1986

Synthesizing Non-Uniform Systolic Designs

Concettina Guerra

Rami Melhem

Report Number:
86-621

SYNTHESIZING NON-UNIFORM SYSTOLIC DESIGNS

Concettina Guerra
Rami Melhem

CSD-TR-621
August 1986

SYNTHESIZING NON-UNIFORM SYSTOLIC DESIGNS

Concettina Guerra

Rami Melhem

Department of Computer Sciences, Purdue University

West Lafayette, Indiana

ABSTRACT

In this paper we propose a method to derive systolic designs with non-uniform data flow. One of the major difficulties in systematic design is in transforming the original sequential specification of a problem into a form suitable to VLSI implementation. Our approach to automatically restructuring a problem is based on a subset of the data dependencies extracted from the original problem specification. By using such dependencies we are able to identify chains of dependent computations which are then converted into recurrence equations. The mapping of the new specification into hardware is also based on data dependencies. We illustrate the methodology by applying it to algorithms using dynamic programming.

I. INTRODUCTION

The development of efficient VLSI algorithms is relevant to many applications including signal and image processing, which justifies the considerable interest for this topic. In recent years, the problem of synthesizing VLSI design - and systolic design, in particular - has also received much attention [1]-[4], [8], [10]-[11], [13]-[24]. Systematic methodologies for the derivation of systolic algorithms have proved useful both in finding new designs and in verifying the correctness of old ones. Moreover, the possibility of automatically generating a number of viable algorithms for the solution of a given problem enables the selection of an optimal algorithm among a wider set of candidates. The optimality can be based on such parameters as completion time T, number of processors P, chip area, etc.[18]. Most of the existing synthesis methods tend to minimize the completion time.

The first and probably most challenging part of the systematic design process is in the transformation of the high-level problem specification into a form better suited to VLSI implementation. Such a form can be, for example, a system of first-order recurrence equations, a uniform recurrence equation, or a nested loop with constant data dependencies. Often, this transformation is obtained by using techniques similar to those used for software compilers: buffering of variables, addition of new variables, etc. However, it is not well understood how to select a good transformation especially for some complex nonnumerical problems. Some authors assume that the problem is already given in the required form, and concentrate on how to map it into hardware.

A fundamental distinction between different approaches to the mapping problem is whether they use data dependence based transformations [1]-[4], [7], [14], [16], [20]-[24], or delay operators applied to the mathematical expression of the algorithm [10]-[11]. For a survey of the various methods see [6].

The transformational approach based on data dependencies has been successfully applied to a variety of algorithms characterized by constant data dependencies. For such algorithms linear

time and space transformations of the computations into VLSI arrays have been derived [14], [20]-[23]. The resulting designs have constant and regular data flow. However, a number of more complex problems with non-constant data dependencies require non linear transformations. A typical example is the optimal parenthesization problem, which uses dynamic programming. In [9] a systolic algorithm for dynamic programming was presented that can be cast in a triangular array of processing elements. The design is quite complex: the data flow in non-constant throughout the array and the action of a processing element varies at different clock cycles.

Attempts have been made to synthesize non-uniform designs. A non-linear transformation for dynamic programming was indicated in [22]. However, the paper does not precisely describe how the transformation is obtained from the algorithm specification. Chen [1] presents a methodology for mapping algorithms expressed as systems of first order recurrence relations into systolic arrays. The synthesis procedure allows designs with non-uniform communication patterns. The mapping is done inductively, starting at the boundary conditions. By using this technique, various designs, corresponding to different communication patterns in the systolic array, are derived for dynamic programming. The procedure, based on a point-by-point mapping, appears to be quite lengthy. In [19] a mathematical model, based on a sequence notation, is used to represent index transformations. The model provides a precise specification of systolic designs. It allows arbitrary interconnections in systolic networks and is not restricted to linear transformations. The paper does not give a constructive methodology to find the transformations.

This paper presents a systematic method for the design of VLSI algorithms. The method consists of transforming the high-level problem specification into a set of mutually dependent recurrence equations. To select the appropriate transformations we propose a two-step refinement procedure. The procedure first determines a coarse timing function of the computations in the original specification of the problem, based on a subset of constant data dependencies; then uses this function to guide the search for an index transformation. Ordered chains of dependent computations are identified in the algorithm, according to such function, and an index transformation

is applied to each single chain. The index transformation must be compatible with the timing function, i.e. all the computations whose operands are available first will be performed first. Subsequently, the mapping of the obtained system of recurrence equations into a VLSI network is performed by applying linear time and space transformations separately to each recurrence subject to global constraints. The resulting design may have non-uniform data flow.

This paper is organized as follows. Section II presents the computational models for VLSI arrays and algorithms and reviews the transformational approach to derive linear time-space transformations for mapping an algorithm into a VLSI array. Section III presents a method which helps deriving from an abstract algorithm a set of mutually dependent recurrence equations. Section IV illustrates the method by applying it to dynamic programming. In section V time-space transformations are used to map the obtained system of mutually dependent recurrence equations into a VLSI array. Finally, section VI shows how to derive a new design for dynamic programming that uses fewer processing elements than the one in [9].

II. COMPUTATIONAL MODELS

A. *Definitions*

We will review the transformational approach [20]-[23] by referring to a specific algorithmic model. We consider a structured set of computations written as a recurrence relation or a nested loop. The set includes input statements, output statements, assignment statements, and conditional assignment statements, where the latter are of the form "IF predicate THEN assignment". Conditional assignments are restricted to those where the predicate depends only on the values of the loop indeces and not on the values of the variables. The recurrence relation or nested loop defines an index set $I^n=\{(i_1,\ldots,i_n) \mid l_1^1\le i_1\le l_1^2,\ldots,l_n^1\le i_n\le l_n^2\}$, subset of the set Z^n of the n-tuples of integers. We assume that the following conditions are satisfied:

CA1 - each variable in the algorithm is associated with an index vector $(i_1,i_2,...,i_n)$, i.e. is an element of an n-dimensional array. In other words, there is a one-to-one correspondence between the n-tuples in I^n and the dimensions of any array used in the algorithm.

CA2 - if S is an assignment statement indexed by $(i_1,i_2,...,i_n)$, and a variable on the right side of S is indexed by $(j_1,j_2,...,j_n)$, then each i_k may depend only on j_k.

CA3 - The dependence vector of a variable is defined as the difference of the index vectors of computations where the variable is used and generated [8]. The data dependence vectors $\bar{d}_1,\ldots,\bar{d}_m$ associated to the variables of a recurrence can be represented as a matrix $D=[\bar{d}_1 \cdots \bar{d}_m]$, whose columns are labelled by the variable names. A variable may have many dependencies each corresponding to a different index vector. In the canonic form we assume that data dependence vectors are constant, i.e. independent of the index vectors.

CA4 - a variable is used exactly once after it is generated.

We will refer to the above specification of an algorithm as to the *canonic form* of the algorithm.

The canonic form does not explicitly specify any ordering among the computations; the lexicographical ordering in I^n is arbitrary and therefore irrelevant to our purposes. Rather, an implicit partial ordering of the computations is given by the data dependencies. A partial ordering $>_D$ defined by the data dependencies is such that $\bar{I} >_D \bar{I}'$ if $\bar{I}=\bar{I}'+\bar{d}_j$ for some $\bar{d}_j \in D$.

B. Mapping a canonic form into a VLSI array

It is possible, under certain conditions, to automatically map a recurrence into a VLSI array [20]-[23]. Consider the following simplified model of a VLSI array. Each processor of the array

is assigned a label $l \in L^{n-1} \subset Z^{n-1}$. The connection pattern of the array is described by the matrix $\Delta=[\delta_1,\delta_2,\ldots,\delta_s]$ which specifies the links among the processors. Precisely, δ_i is the difference vector of the integer labels of adjacent cells in the network.

A linear time-space transformation of the computations indexed by I^n into the VLSI array is defined by:

$$\Pi = \begin{vmatrix} T \\ S \end{vmatrix}$$

where T is a mapping from $I^n \longrightarrow Z$ and S is a mapping from $I^n \longrightarrow L^{n-1}$. The time function T transforms D into TD. Thus, to ensure a correct execution ordering, T must satisfy the following condition:

$$T(\bar{d}_i) > 0 \quad \text{for } each \ \bar{d}_i \in D \tag{1}$$

System (1) may have no solution or several solutions. In this latter case, the one which minimizes the total execution time (defined as the difference between the maximum and minimum value of T) is chosen.

The space mapping S is a mapping from the set of computations to the set of processing elements such that for each $\bar{i},\bar{j} \in I^n$

$$S(\bar{i})=S(\bar{j}) \quad implies \quad T(\bar{i})\neq T(\bar{j}) \tag{2}$$

i.e. concurrent computations cannot be mapped into the same processor. Determining a solution for S which satisfies constraint (2) is equivalent to solving the diophantine equations:

$$SD = \Delta K \tag{3}$$

for which the matrix $\begin{vmatrix} T \\ S \end{vmatrix}$ is non-singular. K is an integer matrix with positive elements. The equations may have no solution or several solutions. If no feasible solution is found, the design

procedure is repeated by starting with a different timing function or else a different interconnection network. If several solutions are possible, the one which is optimal according to some given criterion is chosen.

C. Algorithm transformations

To derive a canonic form from the high-level problem specification, transformations similar to those used for software compilers are generally used [14]. The goal of such transformations is to enhance pipelining and local communication in an algorithm. This is accomplished by (i) adding indeces to existing variables in the algorithm, (ii) renaming variables, or (iii) introducing new variables. Such transformations do not change the algorithm fundamentally, but only the data dependencies among the variables. The new data dependencies in the resulting specification of the problem are therefore not characteristic of the problem itself but of its parallel implementation. Indeed, as is well known, there are in general several ways to transform a given problem specification, according to the previous rules, each way producing a different set of data dependencies. However, not all the possible transformations lead to a feasible VLSI design. Moreover, different transformations may result in different performances. Consider the two following examples:

Example 1. The convolution problem is defined by : given a sequence $x_1, \ldots, x_n$ and a set of weights $w_1, \ldots, w_s$, determine the sequence $y_1, \ldots, y_n$ such that:

$$y_i = \sum_{k=1,s} w_k \cdot x_{i-k}$$

Broadcasting of x and w can be eliminated by adding one more index to all variables x, w, and y. At least two different index transformations can be applied which produce the two recurrences in canonic form below. The first is a backward recurrence, where the variable $y_{i,k}$ is

defined in terms of $y_{i,k-1}$:

$$\begin{aligned} & 1 \le i \le n \ ; \ 1 \le k \le s \\ & w_{i,k} = w_{i-1,k} \\ & x_{i,k} = x_{i-1,k-1} \\ & y_{i,k} = y_{i,k-1} + w_{i,k} \cdot x_{i,k} \end{aligned} \qquad (4)$$

with initial conditions:

$$y_{i,0} = 0;\ w_{0,k} = w_k;\ x_{i-1,0} = x_i;\ x_{0,k-1} = 0$$

The second is forward recurrence, where the variable $y_{i,k}$ is defined in terms of $y_{i,k+1}$.

$$\begin{aligned} & 1 \le i \le n \ ; \ 1 \le k \le s \\ & w_{i,k} = w_{i-1,k} \\ & x_{i,k} = x_{i-1,k-1} \\ & y_{i,k} = y_{i,k+1} + w_{i,k} \cdot x_{i,k} \end{aligned} \qquad (5)$$

with initial conditions:

$$y_{i,s+1} = 0;\ w_{0,k} = w_k;\ x_{i-1,0} = x_i;\ x_{0,k-1} = 0$$

Data dependencies corresponding to the two recurrences introduce two different partial orderings in $I^2=\{(i,k) \mid 1 \le i \le n;\ 1 \le k \le s\}$, which translate into two different schedules for the computations and consequently into different systolic designs. We first derive a systolic design for recurrence (4). Data dependencies for variables y, x and w in (4) are respectively:

$$d_1 = (0\ 1)^t \qquad d_2 = (1\ 1)^t \qquad d_3 = (1\ 0)^t.$$

The coefficients T_1 and T_2 of a linear transformation T must satisfy (1), that is:

$$T_2 > 0 \quad T_1+T_2 > 0 \quad T_1 > 0.$$

Several solution to the above equations are possible; the one which minimizes the execution time is given by:

$$T_1 = 0 \quad T_2 = 1.$$

The resulting timing function is:

$$T(i,\ k) = i+k.$$

A space mapping of the recurrence into a linear array of processing elements obtained from (3) is given by:

$$S(i, k) = k.$$

The resulting design, identical to the one presented by Kung [12], is summarized in table 1. Similarly, by applying the mapping procedure to recurrence (5) we obtain two other designs presented in [12]. These latter are summarized in table 2. Notice that design W2 cannot be generated starting from recurrence (5) and, viceversa, designs R2 and W1 cannot be generated from recurrence (4). Finally, we mention that other designs for convolution are possible [3], [12], but they cannot be generated from recurrences (4) or (5).

Design	Output (y)	Input (x)	Weights (w)
W2	Move in the same direction at different speeds		Stay

Table 1. Systolic design for recurrence (4)

Design	Output (y)	Input (x)	Weights (w)
W1	Move in opposite directions		Stay
R2	Stay	Move in the same direction at different speeds	

Table 2. Systolic designs for recurrence (5)

Example 2. Recursive convolution

This problem can be expressed as: given a sequence of weights $w_1,..,w_s$, determine the sequence $y_1,..,y_n$ such that:

$$y_i = \sum_{k=1,s} w_k \cdot y_{i-k}$$

Of the two recurrences which can be derived by using transformations similar to those used in Example 1, only the forward recurrence has to be considered for a systolic implementation. The backward recurrence does lead to a any reasonable design since it cannot overlap computations of $y_{i,k}$ for different values of index k.

For more complex problems, such as optimal parenthesization using dynamic programming, selecting a good initial transformation is not an obvious task and in fact straightforward transformations such as those applied to the convolution problem fail. It is clear that this part of the synthesis procedure sometimes requires creativity and programming experience. We suggest a way to assist the human designer in this crucial task. Obviously, it is not always possible to transform a given problem into canonic form. For problems which do not have such representation we attempt to rewrite them into a form of many modules each being in canonic form. The method we propose here relies on identifying a set of constant data dependencies in the original formulation of the problem.

III. DERIVING A VLSI ALGORITHM FROM THE HIGH-LEVEL SPECIFICATION OF THE PROBLEM

We assume that the high level specification of a problem is written in the form of a nested loop with the index set I^n defined by:

$$I^n=\{(i_1, \ldots, i_n) \mid l_1^1 \le i_1 \le l_1^2, \ldots, l_n^1 \le i_n \le l_n^2\}.$$

We assume that, unlike the canonic form, the loop contains a variable which is an s-dimensional array. More specifically, we let $s=n-1$ and assume that the loop contains an assignment

statement of the form:

$$c(\bar{i}^s) = f(c(\bar{i}^s - \bar{d}_1^s), \ldots, c(\bar{i}^s - \bar{d}_m^s)) \tag{6}$$

where: $\bar{i}^s=(i_1, i_2,\ldots, i_s)$ is an s-tuple of indices of the loop;

$\bar{d}_j^s=(a_{j,1}, \ldots, a_{j,t-1}, i_t-i_n, a_{j,t+1}, \ldots, a_{j,s})$, $j=1,\ldots,m$;

and where $a_{j,l}$ are integer constants. In other words, the index i_t on the left side of (6) is replaced on the right side by the index i_n. Each vector $\bar{d}_j^s$, $(j=1,\ldots,m)$, represents a non-constant data dependence for variable c, since its t-th component is a function of the two indices i_t and i_n. We can expand $\bar{d}_j^s$ into a number of data dependence vectors, each corresponding to a specific value of index i_n in the t-th component. Then, we associate to each pair $(variable, index\ vector)=(c, \bar{i}_s)$ the set $D_{\bar{i}^s}^c$ of all the data dependence vectors obtained by expanding $\bar{d}_1^s,\ldots,\bar{d}_m^s$.

Our strategy to select the initial index transformation consists of a two-step procedure. Let $I^s = \{(i_1,\ldots,i_s) \mid l_1^1 \le i_1 \le l_1^2, \ldots, l_s^1 \le i_s \le l_s^2\}$. First, determine from the high-level specification of the problem a coarse time function $T: I^s \rightarrow Z$. This function will be used in the subsequent step of the procedure to guide the search for a schedule of computations indexed by I^n. We derive T from a subset of the data dependencies of the algorithm, namely, the subset of constant data dependencies, thus, T provides only a lower bound for an actual timing function for I^s. Furthermore, T depends only on the implicit dependencies of the problem since it is derived before any arbitrary ordering of the computations is introduced. The set $D_{\bar{i}^s}^c$ of data dependencies is non-constant in the computation space. However, the set D^c defined as the intersection of $D_{\bar{i}^s}^c$ for all $\bar{i}^s \in I^s$ contains only constant data dependencies. For a nested loop with constant data dependencies, a linear (or quasi-affine) timing function can be determined by applying the transformational method described in section I. Thus, if D^c is not empty we can derive a linear timing

function $T : I^s \rightarrow Z$ which is compatible with the set D^c, i.e. $\bar{i}^s >_{D^c} \bar{i}'^s$ implies $T(\bar{i}^s) > T(\bar{i}'^s)$. The function T must satisfy (1), that is:

$$T(\bar{d}_i) > 0 \quad \text{for each } \bar{d}_i \in D^c \tag{7}$$

If system (7) has several solutions, the one which minimizes the total execution time is chosen.

It obvious that if τ is an actual timing function for I^s then it must be $\tau(\bar{i}^s) \geq T(\bar{i}^s)$ for each $\bar{i}^s \in I^s$. Moreover, because of the monotonocity of an expanded data dependence vector in $D^c_{\bar{i}^s}$, it must be $\tau(\bar{i}^s) > \tau(\bar{i}'^s)$ if $T(\bar{i}^s) > T(\bar{i}'^s)$.

The partial ordering defined by T on the set of computations I^s defines a partial ordering in a subset of I^n based on the availability of operands. Consider the set $J^n \subset I^n$ consisting of the n-tuples $(\bar{i}^s, i_n)$ for any given $\bar{i}^s \in I^s$ and for $I_n^1 \leq i_n \leq I_n^2$. For any two such n-tuples, we introduce the relation $>_T$ defined by:

$$(\bar{i}^s, i_n') >_T (\bar{i}^s, i_n'') \iff$$

$$Max\{T(\bar{i}^s - \bar{d}_1'), \ldots, T(\bar{i}^s - \bar{d}_m')\} > Max\{T(\bar{i}^s - \bar{d}_1''), \ldots, T(\bar{i}^s - \bar{d}_m'')\}$$

where $\bar{d}_j'$ and $\bar{d}_j''$ $(j=1,\ldots,m)$ are the data dependence vectors in (6) corresponding to values i_n' and i_n'', respectively, of index i_n. Obviously, the relation $>_T$ is a partial ordering in J^n. Thus, J^n can be decomposed into a number of chains, i.e. subsets whose elements are linearly ordered (relative to each other). Of course, there will be only one chain if the relation $>_T$ is linear to start with. Each chain consists of indexed computations which have to be carried out one after the other in a specific linear ordering. Computations belonging to different chains can be carried out independently. There can be many decompositions of a partially ordered set into chains. Minimal chain decompositions can be found by network flow techniques [5]. We are interested in a simpler problem, namely in finding a chain decomposition of $>_T$ such that the computations in a chain are also sorted (either in increasing or decreasing order) according to the index i_n.

At this point, we are ready to restructure the given algorithm. Let us denote by s the number of chains. We partition the computations indexed by J^n into s separate recurrences, each corresponding to a distinct chain. In each recurrence the ordering for index i_n is chosen according to its ordering in the chain. Then, we transform each recurrence into canonic form using the three previous rules: 1) adding missing indices, 2) adding new variables, and 3) renaming old variables. In addition we also add statements between recurrences to correlate variables in distinct recurrences. In fact, a recurrence may use a variable generated in another recurrence. This last step may introduce non-constant data dependencies in the system.

In conclusion, the obtained new specification is expressed as a system of s modules, each module being a recurrence equation in canonic form. Non-constant data dependencies may occur between variables of different modules.

IV. AN APPLICATION TO DYNAMIC PROGRAMMING

Consider the dynamic programming technique applied to such problems as optimal parenthesization, and shortest path. Both problems can be expressed by the recurrence:

$$1 \le i \le n ; i < j \le n$$

$$c_{i,j} = \min_{i<k<j} f(c_{i,k}, c_{k,j}) \tag{8}$$

with initial conditions:

$$c_{i,i+1} = c_i \quad 1 \le i \le n$$

for some function f. Note that no explicit ordering for the indices i, j, and k is specified. If we choose the normal lexicographical ordering for the three indices and we apply some index transformation we can derive a canonic form from the above recurrence; however, the systolic implementation of it will have execution time $O(n^2)$ since it does not overlap the computations of $c_{i,j}$ for different k.

Let $I^3=\{(i,j,k) \mid 1\le i,j\le n; i<k<j\}$ be the index set of the recurrence above; and let $I^2=\{(i,j) \mid 1\le i, j\le n\}$ be the index set defined by variable c. Each pair $(c, (i, j))$ is associated with a distinct set of data dependencies, here represented in matricial form:

$$D^c_{(i,j)} = \begin{vmatrix} 0 & \text{i-k} \\ \text{j-k} & 0 \end{vmatrix}$$

or in the expanded form below where each column corresponds to specific value of the index k:

$$D^c_{(i,j)} = \begin{vmatrix} 0 & \dots & 0 & 0 & -1 & -2 & \dots & \text{i-j+1} \\ \text{j-i+1} & \dots & 2 & 1 & 0 & 0 & \dots & 0 \end{vmatrix}$$

The intersection D^c of all the sets $D^c_{(i,j)}$ $(1\le i, j\le n)$ is non empty and is given by:

$$D^c_{(i,j)} = \begin{vmatrix} 0 & -1 \\ 1 & 0 \end{vmatrix}$$

We derive a time function $T : I^2 \rightarrow Z$ compatible with D^c. Since D^c contains only constant data dependencies, we can apply the methodology discussed in section I to determine a linear time function. Conditions (7) applied to the data dependence vectors in D^c give:

$$T_1 > 0 \quad \text{and} \quad T_2 \le -1$$

The least integer values that satisfy the above equations are:

$$T_1 = 0 \quad \text{and} \quad T_2 = -1.$$

Thus, the optimal time transformation is:

$$T(i,j) = j - i.$$

Next we consider the partial ordering $>_T$ in $J^3=\{(i,j,k) \mid i,j \textit{ are fixed and } i<k<j\}$ defined by:

$$(i,j,k') >_T (i,j,k'') \iff$$
$$Max\{T(i,k'), T(k',j)\} > Max\{T(i,k''), T(k'',j)\}$$

Notice that the minimal elements with respect to $>_T$ are:

$(i, j, (i+j)/2)$ if $i+j$ is even or

$(i, j, (i+j-1)/2)$ and $(i, j, (i+j+1)/2)$ if $i+j$ is odd.

By repeatedly finding minimal elements after removing the previous minimal elements from the set we obtain a decomposition of J^3 in the two chains below (here we only write the third component of the index vectors):

{if (i+j) is even}

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

$(i+j)/2+1, (i+j)/2+2, ..., j-1.$

{if (i+j) is odd}

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

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

We can now rewrite (8) into a form where the execution ordering of index k is specified according to the above chains. The new specification of the problem consists of two recurrences: the first recurrence is a forward recurrence where the index k varies from $(i+j)/2$ to $i+1$ (or from $(i+j-1)/2$ to $i+1$ if $i+j$ is odd); and the second is a backward recurrence where k varies from $(i+j)/2$ to $j-1$ (or from $(i+j+1)/2$ to $j-1$ if $i+j$ is odd) . To transform each recurrence into

canonic form some further manipulation is necessary. We first add missing indices to variables on both sides of (8) and then introduce new recurrence variables to eliminate broadcast of a variable to different destinations. We use different sets of variables for the two recurrences. Each recurrence initializes some of its variables to values generated by the other recurrence. Now equations (8) can be converted into the following system of mutually dependent recurrence equations:

for i :=1 to $n-1$ do $a''_{i,\,i+1,\,i+1} := c_{i,\,i+1}$; $c_{i,\,i+1,\,i+1} := c_{i,\,i+1}$;
for $l := 2$ to $n-1$ do

for $i := 1$ to n do begin
    $j := i+l$;
    if $(i+j)$=*even*) then begin
        $k := (i+j)/2$;
A1: $a'_{i,\,j,\,k} = a''_{i,\,j-1,\,k}$;
A2:   if $k = i+1$ then $b'_{i,\,j,\,k} := c_{i+1,\,j,\,j}$ else $b'_{i,\,j,\,k} := b'_{i+1,\,j,\,k}$;
        $c'_{i,\,j,\,k} := f\,(a'_{i,\,j,\,k}\,,\, b'_{i,\,j,\,k}\,)$; $c''_{i,\,j,\,k} := c'_{i,\,j,\,k}$
        end
    else begin {i+j=odd};
        $k := (i+j-1)/2$;
        $a'_{i,\,j,\,k} := a'_{i,\,j-1,\,k}$;
        if $k = i+1$ then $b'_{i,\,j,\,k} := c_{i+1,\,j,\,j}$ else $b'_{i,\,j,\,k} := b'_{i+1,\,j,\,k}$; (9)
        $c'_{i,\,j,\,k} := f\,(a'_{i,\,j,\,k}\,,\, b'_{i,\,j,\,k}\,)$
        $k := (i+j+1)/2$;
A3:   if $k := j-1$ then $a''_{i,\,j,\,k} := c_{i,\,j-1,\,j-1}$ else $a''_{i,\,j,\,k} := a''_{i,\,j-1,\,k}$;
A4: $b''_{i,\,j,\,k} := b'_{i+1,\,j,\,k}$;
        $c''_{i,\,j,\,k} := f\,(a''_{i,\,j,\,k}\,,\, b''_{i,\,j,\,k}\,)$;
        end

    for $k := \lceil (i+j-1)/2 - 1 \rceil$ downto $i+1$ do begin
        $a'_{i,\,j,\,k} := a'_{i,\,j-1,\,k}$;
        if $k = i+1$ then $b'_{i,\,j,\,i+1} := c_{i+1,\,j,\,j}$ else $b'_{i,\,j,\,k} := b'_{i+1,\,j,\,k}$; module 1
        $c'_{i,\,j,\,k} := h\,(c'_{i,\,j,\,k+1}\,,\, f\,(\,a'_{i,\,j,\,k}\,,\, b'_{i,\,j,\,k}\,)\,)$;
        end;

    for $k := \lfloor (i+j+1)/2 + 1 \rfloor$ to $j-1$ do begin
        if $k = j-1$ then $a''_{i,\,j,\,k} := c_{i,\,j-1,\,j-1}$ else $a''_{i,\,j,\,k} := a''_{i,\,j-1,\,k}$; module 2
        $b''_{i,\,j,\,k} := b''_{i+1,\,j,\,k}$;
        $c''_{i,\,j,\,k} := h\,(\,c''_{i,\,j,\,k-1}\,,\, f\,(a''_{i,\,j,\,k}\,,b''_{i,\,j,\,k}\,))$:
        end;
A5: $c_{i,\,j,\,j} := h\,(c'_{i,\,j,\,i+1}\,,\, c''_{i,\,j,\,j-1})$;
    end;

From the above specification we extract distinct sets D_1 and D_2 of local data dependencies for the two modules.

$$D_1 = \begin{array}{|ccc|} c' & a' & b' \\ 0 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{array} \qquad D_2 = \begin{array}{|ccc|} c'' & a'' & b'' \\ 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{array}$$

Non-constant dependencies between variables of distinct recurrences are defined by statements A1 to A5 in the algorithm. We will refer to such dependencies as global dependencies.

V. MAPPING THE ALGORITHM INTO HARDWARE

A. Time function

Once the algorithm has been converted into the new form, the mapping into a VLSI array can be accomplished in two steps:

1) Finding for each individual module in the algorithm representation a separate time function which is compatible with the local data dependencies and also satisfies the constraints imposed by the global dependencies.

2) Finding a space mapping for each module into the processors of a physical network which is compatible with the time function and satisfies global constraints.

Referring again to the dynamic programming algorithm, we first seek linear time transformations λ and μ for module 1 and 2, respectively. Furthermore, let σ be the time function for computation in A5 outside the two modules. The linearity requires that for the dependence vectors of the modules it must be:

$$\lambda(\bar{d}) > 0 \quad \text{for } \bar{d} \in D_1 \quad \text{and} \quad \mu(\bar{d}) > 0 \quad \text{for } \bar{d} \in D_2$$

that is:

$$\lambda_1 \le -1 \quad \lambda_2 \ge 1 \quad \lambda_3 \le -1$$

$$\mu_1 \le -1 \quad \mu_2 \ge 1 \quad \mu_3 \ge 1.$$

Constraints introduced by A1-A5 lead to the following equations:

$$\lambda(i, j, (i+j)/2) > \mu(i, j-1, (i+j)/2)$$

$$\lambda(i, j, i+1) > \sigma(i+1, j, j)$$

$$\mu(i, j, j-1) > \sigma(i, j-1, j-1)$$

$$\mu(i, j, (i+j+1)/2) > \lambda(i+1, j, (i+j+1)/2)$$

$$\sigma(i, j, j) \ge \max[\lambda(i, j, i+1), \mu(i, j, j-1)]$$

It easy to check that an optimal solution to the above system is given by:

$$\lambda_1 = -1 \quad \lambda_2 = 2 \quad \lambda_3 = -1$$

$$\mu_1 = -2 \quad \mu_2 = 1 \quad \mu_3 = 1$$

$$\sigma_1 = -2 \quad \sigma_2 = 1 \quad \sigma_3 = 1.$$

Hence:

$$\lambda(i,j,k) = -i+2j-k$$

$$\mu(i,j,k) = -2i+j+k$$

$$\sigma(i,j,j) = -2i+2j.$$

The next step is to find a mapping of the computations indexed by I^3 into the processors of a VLSI array.

B. Space Function

The automatic procedure for determining the mapping of computations into the cells of a systolic array is analogous to the one for the time function. Again, we look for separate solutions to the different modules in the algorithm subject to global constraints. We consider a 2-D VLSI array of processing elements modelled by the pair $[L^2,\Delta]$, where L^2 is the set of labels (x,y) assigned to processing elements and Δ is a matrix describing the interconnection network between processing elements. Different interconnection patterns may result in different classes of designs. In the following, we generate the optimal design when Δ is chosen to be:

$$\Delta = \begin{vmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \end{vmatrix}$$

Δ corresponds to a network with unidirectional links, as shown in fig. 1.

Let S', S'', and S be the space functions for module 1, module 2, and statement A5, respectively.

$$S' = \begin{vmatrix} S_{11}' & S_{12}' & S_{13}' \\ S_{21}' & S_{22}' & S_{23}' \end{vmatrix} \qquad S'' = \begin{vmatrix} S_{11}'' & S_{12}'' & S_{13}'' \\ S_{21}'' & S_{22}'' & S_{23}'' \end{vmatrix} \qquad S = \begin{vmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \end{vmatrix}$$

In addition to satisfy (3), the coefficients of S', S'', and S must satisfy the constraints imposed by global dependencies. Precisely, if a global dependence involves two variables belonging to different modules which are computed at times t and t' with $t-t'=d$ then the distance of the cells where the two variables will be mapped cannot be more than d. By distance we mean the length of a path consisting of interconnection links between the two cells. From A1 we have:

$$S' \ |i \ \ j \ \ (i+j)/2|' \ = \ S'' \ |i \ \ j-1 \ \ (i+j)/2 \ |' \ + \ \bar{d}_1; \qquad \bar{d}_1 \in \Delta,$$

since $\lambda(i,j,(i+j)/2) - \sigma(i,j-1,(i+j)/2) = 1$ and, consequently, the two computations must occur either in the same cell or in adjacent cells. Similarly, from A2-A4 we obtain:

$$S' \ |i \ \ j \ \ i+1 \ |^t \ = \ S \ \ |i+1 \ \ j \ \ j \ |^t + \vec{d}_2; \quad \vec{d}_2 \in \Delta; \tag{10}$$

$$S'' \ |i \ \ j \ \ j-1 \ |^t \ = \ S \ \ |i \ \ j-1 \ \ j-1 \ |^t + \vec{d}_3; \quad \vec{d}_3 \in \Delta;$$

$$S'' \ |i \ \ j \ \ (i+j+1)/2 \ |^t \ = \ S' \ \ |i+1 \ \ j \ \ (i+j+1)/2 \ |^t + \vec{d}_4; \quad \vec{d}_4 = \delta_i + \delta_j \ \ \delta_i, \delta_j \in \Delta;$$

$$S \ \ |i \ \ j \ \ j \ |^t \ = \ S \ \ |i \ \ j \ \ i+1 \ |^t + \vec{d}_5; \quad \vec{d}_5 \in \Delta;$$

One solution to above system of equations is:

$$S'_{11} = S'_{13} = 0; \ S'_{12} = 1 \qquad S'_{22} = S'_{23} = 0; \ S'_{21} = 1$$

for the first recurrence and:

$$S''_{11} = S''_{13} = 0; \ S''_{12} = 1 \qquad S''_{22} = S''_{23} = 0; \ S''_{21} = 1$$

for the second recurrence. Thus

$$S'(i,j,k) = S''(i,j,k) = S(i,j,j) = (j,i).$$

The resulting design is identical to the one first introduced in [9]. The corresponding systolic array and the action of a cell at different times is depicted in figure 1.

VI. A NEW DESIGN FOR DYNAMIC PROGRAMMING

As shown in [8], a new design for dynamic programming can be automatically generated if we choose a different interconnection pattern for the network. This design uses fewer processing elements than the one in [9]; precisely $3/8n^2$ instead of $n^2/2$. Consider an array of processing elements whose communication pattern is described by:

$$\Delta = \begin{vmatrix} 0 & 1 & 0 & -1 & -1 \\ 0 & 0 & -1 & 0 & -1 \end{vmatrix}$$

Cells in the array are connected by bidirectional horizontal links as well as by diagonal and vertical links, as shown in fig. 2. An optimal design for dynamic programming is generated for such an array using the same mapping procedure. Again we solve equations (4) subject to global constraints (10). We derive for the first recurrence:

$$S_{11}' = S_{12}' = 0;\ S_{13}' = 1 \qquad S_{22}' = S_{23}' = 0;\ S_{21}' = 1$$

and for the second recurrence:

$$S_{11}'' = S_{12}'' = 1;\ S_{13}'' = -1 \qquad S_{22}'' = S_{23}'' = 0;\ S_{21}'' = 1$$

Thus we have:

$$S'(i,j,k) = (k,i) \text{ and } S''(i,j,k) = (i+j-k,i).$$

These transformations lead to the systolic design of figure 2. The array consists of $3/8n^2$ cells. All cells are identical. However, the action of a cell varies from time to time. It does computation relative to module 1 or module 2 depending on the values of indeces i, j, and k. Also, the direction of data streams varies for the two modules. The transformation of data dependence vectors D_1 into communication vectors Δ is derived from S'. From:

$$\begin{vmatrix} S_{11}' & S_{12}' & S_{13}' \\ S_{21}' & S_{22}' & S_{23}' \end{vmatrix} \begin{vmatrix} c' & a' & b' \\ 0 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \end{vmatrix}$$

we derive that variables $c_{i,j,k}'$ move to the left along the horizontal links, variables $a_{i,j,k}'$ do not move along the array but stay inside the cells, where they are updated. Finally, variables $b_{i,j,k}'$ move up, except at the boundary, where they move along the diagonal (from (10)). The direction of variables in module 2 is derived from the mapping S''. Variables $a_{i,j,k}''$ move to the right along the horizontal links, variables

$b''_{i,j,k}$ move up to the left along the diagonal links. Variables $c''_{i,j,k}$ move with the same pattern as in the other module. The action of a cell at each time is illustrated in figure 2.

VII. CONCLUSION

We have suggested a methodology to assist a human designer into the difficult task of designing a VLSI algorithm starting from a sequential specification of a problem. Among the many possible transformations which can make parallelism explicit in a program the method helps selecting a feasible and optimal sequence of transformations. The methodology appears useful for complex nonnumerical problems for which standard restructuring techniques seem inadequate. The class of obtainable designs is not restricted to designs having constant data flow.

REFERENCES

[1] M. Chen, "Synthesizing Systolic Designs ", in *Proc. Int. Symp. on VLSI Technology, Systems and Applications*, Taipei-Taiwan, 1985.

[2] M. Chen, "The Generation of a Class of Multipliers: A Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI", *IEEE Int. Conf. on Computer Design*, 1985.

[3] P. R. Cappello, and K. Steiglitz, "Unifying VLSI Array Design with Geometric Transformations", *Int. Conf. on Parallel Processing*, pp. 448-457, 1983.

[4] J. M. Delosme, and I. Ipsen, "An Illustration of a Methodology for the Construction of Efficient Systolic Architectures in VLSI", *Proc. Second Int. Sym. on VLSI Tech. Syst. and Appl.*, pp. 268-23, 1985.

[5] L. Ford, and D. Fulkerson, *Flows in Networks*, Princeton Univ. Press, 1962.

[6] J. A. B. Fortes, K. S. Fu, and B. W. Wah, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays", Tech. Rep., Electr. Eng., Purdue University.

[7] J. A. B. Fortes, and D. I. Moldovan, "Parallelism Detection and Algorithm Transformation Techniques Useful for VLSI Architecture Design" *J. Parallel Distrib. Comput.*, May 1985.

[8] C. Guerra, "A Unifying Framework for Systolic Designs", *AWOC Symposium*, Greece, 1986.

[9] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms ", *Proc. of Caltech Conf. on VLSI*, 1979.

[10] L. Johnsson, and D. Cohen, "A Mathematical Approach to Modelling Flow of Data and Control in Computational Networks", *VLSI Systems and Computations*, H.T. Kung , B. Sproull and G. Steele eds., Computer Science Press, pp. 213-225, 1981

[11] L. U. Johnsson, U. Weiser, D Cohen, and A. L. Davis, "Towards a Formal Treatment of VLSI Arrays", *2nd Caltech Conf. on VLSI*, pp. 375-398, 1981.

[12] H. T. Kung, "Why Systolic Architectures?", *Computer*, pp. 37-45, 1982.

[13] H. T. Kung, and W. Lin, "An Algebra for VLSI Algorithm Design", *Proc. Conf.on Elliptic Problem Solvers*, 1983.

[14] R. H. Kunh, "Transforming Algorithms for Single-Stage and VLSI Architectures", *Workshop on Interconnection Networks for Parallel and Distributed Processing*, 1980.

[15] O. H. Ibarra, S. M. Kim, and M. A. Palis, "Designing Systolic Algorithms Using Sequential Machines", *IEEE Trans. on Comp.* , C-35, pp. 531-542, 1986.

[16] M. Lam, and J. Mostow, "A Transformational Model of VLSI Systolic Design", *Computer*, pp. 42-52, 1985.

[17] C. Leiserson, and J. Saxe, "Optimizing Synchronous Systems", *Proc. 22nd Annual Sym. Foundations of Computer Science*, 1981.

[18] G. Li, and B. Wah, "The design of Optimal Systolic Arrays", *IEEE Trans. on Computers*, C-34, pp. 66-77, 1985.

[19] R. Melhem, and C. Guerra, "The Application of a Sequence Notation to the Design of Systolic Computations", Techn. Rep. 568, Dept. Comp. Sc., Purdue University.

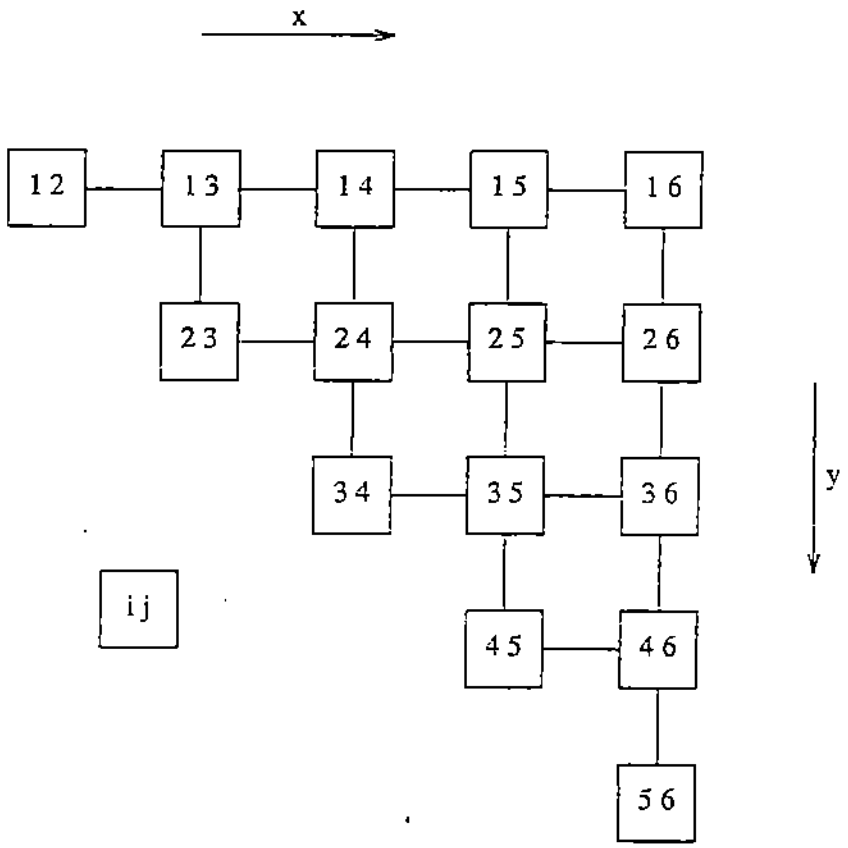
[20] W. L. Miranker, and A. Winkler, "Spacetime Representations of Computational Structures", *Computing*, vol. 32, 1984.

[21] D. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms", *IEEE Trans. on Computers*, C-31, pp. 1121-1126, 1982.

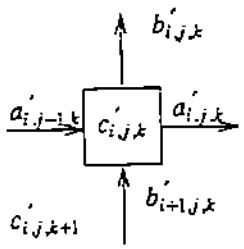
[22] D. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays", *Proc. IEEE*, vol. 71, pp. 113-120. Jan 1983.

[23] P. Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations ", *Proc. 11-th Annual Symp. on Computer Architecture*, pp. 208-214, 1984.

[24] I.V. Ramakrishnan, D. S. Fussell, and A. Silberschatz, "Mapping Homegeneous Graphs on Linear Arrays", *IEEE Trans. on Computers*, vol C-35, pp. 189-209, 1986.



for $k < \lceil (i+j)/2 \rceil$



for $k > \lceil (i+j-1)/2 \rceil$

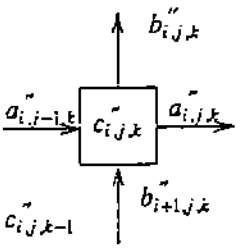
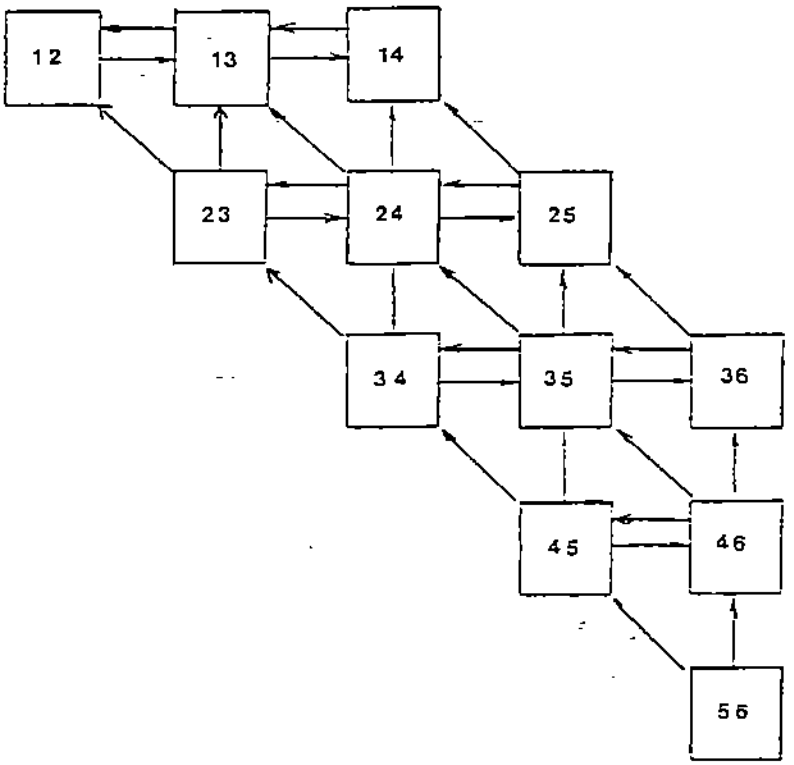
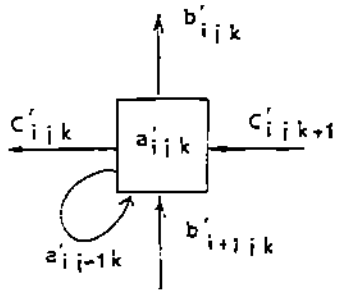
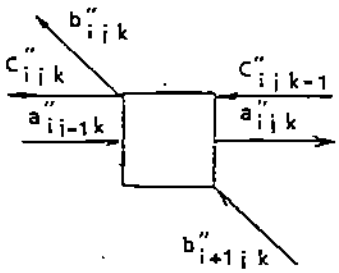


Fig. 1. A systolic array for dynamic programming





FOR $k \leq \lceil (i+j)/2 \rceil$



FOR $k > \lfloor (i+j-1)/2 \rfloor$

Fig. 2. A new design for dynamic programming

Fig. 1. A systolic array for dynamic programming

Fig. 2. A new design for dynamic programming

Index terms - VLSI algorithms, VLSI architectures, systolic algorithms, algorithm transformations, synthesis, mapping.