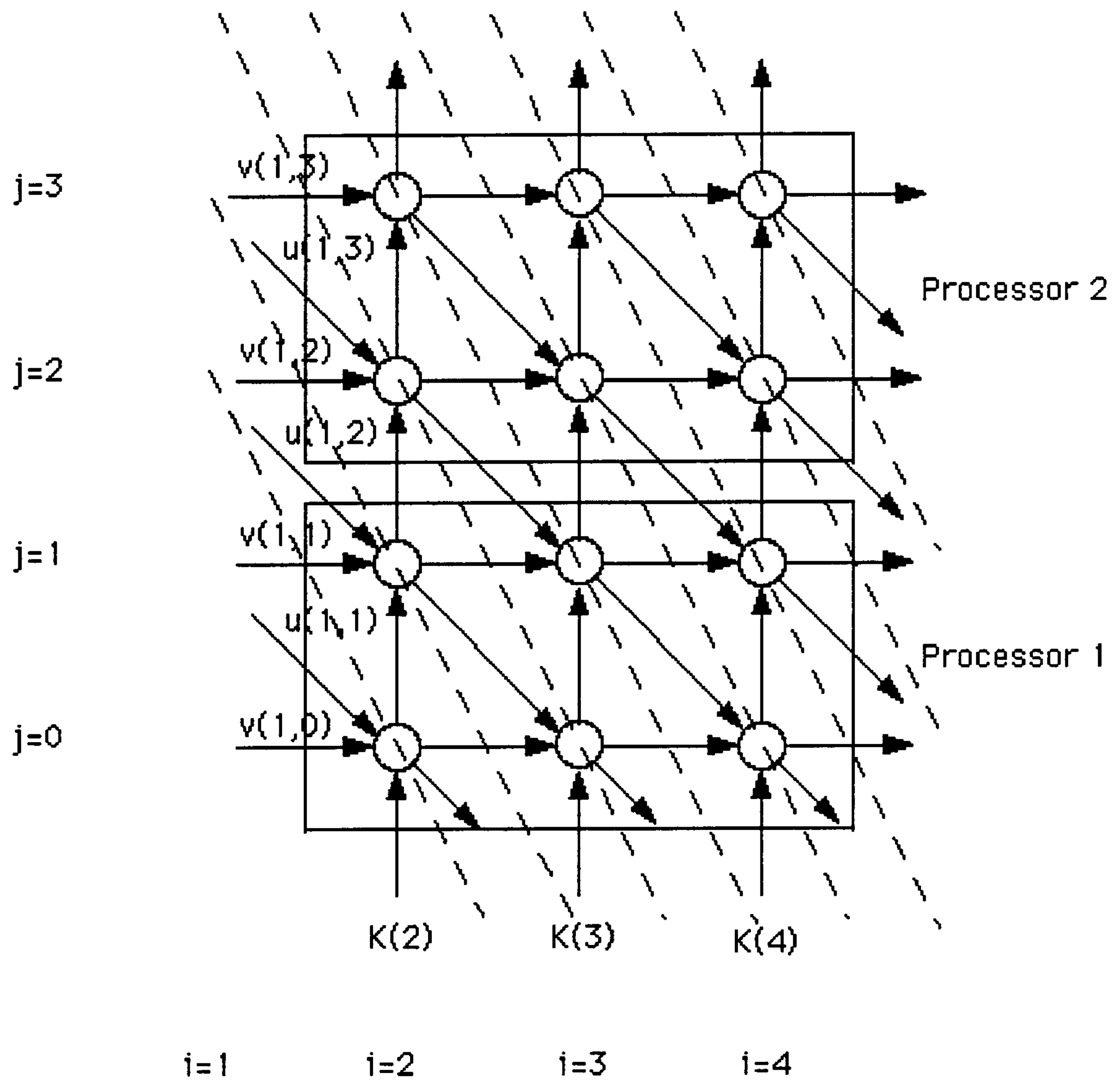

Fig. 10. Schur - after merging every 2 processor into one.

$$\lambda_a^T = [\,0 \;\; 1\,]$$

or the projection direction is:

$$u = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\delta = \lambda_a^T u = [\,2 \;\; 1\,] \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 2$$

In this case, we will be able to merge two processors into one as show in Fig. 10. The new allocation function can be obtain from

$$a(p) = \left\lceil \frac{1}{\delta} \lambda_a \, p - \alpha_a \right\rceil \qquad (5.2)$$

Thus,

$$a(p) = \left\lceil [\,0 \;\; \tfrac{1}{2}\,] \, p + 1 \right\rceil \qquad (5.3)$$

The new array has the same computation time. The total number of PEs is reduced by 2. Moreover, except for extra registers to queue the data, processors in the new array have the same function units. By studying the space-time table (see Table 1 and 2), only one v register is added to each processor unit.

<u>5.1.2 Cluster Array for Algorithm II (Back-Substitution)</u>

From studying the timing and allocation diagram (Fig. 7) and space-time table (Table 3), every processor is active for one out of 2 time units. Also from eq. 5.1

CLK	Processor 1				Processor 2			Processor 3			Processor 4		
1	s_4	x_4	a_{44}	b_4									
2	•	•	•	•	s_3	x_4	a_{34}						
3	s_3	x_3	a_{33}	b_3	•	•	•	s_2	x_4	a_{24}			
4	•	•	•	•	s_2	x_3	a_{23}	•	•	•	s_1	x_4	a_{14}
5	s_2	x_2	a_{22}	b_2	•	•	•	s_1	x_3	a_{13}			
6	•	•	•	•	s_1	x_2	a_{12}						
7	s_1	x_1	a_{11}	b_1									

• processor is inactive due to inefficiency mapping

Table 3. Back-substitution - space-time of Conventional Mapping.

CLK	Processor 1				Processor 2		
1	s_4	x_4	a_{44}	b_4			
2	s_3	x_4	a_{34}				
3	s_3	x_3	a_{33}	b_3	s_2	x_4	a_{24}
4	s_2	x_3	a_{23}		s_1	x_4	a_{14}
5	s_2	x_2	a_{22}	b_2	s_1	x_3	a_{13}
6	s_1	x_2	a_{12}				
7	s_1	x_1	a_{11}	b_1			

Table 4. Back-substitution - space-time of clustering mapping.

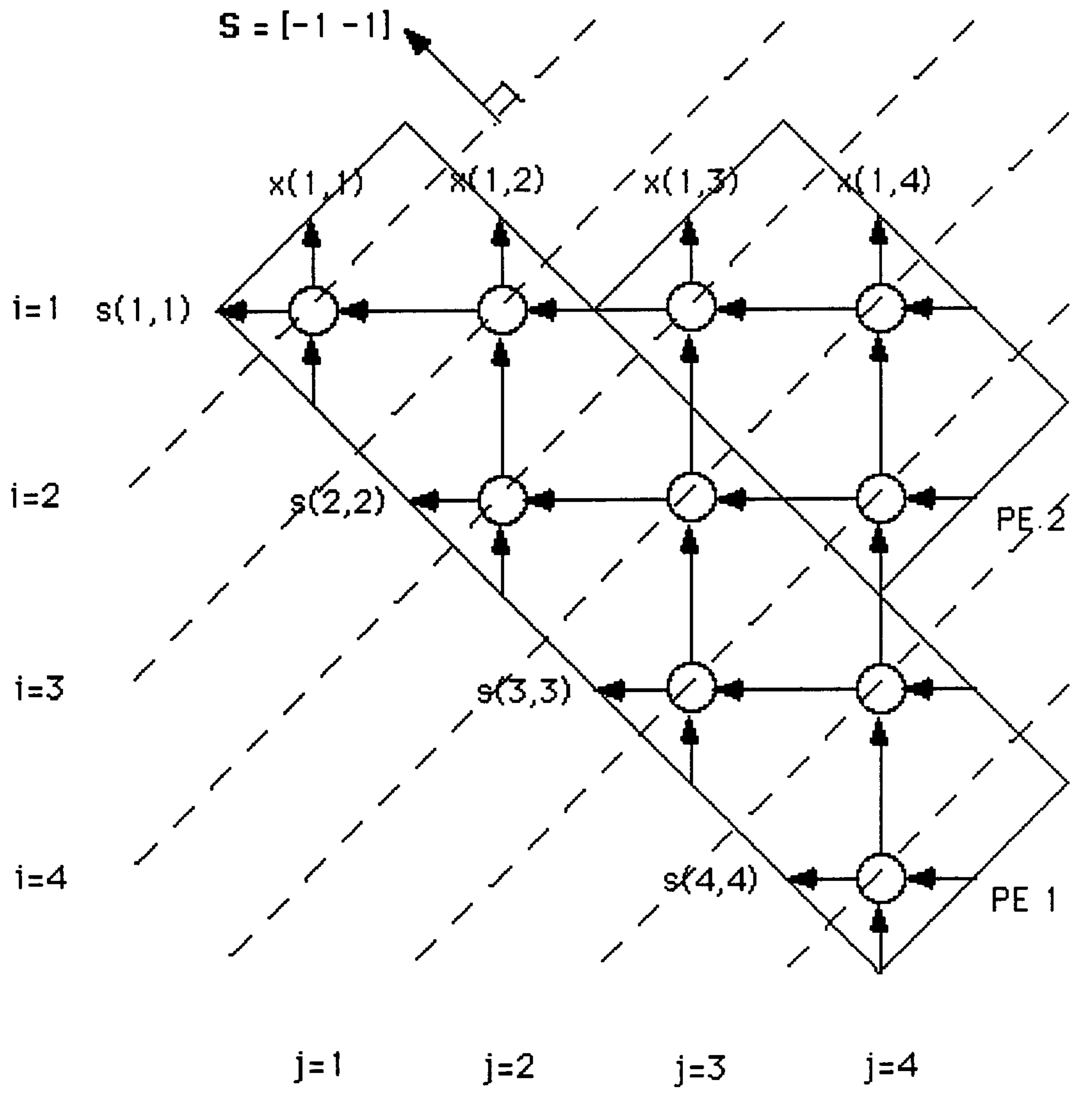

Fig. 11. Back-substitution with clustering mapping.

$$\lambda_a^T = [\,-1 \;\; 1\,]$$

with the projection direction along

$$u = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

$$\delta = \lambda_T^T u = [\,-1 \;\; -1\,] \begin{bmatrix} -1 \\ -1 \end{bmatrix} = 2$$

We can merge 2 neighboring processors and derive a new array which is fully efficient. The new allocation function is changed to:

$$a(p) = \left\lceil [\,-\tfrac{1}{2} \;\; \tfrac{1}{2}\,]\; p + 1 \right\rceil \tag{5.4}$$

The new array has the same computation time and has only 1/2 number of processors as the old array. Moreover, processors in the new array have the same function units without additional registers required (see Table 3 and 4).

5.2 Performance Evaluation of Cluster Toeplitz Solver

Cluster Algorithm I (Schur Algorithm)

Fig. 12. shows the simulated array processors of Schur algorithm by using clustering mapping method. Speed up on the new structure will be the same as before (O(n)). However, the number of processor units, which will be n/2, decrease by factor of δ. The efficiency is

$$E = \frac{S}{P}$$

$$E = \frac{2(n-1)}{4n-5}$$

For large n, E will be close to $\frac{1}{2}$.

Cluster Algorithm II (Back-Substitution)

Simulated Back-substitution solution by using clustering mapping method is shown in Fig. 13. The number of processor units have been decreased by factor of 2. The efficiency is

$$E = \frac{n+1}{2n-1}$$

For very large number of n, E will be close to $\frac{1}{2}$. The clustering mapping method have doubled the efficiency of both Algorithm I and II's arrays by decreasing number of PEs. Moreover, the new arrays even reducing the space complexity by decreasing the total number of registers required (as comparing Table 1,3 to Table 2,4).

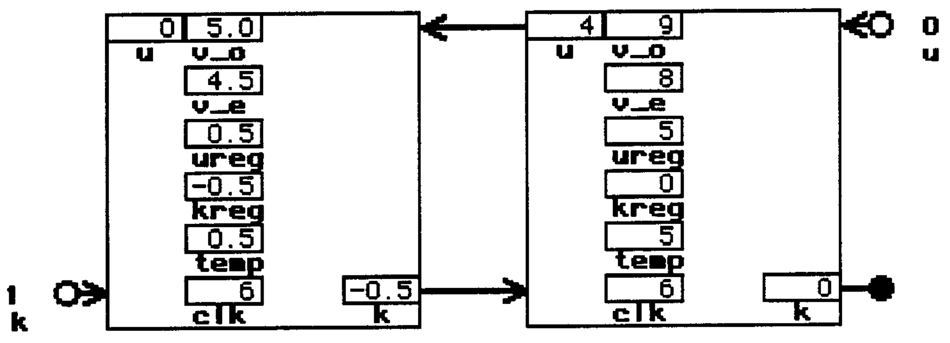

Fig. 12. Schur algorithm - clustering result.

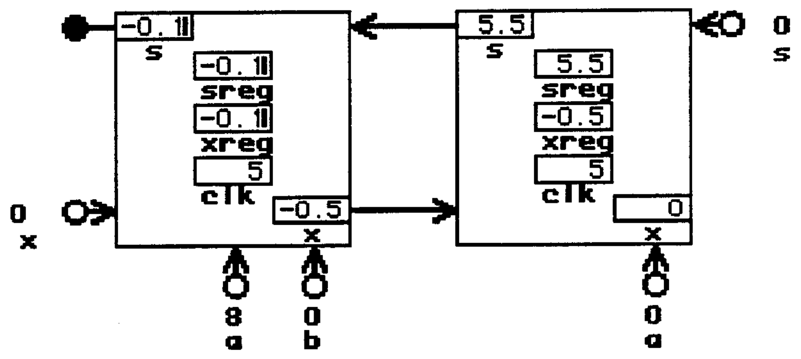

Fig. 13. Back-substitution - clustering result.

Chapter 6. Multi-Rate Array Solution

6.1 Advantage of MRA Solution

For many applications, due to the uniform data propagation of systolic arrays additional delay time for the transmission of data is required which degrades the computational rate and the processor utilization. This additional delay time could be avoided by using the Multi-Rate Array (MRA) structure where the variables are propagated at different rates achieving higher speedup and efficiency [17].

A MRA can be defined as a space-time domain with the following constraints: 1) the data dependencies are spatially and temporally local, and correspond to nearest neighbor interconnections; 2) data dependencies preserve causality; 3) propagation of different variables in the PE can vary. The mapping of an algorithm on a MRA is characterized by the same constraints which are characterizing the mapping of URE on systolic arrays [16].

In MRA, the processing time of the slowest operation is chosen to be the basic time unit, the fast processing operation takes only a fractional part of the basic unit $\frac{1}{f}$. The slowest operation usually involves complex arithmetic calculation with the largest propagation delay and the fast operation can be as simple as propagating values to next processor with short delay. Thus, the hardware can take advantages of different propagation rates to maximize the system throughput. The constraints of this type of MRA [11,12]:

(1) $d_{basic} = \lambda_T^T e_{slow} \geq 1$

(2) $d_{frac} = \lambda_T^T e_{fast} \geq \frac{1}{f}$

(3) $\lambda_T^T \lambda_a \neq 0$ (conflict free condition)

6.2 MRA Solution of Algorithm I (Schur Algorithm)

In previous chapters, allocation function has been chosen so that **v** has been stored locally without any propagation is needed; however, n-1 time units has been required for data pre-loading **u** before the computations begins. By changing the allocation function, the steps of data pre-loading can be eliminated; however, the delay of propagating **v** is different from other variables **u** and **K** (as shown in Ch. 4.4.1).

Closer examination of the Algorithm I that the variable **v** and **u** requires the same hardware complexity both multiplications and additions where the data values **u** and **K** are transmitted variables and the data values **v** is stored locally in previous solutions. The propagation delay of **v** is 2, and the propagation delay of **u** is only 1. By choosing the processing time of the **v** operation to be the basic time unit, the processing time of **u** operation is $\frac{1}{2}$ of the basic unit. In this case, it will not have any performance improvement to chose the longest delay to be the basic time unit 1 without increasing the clock rate because both variables **u** and **v** have the same hardware complexity. Therefore, by adding extra buffers to increase the delay of **v** to be 2 as:

(1) $d_{basic} = \lambda_T^T e_v = 2$

(2) $d_{frac} = \lambda_T^T e_u = \lambda_T^T e_k = 1.$

Allocation Function: Fig. 14 shows the projection vector of $\lambda_a^T =$ [1 0]. Allocation function: a(p) = [1 0] p - 1

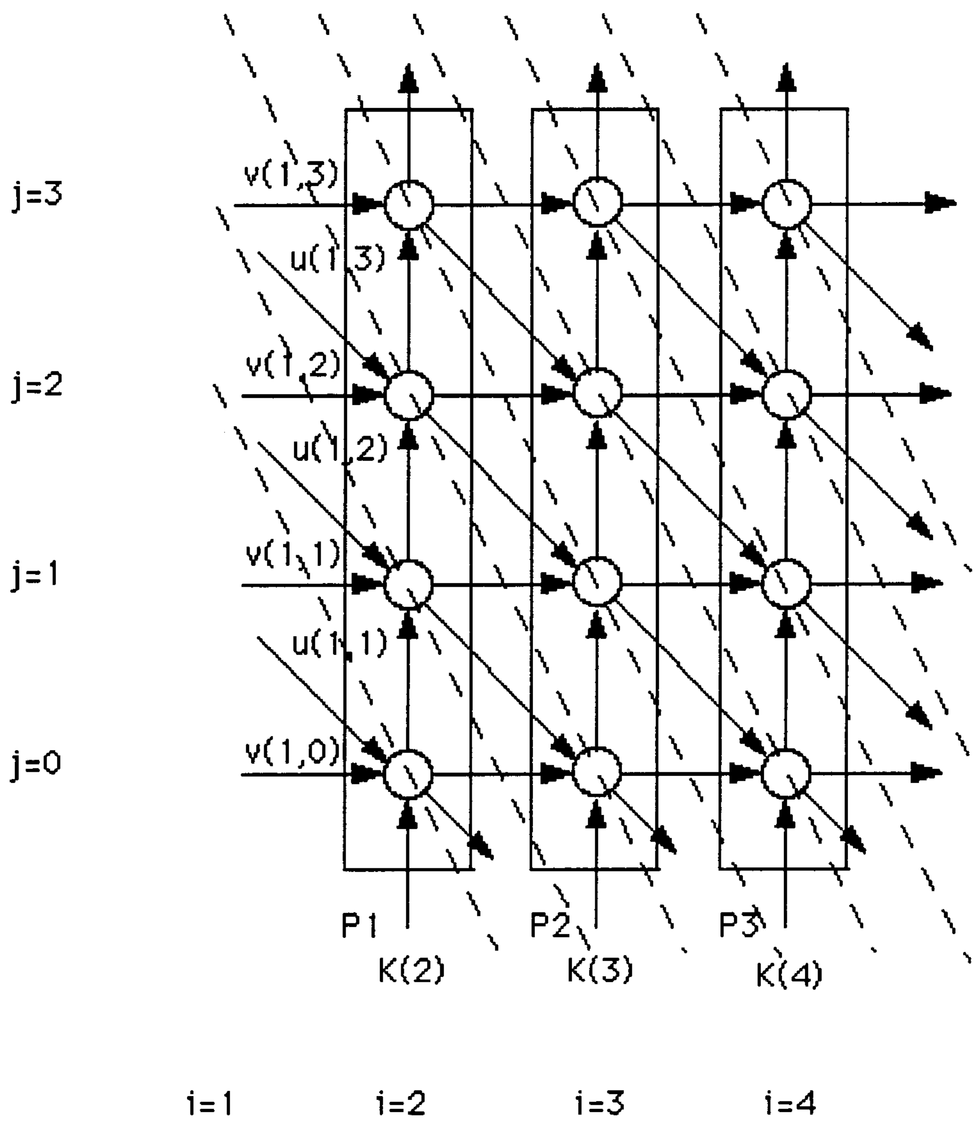

Fig. 14. Allocation function of MRA solution.

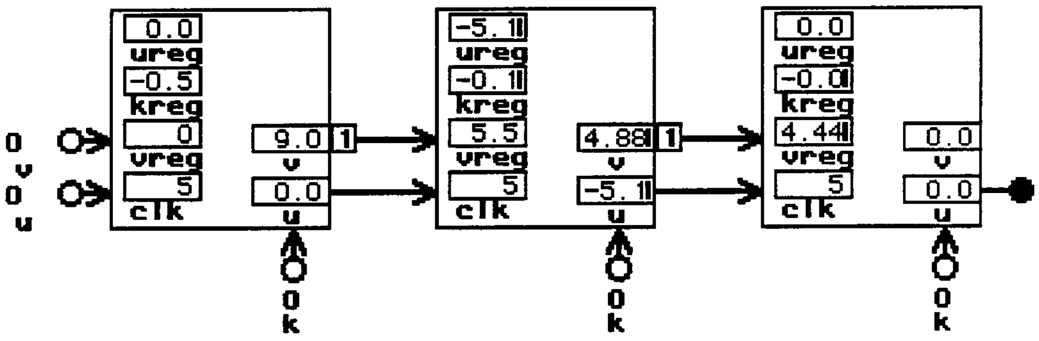

Fig. 15. MRA simulation realization

CLK	Processor 1			Processor 2			processor 3		
1	K_2	u_{20}	v_{20}						
2	K_2	u_{21}	v_{21}						
3	K_2	u_{22}	v_{22}	K_3	u_{30}	v_{30}			
4	K_2	u_{23}	v_{23}	K_3	u_{31}	v_{31}			
5				K_3	u_{32}	v_{32}	K_4	u_{40}	v_{40}
6				K_3	u_{33}	v_{33}	K_4	u_{41}	v_{41}
7							K_4	u_{42}	v_{42}
8							K_4	u_{43}	v_{43}

Table 5. Space-time table of MRA solution.

6.3 Performance Evaluation of MRA Solution

The simulator of MRA shows that by eliminating the data pre-loading **u**, the array requires only 3n-4 steps to finish the computation of nxn matrix system (see Table. 5). With new allocation function, it also requires n-1 processor units. Each processor element requires an extra buffer for output delay of **v** variables. Speed up on the MRA structure will be:

$$S = \frac{T_s}{T_p} = \frac{n(n-1)}{3n-4}$$

The new MRA solution still has speed up of order O(n), but with about 1/3 increasing in the speed. Number of processor units, which now will be n-1. The efficiency:

$$E = \frac{S}{P} = \frac{n}{3n-4}$$

which the efficiency is close to $\frac{1}{3}$ for large of n.

Chapter 7. Combined Architecture

The final solution is given by

$$\mathbf{x} = \mathbf{T}^{-1}\mathbf{y} = \mathbf{U}^{-1}\mathbf{D}(\mathbf{U}^{T})^{-1}\mathbf{y} \qquad (2.5)$$

where $\mathbf{D}$ = diag[$u_1, \ldots, u_{(N+1)}$]; $\mathbf{U}$ is an upper triangular matrix obtained from Schur algorithm (Algorithm I). The solution $\mathbf{x}$ of (2.1) can be solved by two back-substitution (Algorithm II). We have obtained the architectures for the decomposition (Algorithm I) and back-substitution (Algorithm II). Example, we will show how to combine both architectures for 6 X 6 Toeplitz matrix.

$$\mathbf{U} = \begin{bmatrix} u_{10} & u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ 0 & u_{21} & u_{22} & u_{23} & u_{24} & u_{25} \\ 0 & 0 & u_{32} & u_{33} & u_{34} & u_{35} \\ 0 & 0 & 0 & u_{43} & u_{44} & u_{45} \\ 0 & 0 & 0 & 0 & u_{54} & u_{55} \\ 0 & 0 & 0 & 0 & 0 & u_{65} \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ 0 & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ 0 & 0 & a_{33} & a_{34} & a_{35} & a_{36} \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & 0 & a_{66} \end{bmatrix}$$

where $\mathbf{c} = \mathbf{A}^{-1}\mathbf{b}$ for back-substitution. Now let us first consider the first back-substitution step $(\mathbf{U}^{T})^{-1}\mathbf{y}$ which consists

$$\mathbf{U}^{T} = \begin{bmatrix} u_{10} & 0 & 0 & 0 & 0 & 0 \\ u_{11} & u_{21} & 0 & 0 & 0 & 0 \\ u_{12} & u_{22} & u_{32} & 0 & 0 & 0 \\ u_{13} & u_{23} & u_{33} & u_{43} & 0 & 0 \\ u_{14} & u_{24} & u_{34} & u_{44} & u_{54} & 0 \\ u_{15} & u_{25} & u_{35} & u_{45} & u_{55} & u_{65} \end{bmatrix}$$

In order to use the systolic solution of algorithm II for first back-substitution step, we can rearrange the input order as

$$\begin{bmatrix} a_{66} & 0 & 0 & 0 & 0 & 0 \\ a_{56} & a_{55} & 0 & 0 & 0 & 0 \\ a_{46} & a_{45} & a_{44} & 0 & 0 & 0 \\ a_{36} & a_{35} & a_{34} & a_{33} & 0 & 0 \\ a_{26} & a_{25} & a_{24} & a_{23} & a_{22} & 0 \\ a_{16} & a_{15} & a_{14} & a_{13} & a_{12} & a_{11} \end{bmatrix}$$

$$= \begin{bmatrix} u_{10} & 0 & 0 & 0 & 0 & 0 \\ u_{11} & u_{21} & 0 & 0 & 0 & 0 \\ u_{12} & u_{22} & u_{32} & 0 & 0 & 0 \\ u_{13} & u_{23} & u_{33} & u_{43} & 0 & 0 \\ u_{14} & u_{24} & u_{34} & u_{44} & u_{54} & 0 \\ u_{15} & u_{25} & u_{35} & u_{45} & u_{55} & u_{65} \end{bmatrix}$$

and

$$\begin{bmatrix} b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}$$

then the first back-substitution $\mathbf{g} = \mathbf{D}(\mathbf{U}^T)^{-1}\,\mathbf{y}$ can be rearrange as follow

$$\mathbf{g} = \mathbf{D} \begin{bmatrix} a_{66} & 0 & 0 & 0 & 0 & 0 \\ a_{56} & a_{55} & 0 & 0 & 0 & 0 \\ a_{46} & a_{45} & a_{44} & 0 & 0 & 0 \\ a_{36} & a_{35} & a_{34} & a_{33} & 0 & 0 \\ a_{26} & a_{25} & a_{24} & a_{23} & a_{22} & 0 \\ a_{16} & a_{15} & a_{14} & a_{13} & a_{12} & a_{11} \end{bmatrix}^{-1} \begin{bmatrix} b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \end{bmatrix}$$

CLK	U from systolic solution of Algorithm I					
4	u_{10}	-	-	-	-	-
5	-	u_{11}	-	-	-	-
6	-	-	u_{12}	-	-	-
7	-	u_{21}	-	u_{13}	-	-
8	-	-	u_{22}	-	u_{14}	-
9	-	-	-	u_{23}	-	u_{15}
10	-	-	u_{32}	-	u_{24}	-
11	-	-	-	u_{33}	-	u_{25}
12	-	-	-	-	u_{34}	-
13	-	-	-	u_{43}	-	u_{35}
14	-	-	-	-	u_{44}	-
15	-	-	-	-	-	u_{45}
16	-	-	-	-	u_{54}	-
17	-	-	-	-	-	u_{55}
18	-	-	-	-	-	-
19	-	-	-	-	-	u_{65}

u_{10} obtained directly from t_0.

Table 6. Output from systolic solution of Algorithm I.

CLK	Input data **a** for systolic solution of Algorithm II					
1	a_{66}	-	-	-	-	-
2	-	a_{56}	-	-	-	-
3	a_{55}	-	a_{46}	-	-	-
4	-	a_{45}	-	a_{36}	-	-
5	a_{44}	-	a_{35}	-	a_{26}	-
6	-	a_{34}	-	a_{25}	-	a_{16}
7	a_{33}	-	a_{24}	-	a_{15}	-
8	-	a_{23}	-	a_{14}	-	-
9	a_{22}	-	a_{13}	-	-	-
10	-	a_{12}	-	-	-	-
11	a_{11}	-	-	-	-	-

Table 7. Input of systolic solution of Algorithm I.

One should notice that the space-time of data output from Algorithm I solution (Table 6) is not the same as the input of Algorithm II (Table 7). The following procedures shows how to combine the two algorithms.

Procedure I: store the output **U** from systolic solution of Algorithm I into (3n-2 X n) memory cells (as in form of Table 8 starting at CLK=2).

Procedure II: perform parallel shift (see Fig. 16) and produce the output as in the form of Table. 10a which can be directly feed into the input of systolic solution of Algorithm II and also store into a (2n-1 X n) LIFO I which in a last-in-first-out buffer (see Fig. 17).

The output from the systolic solution of Algorithm II can be multiplied by the scaling operator **D** to generate result **g** and store it into a (1 X n) LIFO II (see Fig. 17). Now, the second step of the back-substitution $\mathbf{x} = \mathbf{U}^{-1}\mathbf{g}$ computation can start. **U** is available from LIFO I as in form of Table 10b and **g** is available from LIFO II. This is shown in Fig. 17.

For the clustering method, we use the same two procedures described above except one has to separate the output data **U** from the Algorithm I and then merge the output **U** rearranged in the parallel shift. The separation and merging can be performed by using transmission gates controlled by Flip-Flops. In this case, the size of LIFO I is reduced into half to (2n-1 X $\frac{n}{2}$).

Operation Overview

The following steps illustrates the above procedures (also see Fig. 17):

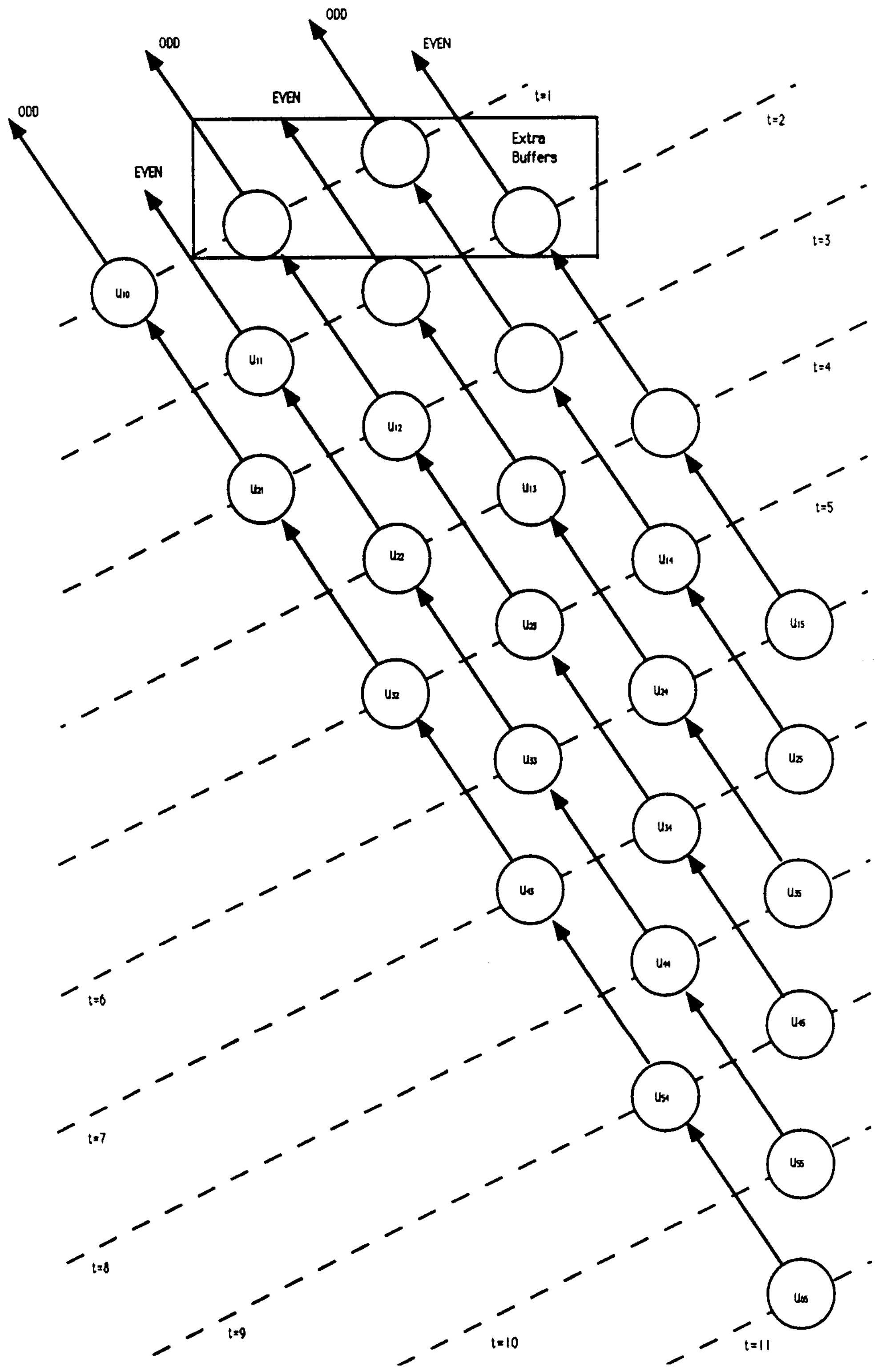

Fig. 16 Parallel shift for manipulating the data **U** which output from Algorithm I's solution to feed into Algorithm II.

CLK	Order and position of **U** required for first back-substitution step					
1	u10	-	-	-	-	-
2	-	u11	-	-	-	-
3	u21	-	u12	-	-	-
4	-	u22	-	u13	-	-
5	u32	-	u23	-	u14	-
6	-	u33	-	u24	-	u15
7	u43	-	u34	-	u25	-
8	-	u44	-	u35	-	-
9	u54	-	u45	-	-	-
10	-	u55	-	-	-	-
11	u65	-	-	-	-	-

Table 8a. Required order of **U** for first back-substitution step.

CLK	Order and position of **U** for second back-substitution step					
1	u65	-	-	-	-	-
2	-	u55	-	-	-	-
3	u54	-	u45	-	-	-
4	-	u44	-	u35	-	-
5	u43	-	u34	-	u14	-
6	-	u33	-	u24	-	u15
7	u32	-	u23	-	u25	-
8	-	u22	-	u13	-	-
9	u21	-	u12	-	-	-
10	-	u11	-	-	-	-
11	u10	-	-	-	-	-

Table 8b. Required order of **U** for second back-substitution step.

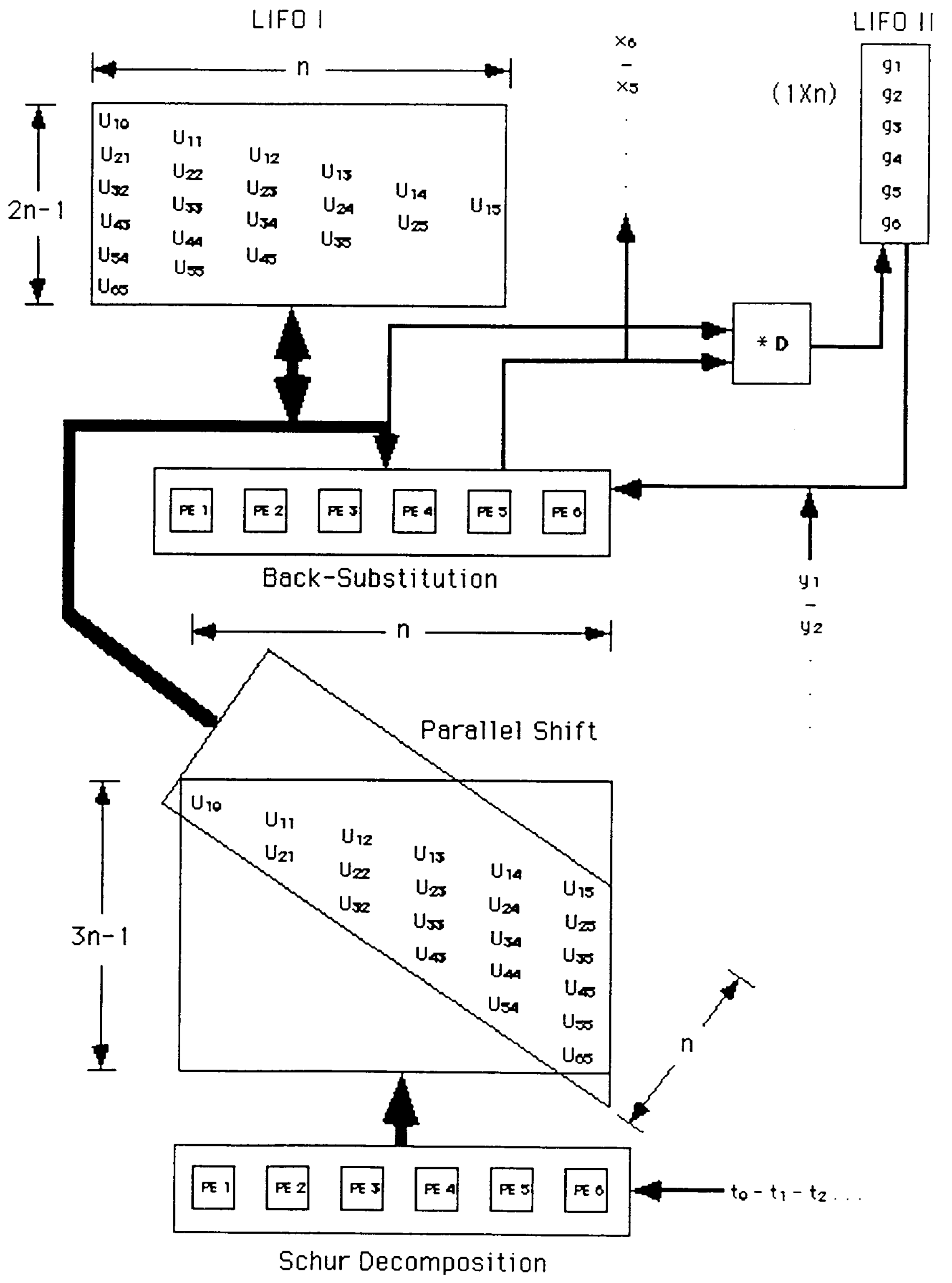

Figure 17. Overview of combined architecture.

Stage I: Decompose **T** to obtain **U**. This will takes 4n-5 cycles, and the first output will start after n-1 cycle (see Table 6).

Stage II: The output data **U** from the decomposition (Algorithm I) can be stored into 3n-2 memory cells immediately in one cycle.

Stage III: Parallel shift **U** for data manipulation, this takes 2n-1 cycles.

Stage IV: The output of parallel shift **U** enters the linear pipeline array for the back-substitution (Algorithm II) in one cycle. Note that the data is also be stored into LIFO I at the same time. First output of back-substitution array is produced and following data outputs in every two cycle after (see Table 3).

Stage V: Now we multiply $(\mathbf{U}^T)^{-1}\mathbf{y}$ with the scalar **D** to produce **g** which can performed after the output from the first back-substitution. This takes one cycle.

Stage VI: Stores **g** into LIFO II in one cycle.

Stage VII: To obtain **x**, the second back-substitution $\mathbf{U}^{-1}\mathbf{g}$ will take 2n-1 cycles.

Based on this, input data of **T** will be provided every two cycles beginning at Stage I (clock = 1) and **y** also is available in every two cycles starting at Stage IV 4n-3 cycles later. The output of **x** computed every two cycles beginning after Stage VII (clock = 6n-1). The total computation steps requires then is (4n-5) + 1 + (2n-1) + 1 + 1 + 1 + (2n-1) = 8n-3. Time complexity of complete solution is in order of O(n). With clustering method, the solution requires only n/2 processing elements instead of n developed elsewhere [1].

CLK	**U** from Cluster Array of Algorithm I		
4	*u_{10}*	-	-
5	u_{11}	-	-
6	-	u_{12}	-
7	u_{21}	u_{13}	-
8	-	u_{22}	u_{14}
9	-	u_{23}	u_{15}
10	-	u_{32}	u_{24}
11	-	u_{33}	u_{25}
12	-	-	u_{34}
13	-	u_{43}	u_{35}
14	-	-	u_{44}
15	-	-	u_{45}
16	-	-	u_{54}
17	-	-	u_{55}
18	-	-	-
19	-	-	u_{65}

u_{10} obtained directly from t_0.

Table 9a. Output from Cluster Array of Algorithm I.

CLK	Order and position of **U** required for first back-substitution step on Cluster Array		
1	u_{10}	-	-
2	u_{11}	-	-
3	u_{21}	u_{12}	-
4	u_{22}	u_{13}	-
5	u_{32}	u_{23}	u_{14}
6	u_{33}	u_{24}	u_{15}
7	u_{43}	u_{34}	u_{25}
8	u_{44}	u_{35}	-
9	u_{54}	u_{45}	-
10	u_{55}	-	-
11	u_{65}	-	-

Table 9b. **U** for Cluster Array of first back-substitution step.

Summary

Effective pipelinability, regularity and synchronization of systolic array structure provide an ideal implementation environment. Different solutions of Toeplitz have been obtained at the performance is summarized into Table 10 and 11. These three solutions (systolic, clustering, and MRA) have achieved linear computational complexity, space, and speed up. Efficiency vs n (size of matrix) for different solutions have been plotted on Fig. 18 and Fig. 19. The clustering mapping method has the highest efficiency as compared to other cases.

The clustering mapping method increasing the efficiency by a factor of δ; however, it doesn't has 100% efficiency which is because the array have to pre-load and un-load the data. The processors will be idle for some time units to load the first data once it finishes all computations, it becomes idle again but not all the processors stop at the same time. Hence, the utilization of a processor should be defined as the ratio of its active time over the whole computation time. Indeed, in many applications, the samples of real-time process come in infinitely. The time correspond to the execution of an infinity of identical computations, speed up and efficiency are then in steady state. Speed up (S) will be close to number of processors (P). Efficiency (E) will be close to 100%. Since in this case one neglects the amount of time spent in loading the data and unloading the results [2,3].

Multi-Rate Array solution of Schur algorithm reduces the number of computational steps by 25%; however, it requires n-1 processing elements as in comparison to n/2 of clustering mapping method. Since the computations of **u** and **v** variables have the same complexity, MRA solution cannot take advantage of short propagation rate of **u** to provide additional two fold of speed up. There may have some cases that clustering method and MRA can be combined to both reduce number of processing elements and enhance the speed. However, in solving the Toeplitz system, solution of MRA has δ equal to one, the number of processing elements cannot be reduced by using clustering mapping method.

	Systolic Solution	Clustering Mapping	Multi-Rate Array
No. of PEs	n	$\frac{n}{2}$	$n-1$
No. of Registers per PE	3	4	3
No. of Delay Buffers per PE	0	0	1
No. of Steps	$4n-5$	$4n-5$	$3n-4$
Speed Up	$\frac{n^2-n}{4n-5}$	$\frac{n^2-n}{4n-5}$	$\frac{n(n-1)}{3n-4}$
Efficiency	$\frac{n-1}{4n-5}$	$\frac{2(n-1)}{4n-5}$	$\frac{n}{3n-4}$
Computational Complexity	$O(n)$	$O(n)$	$O(n)$
Order of Space	$O(n)$	$O(n)$	$O(n)$
Order of Speed Up	$O(n)$	$O(n)$	$O(n)$
Pipeline Rate	$\frac{1}{2}$	1	1

Table 10. Summary of solutions of Schur algorithm.

	Systolic Solution	Clustering Mapping
No. of PEs	n	$\frac{n}{2}$
Max. No. of Registers per PE	4	4
No. of Steps	2n-1	2n-1
Speed Up	$\frac{n(n+1)}{2(2n-1)}$	$\frac{n(n+1)}{2(2n-1)}$
Efficiency	$\frac{n+1}{2(2n-1)}$	$\frac{n+1}{2n-1}$
Computational Complexity	O(n)	O(n)
Order of Space	O(n)	O(n)
Order of Speed Up	O(n)	O(n)
Pipeline Rate	$\frac{1}{2}$	1

Table 11. Summary of solutions of Back-substitution.

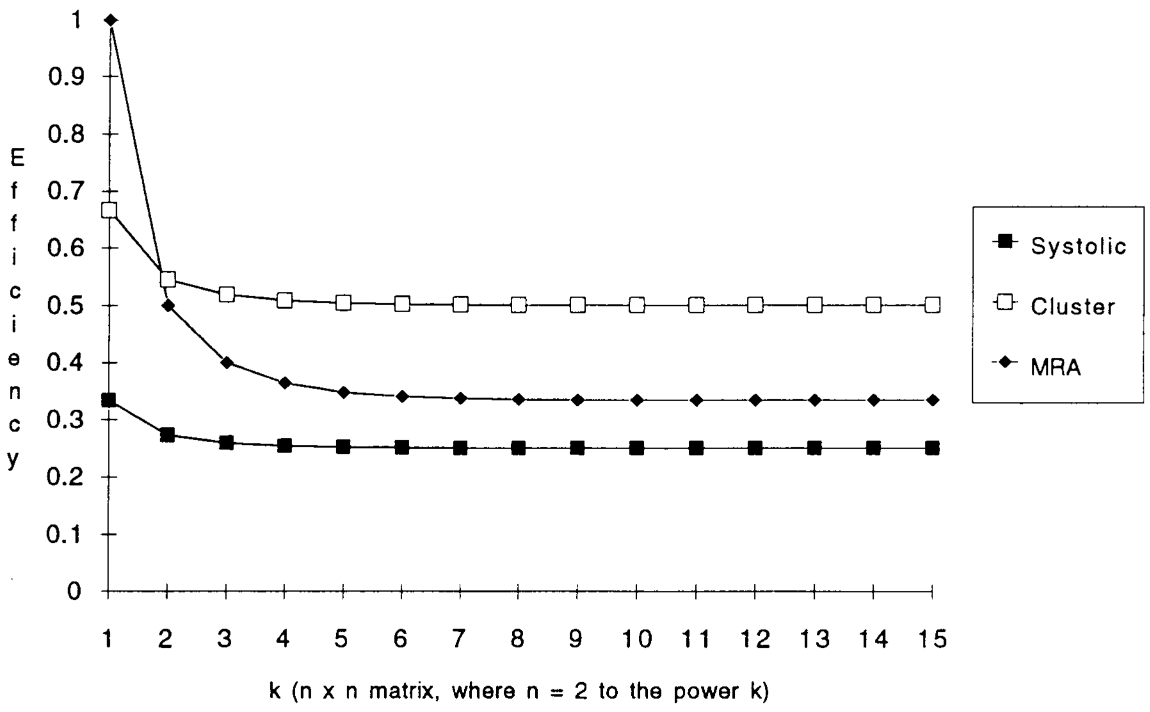

Fig. 18 Efficiency Vs n for Algorithm I (Decomposition)

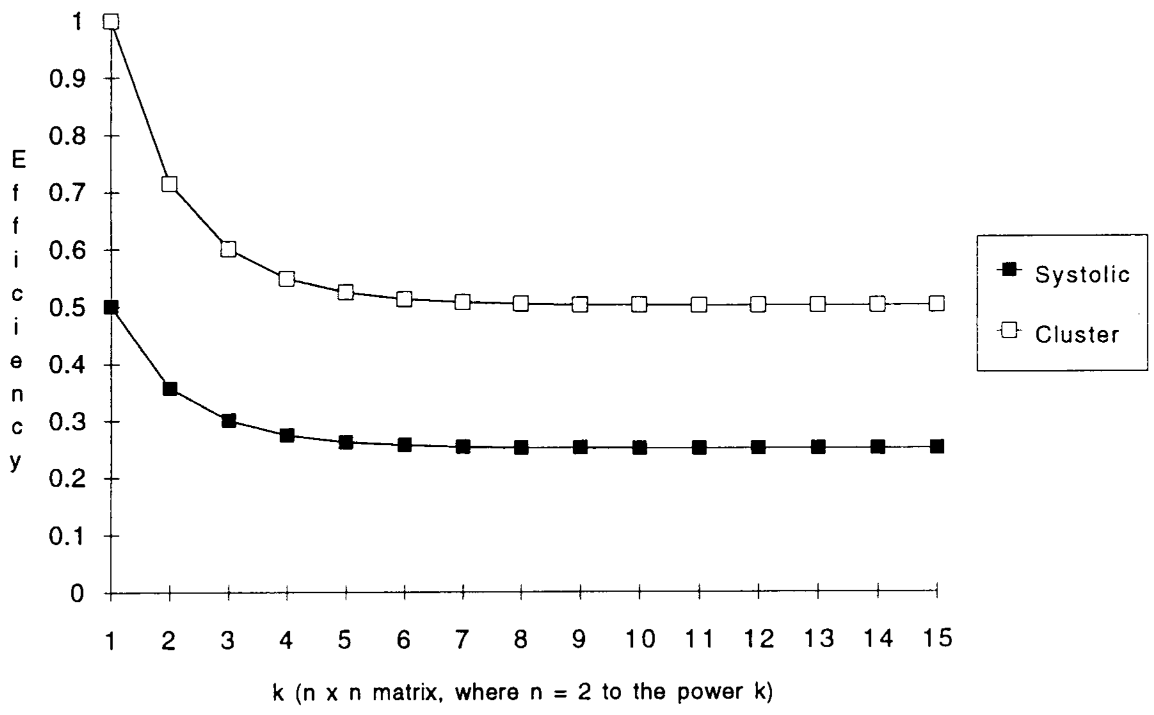

Fig. 19. Efficiency Vs n for Algorithm II (Back-substitution)

References

[1] Sun-Yuan Kung and Yu Hen Hu, "A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems", IEEE Trans. on ASSP, Vol. 31, No. 1, Feb. 1983.

[2] Xiaoxiong Zhong and Sanjay Rajopadhye, "Synthesizing fully efficient systolic arrays", Computer Science Dept., University of Oregon., 1991.

[3] J. P. Charlier, M. Vanbegin and P. Van Dooren, "Systolic algorithms for digital signal processing", Philips Journal of Research, Vol. 43, pp. 268-290, Nos 3/4, 1988.

[4] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed-size systolic arrays", IEEE Trans. Computers, Vol. C-35, No. 1, pp. 1-12, Jan. 1986.

[5] W. L. Miranker and A. Winkler, "Spacetime representations of computational structures", Computing, Vol. 32, pp. 93-114, 1984.

[6] H. T. Kung, "Why systolic architectures", IEEE Computer, pp. 37-45, Jan. 1982.

[7] Yoav Yaacoby and Peter R. Cappello, "Scheduling a system of affine recurrence equations onto a systolic array", Dept. of Electrical & Computer Engineering, and Dept. of Computer Science, U. of California, Santa Barbara, 1988.

[8] Sanjay V. Rajopadhye and Richard M. Fujimoto, "Synthesizing systolic arrays form recurrence equations", Parallel Computing, 1988.

[9] R. M. Karp, R. E. Miller, and S. Winogard, "The organization of computations for uniform recurrence equations", JACM 14, 3, pp. 563-590, July 1967.

[10] Sanjay V. Rajopadhye, "Synthesizing systolic arrays with control signals from recurrence equations", Distributed Computing, pp. 88-105, May 1989.

[11] S. Kiaei and L. Aihua, "Analysis of multi-rate/bounded broadcast", Technical Report, Electrical and Computer Engineering Dept., Oregon State University, 1989.

[12] L. Aihua and S. Kiaei, "VLSI design of multi-rate arrays for DSP algorithms", 1990 Int. Conf. on ASSP, New Mexico.

[13] P. Quinton, "The systematic design of systolic arrays", IRISA Research Report, No. 193, 1983.

[14] Sailesh Rao, "Regular iterative algorithms and their implementations on processor arrays", PhD thesis, Standard University, Information System Lab., Standard, CA, Oct. 1985.

[15] A. V. Oppenheim, Ed., Applications of Digital Signal Processing, Englewood cliffs, NJ: Prentice-Hall, 1978.

[16] Patrick M. Lenders and Sayfe Kiaei, "Synthesis and Automatic Derivation of Multi-Dimensional MRA's", Submitted to IEEE Transaction on Parallel and Distributes Computing, 1991.

[17] Sayfe Kiaei, "Synthesis and Systematic Derivation of Multi-Rate VLSI Arrays", Dept. of Electrical and Computer Engineering, Oregon State University, 1991.

[18] Christophe P. Rialan and Louis L. Scharf, "Fast algorithms for computing QR and Cholesky factors of Toeplitz operators", IEEE Trans. on ASSP, Vol. 36, No. 11, pp. 1740-1747, Nov. 1988.

[19] J. R. Bunch, "Stability of Methods for solving Toeplitz systems of equations", SIAM J. Sci. Statist. Comput., pp. 349-364, Vol. 6, 1985.

[20] Golub H. Golub and Charles F. Van Loan, "Matrix computations", The Johns Hopkins University Press.

[21] F. de Hoog, "A new algorithm for solving Toeplitz systems of equations", Lin. Algebra Appl., pp. 122-138, 1987.

[22] Jean Marc Delosme and Ilse C. F. Ipsen, "Efficient systolic arrays for the solution of Toeplitz systems: An illustration of a methodology for the construction of systolic architectures in VLSI", Research Report YALEU/DCS/RR-370, June 1985.

[23] Gregory S. Ammar and William B. Gragg, "Numerical experience with a superfast real Toeplitz solver", Naval Postgraduate School, NPS-53-89-008, Feb. 1989.

[24] Ben Manuto and Sanjay V. Rajopadhye, Systolic Array Simulator, Computer Science Department, University of Oregon.

Appendices

Appendix A. Simulation Code of Systolic Solution for Schur Algorithm

```
(Toeplitz_Matrix

 ;processor type-a declaration.
 ((ptype-a
  (var
   (vreg float 0)
   (ureg float 0)
   (kreg float 0)
   (clk int 0))
  (inputs
   (v float 0)
   (u float 0)
   (k float 0))
  (outputs
   (u float 0 (-1 0) 0)
   (k float 0 (1 0) 0))
  (code

   ;initialize v registers at clk 0.
   ((if (equal? clk 0)
     (begin
       (set! vreg v.in)
       (set! ureg u.in)
       (set! kreg k.in)
        (set! u.out (+ ureg (* kreg vreg)))
        (set! vreg (+ vreg (* kreg ureg)))
       (set! ureg u.out)
       (set! k.out kreg)
        (set! clk (+ clk 1)))
     (begin
       (set! ureg u.in)
       (set! kreg k.in)
        (set! u.out (+ ureg (* kreg vreg)))
        (set! vreg (+ vreg (* kreg ureg)))
       (set! ureg u.out)
       (set! k.out kreg)
        (set! clk (+ clk 1)))
     ))
 ))

;declaration of processor type-b.
(ptype-b
 (var
  (vreg float 0)
  (ureg float 0)
  (kreg float 0)
  (clk int 0))
 (inputs
  (v float 0)
  (u float 0)
  (k float 0))
 (outputs
  (u float 0 (-1 0) 0)
  (k float 0 (1 0) 0))
(code

  ;initialize v registers at clk 0.
  ((if (equal? clk 0)
     (begin
       (set! vreg v.in)
       (set! ureg u.in)
        (set! kreg (* (/ (- 0 ureg) vreg) k.in))
        (set! u.out (+ ureg (* kreg vreg)))
        (set! vreg (+ vreg (* kreg ureg)))
       (set! ureg u.out)
       (set! k.out kreg)
        (set! clk (+ clk 1)))
     (begin
       (set! ureg u.in)
        (set! kreg (* (/ (- 0 ureg) vreg) k.in))
        (set! u.out (+ ureg (* kreg vreg)))
        (set! vreg (+ vreg (* kreg ureg)))
       (set! ureg u.out)
       (set! k.out kreg)
        (set! clk (+ clk 1)))
     ))
 )))


((instantiate ptype-a (line (2) (4)))
 (instantiate ptype-b (point (1))))
```

```
() ; no connections need be specified.

 ((input-data
   ((v gets v-inputs from (1) to (4)))
  (input (i = 1)
         (j = pp.i)))

  (input-data
   ((u gets u-inputs at (4)))
  (input (i = 1)
         (j = t)))

  (input-data
   ((k gets k-inputs at (1)))
  (input (i = 1)
         (j = t))))

 (output-data
  ((k at (4)))))
```

Sample data

```
(v-inputs
 (6 7 8 9))

(u-inputs
 (3 0 4 0 5 0 0 0 0 0 0 0 0 0 0 0 0))

(k-inputs
 (0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0))
```

Appendix B. Simulation Code of Clustering Mapping Method for Schur Algorithm

```
(Toeplitz_Matrix

 ;declaration of processor type-a.
 ((ptype-a
  (var
   (v_o float 9)
   (v_e float 8)
   (ureg float 0)
   (kreg float 0)
   (temp float 0)
   (clk int 0))
  (inputs
   (u float 0)
   (k float 0))
  (outputs
   (u float 0 (-1 0) 0)
   (k float 0 (1 0) 0))
  (code
   ((if (equal? (modulo clk 2) 0)

      ;even cycle operation.
     (begin
      (set! kreg k.in)
      (set! temp (+ ureg (* kreg v_e)))
      (set! v_e (+ v_e (* kreg ureg)))
      (set! ureg temp)
      (set! u.out ureg)
      (set! clk (+ clk 1)))

      ;odd cycle operation.
     (begin
      (set! ureg u.in)
      (set! temp (+ ureg (* kreg v_o)))
      (set! v_o (+ v_o (* kreg ureg)))
      (set! ureg temp)
      (set! k.out kreg)
      (set! clk (+ clk 1)))
     ))
  ))
;declaration of processor type-b.
(ptype-b
(var
  (v_o float 7)
  (v_e float 6)
  (ureg float 0)
  (kreg float 0)
  (temp float 0)
  (clk int 0))
 (inputs
  (u float 0)
  (k float 0))
 (outputs
  (u float 0 (-1 0) 0)
  (k float 0 (1 0) 0))
(code

  ((if (equal? (modulo clk 2) 0)

      ;even cycle operation.
     (begin
        (set! kreg (* (/ (- 0 ureg) v_e) k.in))
        (set! temp (+ ureg (* kreg v_e)))
        (set! v_e (+ v_e (* kreg ureg)))
       (set! ureg temp)
       (set! u.out ureg)
        (set! clk (+ clk 1)))

      ;odd cycle operation.
     (begin
       (set! ureg u.in)
        (set! temp (+ ureg (* kreg v_o)))
        (set! v_o (+ v_o (* kreg ureg)))
       (set! ureg temp)
       (set! k.out kreg)
        (set! clk (+ clk 1)))
     ))
 )))


((instantiate ptype-a (point (2)))
 (instantiate ptype-b (point (1))))
```

```
() ; no connections need be specified.

 ((input-data
   ((u gets u-inputs at (2)))
  (input (i = 1)
         (j = t)))

  (input-data
   ((k gets k-inputs at (1)))
  (input (i = 1)
         (j = t))))

 (output-data
  ((k at (2)))))
```

Sample data

```
(u-inputs
 (0 3 0 4 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0))

(k-inputs
 (0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0))
```

Appendix C. Simulation Code of MRA Solution for Schur Algorithm

```
(Toeplitz_Matrix

 ;declaration of processor type-a.
 ((ptype-a
  (var
   (ureg float 0)
   (kreg float 0)
   (vreg float 0)
   (clk int 0))
  (inputs
   (v float 0)
   (u float 0)
   (k float 0))
  (outputs
   (v float 0 (1 0) 1)
   (u float 0 (1 0) 0))
  (code

   ;check divided by zero.
   ((if (equal? (+ v.in 0.0) 0.0)
       (begin
         (set! kreg kreg)
         (set! u.out (+ u.in (* kreg v.in)))
         (set! v.out (+ v.in (* kreg u.in)))
         (set! ureg u.out)
         (set! vreg v.out)
         (set! clk (+ clk 1)))
       (begin
         (set! kreg (+ (* (/ (- 0 u.in) v.in) k.in) kreg))
         (set! u.out (+ u.in (* kreg v.in)))
         (set! v.out (+ v.in (* kreg u.in)))
         (set! ureg u.out)
         (set! vreg v.out)
         (set! clk (+ clk 1)))
     ))
   )))


 ((instantiate ptype-a (line (1) (3))))
() ; no connections need be specified.

 ((input-data
   ((k gets k-inputs from (1) to (3)))
  (input (i = t)
        (j = pp.i)))

  (input-data
   ((u gets u-inputs at (1)))
  (input (i = 1)
        (j = t)))

  (input-data
   ((v gets v-inputs at (1)))
  (input (i = 1)
        (j = t))))

 (output-data
  ((u at (3)))))
```

Sample data

```
(v-inputs
 (6 7 8 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

(u-inputs
 (3 4 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

(k-inputs
 (1 0 0)
 (0 0 0)
 (0 1 0)
 (0 0 0)
 (0 0 1)
 (0 0 0)
 (0 0 0))
```

Appendix D. Simulation Code of Systolic Solution for Back-Substitution

```
(Back_Substitution

 ;processor type-a declaration.
 ((ptype-a
  (var
   (sreg float 0)
   (xreg float 0)
   (clk int 0))
  (inputs
   (s float 0)
   (x float 0)
   (a float 0))
  (outputs
   (s float 0 (-1 0) 0)
   (x float 0 (1 0) 0))
  (code
   ((if (equal? clk 0)
      (begin
        (set! xreg x.in)
        (set! sreg s.in)
         (set! sreg (+ sreg (* a.in xreg)))
        (set! x.out xreg)
        (set! s.out sreg)
         (set! clk (+ clk 1)))
      (begin
        (set! xreg x.in)
        (set! sreg s.in)
         (set! sreg (+ sreg (* a.in xreg)))
        (set! x.out xreg)
        (set! s.out sreg)
         (set! clk (+ clk 1)))
      ))
  ))

  ;declaration of processor type-b.
  (ptype-b
  (var
   (sreg float 0)
   (xreg float 0)
   (clk int 0))
  (inputs
   (s float 0)
   (x float 0)
   (a float 0)
   (b float 0))
  (outputs
   (s float 0 (-1 0) 0)
   (x float 0 (1 0) 0))
 (code
   ((if (equal? (+ a.in 0.0) 0.0)
       (begin
         (set! xreg x.in)
         (set! sreg s.in)
         (set! sreg 0)
         (set! xreg sreg)
         (set! s.out sreg)
         (set! x.out xreg)
          (set! clk (+ clk 1)))
       (begin
         (set! xreg x.in)
         (set! sreg s.in)
          (set! sreg (/ (- b.in sreg) a.in))
         (set! xreg sreg)
         (set! s.out sreg)
         (set! x.out xreg)
          (set! clk (+ clk 1)))
       ))
  )))


((instantiate  ptype-a  (line  (2)  (4)))
  (instantiate  ptype-b  (point  (1))))

() ; no connections need be specified.

((input-data
  ((s gets s-inputs at (4)))
 (input (i = 1)
        (j = t)))

 (input-data
  ((a gets a-inputs from (1) to (4)))
  (input (i = t)
         (j = pp.i)))

  (input-data
   ((x gets x-inputs at (1)))
  (input (i = 1)
         (j = t)))

  (input-data
   ((b gets b-inputs at (1)))
  (input (i = 1)
         (j = t))))

 (output-data
 ((s at (1)))))
```

Sample data

```
(s-inputs
 (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

(a-inputs
 (1 0 0 0)
 (0 3 0 0)
 (2 0 6 0)
 (0 5 0 10)
 (4 0 9 0)
 (0 8 0 0)
 (7 0 0 0)
 (0 0 0 0)
 (0 0 0 0)
 (0 0 0 0))

(x-inputs
 (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

(b-inputs
 (1 0 2 0 3 0 4 0 0 0 0 0 0 0 0 0 0 0))
```

Appendix E. Simulation Code of Clustering Mapping Method for Back-Substitution

```
(Back_Substitution

 ;processor type-a declaration.
 ((ptype-a
  (var
   (sreg float 0)
   (xreg float 0)
   (clk int 0))
  (inputs
   (s float 0)
   (x float 0)
   (a float 0))
  (outputs
   (s float 0 (-1 0) 0)
   (x float 0 (1 0) 0))
  (code
   ((if (equal? (modulo clk 2) 0)
      (begin
        (set! xreg x.in)
         (set! sreg (+ sreg (* a.in xreg)))
        (set! s.out sreg)
         (set! clk (+ clk 1)))
      (begin
        (set! sreg s.in)
         (set! sreg (+ sreg (* a.in xreg)))
        (set! x.out xreg)
         (set! clk (+ clk 1)))
      ))
  ))

  ;declaration of processor type-b.
  (ptype-b
  (var
   (sreg float 0)
   (xreg float 0)
   (clk int 0))
  (inputs
   (s float 0)
   (x float 0)
  (a float 0)
  (b float 0))
 (outputs
  (s float 0 (-1 0) 0)
  (x float 0 (1 0) 0))
(code
  ((if (equal? (modulo clk 2) 0)
     (begin
       (set! xreg x.in)
        (set! sreg (/ (- b.in sreg) a.in))
       (set! xreg sreg)
       (set! s.out sreg)
        (set! clk (+ clk 1)))
     (begin
       (set! sreg s.in)
        (set! sreg (+ sreg (* a.in xreg)))
       (set! x.out xreg)
        (set! clk (+ clk 1)))
     ))
 )))


((instantiate  ptype-a  (point  (2)))
 (instantiate  ptype-b  (point  (1))))

() ; no connections need be specified.

((input-data
  ((s gets s-inputs at (2)))
 (input (i = 1)
       (j = t)))

 (input-data
  ((a gets a-inputs from (1) to (2)))
 (input (i = t)
       (j = pp.i)))

 (input-data
  ((x gets x-inputs at (1)))
 (input (i = 1)
       (j = t)))

 (input-data
   ((b gets b-inputs at (1)))
   (input (i = 1)
          (j = t))))

  (output-data
   ((s at (1)))))
```

- -

Sample data

- -

```
(s-inputs
 (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

(a-inputs
 (1 0)
 (3 0)
 (2 6)
 (5 10)
 (4 9)
 (8 0)
 (7 0)
 (0 0)
 (0 0)
 (0 0))

(x-inputs
 (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

(b-inputs
 (1 0 2 0 3 0 4 0 0 0 0 0 0 0 0 0 0 0))
```