AUTOMATIC SYSTOLIC KALGORITHM DESIGN I
(Basic Synthesis Techniques)

G M MEGSON

Rapporteur: P Ezhilchelvan
D Comish

AUTOMATING SYSTOLIC ALGORITHM DESIGN I :
(Basic Synthesis Techniques)

G.M. Megson
Computing Laboratory
University of Newcastle-Upon-Tyne
Claremont Tower, Claremont Rd,
Newcastle-Upon-Tyne
NE1 7RU
U.K.

1.0 Introduction :

In the last decade the rapid development of VLSI computing techniques has had a significant impact on the development of novel computer architectures. One class of architectures, the so-called systolic arrays, have gained popularity because of their abilities to exploit massive parallelism and pipelining to produce high performance. Informally a systolic system can be envisaged as an array of synchronised processors (or cells) which process the data in parallel by passing it from cell to cell in a regular rhythmic pattern [1]. Systolic arrays have been designed for a wide variety of problems of which table I is only a small selection. Initial designs were ad-hoc and relied substantially on the skill and intuition of the designer. In the systolic paradigm every algorithm requires a specialized systolic design in which the communication data streams, cell definitions, and input-output are customized. Consequently the terms array and algorithm are often synonymous. The early designs contributed significantly to the foundation of systolic synthesis as a design methodology and recently a great deal of research effort has been devoted to the problem of generating systolic arrays in a systematic manner.

A number of very useful papers on synthesis can now be found in the literature (see [1][12][6]). This paper is intended to compliment existing work and present the basics of synthesis in a straightforward and intuitive way. One of the unfortunate features of the current literature is that relatively straightforward ideas and concepts are often expressed in an overly formal and stylized manner. While on the one hand these developments must be applauded, for adding much needed mathematical formulation and rigor to the subject and addressing the early criticism that systolic array design was a 'black-art', one must also criticize them for failing to make the inherent simplicity of the technique more explicit. Indeed in some cases the impression is that the formulation is more important than the actual method or its uses. Where the teaching of a subject and the 'pull-through' effect of transferring research to undergraduate and postgraduate courses is considered methods of presentation are of the utmost importance.

This paper is the first of two short tutorials, it describes some of the history behind the development of current design techniques, introduces the basic principles of synthesis methods in a simple and intuitive way, places emphasis on clarifying points which are rather vague in the existing literature, and indicates where the

current research trends are likely to lead in the future. In particular we will take the opportunity to expose one or two mis-representations regarding systolic algorithm design which have affected the acceptance of the method as little more than another special-purpose technique restricted to VLSI and ASIC design. The second paper is more research orientated and describes DTAGS, a graphical and interactive tool for performing the synthesis process [21]. DTAGS is currently under development at Newcastle and is still at quite a primitive state. However, it has been used successfully to design several new systolic algorithms ([22]) and can be employed effectively to teach the basics of synthesis methods. If nothing else it demonstrates practically the gulf that exists between current theory, so-called academic tools, and robust, safe, software for exploiting commercially available multi-processors.

Area	Application
SIGNAL PROCESSING	Signal processors for recursive filtering, implementation of Kalman filters, Discrete Fourier Transform (DFT), Convolution (multi-dimensional), Linear Algebra machines
NUMERICAL PROBLEMS	Finite Element analysis, Singular Value Decomposition, Linear time solution of Toeplitz systems, Orthogonal equivalence transformations, Least-Squares (adaptive beamforming), Eigenvalues and generalized inverses, Iterative algorithms
SHAPES & PATTERNS	Pattern Matching, Feature extraction, Pattern Classification, Stereo matching, Algorithms for recti-linear polygons, B-Splines, Neural network simulations and training.
WORDS & RELATIONS	Largest common subsequence problem, Dictionary machines, Relational Database Operations, Connected Word recognition
AUTOMATA	Tree Acceptors, Trellis Automata, Binary Tree Automata, Design Rule Checking
GENERAL	Shortest Path problem, Algebraic path problem (including matrix inverse), Fundamental sorting problems, Linear-time Greatest Common Divisor (GCD), Priority Queues, Stacks, Table generation, Simplex algorithm, Assignment problem, Knapsack problem.

Table I : Some applications of systolic algorithms

2.0 An Historical Perspective :

Figure 1, illustrates an approximate time-line for the development of the systolic processing paradigm. There are two strands, the first which acknowledges the existence of simultaneous efforts in automatic extraction of parallelism using compiler extensions and based primarily on the notion of data dependence analysis to determine partial orders of instructions. Such techniques can largely be summarized by so-called loop restructuring mechanisms for source to source transformation of programs. This 'software' strand is not intended to be comprehensive but simply to establish the origins of nested-loop analysis using a recurrence formulation which is now the basis for many systolic synthesis methods. The second strand deals with the development of systolic arrays and can be broken down into four distinct phases. The first or 'pre-systolic' phase covers developments upto the first appearance of the term 'systolic' in the literature (circa 1979). In particular it is important to acknowledge the existence of very similar pre-systolic paradigms such as cellular automata where cells define finite state machines (or automatons) which communicate with each other. These ideas and the development of Very Large Scale Integration (VLSI) manufacturing techniques made it feasible to build silicon chips with many simple cells and created a technological environment in which systolic processing could flourish.

The second phase we shall term 'pure-systolic'. Roughly this phase covers the early period of systolic computation form 1978-1984. During this time the basic 'axioms' of systolic design such as regular arrays of simple cells with nearest neighbour communication, and recurrent synchronous movement of data were established. In particular the design philosophy was shown to work well for a wide range of compute-bound applications in signal and image processing (FFT's, digital filters, solution of linear systems, differential equations and so on). Bit serial arrays using fixed-point arithmetic began to emerge for computationally intensive tasks such as convolution, and the solution of Topelitz systems which formed earlier proto-types for commercial chips such as the NCR GAPP [23]. It was soon clear that systolic principles could be used to develop algorithmically specialized arrays for performing computationally intensive tasks in real-time. The limitations of high input-output bandwidth and complexity of basic cells for practically important problems (e.g singular value decompositions) established design 'heuristics' based on limitation of technology and lead to the development of programmable systolic devices (such as the wavefront array[24], WARP [25], and specialized processor arrays for developing systolic systems such as MICMACS [26]). As more examples emerged and the utility of the approach was established a movement towards more systematic design methods began. Early methods such as re-timing [27], cuts [28] and signal flow graph notation concentrated on transformations of one array into another by re-distribution of the circuit delays. Primitive representations of algorithms in space-time were developed and established important links between dependency graphs and array geometry using projections.

The third phase covers the period 1984-1990 and is characterized by a concerted movement towards the development of a theoretical framework based on

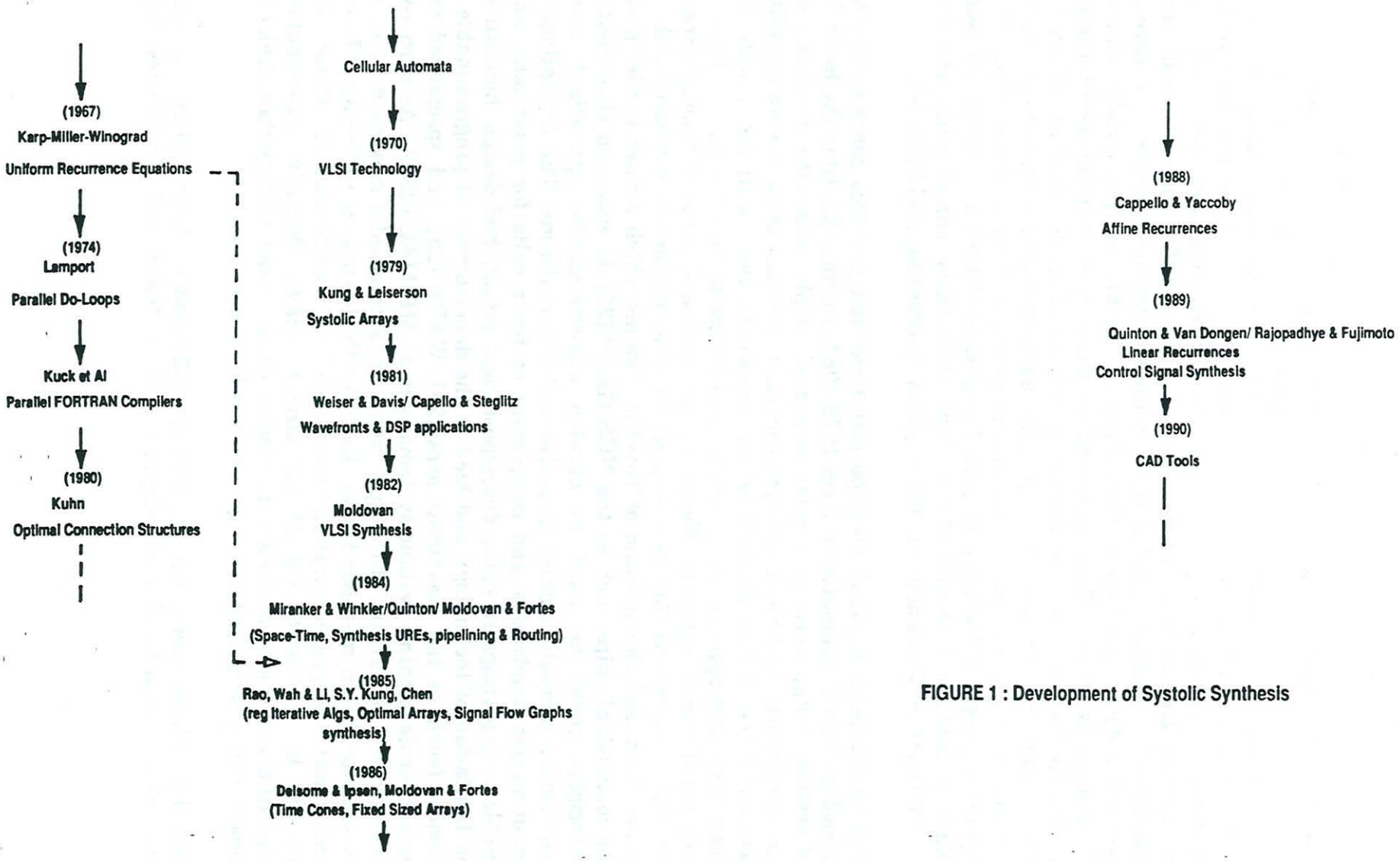


FIGURE 1 : Development of Systolic Synthesis

dependence manipulation and the association of geometry with a computation by mapping nested loop programs into Euclidean space. Sections 3-6 below outline the basic approach. These important developments have utilized concepts from source to source program transformation methods and the formulation of loop computations as uniform [2], Affine [13][15], and more recently Linear recurrence equations [11]. They have been shown to be effective for a range of important applications. In particular the systematic re-timing of data dependency graphs to remove non-local and broadcast connections using pipelining and routing techniques is of particular importance [8]. The theoretical framework for systolic design has allowed considerable progress to be made in extending the range of established designs. Much more complicated 'hybrid' designs which rely on more sophisticated cells than envisaged for 'pure' arrays are now possible. The idea of 'soft' systolic arrays which describe an abstract model of the computation to be mapped on to existing parallel architectures are also beginning to attract interest [3][9].

The fourth phase brings us to the present where considerable effort is now being expended in the direction of CAD tools ([2][10][17][18][19]) and systolic compilers ([4][5]). The basic synthesis method serves as a back-bone for developing an integrated set of CAD tools but many questions remain unanswered. Section 7.0 discusses some of these extensions and suggests directions for future research. For example mapping arrays dependent on the problem size onto fixed sized architectures with restricted topologies. Automatically generating code for parallel machines or producing designs for direct silicon implementation. The extension of the basic synthesis method to allow development of complex and composite designs involving a number of systolic designs pipelining together is also an interesting avenue of research.

3.0 Space-Time and Euclidean Spaces :

In systolic design an algorithm is formulated as a nested-loop program with the form

```
For I1 := L1 to U1 do
  For I2 :=  L2 to U2 do
    For I3 := L3 to U3 do
                :
                :
      For In := Ln to Un do
        Begin
                :
                :
          Statement;
                :
                :
        End;
```

the Ij are loop variables or indices, and Lj, Uj are the upper and lower bounds of loop j. A statement is simply an assignment with the form

$$f(p)=g(f_1(q_1),f_2(q_2),....f_m(q_m)); \tag{3.1}$$

where p is an n component vector of the form $(I1,I2,\cdots,In)$, $m>0$ and the q_i can be defined as

$$q_i=Ap+b \tag{3.2}$$

with A an $n\times n$ matrix and b an $n\times 1$ vector. For example the statement

$$a_{ij}=a_{ij-1}+b_{i-1j} \tag{3.3}$$

has $p=(i,j)$, $q_1=(i,j-1)$ and $q_2=(i-1,j)$ $(n=2)$. The function g is addition and the f_i functions simply index the variables a and b respectively $(m=2)$. Equation (3.3) is an example of a recurrence equation. In general a single index recurrence can be written as

$$x_{i+1}=d_1x_i+d_2x_{i-1}+\cdots+d_mx_{i-m+1} \tag{3.4}$$

which is linear when the d_i are constant and non-linear otherwise. It should be clear that the left hand side of (3.4) depends *strictly* (i.e only on) preceding values of x which can be assumed to have been calculated previously. The extension to n-index recurrences follows trivially by adding extra subscripts to the variables.

A geometric interpretation of the nested-loop program is obtained by unrolling the computation into Euclidean space. For n loops we require an n dimensional space and an n component vector describes a unique point in the space. The computations associated with the assignments in the loop-body are mapped to the points described by p. For example if $i,j=1(1)2$ in (3.3), point (1,1) computes $a_{1,0}+b_{0,1}$, point (2,2) $a_{2,1}+b_{1,2}$, point (1,2) $a_{1,1}+b_{0,2}$ and so on. Observe that each point requires values computed at other points. Connecting the points that require data from (or depend) on each other produces a data dependency graph (DG) representation of the computation. The set of values that the recurrence indices can take defines the number and position of points (or domain) of the computation. Normally one assumes that loop bounds $L1$ and $U1$ are constants and that Lj, Uj for $1<j\leq n$ are expressions involving constants or Ik values where $1\leq k<j$, this ensures that the domain is finite. If a point requires a value that belongs to a point outside the domain we define the value as an input. Similarly results of computations that are sent to points outside the domain become outputs. Finally, a Uniform Recurrence Equation (URE) is defined when

$$q_i=p\pm w_i \tag{3.5}$$

so that $A=I$ (or in general a diagonal matrix) and $b=\pm w_i$ in (3.2) and w_i is a constant n component vector (where the sign can be chosen to indicate the direction of the data flow). It should be clear that w_i represents a constant data dependency which is assumed to be independent of the problem instance.

URE's play an important role in systolic synthesis, this is because it is exactly

these kind of recurrences that allow a direct mapping of computations onto locally connected architectures. In addition, it is also useful to consider a number of further restrictions. Firstly, confine m in (3.2) to a small constant so that nodes of the DG have a limited number of input-outputs (this translates to the practical problem of building cells with potentially unbounded fan-in and fan-out). Secondly the elements of the dependence vectors are kept small so that each node in the DG can communicate with nodes within a small neighbourhood of bounded radius (this prevents long connections and clock-skew problems). Finally, we assume that the functions at each of the nodes in the DG are the same (this is not necessary but simplifies the discussion).

4.0 Scheduling and Allocation :

Given a DG for a particular problem we can derive systolic arrays for performing the computation by determining a pair $(t(p),a(p))$ where $t(p)$ and $a(p)$ are timing and allocation functions which determine the time and place that the computation associated with point p must be evaluated. It is known that if a URE is computable a linear timing function exists, consequently we can write

$$t(p)=\lambda^t p+\alpha \tag{4.1}$$

and

$$a(p)=Mp \tag{4.2}$$

where λ^t is an n component vector, α is a scalar, and M is an $(n-1)\times n$ matrix. Equation (4.1) implies that $t(p)$ is a hyperplane of the n dimensional space and a family of parallel planes exist for $t=1,2,\cdots$. All the points lying on a particular hyper-plane can be computed in parallel. It follows that the normal (i.e. λ) to the set of planes represents the direction of a time axis. Hence the Euclidean space can be interpreted as a space-time system with 1 time dimension and $n-1$ space dimensions.

4.1 Timing :

To derive a timing schedule λ and α must be determined, this is achieved by embedding the domain of the computation inside a convex polytope (or cone) and deriving a suitable linear programming (LP) problem. A *convex cone (S)* is defined as a subset of n-dimensional space if and only if for two points p and q in S a point $\alpha p+\beta q$ also in S is produced when $\alpha\geq 0$, $\beta\geq 0$. A *ray* is defined as $q=\alpha p$ for $\alpha\geq 0$ and any point p in the n dimensional space. Provided the loops of the program satisfy our assumptions above the domain of computation can be defined by a set of linear inequalities (half-spaces, or directed hyper-planes) and forms a finite convex polyhedron. The corners of the polyhedron correspond to the 2^n possible combinations of the lower and upper loop bounds and are termed *vertices*. Now it is known that a finite set of half-spaces can be used to define a convex polyhedron. It follows that any convex domain can be embedded into a cone of suitable size and that a timing schedule can be determined by solving an LP constructed from the rays, vertices, and data dependencies in the cone. For simplicity we will choose a cone with an apex at the origin and define rays so that they all pass through the origin. In general it is better to use the so-called extremal rays determined by the

vectors lying on the faces of the domain defined by the half-spaces (because the number of rays is smaller and limits the size of the LP).

Now, if we define unit time to be the cost of evaluating the most complex computation of any of the nodes in the domain then

$$t(p) \geq t(q)+1 \tag{4.3}$$

as point p depends on the results of point q. Now put $q=p-w$ and substitute in (4.1) to produce

$$\lambda^t p+\alpha >= \lambda^t(p-w)+\alpha+1 \rightarrow \lambda^t w \geq 1 \tag{4.4}$$

which yields a single equation in λ and α for each dependency. Similarly, we can write

$$\lambda^t v+\alpha \geq 0 \tag{4.5}$$

for each vertex v which asserts that all computation must occur at times $t \geq 0$. Finally, for two points p and q on the same ray r we can write

$$p=\beta r, \quad q=\gamma r, \quad \beta>0, \gamma>0, \beta>\gamma$$

and by (4.3) $t(p)>t(q)$ hence

$$\lambda^t p>\lambda^t q \rightarrow \beta\lambda^t r>\gamma\lambda^t r$$

so that

$$\lambda^t r>0 \tag{4.6}$$

otherwise we can find a case when $t(q)>t(p)$ which is not permitted. A schedule $t(p)$ can now be found by solving the LP problem defined as

$$minimize \quad \lambda^t g+\alpha=t(g) \tag{4.7}$$

subject to (4.4), (4.5), (4.6) for all rays, vertices, and dependencies. The point g (normally) being chosen as the point that is to be computed last. It is important to note that there are generally less vertices, rays, and dependencies than actual points in the domain so the LP is independent of the problem size.

4.2 Allocation :

Geometrically an allocation is a projection of points in n dimensions into a space with $n-1$ dimensions. For example, when $n=2$

$$P=\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix} \quad Q=\begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix} \tag{4.8}$$

are projections. Applying (4.2) shows that P removes the j axis, and Q removes the i axis consequently P is a projection along the i-axis while Q projects along the j-axis. A projection in an arbitrary direction described by the vector $d=(i\,j)$ is produced by applying a rotation to move point (i,j) in space onto one of the axes and then projecting in a direction parallel to the axis and towards the origin, hence

$$M=QRd$$

projects an arbitrary point (i,j) when R is defined as

$$R=\frac{1}{\sqrt{i^2+j^2}}\begin{bmatrix} i & j \\ -j & i \end{bmatrix}$$

for $n=2$. In general M is an $(n-1)\times n$ matrix and R is constructed from a sequence of $n-1$, 2×2 Givens rotations embedded into $n\times n$ matrices so that

$$R=R_2\cdots R_n$$

where R_i annihilates the i th component of d. A projection $\bar{P}$ in the Rd direction maps d to the origin and consequently a projection of the computation is found by applying $M=\bar{P}R$ to all the nodes in the DG.

5.0 Non-Uniformity :

So far we have considered only uniform recurrences, when A is not diagonal a non-uniform recurrence results. The practical implications of non-uniformity are that data is broadcast simultaneously to a number of cells possibly over long distances. Fortunately it is possible to remove non-uniformity from recurrences by systematically applying two methods *pipelining* and *routing*.

5.1 Pipelining :

Suppose we are a given a non-uniform recurrence with the form of (3.2). Further suppose that we can find a non-null vector v such that $Av=0$ (that is v belongs to the kernal of A). It follows that

$$Ap+b=Ap-Av+b=A(p-v)+b \tag{5.1}$$

For example, consider the recurrence

$$y_{ij}=y_{ij-1}+a_{ij}x_{0j} \tag{5.2}$$

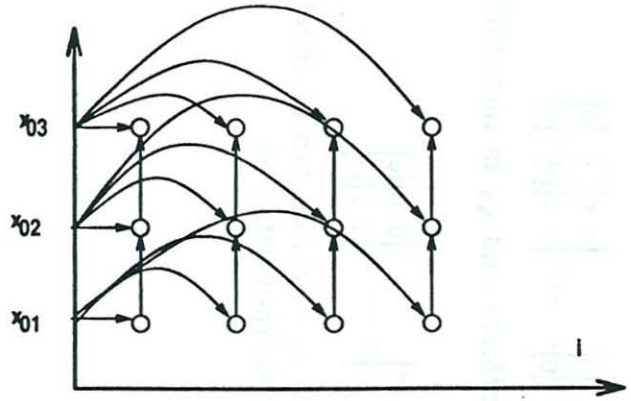
we can write the dependencies as

$$q_1=\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\-1\end{bmatrix},\quad q_2=\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix},\quad q_1=\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$$

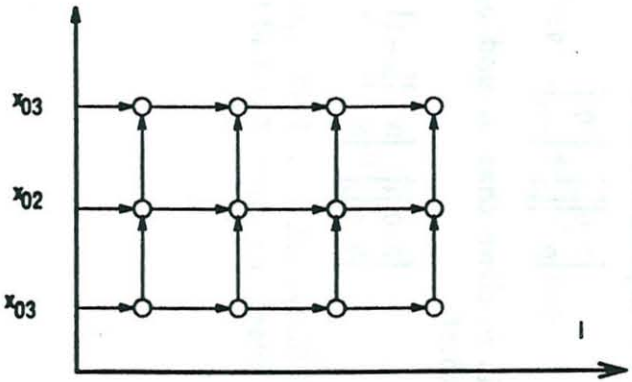
from which it is clear that q_1 and q_2 are uniform and q_3 is non-uniform. Thus according to (5.1)

$$\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}=\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}i-1\\j\end{bmatrix}=\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}i-2\\j\end{bmatrix}=\cdots=\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}0\\j\end{bmatrix} \tag{5.3}$$

where $v=(\gamma,0)$ for $\gamma=1,2,\cdots$. Consequently the x_{0j} variable can be pipelined in the i direction to reach (i,j) from $(0,j)$ and (5.2) can be re-written as



a) before pipelining



b) after pipelining

FIGURE 2 : Pipelining

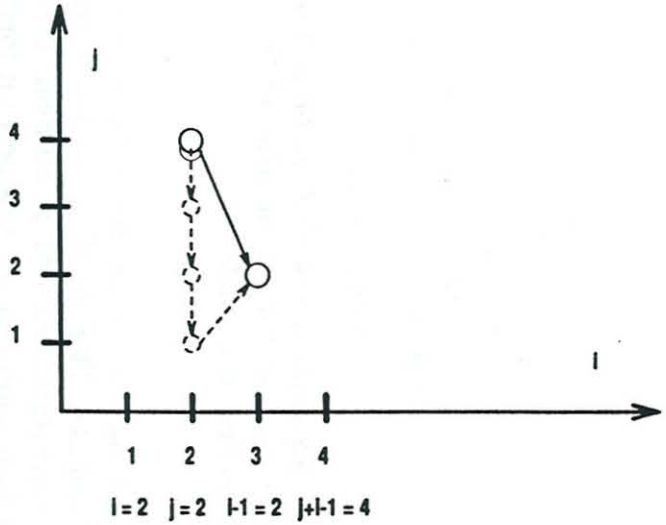


FIGURE 3 : Data Routing

$$y_{ij} = y_{ij-1} + a_{ij} X_{ij} \tag{5.4a}$$

$$X_{ij} = \begin{cases} X_{i-1j} & i > 1 \\ x_{0j} & i = 1 \end{cases} \tag{5.4b}$$

which is a system of uniform recurrence equations, the x_{0j} being an input to the domain. Both X_{ij} and Y_{ij} are computed at point (i,j) in space-time, with X_{ij} performed before Y_{ij} because the latter depends on the former. A visual interpretation of the transformation is shown in Figure 2. More formally we can rewrite the equation

$$f(p) = g(\cdots f_r(Aq+b)\cdots) \tag{5.5}$$

by the equivalent system

$$f(p) = g(\cdots F_r(p)\cdots) \tag{5.6a}$$

$$F_r(p) = \begin{cases} F_r(p-v+b) & \textit{for } (p-v+b) \textit{ in the domain} \\ f_r(Aq+b) & \textit{for } Aq+b \textit{ not in the domain} \end{cases}$$

5.2 Routing :

When no non-null vector v, which satisfies $Av=0$ exists, pipelining cannot be applied and routing must be used instead. For instance suppose we have the recurrence

$$y_{ij} = y_{ij-1} + x_{i-1j+i-1} \tag{5.7}$$

where x is non-uniform and $v=0$ is the only vector satisfying $Av=0$. The x dependency shows that point (i,j) depends on data at $(i-1,j+i-1)$ hence a connection defined by the vector $(1,-i+1)$ is required between the two points. Now any n dimensional point can be written as the linear combination of n linearly independent vectors that form a basis spanning the space. Thus a general point (i,j) can be written as

$$\begin{bmatrix} i \\ j \end{bmatrix} = i\begin{bmatrix} 1 \\ 0 \end{bmatrix} + j\begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{5.8}$$

similarly we can write

$$\begin{bmatrix} 1 \\ -i+1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + i\begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{5.9}$$

so that data is routed between the two original points using only nearest neighbour connections incorporating i steps in the $(0,-1)$ direction followed by one step in the $(1,1)$ direction. Figure 3 illustrates the principle graphically. Unfortunately re-writing the recurrences is non-trivial because additional control must be added to existing nodes to route the data. On contrast pipelining only involves a single direction and is to be preferred whenever possible because of it's simplicity.

6.0 Design Methodology :

We are now in a position to state a simple methodology for designing systolic algorithms, for simplicity this procedure may be stated simply as follows

Step 0 : Fully Index the problem by adding zeroes to the variables of the loop statements so that every variable has n indexes.

Step 1 : Separate the equations into three groups called input, computation and output equations respectively. (the resulting system is often called a *canonical* or *normal* form).

Step 2 : Remove the non-uniformities from the problem to produce a system of URE's - tuple the system into a single URE.

Step 3 : Set up the LP problem from the rays, vertices, and dependencies of the problem, and produce a set of feasible timing functions $t(p)$.

Step 4 : derive a set of feasible allocations $a(p)$ for each $t(p)$ by selecting an appropriate projection direction.

Step 5 : Evaluate the design described by each $(t(p), a(p))$ pair using and objective function relevant to the application domain. If satisfied Stop otherwise repeat steps 2-4 with different pipelining and routing vectors.

The design algorithm is best illustrated by recourse to a suitable example - in this case we will choose the well-known convolution problem which can be written in for-loop notation as follows

```
For i := 1 to n-k+1 do
 Begin
  y[i] = 0;
  For j := 1 to k do
    y[i] = y[i] + w[i]*x[i+j-1];
 End;
```

The w values are called weights and $k<<n$ is usual. Our objective is to define a systolic array which computes the result with the smallest number of time steps possible and with the least number of cells. In addition we might also want to restrict the number of input and outputs to the array or make particular data sequences stationary or non-stationary. For example, uni-directional arrays are amenable to fault tolerant implementation techniques. A fully indexed recurrence formulation of the computation has the form

$$y_{i,j} = y_{i,0} + w_{0,j} x_{0,i+j-1} \qquad (6.0)$$

from which it is clear that all the variables on the right hand side have non-uniform indices.

For the y variables we can write

$$\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}=\begin{bmatrix}i\\ 0\end{bmatrix}\rightarrow v=\begin{bmatrix}0\\ -1\end{bmatrix} \tag{6.1}$$

Similarly for the w values

$$\begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}=\begin{bmatrix}0\\ j\end{bmatrix}\rightarrow v=\begin{bmatrix}-1\\ 0\end{bmatrix} \tag{6.2}$$

and for x we derive

$$\begin{bmatrix}0 & 0\\ 1 & 1\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ -1\end{bmatrix}=\begin{bmatrix}0\\ i+j\end{bmatrix}+\begin{bmatrix}0\\ -1\end{bmatrix}\rightarrow v=\begin{bmatrix}1\\ -1\end{bmatrix} \tag{6.3}$$

hence we can re-write (6.0) as follows

$$Y_{ij}=Y_{ij-1}+W_{ij}X_{ij} \tag{6.4a}$$

$$W_{ij}=\begin{cases}W_{i-1j} & i>1\\ w_{0j} & i=1\end{cases} \tag{6.4b}$$

$$X_{ij}=\begin{cases}X_{i-1j+1} & i>1\\ x_{0,i+j-1} & i=1\end{cases} \tag{6.4c}$$

where $Y_{i,0}=0$. A geometric representation of the domain and DG for $k=3$ and $n=4$ is shown in Figure 4. Now using (4.7) the LP problem can be written as

$$\begin{bmatrix}1 & 3 & 1\\ 1 & 1 & 1\\ 4 & 3 & 1\\ 4 & 1 & 1\\ 0 & 1 & 0\\ 1 & 1 & 0\\ 1 & 0 & 0\\ 1 & -1 & 0\\ 1 & 3 & 0\\ 1 & 2 & 0\\ 1 & 1 & 0\\ 2 & 1 & 0\\ 2 & 3 & 0\\ 3 & 1 & 0\\ 3 & 2 & 0\\ 3 & 3 & 0\\ 4 & 1 & 0\\ 4 & 2 & 0\\ 4 & 3 & 0\end{bmatrix}\begin{bmatrix}\lambda_1\\ \lambda_2\\ \alpha\end{bmatrix}\geq\begin{bmatrix}0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\end{bmatrix} \tag{6.5}$$

and

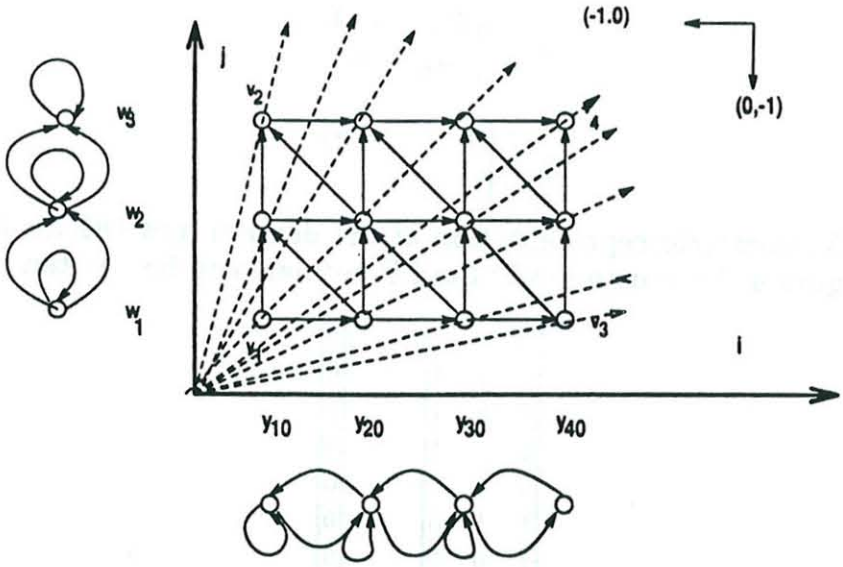


FIGURE 4 : Convolution Problem

$$minimise \quad 4\lambda_1+3\lambda_2+\alpha \tag{6.6}$$

with the first four rows derived from the vertices, the next three from dependencies, and the rest from rays. A solution to this equation gives the optimal timing function as $\lambda_1=1$, $\lambda_2=2$, $\alpha=-2$ so that

$$t(p)=i+2j-2 \tag{6.7}$$

so that $t((1,1))=1$, $t((2,1))=2$, $t((1,2))=3$ etc etc. For $n-k+1$ values of y each with k weight terms the domain contains $(n-k+1)\times k$ points hence substitution of $(n-k+1,k)$ into (6.7) gives a total time of $T=(n-k+1)+2k-2=n+k-1$ steps. A step of course is the cost of performing the scalar inner product associated with the loop body.

Figure 4 also shows the arrays resulting from simple projections along the two co-ordinate axes associated with the program loops. It should be clear that a projection parallel to the i-axis produces an array with k cells while a projection along the j-axis requires n cells. In the former the w values remain stationary while in the latter the y values do not move. The k cell array also has the advantage of uni-directional data flow compared with the n cell array requires bi-directional data movement. Both arrays require the same computing time as governed by (6.7) consequently we should choose the array with the least number of cells.

6.1 Design Principles :

At each stage in the design process we make decisions which reduces the number of possible designs. In the design process outlined above design decisions correspond to reductions in degrees of freedom. For example when we fully index the system zeroes can be placed in any position to pad up variables with less than n indices. However, it is normal practice to associate each loop index with a given dimension and then place zeroes in a way that allocates corresponding variable subscripts to the same dimension wherever possible. One reason for adopting this procedure is to limit the amount of non-uniformity introduced as a result of full indexing. Once a system has been fully indexed there usually exists a large number of possible pipelining (or routing) vectors each of which controls the direction of data movement and hence data flow in the final design. Similarly, the LP problem by definition will produce a timing function within a constant bound of being optimal for a single URE. For a system of URE's an equivalent URE is produced by a process called tupling (which basically amounts to treating the statements in a loop-body as a single compound statement). However standard simplex-type procedures work on the principle that the feasible solutions of the LP form a convex polytope. The feasible solutions of the timing function correspond to the vertices of this polytope and so a systematic procedure can be used to generate a number of different but feasible solutions. Any of these alternative functions can be used and are often useful when one wants to pipeline designs together (so that mutually compatible timing functions can be produced). For each timing function we can apply a large range of possible projection functions. For example, interpreting the direction vector of each point in the space as direction of

projection produces a possible design. In practice one often confines attention only to the directions associated with the co-ordinate axes, or a linear combination of the axes where the co-efficients of the combination are integer valued (this is because most useful arrays result from these projections). Alternatively we can choose a direction parallel to the normal vector of the timing hyperplane - this projection has the virtue of exploiting as much parallelism as possible (but the disadvantage of using the most processors and often complex dataflow). Some possible projections are also invalid, for instance projecting in a direction perpendicular to the time axis is not permitted otherwise a case when $t(p)=t(q)$ and $a(p)=a(q)$ for two points on the same hyperplane occurs. That is two computations must be performed simultaneously on the same processor which is impossible.

7.0 Extensions :

The above tutorial has given a very basic introduction to the techniques of systolic synthesis, it shows that a potentially enormous number of designs which trade-off cells, computation time, and data flow complexity exist and can be used to satisfy practical engineering constraints. Many extensions which improve the power of the technique or reduce the computational cost of the algorithms required to generate various parts of the design exist. For example we can consider using affine or non-linear timing and projection functions. Where a number of arrays are designed independently and then pipelined together piece-wise linear timing are also useful. Domains can be used as building blocks to produce composite algorithms or to construct composite domains with different operations at the nodes. Indeed in our simple examples above only one computation is possible at each of the nodes. When the operations at different nodes are permitted projections can generate architectures where the cells change function over time. Changing from one function to another requires some control of the cells. Systematically generating control flow is currently an active and interesting area of research.

The number of cells in a design are always related in some way to the loop-bounds of the program. It follows that area efficient designs are most likely to result from projections involving the axes describing the smallest faces of the convex polyhedral domain. Provided the bounds are sufficiently small and the resulting cells simple enough it may be possible to produce a dedicated array for evaluating the problem. When all the loop sizes are all significantly large or unbounded (e.g while, repeat loops) the practicality of building a dedicated array is reduced. Instead one looks towards programmable arrays with fixed numbers of processors and partitioning the DG so that it is projected onto the available connection topology of the underlying architecture becomes important. These additional topics are beyond the scope of this tutorial but work on all these areas is being actively pursued. In conclusion, the days of ad-hoc design appear numbered. Indeed it is disconcerting to find that a sizeable section of the research community continue to use heuristics methods. One possible explanation for this fact is the lack of software tools for the synthesis method. As such tools emerge and become more widespread we can expect the situation to change for the better.

References :

[1] Kung H.T, Leiserson C.E. 11980, *Systolic arrays for VLSI*, Chapter 8, Introduction to VLSI systems, Addison-Wesley, Reading Mass, 1980.

[2] Fortes J.A.B., Fu K.S., Wah B.W., 1988, "Systematic design approaches for algorithmically specified arrays". In Computer Architecture concepts and systems, eds Milutinovic, pp454-494. North Holland, Elseiver New York.

[3] Gachet P., Joinnault B., Quinton P., 1986, "Synthesizing systolic arrays using DIASTOL". In Moore, McCabe, Uruquart, Int workshop on systolic arrays. Adam-Hilger, pp25-36.

[4] Ibarra O.H., Sohn S.M., 1990, "On mapping systolic algorithms onto the hypercube". IEEE parallel and distributed computing, vol 1, 1, pp48-64.

[5] Kung S.Y., Jean S.N., 1989, "Array Compiler Design for VLSI/WSI Systems". Proc International Conference on Systolic Arrays, pp663-667.

[6] Lengauer C., 1988, "On the projection problem in Systolic Design". Report CMU-CS-88-102, Carnegie-Mellon University, Pittsburgh.

[7] Li G.H., Wah B.W., 1985, "The design of optimal systolic arrays". IEEE Trans Computers, C-34, 1, pp66-77.

[8] Miranker W.L., Winkler A., 1982), "Space-Time representations of computational structures". IBM Research report RC9775, Computing, 32, pp93-114, 1984.

[9] Moldovan D.I., 1983, "On the design of algorithms for VLSI systolic arrays". Proc IEEE, Vol 71, no 1, pp113-120.

[10] Moldovan D.I., Fortes J.A.B., 1986, "Partitioning and mapping algorithms into fixed sized systolic arrays". IEEE Trans on Computing, Vol C-35, no 1, pp1-12.

[11] Moldovan D.I., 1987, "ADVIS : A software package for the design of systolic arrays". IEEE Transactions on computer aided design, CAD-6, 01 Jan, pp33-40.

[12] Quinton P., Van Dongen V., 1989, "The mapping of Linear Recurrence Equations on Regular Arrays". J VLSI Signal processing, 1, pp95-113.

[13] Quinton P., 1984 "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations". Proc 11th Symp on Computer Architecture, IEEE Computer Society Press. New York pp208-214.

[14] Rajopadhye S.V., Fujimoto R.M., 1990, "Automating systolic array design". Integration, the VLSI Journal, 9, pp225-242.

[15] Rao S.K., 1985, "Regular Iterative Algorithms and their implementation of processor arrays". PhD Thesis, Stanford University.

[16] Yaccoby Y., Cappello P.R., 1988, "Scheduling a system of affine recurrence equations on regular arrays". Inter Conf on Systolic Arrays, San Deigo, pp373-381.

[17] Cappello P.R., Steiglitz K., 1984, "Unifying VLSI array design with linear transformations of space-time". Advances in computing research, Vol 2, pp23-65, JAI Press Inc.

[18] Engstrom B.R., Cappello P.R., 1987, "The SDEF Systolic programming system". Proc of the 1987 Int Conf on parallel processing, Aug 17-21 (1987), Editors. pp645-652, Publishers.

[19] Lange A.A.J de, 1991, "Design and implementation of highly pipelined parallel VLSI systems". Delft University of Technology, Jan 1991.

[20] Vehlies U., 1990, "DECOMP - A Program for Mapping DSP-Algorithms onto Systolic Arrays". International Workshop on Algorithms and Parallel VLSI Architectures, Pont-a-Mousson, France.
[21] Megson G.M., Comish D., 1991, "Systolic Algorithm Design Environments". 2nd International Specialist Seminar on the design and application of parallel digital processors, IEE pub No 334, pp100-104.
[22] Megson G.M., Brudaru O., Comish D., 1991, "Systolic designs for Aitken's root finding method", to appear Journal of Parallel Computing.
[23] NCR Commercial Note, 1984, GAPP: Geometric Arithmetic Parallel Processor.
[24] Kung S.Y., 1984, "On Supercomputing with systolic/wavefront array processors", Proc IEEE, 72, pp867-884.
[25] Deutch J., Maulik P.C., Mosur R., Printz H., Ribas H., Senko J., Tseng P.S., Webb J.A., Wu I.C., 1987, "Performance of WARP on the DARPA architecture Benchmarks". Proc Int Conf and Exhibition Parallel processing for computer vision and display, Leeds University.
[26] Charot F., Frison P., and Quinton P., " Systolic architectures for speech recognition", IEEE Trans ASSP, 34, pp765-779, 1986.
[27] Leiserson C.E., Saxe J.B., 1983, "Optimizing synchronous circuits", J. VLSI and Computer Systems, 1, pp41-68.
[28] Kung H.T., Lam M.S., 1984, " Wafer-scale integration and two-level pipelined implementations of systolic arrays", Journal of parallel and distributed computing, 1, pp32-63.