Chapter 5

Advanced Systolic Design

Dominique Lavenier
Patrice Quinton
Sanjay Rajopadhye
IRISA, Campus de Beaulieu
35042 Rennes Cedex
France
email: {lavenier,quinton,rajopadh}@irisa.fr

Abstract

Systolic arrays are locally connected parallel architectures, whose structure is well-suited to the implementation of many algorithms, in scientific computation, signal and image processing, biological data analysis, etc. The nature of systolic algorithms makes it possible to synthesize architectures supporting them, using correctness preserving transformations, in a theoretical framework that has a deep relationship with loop parallelization techniques. This opens the way to new very high-level architecture synthesis techniques, which will be a major step in mastering the use of IC technologies in the future. This chapter has two parts. First, we present the current state of the art in systolic synthesis techniques, and the second part surveys various ways of implementing systolic algorithms and architectures using programmable architectures, FPGAs, or dedicated architectures.

5.1 Introduction

The term systolic arrays was coined by Kung and Leiserson in 1978 to describe application specific VLSI architectures that were regular, locally connected and massively parallel with simple processing elements (PEs). The idea of using such regular circuits was even present in von Neuman's cellular automata in the fifties, Hennie's iterative logic arrays in the sixties, and also in specialized arithmetic circuits (Lyon's bit-serial multiplier [1] is clearly a linear systolic array). However, the emergence of VLSI technology in the late seventies and early eighties made the time ripe for introducing such architectures in order to highlight the characteristics appropriate to the technology.

Systolic arrays immediately caught on, since they involved a fascinating interplay between algorithm and architecture design. When researchers started investigating automatic synthesis, a third stream joined this confluence, namely the analysis, manipulation and transformation of programs. Some of the early designs are classics. The Guibas et al. array for optimal string parenthesization [2] is one of our all time favorites.

The early eighties was a period of intense activity in this area. A large number of "paper designs" were proposed for a wide variety of algorithms from linear algebra, graph theory, searching, sorting, etc. There was also much work on automatic synthesis methods, using *dependency analysis*, *space-time* transformations of inner loops, and also other formalisms such as recurrence equations.

Then, the technological evolutions in the late eighties seemed to invalidate the assumptions of the systolic model, namely that (i) locality, regularity and simplicity were primordial for VLSI and (ii) an elementary computation could be performed in the same time that it took to perform an elementary communication. The first one meant that with the improved CAD tools and increasing levels of integration, circuit designers were not obliged to *always* follow that dictates of the systolic model (which was itself applicable to only a part — albeit, a computationally significant one — of the complete application). The second one meant that the systolic space-time mapping methods were not directly applicable for general purpose parallel machines where the communication latency was an order of magnitude higher than computation speed.

A number of recent developments now lead us to believe that it is time to revive the field. First of all, technology has evolved. With the growing levels of integration, it is feasible to actually implement a number of the old "paper designs", particularly if the elementary operations are not floating point, but come from a simpler algebraic structure (such as semi-rings with additions and comparisons, etc.) Second, the circuits of today are much more complex, and we need to again question the arguments against the regularity, locality and simplicity of systolic arrays. Can an irregular circuits achieve *better* performance? Will they do so *reliably*? What will be the *design time*? A third and very important reason is the emergence of FPGAs, programmable logic and reconfigurable computing. The basic technology underlying FPGAs is regularity, locality and simplicity. It is therefore not surprising that some of the impressive successes in this domain are systolic designs. Fourth, there is the growing understanding that the techniques of systolic synthesis have a close bearing on automatic loop parallelization for general purpose parallel machines. These techniques can thus be seen as very high level synthesis methods, applicable for software *as well as* hardware, and thus form a foundation for codesign, for a certain class of problems. Coupled with the fact that recent advances in integration permit one or more programmable processor cores to be implemented "on-chip", this makes such techniques well suited for *provably correct* codesign. Finally, we believe that the increasing complexity of VLSI system design naturally leads towards formal design methods involving correctness-preserving transformation of high-level specifications.

This chapter presents two aspects of advanced systolic design, namely advanced methods for systolic array design (Secs 5.2–5.2.5), but also design techniques and case studies of advanced, state of the art systolic arrays (Sec 5.3).

In the first part we use a formalism called systems of recurrence equations (SREs) to describe both the initial specification as well as the final array. The entire process of systolic design is viewed as the application of a series of transformations to the initial specification, until one obtains a description of the final array. We desire that all the details, including control, I/O, interface etc. be specified in the same formalism, and that a simple translation step should yield a description that is suitable for a conventional CAD tool (say VHDL). For each transformation we are concerned with two aspects: (i) the manipulation of the SRE to obtain a provably equivalent SRE (à la 'correctness-preserving' program transformations), and (ii) the choice of the transformation, usually with a view to optimizing certain cost criteria. In Section 5.2 we present the basic notations of recurrence equations and their domains. In Section 5.2.2 we describe the foundations of the first aspect, namely the formal manipulation of recurrence equations. In sections 5.2.3 to 5.2.5, we explain how these various transformations can be used to transform a specification into an abstract architecture: section 5.2.3 details scheduling techniques, section 5.2.4 deals with allocation of computations on processors, and finally, section 5.2.5 is concerned with localization and serialization transformations.

In the second part, we illustrate four different ways to implement systolic arrays: general-purpose programmable architectures (Section 5.3.1), application oriented programmable architectures (Section 5.3.2), reconfigurable architectures (Section 5.3.3), and special-purpose architectures (Section 5.3.4), and we illustrate each one with one case study. Although some of the designs presented are not new, a close study of their design is instructive. As the technology of integrated circuits is still in full motion, we believe that lessons have to be learned from these examples, in order to take the best advantage of future architectural opportunities.

5.2 Systolic Design by Recurrence Transformations

Systems of Recurrence Equations (SREs) play an important part in the design process. They are useful as (i) behavioral descriptions of the final arrays, and also (ii) high level specifications of the initial algorithm. For example, consider the system of equations given below.

$$X(t,p) = \begin{cases} \{p=0, -n+1 \leq t < 0\} & : & 0 \\ \{p=0, t \geq 0\} & : & x_t \\ \{0 < p < n, 2p \leq t\} & : & X(t-2, p-1) \end{cases} \tag{1}$$

$$W(t,p) = \begin{cases} \{t=0\} & : & w_p \\ \{0 < t\} & : & W(t-1, p) \end{cases} \tag{2}$$

$$Y(t,p) = \begin{cases} p=0, t \geq 0 & : & W(t,p) * X(t,p) \\ n > p > 0, t \geq p & : & Y(t-1, p-1) + W(t,p) * X(t,p) \end{cases} \tag{3}$$

If we interpret the index t as the time and the index p as the processor, this system can be viewed as a specification of the values that will appear in the 'registers' `X`, `W` and `Y` of an linear array with n-processors as shown in Fig. 5.1. From Eqn. (1) we see that (other than in the boundary processor $p = 0$), the value in the `X` register of processor p at time instant t is the same as that in the `X` register of processor $p-1$ at time $t-2$ (this corresponds to an interconnection between adjacent processors with a delay of 2 cycles.) Similarly, the `W` values stay in the processors after initialization, and the `Y` values propagate between adjacent processors with a 1-cycle delay (after being incremented by the `W*X` product in each processor). Note that the equations do not clearly specify how the `W` registers are

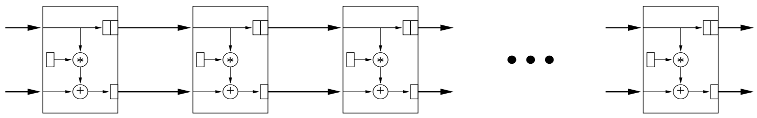

Figure 5.1 The systolic array for Eqns. (1–3).

'loaded' at $t = 0$, nor do they describe the initial values in the `X` registers of the processors other than the first one. These details and the control signals to ensure correct loading and propagation of initial values can be included, but have been omitted in the interests of simplicity. It is well known that this array computes the convolution of the x stream with the coefficients w.

The same SRE formalism, augmented with *reduction operations*, can serve as a very high level, mathematical description of the algorithm for which a systolic array is to be designed. For example, the n-point convolution of a sequence of samples $x_0, x_1, \ldots$ with the weights $w_0 \ldots w_{n-1}$ produces the sequence $y_0, y_1, \ldots$ given by the following equation (assuming that $x_k = 0$, for $-n < k < 0$).

$$y_i = \sum_j w_j x_{i-j} \tag{4}$$

5.2.1 Recurrence Equations: Definitions and Notation

We present the fundamental definitions of recurrence equations, and the domains over which they are defined. In what follows, $\mathbf{Z}$ denotes the set of integers, and $\mathbf{N}$ the set of natural numbers.

Definition 5.2.1 *A* ***Recurrence Equation*** *defining a function (variable) X at all points, z, in a domain, D, is an equation of the form*

$$X(z) = D^X \quad : \quad g(\ldots X(f(z)) \ldots) \tag{5}$$

where

- *z is an n-dimensional* ***index variable***.
- *X is a* ***data variable****, denoting a function of n integer arguments; it is said to be an n-dimensional variable.*
- *$f(z)$ is a* ***dependency function*** *(also called an* ***index*** *or* ***access*** *function), $f : \mathbf{Z}^n \rightarrow \mathbf{Z}^n$;*
- *the "$\ldots$" indicate that g may have other arguments, each with the same syntax;*
- *g is a strict, single-valued function; it is often written implicitly as an* expression *involving operands of the form $X(f(z))$ combined with basic operators and parentheses.*

- D^X *is a set of points in* $\mathbf{Z}^n$ *and is called the* ***domain*** *of the equation. Often, the domains are* ***parameterized*** *with one or more (say, l) size parameters. In this case, we represent the parameter as a vector,* $p \in \mathbf{Z}^l$, *and use p as an additional superscript on D.*

A variable may be defined by more than one equation. In this case, we use the syntax shown below:

$$X(z) = \begin{cases} & \vdots \\ D_i & : \quad g_i(\ldots X(f(z))\ldots) \\ & \vdots \end{cases} \tag{6}$$

Each line is called a ***case****, and the domain of X is the union of the domains of all the cases,* $D^X = \bigcup_i D_i$ *(actually, the convex hull of the union, for analysis purposes). The D_i's must be disjoint. Indeed, the domain appearing in such an equation must be subscripted (annotated) with (i) the name of the variable being defined, (ii) the branch of the case and also (iii) the parameters of the domain. When there is no ambiguity, we may drop one or more of these subscripts.*

Finally, the expression defining g, may contain ***reduction operators****, i.e., associative and commutative binary operators applied to a collection of values such as addition ($\sum$), multiplication (Π), minimum ($\min$), maximum ($\max$), boolean or ($\bigvee$), boolean and ($\bigwedge$), etc. These operators are subscripted with one or more* ***auxiliary*** *index variables, z', whose scope is local to the reduction. A domain for the auxiliary indices may also be given.*

Definition 5.2.2 *A recurrence equation (5) as defined above, is called an* ***Affine Recurrence Equation*** *(ARE) if every dependence function is of the form,* $f(z) = Az + Bp + a$, *where A (respectively B) is a constant $n \times n$ (respectively, $n \times l$) matrix and a is a constant n-vector. It is said to be a* ***Uniform Recurrence Equation*** *(URE) if it is of the form,* $f(z) = z + a$, *where a is a constant n-dimensional vector, called the dependence vector.* URE*s are a proper subset of* ARE*s, where A is the identity matrix and $B = 0$.*

Definition 5.2.3 *A* ***system*** *of recurrence equations (SRE) is a set of m such equations, defining the data variables $X_1 \ldots X_m$. Each variable, X_i is of dimension n_i, and since the equations may now be mutually recursive, the dependence functions f must now have the appropriate (not necessarily square) matrices. We are interested in systems of* ARE*s (SARE*s*) where all dependence functions are affine, and also a proper subset, systems of* URE*s (SURE*s*). Note that in a* SURE, *the domains of all variables must have the same number of dimensions, since A has to be the identity matrix.*

Domains

An important part of the SRE formalism is the notion of domain, i.e., the set of indices where a particular computation is defined. The domain of the *variables* of an SRE are usually specified explicitly. The domains that we use are *polyhedra* (i.e., the set of integral indices that satisfy a finite number of linear (in)-equality constraints), or a union of finitely many polyhedra. The domains may have one or more *size parameters*. Often, other families of domains such as lattices, sparse polyhedra (the intersections of lattices and polyhedra) and linearly bounded lattices (the image of polyhedra by affine functions) are used, but will not be discussed here for the sake of simplicity. The domains are often *parameterized* by a number of size parameters. An important goal of the research in this field is to elaborate parameter-independent analysis techniques and architectures.

A subtle point to note is that in an SRE each (sub) expression itself can be assigned a domain, and this can be deduced using simple rules. For example, let us do this for each sub-expression on the right-hand side (rhs) of Eqn. 4. First, note that the expression within the summation has a 2-dimensional domain, since it involves two indices i and j. The j index is local, and not visible outside the summation, the domain of the expression $\sum_j w_j x_{i-j}$ is thus the *projection* along j of the domain of the expression $w_j x_{i-j}$. We now reason as follows:

- The domain of w is $\{i|0 \leq i < n\}$.
- The domain of the expression w_j is the set of (2-dimensional) index points $[i, j]$ for which w_j is defined, i.e., points such that w_j belongs to the domain of w. This imposes the constraint that j must be between 0 and n (and no constraint on i), and gives us the 'infinite strip', $\{i, j|0 \leq j < n\}$.
- The domain of x is $\{i| - n < i\}$ (remember it is padded by $n - 1$ zeros).
- The domain of x_{i-j} is the set of points $[i, j]$ such that $i - j$ belongs to the domain of x, yielding $\{i, j| - n < i - j\}$.
- The expression $w_j x_{i-j}$ is the product of the two sub-expressions, w_j, and x_{i-j}. It is thus defined only where both operands are present, and hence its domain is the *intersection* of the respective domains of w_j and x_{i-j}, namely the domain $\{i, j|0 \leq j < n, j - n < i\}$.
- Finally, the domain of the entire rhs is the *projection* of the domain of $w_j x_{i-j}$ on the i axis, and is given as $\{i|-n < i\}$. Observe that this includes the domain of the variable y, namely the half line $\{i|0 \leq i\}$, and hence the equation is 'well formed', i.e. there exists a valid definition for every point in the domain of y.

Finally, we precisely define which SREs correspond to a systolic array. Actually, it is not easy to give a formal definition without being either too restrictive or overly general. The following definition corresponds to a restrictive version. Some of the restrictions below can be relaxed (yielding, what may not be an efficient or practically viable systolic array).

Definition 5.2.4 *A **systolic array** is defined as a* SRE *where*

- *All variables, except possibly inputs and outputs, have the same number of dimensions.*

- *All dependency functions (except possibly those involving input and output variables) are uniform.*
- *Each instance of each input variable is used exactly once (typically, but not exclusively at boundary processors).*
- *One index variable, can be interpreted as the* time, *i.e., all dependency vectors are strictly negative in this dimension.*
- *The other indices are interpreted as processor coordinates.*
- *The spatial components of all dependency vectors are interpreted as interconnections between processors.*

5.2.2 Synthesis by Formal Manipulation of SREs: A Walk-through

We now describe the various transformations that can be applied to an SRE. We will focus primarily on describing the basic formal manipulations and on ensuring that the resulting SRE is computationally equivalent to the original one. Another subject, that we will not consider here, is the choice of the particular transformation to apply. As a running example, we will use the convolution SRE of Eqn. 4, and we will illustrate how the SRE of Eqns. 1–3 are systematically derived from it.

Serialization of Reductions

In this transformation we replace the reduction operations by a sequence of binary operations. This is needed because the hardware elements that one uses to implement the algorithm have bounded 'fan-in'. The choice of the transformation involves the 'direction of accumulation': either the *increasing* or *decreasing* order of j, for the convolution example (recall that j is the local index introduced by the reduction), and also the name of the temporary variable in which to accumulate the partial results. Suppose we accumulate in the increasing j direction, and use the name Y for our new variable, we will obtain the following SRE:

$$y_i = Y(i, n-1) \tag{7}$$

$$Y(i,j) = \begin{cases} \{j = 0, 0 \leq i\} & : \quad w_j * x_{i-j} \\ \{0 < j < n, 0 \leq i\} & : \quad Y(i, j-1) + w_j * x_{i-j} \end{cases} \tag{8}$$

To understand why this SRE is equivalent to the reduction, and how it can be derived automatically, consider the expression within the reduction of Eqn. 4, namely $w_j x_{i-j}$. At each point $[i, j]$ in its domain, we associate, in addition to the product of the w and x terms, a *partial sum*, which is the accumulation of the terms 'up to j'. The domain of the auxiliary variable is thus the same as that of $w_j x_{i-j}$. Moreover, we see that the accumulation corresponds to a recurrence defining $Y(i, j)$ with the (uniform) dependency $Y(i, j-1)$, and this follows from our choice of the accumulation direction. The initialization of the Y variable (the clause in its recurrence where there is no dependency on Y) corresponds to the (sub) domain where $j = 0$, and the result is available in the $j = n$ subdomain (this is reflected in the $Y(i, n-1)$ dependency in Eqn. 7). These 'boundaries' of the domain of Y can be deduced by translating it by ± 1 in the j direction and determining the difference.

Alignment

In the SRE (7–8), variables x and w, defined over 1-dimensional domains, are used in Eqn. 8 defining Y, a 2-dimensional variable. We therefore *align* x and w with the 2-dimensional domain of Y. In particular, let us choose to align w with the $i = 0$ line, and x with the $j = 0$ line. Since x and w are inputs of the SRE, we will introduce two auxiliary variables, say W and X, yielding the following SRE.

$$y_i = Y(i, n-1) \tag{9}$$

$$Y(i,j) = \begin{cases} \{j = 0, 0 \le i\} & : \ W(0,j) * X(i-j, 0) \\ \{0 < j < n, 0 \le i\} & : \ Y(i, j-1) + W(0,j) * X(i-j, 0) \end{cases} \tag{10}$$

$$W(i,j) = \{ \ \{i = 0, 0 \le j < n\} \ : \ w_j \tag{11}$$

$$X(i,j) = \{ \ \{j = 0, 0 \le i\} \ : \ x_i \tag{12}$$

We will see later that alignment is a particular case of a general transformation called *change of basis*.

Localization

This SRE does not have uniform dependencies and localization enables us to derive an equivalent SURE. To illustrate this transformation, note that although the dependency of Y on X is not uniform, it is *many to one*. Indeed, *all* points in the domain of Y, lying on the straight line $i - j = c$ depend on $X(c, 0)$. Moreover, the point $[c, 0]$ also lies on this line. As a result, we can introduce an auxiliary variable, X' in our SRE whose domain is $\{i, j | 0 \le j < n, 0 \le i\}$ and 'pipeline the X value to all the points where it is needed'. Using a similar argument for W, we obtain the following SRE.

$$y_i = Y(i, n-1) \tag{13}$$

$$Y(i,j) = \begin{cases} \{j = 0, 0 \le i\} & : \ W'(i,j) * X'(i,j) \\ \{0 < j < n, 0 \le i\} & : \ Y(i, j-1) + W'(i,j) * X'(i,j) \end{cases} \tag{14}$$

$$W(i,j) = \{ \ \{i = 0, 0 \le j < n\} \ : \ w_j \tag{15}$$

$$X(i,j) = \{ \ \{j = 0, 0 \le i\} \ : \ x_i \tag{16}$$

$$W'(i,j) = \begin{cases} \{i = 0, 0 \le j < n\} & : \ W(i,j) \\ \{i > 0, 0 \le j < n\} & : \ W'(i-1, j) \end{cases} \tag{17}$$

$$X'(i,j) = \begin{cases} \{j = 0, 0 \le i\} & : \ X(i,j) \\ \{0 < j < n, 0 \le i\} & : \ X'(i-1, j-1) \end{cases} \tag{18}$$

The legality of this transformation can be shown by an inductive argument, provided that the new pipelining and *initialization* dependency 'spans' the set of points that depend on a given value – any point that needs a given value can be reached by composing one copy of the initialization dependency and a positive number of copies of the pipelining dependency.

SREs can be viewed as *functional programs.* In particular, they are *referentially transparent* and one can substitute equals for equals. Hence, manipulations that one performs on mathematical equations (epitomized by phrases such as "substituting for x and simplifying" that one finds in mathematical discourse) can equally well be performed on SREs. For example, we can substitute the uses of $W(i, j)$ and

$X(i,j)$ (on the rhs of Eqns. 17 and 18) by their definitions, and then Eqns. (15–16) can be dropped since they are not used in any other equation. This (followed by dropping the primes on the remaining variables) yields the SRE given below. Although this seems utterly obvious since we are manipulating SREs by hand, such transformations can be performed automatically on *any* SRE.

$$y_i = Y(i, n-1) \tag{19}$$

$$Y(i,j) = \begin{cases} \{j=0, 0 \le i\} & : \quad W(i,j) * X(i,j) \\ \{0 < j < n, 0 \le i\} & : \quad Y(i,j-1) + W(i,j) * X(i,j) \end{cases} \tag{20}$$

$$W(i,j) = \begin{cases} \{i=0, 0 \le j < n\} & : \quad w_j \\ \{i > 0, 0 \le j < n\} & : \quad W(i-1,j) \end{cases} \tag{21}$$

$$X(i,j) = \begin{cases} \{j=0, 0 \le i\} & : \quad x_i \\ \{0 < j < n, 0 \le i\} & : \quad X(i-1,j-1) \end{cases} \tag{22}$$

Change of basis

Finally, we may apply a *reindexing transformation* (also called a *change of basis* or a *space-time mapping*) to the variables of an SRE. The transformation, $\mathcal{T}$ must admit a *left inverse* for all points in the domain of the variable. When applied to the variable X of an SRE defined as follows:

$$X(z) = \begin{cases} & \vdots \\ D_i^X & : \quad g_i(\ldots Y(f(z)) \ldots) \\ & \vdots \end{cases}$$

we obtain an equivalent SRE as follows:

- Replace each D_i^X by $\mathcal{T}(D_i^X)$, its image by $\mathcal{T}$.
- In the occurrences of a variable on the rhs of the equation for X, replace the dependency f by $f \circ \mathcal{T}^{-1}$, the composition[1] of f and $\mathcal{T}^{-1}$.
- In all occurrences of the variable X on the rhs of *any* equation, replace the dependency f by $\mathcal{T} \circ f$.

The occurrences of X on the rhs of the equation for X itself constitute a special case where the last two rules are *both* applicable, and we replace the dependency f by $\mathcal{T} \circ f \circ \mathcal{T}^{-1}$. Similar transformation rules can be developed for SREs with reduction operations, but are beyond the scope of this overview.

For our example, let us choose a transformation $\mathcal{T}(i,j) = i+j, j$, which maps the point $[i,j]$ to $[i+j,j]$. Here, $\mathcal{T}^{-1}(j,k) = (i-j,j)$ is the required inverse. Let us apply it to *all* the variables, X, W and Y. This gives us the following SRE:

$$y_i = Y(i+n-1, n-1) \tag{23}$$

$$Y(i,j) = \begin{cases} \{j=0, j \le i\} & : \quad W(i,j) * X(i,j) \\ \{0 < j < n, j \le i\} & : \quad Y(i-1,j-1) + W(i,j) * X(i,j) \end{cases} \tag{24}$$

$$W(i,j) = \begin{cases} \{i=j, 0 \le j < n\} & : \quad w_j \\ \{i > j, 0 \le j < n\} & : \quad W(i-1,j) \end{cases} \tag{25}$$

[1]Recall that function composition is right associative, i.e., $(g \circ h)(z) = g(h(z))$.

$$X(i,j) \quad = \quad \begin{cases} \{j=0, j \leq i\} & : \quad x_i \\ \{0 < j < n, j \leq i\} & : \quad X(i-2, j-1) \end{cases} \tag{26}$$

We now see that if we simply rename the new indices as t and p, we obtain the SRE of Eqns. (1–3) which describes the array of Fig. 5.1. As mentioned above, the transformation, $\mathcal{T}$ must admit an integral left inverse (for all points in the domain), and hence it is restricted to what are called *unimodular* transformations[2]. Often, one may desire to apply other transformations. A particular example is non-unimodular non-singular integer transformations. This leads to SREs whose domains are "sparse" (eg. all even points from 1 to n), but a detailed discussion is beyond the scope of this chapter.

We emphasize that what we have presented for the convolution example is a particular series of transformations. In general, the transformations can be applied in different orders, and indeed, infinitely many equivalent SREs can be derived from the original specification.

5.2.3 Scheduling

We have so far discussed the formal manipulations for transforming SREs and obtaining provably equivalent SREs. We now address the second question, what transformation to apply. In general, the answer to this question consists of two interrelated parts: the choice of a scheduling, and the choice of a processor allocation.

Scheduling techniques are based on representing each system of recurrence equations with *dependence graphs*, defined as follows. The *dependence graph* (or *reduced dependence graph*) $G = (V_G, E_G)$ of a SRE is the graph whose vertices V_G are the variables of the system, and there exists an arc $e \in E_G$ between two vertices, Y and X if and only if there exists an equation where X is the left-hand side, and Y is an argument of the right-hand side. The arcs of this graph are labeled with information describing the dependencies. For example, in the system of Eqns. (23–26), the nodes of the dependence graph (shown in figure 5.2) are y, x, w, Y, X, and W. There exists an arc from Y to X, since the definition of Y depends on that of X.

Scheduling SUREs

SUREs (Systems of Uniform Recurrence Equations, see Def. 5.2.3) are of particular interest in order to study scheduling. In a SURE, the dependencies can be represented by n-dimensional vectors giving the translation between the index points of dependent instances of two variables. Thus, dependencies can be summarized in a *dependence matrix* D whose columns are the dependence vectors of the SURE.

In the example of Eqns (23–26), if we consider only variables which are neither inputs (x, and w) nor output (i.e., y), we can see that the system is uniform: $Y[i,j]$ depends on $X[i,j]$, $W[i,j]$ and $Y[i,j-1]$, $X[i,j]$ depends on $X[i-1,j-1]$, and finally, $W[i,j]$ depends on $W[i-1,j]$. We notice also that the dependency between $Y[i,j]$ and $X[i,j]$ is somehow "artificial", in the sense that $Y[i,j]$ depends in fact

[2] A matrix is said to be unimodular if its determinant is ± 1. Hence an integral unimodular matrix admits an integral inverse.

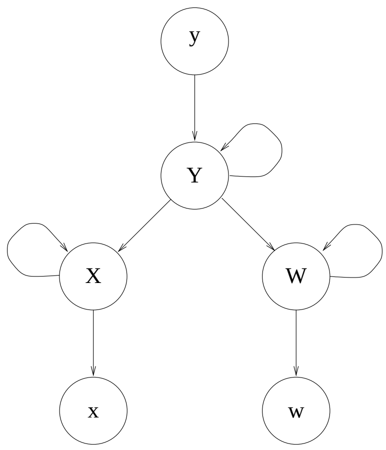

Figure 5.2 Dependence graph associated with the system of Eqns. (23–26)

on $X[i-1, j-1]$, through the definition of X. A similar remark shows that $Y[i, k]$ depends, in fact, on $W[i-1, k]$.

Thus, the following dependence matrix summarizes all the information which is needed to schedule this system of equations:

$$D = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} .$$

The dependence graph can be analyzed via the so-called *dependence relation*. Consider the set $\mathcal{I}$ of pairs (X, x), where X is a variable, and $x \in D^X$. Such a pair is called an *instance* in the following. Given two instances (X, x) and (Y, y) we say that (X, x) depends directly on (Y, y), denoted by $(X, x) \stackrel{1}{\leftarrow} (Y, y)$, if and only if there exists an instance of equation $X(x) = f[\ldots, Y(y), \ldots]$. The relations "depend in t steps on" $(X, x) \stackrel{t}{\leftarrow} (Y, y)$, and "depend on" $(X, x) \leftarrow (Y, y)$ are defined in the usual way, by transitivity. The *extended dependence graph* of the SARE, denoted Γ, is the graph of the dependence relation (see for example Fig. 5.3). It describes exactly the set of variable instances to be calculated in order to compute the SARE.

A *schedule* is a mapping $t : \mathcal{I} \to \mathbf{Z}$ such that

1. $(X, x) \leftarrow (Y, y) \Rightarrow t(X, x) > t(Y, y)$ (causality condition).

2. $t(X, x) \geq 0$, $\forall (X, x) \in \mathcal{I}$ (positivity condition).

In this definition, the causality condition guarantees that dependent computations are performed in a consistent order, whereas the positivity condition ensures that we can set a starting time for the computations.

Observe that a scheduling function may not exist. This is the case, for example, when the extended dependence graph contains a cycle. This motivates the *computability analysis* of recurrence equations, which is directly related to scheduling. A variable X is said to be *computable* – or, *explicitly defined* – if and only if for all $x \in D$, there exists a scheduling function t such that $t(X, x) < +\infty$.

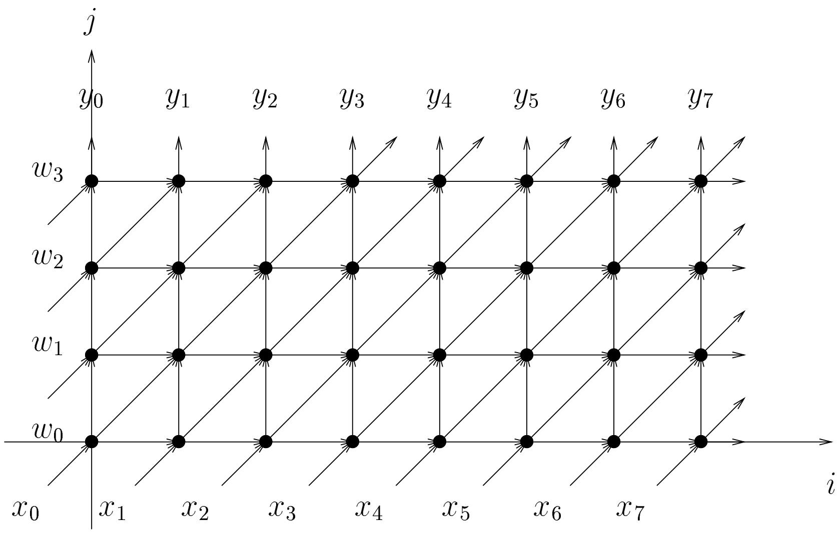

Figure 5.3 Extended dependence graph for the SURE of the convolution algorithm, Eqns. (23–26), for $n = 3$.

The following theorem (due to Karp, Miller and Winograd [3]), sets a framework for the study of scheduling, in the case of a single recurrence equation:

Theorem 1 *Given a uniform recurrence equation*

$$X(z) = f[X(z - d_1), \ldots, X(z - d_s)], \tag{27}$$

defined on the domain $D = \mathbf{N}^n$, the following conditions are equivalent:

1. *$X(z)$ is computable.*
2. *There exists no semi-positive vector[3] $(u_1, \ldots, u_s)$ such that*

$$\sum_{j=1}^{s} -u_j d_j \geq 0 \quad .$$

3. *If $\tau = (\tau_1, \ldots, \tau_n)$ is a vector of $\mathbf{Z}^n$, the system of inequalities $d_j.\tau \geq 1$, $1 \leq j \leq s$, and $\tau_i \geq 0$, $1 \leq i \leq n$ has a solution.*
4. *For all $z \in \mathbf{N}^n$, the following linear programs have an optimal common value $m(z)$:*

$$I \left\{ \begin{array}{l} z - \sum_{j=1}^{s} u_j d_j \geq 0, \\ u_j \geq 0, 1 \leq j \leq s, \\ \max \sum_{i=1}^{s} u_j \end{array} \right. \qquad II \left\{ \begin{array}{l} \tau_i \geq 0, 1 \leq i \leq n, \\ d_j.\tau \geq 1, 1 \leq j \leq s, \\ \min z.\tau \end{array} \right.$$

[3] A vector is said to be semi-positive if it is non null and all its components are non negative.

Condition *3* has a simple geometric interpretation: the extremities of vectors d_i, $1 \leq i \leq s$ must all be strictly on the same side of the hyperplane $\tau.z = 0$, as shown in Fig. 5.4.

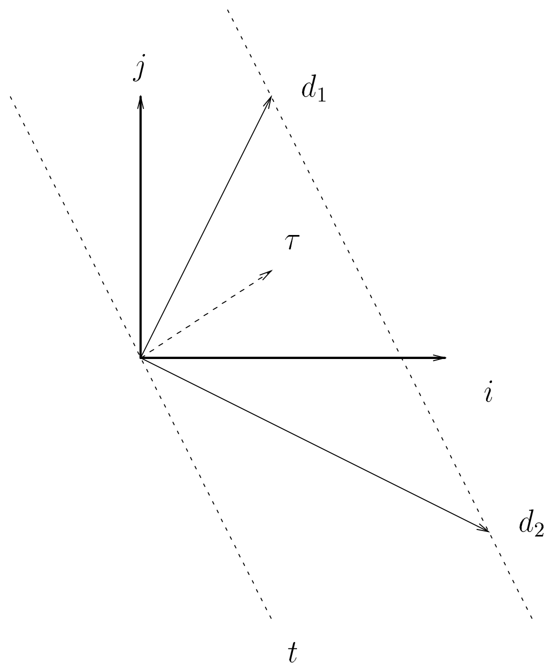

Figure 5.4 Illustration of theorem 1. Points are indexed by i and j, and dependence vector are d_1 and d_2. The vector τ is normal to the iso-temporal lines.

Moreover, the τ vector for which the solution of the linear program *II* is obtained defines a *linear schedule*. In other words, we can compute $X(z)$ at time $\tau.z$. Fig. 5.4 depicts a two dimensional case, where the dashed lines (often called *iso-temporal lines*) represent calculations done at successive instants t and $t + \tau.d_i$.

In practice, one is often interested in linear (or affine) schedules, as then, the velocity of the data is constant, and the implementation of the corresponding parallel program is easier.

Let us illustrate the application of this technique to the convolution example of Eqns. (23–26). As far as scheduling is concerned, we may consider the dependency information of this system as being equivalent to that of the following single equation.

$$\begin{pmatrix} Y(i,k) \\ W(i,k) \\ X(i,k) \end{pmatrix} = g(Y(i,j-1), W(i-1,j), X(i-1,j-1)) \tag{28}$$

Applying condition 3 of Theorem 1 leads to the following inequalities:

$$\begin{gathered} \tau_1 \geq 1 \\ \tau_2 \geq 1 \\ \tau_1 + \tau_2 \geq 1 \end{gathered}$$

A valid schedule is therefore $t(i,j) = i + j$.

Linear schedules

In practice, systems of recurrence equations are seldom SUREs, and moreover, there is no reason why equations are aligned in such a way that a common schedule exist, e.g., one may have to align two variables in opposite directions in order to be able to compute them. For these reasons, one has to associate different linear schedules to different variables.

Consider our standard SRE (Def. 5.2.1) parameterized by an integer p

$$X(z,p) = \begin{cases} & \vdots \\ D_i^X & : \quad g_i(\ldots Y(f(z,p))\ldots) \\ & \vdots \end{cases}$$

where $f(z) = Az + Bp + C$. To each edge e of the reduced dependence graph G describing the dependence between variables Y and X of this equation, let us associate the domain D_i^X, and the index function f. The edge e can therefore be represented by a 4-tuple (Y, X, D_i^X, f), where D_i^X is a sub-domain of D^X where the dependence effectively occurs.

Definition 5.2.5 (Affine by variable parameterized schedule) *An affine by variable parameterized schedule has the form*

$$t_X(z) = \tau_X z + \sigma_X p + \alpha_X \tag{29}$$

where τ_X and σ_X are fixed vectors with rational coordinates, and α_X is a rational number.

For the sake of simplicity, we will consider only rational timing functions. All the following results can be extended to the case where $t_X(z) = \lfloor \tau_X z + \sigma_X p + \alpha_X \rfloor$ without difficulty.

Def. 29 assumes that the schedule depends linearly on the parameter. The problem is to characterize the values of τ_X, σ_X and α_X such that the causality and positivity conditions are satisfied, and then to select one particular solution so as to optimize some quality criterion like maximal throughput or minimum latency.

In order to satisfy the causality condition for each edge (Y, X, D_i, f) of E_G one must satisfy the condition:

$$\forall z \in D_i^X : t_X(z) \geq t_Y(I(z)) + 1 \quad . \tag{30}$$

The above definition of timing-function maps directly to the structure of the equations: it is implicitly assumed that the evaluation of the system is to be performed on an architecture where the combinational operators are separated by at least one delay. However, this hypothesis excludes practical cases when one may wish to cascade several combinational elements. Changing the way delay will be mapped is not a problem provided the information is known statically. The real problem is to ensure that condition (30) is satisfied on and only on domain D_i^X.

When one substitutes appropriate values for z in (30), one obtains systems of linear inequalities in the coefficients of t_X and t_Y. This situation is similar with the positivity constraints, $t_X(z) \geq 0$. The problem is that there is a very large number (perhaps an infinity) of values of z and $I(z)$ which result in a large number of inequalities. We need to express all these inequalities using a finite description.

There are two basic methods for that. One uses a fundamental result of the theory of linear inequalities, the affine form of Farkas' lemma [4], and is called *the Farkas method*. The other one, called the *vertex method*, uses the dual representation of polyhedra, and is summarized as follows.

There are two ways of describing a polyhedron. One is as the set of points which satisfy a finite set of linear inequalities, and the other one is as the convex hull of a finite set of points. For each polyhedron, there exists a finite set of points, called its *vertices*, which together generate the polyhedron, none of them being a convex combination of the other. Such a set of points is a minimal generating system. If the polyhedron is unbounded, some of the vertices will be at infinity, they are called *rays* (a ray is an infinite direction in the polyhedron). There are standard algorithms for going from one representation to the other one [5].

The key idea behind the vertex method is the fact that a linear inequality is satisfied at *all* points of a convex polyhedron if and only if it is satisfied at all points of the generating system (which is finite). Hence the method: find generating systems for the polyhedra D_i^X, write (30) at each of these points, and solve for the unknown coefficients. The method is summarized as follows:

- Compute a generating system for the polyhedron D of any edge (Y, X, D, f) of the dependence graph, and for all polyhedra D^X, $X \in \mathcal{V}$.
- Write instances of (30) at all vertices of polyhedra D of each edge.
- Write instances of $t_X(z) \geq 0$ at all vertices and rays of D^X,
- solve the resulting finite system of linear inequalities by optimizing an appropriate criterion. For example, minimizing the total computation time for a system of equations can be achieved by writing the total time as a linear function of the parameters, and expressing that this function is greater than the schedule of all instances of all variables.

It should be noted that using such a method leads quickly to rather complex linear problems. For example, scheduling the convolution given by the SRE (23–26) using the vertex method leads to a linear program of 37 constraints and 15 variables. The schedule obtained is

$$t_Y(i, j, n) = j + 1$$
$$t_y(i, n) = n + 1$$

The results presented in this section are based on work in the eighties, and more recently in the early nineties, when the work was picked up by the loop parallelization community. Undoubtedly, the most important work on this subject is that of Karp, Miller and Winograd [3], which predates systolic arrays by well over a decade! They studied (in fact, defined) SUREs over a particular class of domains (the domain of all variables is the same and is the entire positive orthant), and gave techniques for determining (one- and multi-dimensional) affine schedules[4]. This work is only recently being matched and improved.

Quinton, Rao, Roychowdhury, Delosme-Ipsen [6, 7, 8, 9] and a number of others worked on schedules for UREs and SUREs. Rajopadhye et al. and Quinton-van Dongen [10, 11, 12] developed the basic results of the vertex method for AREs. Mauras et al. [13] extended this result to *systems* of AREs by using *variable dependent* (also called affine-by-variable) schedules. Rajopadhye et al. [14] proposed *piecewise* affine schedules for SAREs. Feautrier [15] gave the alternative formulation using Farkas' lemma, for determining (one-dimensional, variable dependent) affine schedules for a SARE. He further extended the method to multidimensional schedules [16].

As for the optimality of schedules, basic results (for SUREs) were of course obtained by Karp et al. [3] and later by Rao, Shang-Fortes, Darte et al. and Darte-Robert [7, 17, 18, 19], among others. For SAREs, the problem was addressed by Feautrier, Darte-Robert and Darte-Vivien [15, 16, 20, 21].

Finally, some theoretical results about the undecidability of scheduling are also available. Joinnault [22] showed that scheduling a SURE whose variables are defined over arbitrary domains is undecidable. Quinton-Saouter showed [23] that scheduling even parameterized families of SAREs (each of whose domains is bounded) is also undecidable.

5.2.4 Allocation Functions

The second important aspect in the choice of the space time transformation is the *allocation function*, namely, the function specifying the mapping of the computation, i.e., an index point z, in the domain, D of the SRE to a processor index. In this section we will discuss the nature and choice of the allocation function and how it can be combined with the schedule to obtain an appropriate change of basis transformation. We assume that we have a SURE (variables have been aligned, and non-local dependencies are uniformized, see Section 5.2.2), and a (1-dimensional, affine) schedule specified by the pair (τ, α) has been chosen. For a k-dimensional SRE, the allocation function is $a : \mathbf{Z}^k \rightarrow \mathbf{Z}^{k-1}$, i.e., it maps a k-dimensional index point to a $k-1$-dimensional processor. We consider linear (or affine) transformations, and hence the allocation function is represented by a $(k-1) \times k$ matrix and a $(k-1)$-vector.

Consider the allocation function $a(i,j) = j$ that we used for the convolution example in Section 5.2.2. Note that it is not unique — the function $a(i,j) = 1 - j$ gives us *the same architecture*, except that the processors are now labeled $-1 \ldots -n$. Thus, although the allocation function is specified by an $(k-1) \times k$ matrix and a $k-1$-vector (i.e., by $k^2 - 1$ integers), they are not really independent. In fact, the allocation function is *completely* specified by means of what is called the *projection vector*, $\vec{u}$ (by convention, $\vec{u}$ is a *reduced* vector: the gcd of its components is 1). The intuition is that any two points in D are mapped to the same processor if and only if their difference is a scalar multiple of $\vec{u}$. Since $\vec{u}$ is reduced, we also see that for any $z \in D$, the *next* point mapped to the same processor is one of $z \pm \vec{u}$.

[4] Today we know them to be multi-dimensional schedules, but this understanding was slow in coming.

Now, the *time* between two such successive points mapped to the same processor is $\tau^T\vec{u}$. This is true for all processors: they are all active exactly once every $\tau^T\vec{u}$ clock cycles, and $\frac{1}{\tau^T\vec{u}}$ is called the *efficiency* of the array.

An important constraint that must be satisfied is that no two index points are mapped to the same processor at the same time instant, which holds if and only if $\tau^T\vec{u} \neq 0$. Before we discuss the factors influencing the choice of $\vec{u}$, we will summarize how a change of basis transformation, T is constructed, given τ and $\vec{u}$, such that $\tau^T\vec{u} \neq 0$. We need to construct an $n \times n$ matrix (this may or may not be unimodular, and in the latter case, the domains of the SREs we obtain on applying this change of basis will be what are called "sparse polyhedra") such that its first row is τ^T, and whose remaining $n-1$ rows constitute a matrix π such that $\pi\vec{u} = 0$. This can be done by first *completing* $\vec{u}$ into a unimodular matrix S (say by using the right Hermite normal form [4]), so that $\vec{u} = S\vec{e_1}$ (here, $\vec{e_i}$ is the i-th unit vector). Then the matrix consisting of the last $n-1$ rows of S^{-1} is a valid choice for π, and hence $\begin{bmatrix} \tau^T \\ \pi \end{bmatrix}$ is the required transformation, and it is easy to see that it is non-singular, and that its determinant is $\tau^T\vec{u}$.

Choosing the Allocation Function

Unlike the schedule vector, τ, which must belong to the a polyhedral region specified by a set of linear inequalities, there are no such linear constraints that we can impose on the projection vector $\vec{u}$. Indeed, the space of permissible values of $\vec{u}$ is the entire set of primitive vectors in $\mathbf{Z}^n$, except *only* for the normals to τ. In general, we seek the allocation function that yields the "best" array, and the natural choice for the cost function is the number of processors — the number of integer points in the projection of D. Unfortunately, this is not a linear function. Moreover, it is only an approximate measure of the number of processors — in arrays with efficiency $\frac{1}{k}$, one can always cluster k processors together so that the final array is fully efficient, at the cost of some control overhead. Thus, the number of processors can be systematically reduced (albeit at the cost of additional control and temporary registers). In practice however, a number of other constraints can be brought into play, rendering the space manageable, and indeed, it is often possible to use an exhaustive search.

First of all, for many real time signal processing applications, the domain of computation is infinitely large in one direction, and immediately, since we desire a finite sized array, (and we insist on linear transformations) the projection *has* to be this direction.

A second constraint comes from the fact that the I/O to the array must be restricted to only boundary processors, otherwise we lose many of the advantages of regularity and locality.

In addition, we could also impose that $\tau^T\vec{u} = \pm 1$, which would ensure arrays with 100% efficiency. As mentioned above, arrays that do not achieve this can be "post-processed" by clustering together $\tau^T\vec{u}$ neighboring processors, but this increases the complexity of the control. Moreover, it is also difficult to describe such arrays using only the formalism of linear transformations (the floor and ceiling operations involved in such transformations are inherently non-linear).

Finally, we could impose the constraint that the interconnections in the derived array must be of a fixed type (such as 4, 6 or 8 nearest neighbors, or we may even permit "one-hop" interconnections). It turns out that this constraint is surprisingly effective. When deriving 1-dimensional arrays (from 2-dimensional recurrence equations), if we allow only nearest neighbor interconnections, there are no more than 4 possible arrays that one can derive. Similarly there are no more than 9 2-dimensional arrays (with only north, south east and west connections), the number goes up to 13 if one set of diagonal interconnections is allowed, and to 25 if 8 nearest neighbors are allowed). Moreover, these allocation functions can be efficiently and systematically generated.

5.2.5 Localization and Serialization

We now address one of the classic problems of systolic design, namely localization (and its dual, problem serialization). The key idea is to render uniform all the dependencies of a given SRE. Clearly, this is closely related to the problem of alignment (i.e., defining the SRE such that all variables have the same number of dimensions). For example, consider the SRE defined below (we have deliberately chosen distinct index names for each variable)

$$X(i,j) \quad = \quad \{0 \leq i,j < n\} \ : \ Y(j-1,i-1) \tag{31}$$

$$Y(p,q) \quad = \quad \{0 \leq p,q < n\} \ : \ A_{p,q} \tag{32}$$

At first glance, the $X \to Y$ dependency is not uniform. However, this can be made uniform if the variable Y (and the input A) is aligned so that the p dimension is aligned with j and q with i, i.e., we perform a change of basis on Y with transformation, $\mathcal{T}(p,q) = (q,p)$, and then *rename* the indices (p,q) to (i,j). Alignment does not help with *self dependencies*, and in the rest of this section, we will consider only such dependencies.

Consider a (self) dependency, $X \to X(f(z))$ in an ARE, thus $f(z) = Az+a$ (for simplicity, we ignore the parameters). Applying a change of basis transformation $\mathcal{T}$ will yield a new dependency $\mathcal{T} \circ f \circ \mathcal{T}^{-1}$. This will be uniform, i.e., its linear part will be Id if and only if $A = \mathrm{Id}$, i.e., the original dependency is itself uniform. Hence, change of basis transformations cannot serve to localize a dependency, and we need to develop other transformations.

The fundamental such transformation is (null space) *pipelining*. Recall the convolution example and its localization in Sec. 5.2.2. We could view the transformation as consisting of two steps, the first consists of defining a new variable, say X' whose domain is also D (the same as that of X), and whose value at any point z is that of $X(f(z))$. then we can simply replace the $X(f(z))$ on the rhs of the equation for X by $X'(z)$ and get a semantically equivalent SRE. Let us focus on the equation for X', which does not do any "computation" but will be only used for propagating the values appropriately. Now, if we find a constant vector ρ such that $\forall z \in D$, $X'(z) = X'(z+\rho)$, and another constant vector ρ' such that, adding "enough" scalar multiples of ρ to any point $z \in D$ eventually yields a point z' such that $f(z) = z' + \rho'$, then we can replace the affine dependency f by a uniform dependency ρ. This can be proved by a simple inductive argument that we omit. Hence, our problem can be reduced to the following:

- Determining the pipelining vector(s) ρ.

- Resolving the problem of "initializing" the pipelining, i.e., determining the vector(s) ρ' and also *where* it is to be used.

The first problem is resolved by observing that the pipelining vectors must "connect up" all the points that depend on the *same* value, in other words, two points z and z' could be (potentially) connected by a dependency $\rho = z - z'$ if $f(z) = f(z')$, i.e., $A(z - z') = 0$, i.e., $z - z'$ must belong to the null space A. Indeed, any *basis vector* of the null space of A is a valid candidate for the pipelining vector.

Now, to initialize the pipelines, once we have chosen the pipelining vectors ρ we need to ensure that for all points $z \in D$, there exists a scalar, k (which could be a function of z), such that $z - k\rho$ is a *constant* distance, ρ' from $f(z)$. In other words, $A(z + k\rho) + a - z + k\rho = \rho'$. Multiplying both sides by A and simplifying shows that this is possible if and only if $A^2 = A$, i.e., A is a *projection*. For illustration, consider the two recurrences below.

$$
\begin{aligned}
X(i,j) &= \{0 < i,j < n\} \ : \ X(i,0) && (33)\\
Y(i,j) &= \{0 < i,j < n\} \ : \ Y(0,i) && (34)
\end{aligned}
$$

In the first one, we see that all the index points in the i-th column require the same argument. The dependency functions are, respectively, $A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ and $A_2 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$. For both of them, the null space of A is 1-dimensional, and spanned by $\pm[0,1]$; this is thus the pipelining vector. However, while $A_1^2 = A_1$, we see that $A_2^2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \neq A_2$. Hence the first recurrence can be pipelined (using null space pipelining as outlined above), while the second one cannot (see Fig. 5.5). For Eqn. 33, we obtain the following, provably equivalent, SRE.

$$
\begin{aligned}
X(i,j) &= \{0 < i,j < n\} \ : \ P_X(i,i) && (35)\\
P_X(i,j) &= \begin{cases} \{0 < i,j < n\} & : \ P_X(i,j-1) \\ \{0 < i < n; j = 0\} & : \ X(i,j) \end{cases} && (36)
\end{aligned}
$$

It was derived by

- defining a new equation for a pipelining variable, P_X, whose domain is that of the (sub) expression where the dependency occurs, *plus* a "boundary" $\{j = 0\}$, for the initialization. P_X depends on itself with the uniform dependency ρ everywhere, except for the boundary, where it depends on X with the dependency ρ'.
- replacing, in the original equation for X, the $X(i,0)$ by $P_X(i,j)$. Thus all affine dependencies have been replaced by uniform ones.

The natural question to ask is what happens when null space pipelining is not possible. In this case, a number of solutions exist. First, we could try *multi-stage* pipelining, where we initialize a pipelining variable, not with the original variable, but with *another* pipelining variable. Consider the following example.

$$
X(i,j) = \{0 < i,j < n\} \ : \ X(0,i) + X(0,j) \qquad (37)
$$

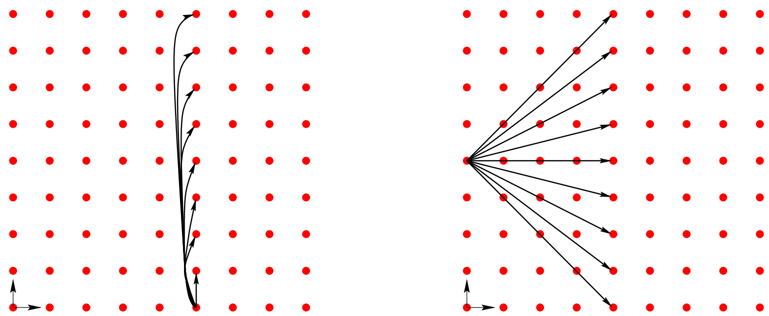

Figure 5.5 Illustration of the conditions for null space pipelining. The diagrams show the dependencies of Eqns. 33 (left) and 34 (right). In both cases, the pipelining vector is $\pm[0,1]$, since all the points in a given column depend on the *same* point. For Eqn. 33, these *producer* points are "close" to the *consumers* (because the dependency is idempotent), and the pipeline can be correctly initialized, while for Eqn. 34, this is not the case, so null space pipelining is not possible.

Observe that the first dependency is the same as that in Eqn. 34, for which, null space pipelining is impossible. The second dependency, however, is similar to the one in Eqn. 33 (the pipelining is along the row rather than along the columns) as illustrated in Fig. 5.6, and can be pipelined by introducing a local variable P_2 defined as follows.

$$P_2(i,j) \quad = \quad \begin{cases} \{0 < i, j < n\} & : \; P_2(i-1,j) \\ \{0 < j < n; i = 0\} & : \; X(i,j) \end{cases} \tag{38}$$

But now, we may also pipeline the second dependency by going "piggyback" on this pipeline. In particular, we note that the (potential) pipeline for the first dependency *intersects* the second pipeline on the $i = j$ line. Hence, we can initialize the first pipeline at this "boundary" but, the variable used for initialization is P_2, and not X. On the other side of this line, we pipeline in the opposite directions, as follows.

$$P_1(i,j) \quad = \quad \begin{cases} \{0 < i < j < n\} & : \; P_1(i,j-1) \\ \{0 < j < i < n\} & : \; P_1(i,j+1) \\ \{0 < i = j < n\} & : \; P_2(i,j) \end{cases} \tag{39}$$

All that remains is to replace the two dependencies in Eqn. 37 by $P_1(i,j)$ and $P_2(i,j)$, and we have a SURE.

The next question concerns what to do when such multi-stage pipelining is not possible. At this point, we may still try to pipeline, but by sacrificing the primary advantage of null space pipeline — data values are propagated *only through* those points that use them. If we relax this constraint, we could localize Eqn. 34 by defining *two* variables, P_Y^1 and P_Y^2. The first one corresponds to propagation along the pipelining vector $[0,1]$, and is initialized (at the $j = 0$ boundary), but with the P_Y^2 variable. This is similar to multi-stage pipelining, except that we do not have the P_Y^2 variable pipelined "for free", but we must actually construct its pipeline

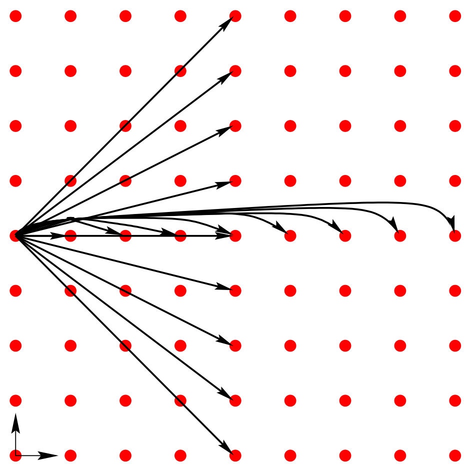

Figure 5.6 Illustration of multi-stage pipelining. The first dependency $[i,j] \rightarrow [0,j]$ can be pipelined (propagation vector $[1,0]$), but the second one $[i,j] \rightarrow [0,i]$ cannot (the propagation vector is $\pm[0,1]$, but the pipelined cannot be initialized). In multistage pipelining, the second pipeline is initialized with the *first* one at the *intersection* of the two pipelines.

(also called its routing). The following is (one of many possible) solutions.

$$P_Y^1(i,j) = \begin{cases} \{0 < i,j < n\} & : P_Y^1(i,j-1) \\ \{0 < i < n; j = 0\} & : P_Y^2(i,j) \end{cases} \tag{40}$$

$$P_Y^2(i,j) = \begin{cases} \{0 < i,j < n\} & : P_Y^2(i-1,j+1) \\ \{0 < j < n; i = 0\} & : Y(i,j) \end{cases} \tag{41}$$

What these two equations achieve can be explained as follows. The variable P_Y^2 is used to propagate the values of Y from the first column in the south-east direction (with the $[-1,1]$ dependency vector.) This propagation passes through half the index points in the domain. Once the values reach the bottom row, they are used to initialize the standard pipeline (of P_Y^1) which propagates them up the column to the index points that will use these values. There are many choices for the propagation/routing vector. Indeed, it suffices to chose it so that it does not belong to the *range space* of A. Thus, any SAREs where the dependencies are singular can always be rendered uniform. However, the choice of the propagation/routing vectors can affect the schedule (optimality and/or even the existence of a schedule).

The final question concerns how to treat non-singular dependencies. One simple, but naive approach is to extend the dimension of the index space, and in this extended index space, one can always make the dependency non-singular. However, a much more interesting and elegant solution is possible in certain cases. If the non-singular dependency matrix A is such that, for some integer k, $A^k = I$, then we can "cut" the domain into k pieces by means of $k-1$ hyperplanes, and then "superpose" each of these pieces so that the dependency become uniform. To

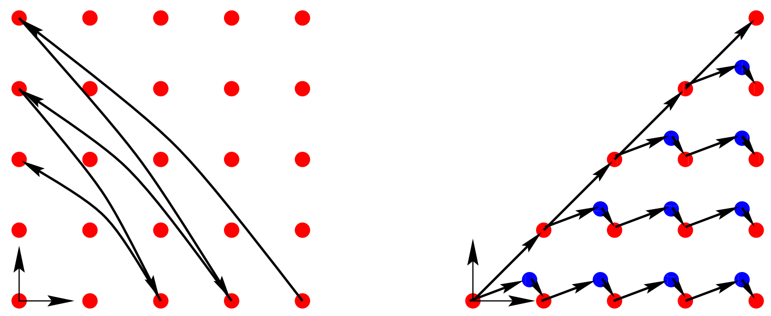

Figure 5.7 Illustration of localization by folding.

illustrate, consider the RE below.

$$X(i,j) = \begin{cases} 0 \le i = j < n & : X(i-1, j-1) \\ 0 \le j < i < n & : X(j,i) \\ 0 \le i < j < n & : X(j-1, i) \quad . \end{cases} \tag{42}$$

Here, the dependency corresponds to the transpose, i.e., $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and since $A^2 = I$, there must be a line (namely the diagonal, $i = j$) such that folding with respect to it will render the dependencies uniform. Indeed, we obtain the following SRE (see Fig. 5.7).

$$X(i,j) = \begin{cases} \{i = j\} : & \Rightarrow \; X(i-1, j-1) \\ \{i > j\} : & \Rightarrow \; Y(i,j) \end{cases} \tag{43}$$

$$Y(i,j) = X(i-1, j) \tag{44}$$

The results in this section are based on the work in the late eighties, notably by Rajopadhye et al. Quinton-Van Dongen, Wong-Delosme, Roychowdhury, and Yaacoby-Cappello [10, 24, 12, 25, 9, 26]. It is also interesting to note that similar ideas underlie much of the recent work on communication optimization in parallelizing loop nests and in data distribution.

5.3 Advanced systolic architectures

In the second part of this chapter, we describe through several examples, specific techniques used in advanced systolic arrays. In the methodology described in the first part, we have viewed systolic arrays as a dedicated implementation of a single, given algorithm. In practice, the design is (should be) *application-specific*, and not *algorithm-specific*, and this introduces a number of problems. A "naive" or abstract systolic array, such as one derived by our transformations, can be efficiently implemented in many different ways, depending on the granularity of the elementary calculations, and/or the versatility of the architecture which is targeted. In particular, the target technology may be one of the following:

- general purpose programmable parallel architectures,

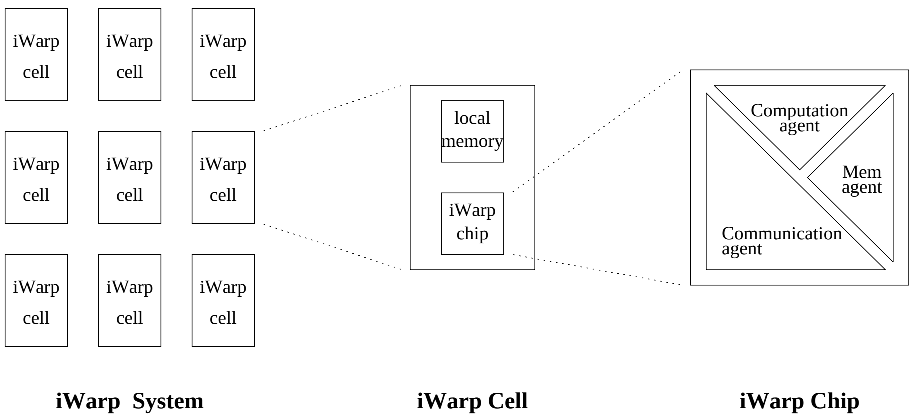

Figure 5.8 An iWARP system is composed of identical iWARP cells. A cell contains an iWARP chip with a local memory. A chip is divided into three sub-systems: a computation agent, a memory agent and a communication agent.

- application oriented programmable architectures,
- reconfigurable FPGA based architectures,
- dedicated (custom) architectures.

5.3.1 General-purpose systolic programmable architectures

General-purpose systolic programmable architectures, are designed to support a wide range of systolic and parallel algorithms. Their main characteristic is that processors are only connected to their neighbors, and only systolic communications are allowed. Most often, these architectures have a special instruction set for communication. There is no general interconnection network: distant communications are implemented by the routers of the intervening processors. The communication/computation ratio is in general close to 1, and such architectures are ideal to implement fine grain parallel algorithms (indeed, it is this feature that would distinguish them from general purpose parallel machines based on a mesh topology such as the Intel Paragon, the Cray T3D, T3E, etc.)

The most famous example of such an architecture is the iWARP [27], a circuit designed by Intel Corporation in 1988 in cooperation with Carnegie Mellon University (CMU). Its architecture is derived from the original WARP project [28] started in 1984 at CMU. The iWARP chip was designed as a building block for developing powerful and programmable parallel systems for high speed signal, image and scientific computing. An iWARP system is simply an assembly of iWARP cells (iWARP chip and memory) connected by means of a dedicated high throughput communication channel. Fig. 5.8 illustrates the iWARP system concept.

The iWARP chip consists of three autonomous sub-systems. The *computation agent* executes programs and contains floating point units with a peak performance of 20 and 10 MFLOPS for single or double precision operations respectively, as well as an integer ALU that performs 20 millions integer or logical operations per second. The *communication agent* implements the iWARP's communication system and operates independently of the computation agent. Finally, the *memory agent* provides an interface to the local memory.

The iWARP processor has many of the more traditional architectural features found in distributed memory machine, such as support for non-neighbor communication, message routing hardware, word-level-flow control between neighboring cells and spooling (a DMA-like mechanism). But what makes the iWARP architecture innovative and very well suited to systolic computations, are the two following additional features: program access to communication and logical channels.

In order to implement systolic communications, an application program can directly access the input and output of the message queues located in the communication agent. These queues are bound to special registers (the *gates*) in the register space of the instruction set. Reading from a gate corresponds to receiving a data from the queue. Similarly, writing to the gate corresponds to sending a data to the queue. Typically, this mechanism is used to smooth the flow of data between cells and to delay one stream of data with respect to the other ones. Such direct access to the message queues is also a way of optimizing the communication protocol. The data can be transmitted using an application-specific protocol which directly analyzes the header of the messages and explicitly tells the communication system where to deposit the messages. This protocol management can be directly used by parallel program generators such as Apply [29], AL [30, 31] or C-stolic [32].

The iWARP's second interesting communication innovation is the implementation of logical channels between processors. Logical channels provide both a higher degree of connectivity than physical links, and a mechanism for delivering guaranteed communication bandwidth for certain classes of messages. A high degree of connectivity is useful for systolic communication: a cell may need to have simultaneous links with several cells. Without logical channels, algorithms that require more physical connections than those provided by hardware cannot be implemented. The iWARP logical channels overcome this problem by proposing multiple *logical* connections on the same physical support. The ability to deliver guaranteed communication bandwidth makes the implementation of systolic algorithms, often characterized by their high throughput and word-level transfer, very efficient. By reserving a channel exclusively to dedicated communications — no other message can pass through this channel — one may guarantee a minimum bandwidth in order to ensure effective low-cost fine-grain communication.

Above all, an iWARP system is a MIMD machine, and no specific language is fundamentally attached to this machine. The W2 language, previously designed for the Warp machine, can be used, as well as many other parallel languages. But imperative languages, such as C, used in conjunction with a communication routine library remain a very simple and effective approach. In that case, the system runs in SPMD mode: each cell executes the same program independently, and synchronizations are ensured through systolic communications.

Other general-purpose systolic architectures are MGAP [33, 34], and Saxpy-1M [35], although they could be justifiably be classified as application oriented programmable. Johnson et al. [36] present an excellent survey of general purpose systolic arrays.

5.3.2 Application oriented programmable architectures

Intermediate between fully programmable architectures and dedicated systolic arrays are *application oriented programmable architectures.* Such architectures target a particular domain of applications, such as image processing, cryptography, or neural network simulation. However, the great variety of algorithms pertaining to the domain calls for a programmable approach.

The SYSTOLA 1024 architecture illustrates this concept. SYSTOLA 1024 ([37, 38]) is a 32×32 systolic programmable architecture oriented towards image processing applications. It is an add-on board for personal computers with PCI slots. SYSTOLA 1024 is based on the concept of Instruction Systolic Array (ISA).

The structure of SYSTOLA 1024 is shown in Fig. 5.9. It consists of a square array of processors, each processor being controlled by instructions, row and column selectors. The whole array is controlled by a global clock. Instructions are input in the upper left corner, move rightwards and downwards step by step so that all processors of a diagonal of the array execute the same instruction. Selectors move in the same way as instructions, and are used to disable processors selectively: a processor executes the instruction it receives only if both selector bits are 1.

Each processor has read and write access to its own memory. One communication register is accessible by four neighbors, thus allowing neighbor to neighbor communications. In addition, broadcast of information on a column of processors is allowed.

A board contains 4×4 chips. Each chip is a 8×8 array of bit-serial RISC processors. Input and output to the array are done by a cascaded memory. Intelligent memory units, called interface processors are at the North and the West of the array. They access on-board RAM (northern board RAM, or western board RAM), which can communicate with the PC through a PCI interface.

The data transfer between the units of this memory hierarchy is supervised by a controller which either receives instructions from the PC, or operates autonomously. In the second case, it receives controller instruction from an instruction queue on the board, loaded from the PC before the application is started. The ISA gets its instruction from the ISA program memory, also loaded in advance.

The sequence of activities of the parallel computer is determined by so-called configurations, that is to say, controller programs and ISA programs. A programming environment called `ISATOOLS` allows the application programmer to develop parallel programs for this architecture, using an adaptation of the Pascal language.

Each processor is capable of 3 Mips (16 bit instructions), which leads to a peak performance of 3125 Mips for the whole array. A typical application performs between 1000 and 3000 Mips. The array can reach 142 Mflops on 64 bit IEEE floating point operations, and 237 Mflops on 32 bit numbers. Typical acceleration rates of 60 are reached in numerical applications.

Although SYSTOLA 1024 was targeted for image processing, it can be used to accelerate a much broader class of algorithms: solution of large linear equation systems (for finite element methods), pattern matching, graph algorithms, neural networks, computer graphics, image reconstruction for computer tomography, speech recognition, and microbiology.

Other architectures in this class of application oriented programmable architectures include: (i) Adaptive Solutions' CNAPS [39], designed for accelerating

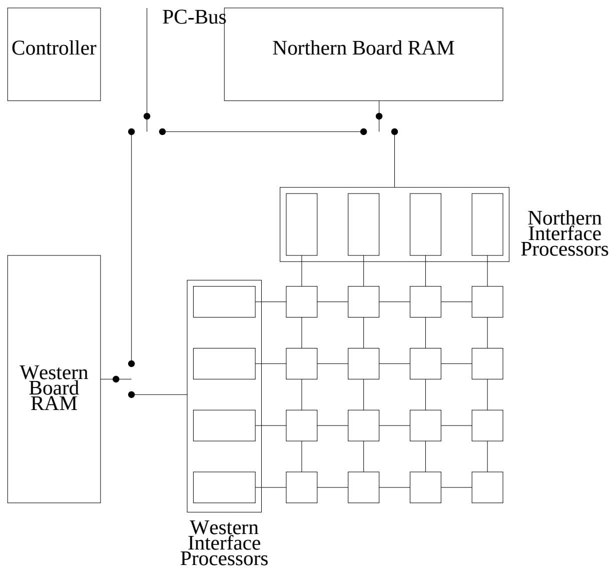

Figure 5.9 Architecture of the Systola 1024 architecture

neural networks; (ii) Synapse-1, designed by Siemens [40], another systolic architecture for neural networks; (iii) Kestrel [41], a programmable systolic architecture dedicated to DNA processing; (iv) MicMacs [42, 43] a programmable systolic array for string matching algorithms; (v) MOVIE [44] and Sympathy2 [45], designed for image processing applications.

5.3.3 Reconfigurable architectures

Another popular approach for the implementation of systolic arrays is to use reconfigurable architectures, using FPGA chips. Examples of such architectures are PeRLe-1 [46], Splash-2 [47] or Space-2 [48]. Although these systems have not specially been designed to support systolic computations, their architecture make them particularly suitable for this purpose: they are made of a regular arrangement of FPGA chips on which systolic arrays/algorithms can be efficiently implemented.

Example of systolic arrays designed using FPGA boards are:

- systolic filter for fast DNA bank scanning [49],
- 2D convolution [50],
- text searching [51],
- numerical algorithms [52], division and other arithmetic operations [53].

Fitting a systolic array on such architectures requires the following problems to be solved:

- Good partitioning strategies: the boards have limited resources as compared to the size of the systolic array one wishes to implement.

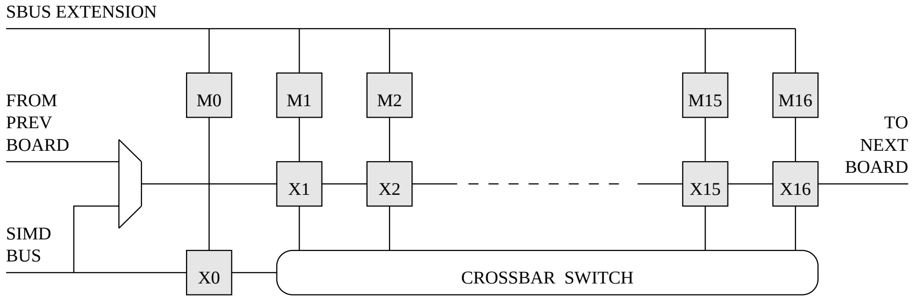

Figure 5.10 SPLASH-2 array board architecture: up to 16 such boards can be connected together to form a larger array. One board contains 17 Xilinx XC4010 FPGA chip connected linearly and through a crossbar switch.

- Efficient synthesis software: the challenge is to be able to rapidly map systolic algorithms to the reconfigurable platform. To this end, high clock speed and quick synthesis may be more important to space optimized designs. New methods need to be invented, as since current, design methodology mimics ASIC design, which is not appropriate.

We illustrate this class using the SPLASH-2 architecture, which is an attached processor system. It has been designed to connect up to 16 array boards to a SPARCstation-2 through a specific interface board. Each array board contains 17 Xilinx XC4010 FPGA chips. 16 of them form a linear array, and the 17th provides a broadcast capability to the other 16 chips. These 17 FPGA chips can access 512 Kbytes of memory (see Fig. 5.10). In addition, a reconfigurable crossbar switch links all the FPGA chips together. At run time, five crossbar configurations can be simultaneously stored and activated instantaneously.

The primary models of computation that were intended to be supported by the SPLASH-2 architecture were a SIMD and a linear (not necessarily systolic) architecture. Viewed as a SIMD machine, instructions can be broadcasted through the X0 chip to the other FPGA components. Viewed as a linear array, data can circulate through the linear data-path of one board, then reach the first FPGA chip of the next board, and so on. The presence of the crossbar switch allows the programmer to get beyond the rigid structure of a linear data-path.

The SPLASH-2 machine, as well as the PAM[5] family boards (PeRLe-0 and PeRLe-1) are the first well-known FPGA structures on which many applications have been tested. Applications of image processing, scientific calculation, data compression and encryption, and molecular biology have been experimented. These experiments have shown that supercomputer performance can be achieved, especially on computations which exhibit high regularity. In most cases, the time-consuming part of these applications can be implemented on systolic architectures.

This is not really surprising since the FPGA and the systolic concepts share some basic features: they are both composed of identical and locally connected cells. Systolic arrays can then be considered as adapted structures for FPGA arrays, even if the cell granularity differs. A systolic processor is made up of several elementary

[5] Programmable Active Memory.

FPGA cells, and thus the same pattern of FPGA cells can be replicated inside a component, and the same configuration applied to every FPGA chip of the array. This last remark has a major consequence on the design process: it shortens the place-and-route stage which appears as the main limitation of the FPGA technology. In the case of a systolic architecture, the place-and-route operation of a single processor can be performed rapidly as only a few FPGA cells have to be configured, and the result is replicated immediately across the array. In practice, defining a placement is sufficient, since experience has shown that a good placement leads to a good and fast routing.

Recently, a number of researchers have explored incorporating an FPGA or similar programmable hardware on general purpose programmable processors. Although they are not systolic architectures per se, these architectures use regularity and locality within the reconfigurable part. For example, RaPiD (Reconfigurable Pipelined DataPath) [54], is a configurable computing solution for regular computation-intensive applications. It fills the gap between static systolic arrays, which are too inflexible, and programmable systolic arrays like the WARP, which more closely resemble multiprocessors with too much control overhead.

5.3.4 Special-purpose (custom) architectures

Implementing a systolic algorithm on a special-purpose architecture is surely the best way to reach high performance. This comes however at the price of a longer design time. We illustrate this technique by summarizing the design of the SAMBA machine [55].

SAMBA is a full-custom parallel hardware accelerator dedicated to the comparison of biological sequences such as the scan of DNA or protein data-bases. It implements a parameterized version of a well-known biological sequence comparison algorithm — the Smith and Waterman algorithm — which is based on dynamic programming. This algorithm is known to provide good results but not often used when large data sets such as genetic data banks are involved. Other, heuristic algorithms are generally preferred for their speed, even though they are less reliable. The primary goal of SAMBA was to develop a systolic architecture for implementing the full Smith and Waterman algorithm in a very efficient way.

The second objective was to design the machine in the most systematic way as possible in order to **rapidly** get a **correct** hardware. A few decisions — that were never challenged later — were made at the beginning of the project. For example, no attempts to improve the basic comparison algorithm was made. In addition, it was decided that the architecture would be designed using available CAD tools without any attempt to adapt these tools to the particular problem. The general architecture of SAMBA and the global design methodology were thus fixed and remained unchanged during the full duration of the project.

Thus, it was soon decided that the basic architecture of SAMBA would be a linear systolic array of hardwired full-custom VLSI chips connected to a standard workstation through a reconfigurable interface. This linear structure was imposed by the length of the biological sequences. The hardwired full-custom solution was dictated by performance. Finally, the reconfigurable interface was found to be the easiest way to fill the gap between a programmable von Neuman machine (the workstation) and a fully dedicated hardware (the array). Furthermore, FPGA boards already existed and could be efficiently used for this purpose. For SAMBA, the PeRLe-1 FPGA board was used for housing the workstation/array interface.

The main part of the SAMBA project was then the design of the systolic array and, more precisely, of the full-custom VLSI chip. The design methodology that was followed can be summarized as follows:

Functional specification: the specification of the algorithm, including its parallel version was written using the C-stolic language [56] (an extension of C allowing parallel systolic algorithms to be simulated on Unix workstations). The C-stolic compiler produces three outputs: a C program emulating one processor of the systolic array, a C program emulating the interface between the host and the systolic array, and a C program for the host workstation.

Functional simulation: the functional specification was simulated on test data obtained from various data sets.

Architectural specification: the program run by the processors of the systolic array was translated manually from C-stolic code into VHDL code. Only the subset of VHDL accepted by the COMPASS VLSI synthesis software was used, in order to be able to obtain a hardware specification.

Validation of the architecture: the architectural specification was run on the same data as the functional simulation, and validated by comparison.

Synthesis of the VLSI chip: the VLSI chip was synthesized using the COMPASS hardware synthesis software, and sent to fabrication. The chips were fabricated by the ES2 company through the French multi-chip project.

Board design: the systolic array was designed by assembling the chips in printed boards.

Interface design: Due to the flexibility of the PeRLe-1 board, the interface was designed in two steps. First, the PeRLe-1 board was configured as a very simple link between the host and the systolic array, in order to test the full functionality of the chips and of the architecture. In this configuration, the interface program produced by the C-STOLIC compiler was run by the host. Then, in order to reach the full speed of the systolic array, the PeRLe-1 board was configured to implement the final interface.

The total design effort amounted to 11 person/months: 1 month for functional specification and simulation, 1 month for architectural specification and simulation, 2 months for architectural synthesis, 2 months for chip testing, 0.5 month for board design, 4 months for interface design, and 0.5 month for final integration. The chip, as well as the complete machine, was correct at the first fabrication. It must be pointed out that the most difficult part (and the longest one) was the design of the interface.

Like SAMBA, a number of custom architectures have been proposed and built for a number of applications, and it is impossible to enumerate them all. Image compression is an interesting area where systolic algorithms can be applied. For example, Baglietto et al. [57] present a chip for block-matching. The MA7170

Systolic correlator [58] is a single chip architecture designed by InMos in the sixties. Demassieux [59] describes the implementation of a Median filter on VLSI.

To conclude our discussion of custom systolic architectures, we may list a few rules[6], taken from the SAMBA experience, which can help designing such architectures rapidly and correctly.

- Never work simultaneously on algorithm and architecture: choosing an existing stable application is the best way to converge rapidly to an appropriate architecture.
- Do not design simultaneously tools and architecture: CAD tools are complex and need extensive verification before being operational for real designs.
- Make sure that the anticipated speed-up beats the current general purpose technology by *at least* one order of magnitude: designing a special-purpose architecture takes time. The architecture must not become obsolete due to the technology progress (doubling every 18 months or less) during the time it takes for the design.
- Design a balanced architecture: remember that the speed of a computer system is dictated by its slowest element! Within a complete systolic system, a high speed array is inefficient if it cannot be fed with input data.
- Get the help of end-user: this is probably the most important rule. The success of a special-purpose machine depends essentially of the quality of the service it provides. And only the future users are able to specify exactly what they expect the machine to do.

5.4 Conclusion

In this chapter, we have presented two aspects of advanced systolic design, namely advanced design methods for systolic design and also the design and implementation of advanced systolic arrays.

The theory summarized in the first part is more or less the basis of several transformation systems and tools for regular array synthesis. Without any pretension to be exhaustive, let us cite a few of these systems. A number of classic systems such as SDEF, VACS, etc., were developed as research prototypes at various universities. Among the earliest software based on such transformations was Diastol [60]. This program accepted a few recurrence equations, and produced the sketch of a systolic architectures using the space-time transformation technique. More recent attempts such as the MMAlpha system [61, 62] are based on applicative languages, and correspond closely to the theory that we have presented. The Cathedral IV software [63] is an attempt to produce regular designs from algorithms specified in Silage (Silage is a single assignment language used as the starting point for synthesis of signal processing architectures. A commercial version of this language, called DFL, is used by Mentor Graphics.) Presage [64] was a prototype tool written in Prolog, also meant for time-space transformations. One characteristics

[6] Most of these rules may also hold for designing other (non systolic) custom architectures.

of this software was its ability to explore in a systematic way the solution space, using the backtracking mechanism of Prolog. Another attempt in this direction was presented by Baltus [65], where the solution space was explored using branch and bound techniques. HiFi [66] is developed in Delft, and covers a large spectrum of synthesis techniques, including clustering and partitioning.

One important limitation of the theoretical model is its inability to model partitioning (the problem of designing systolic arrays under resource constraints). There has been considerable work on the *choice* of the partitioning (to satisfy certain resource constraints and to optimize certain cost functions). However, these results are still not satisfactorily incorporated into the framework of design by correctness-preserving transformations of (very) high level specifications in the form of SREs. The key difficulty is that after partitioning, the resulting domains are no longer polyhedral and the dependencies are no longer affine.

There is a wide range of target technologies for implementing systolic architectures. In the future, there will be many more choices and, given the growing complexity of designs, it is important to be able to learn from the lessons of previous designs. For this reason we have described a number of implementation examples in the second part.

The next generations of chips will integrate hundred of millions of transistors. How will microprocessors and other VLSI implementations of DSP and multimedia systems use these resources? One possibility, which is now moving beyond the research prototype to commercial availability, is processors with a reconfigurable area on which custom (co) processing will take place. The success of this concept will mainly depend on two points: the efficiency of the co-processor, and the programming ease and availability of tools.

The efficiency will be provided by the parallelization of some parts of the programs and, as already mentioned, systolic arrays are largely used for this purpose. Experiments carried on the SPLASH-2 and PAM projects are good illustrations of this idea. They have shown that systolic structures fit well onto FPGA components and that they deliver high performance.

On the other hand, programming a reconfigurable architecture is still delicate. It requires knowledge of both hardware, computer architecture and digital design. For example, the translation of a few lines of imperative code into its equivalent in hardware is not obvious for usual programmers. Undoubtedly the challenge is then to make this translation as automatic as possible, even if the efficiency (from a resource allocation point of view) cannot reach that of a hand-crafted solution. The main problem to solve for reconfigurable co-processors to be successful is therefore to design compilers which map sections of code to FPGA in a reasonable amount of time without sacrificing efficiency to much. The long research on automatic synthesis of VLSI systolic arrays can find a new field of application in compilation and optimization for reconfigurable co-processors.

Acknowledgment The authors gratefully acknowledge their colleagues in Irisa and elsewhere who have contributed to the development of the concepts presented in this chapter.

REFERENCES

[1] Lyon, R.F., "Two's complement pipeline multipliers," *IEEE Transactions on*

Computers, vol. COM-24, pp. 418–425, April 1976.

[2] L. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," in *Proc. Conference on Very Large Scale Integration: Architecture, Design and Fabrication*, pp. 509–525, January 1979.

[3] R. Karp, R. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the Association for Computing Machinery*, vol. 14, no. 3, pp. 563–590, July 1967.

[4] A. Schrijver, *Theory of Integer and Linear Programming.* John Wiley and Sons, 1988.

[5] H. Le Verge, "A note on Chernikova's algorithm," RR 635, IRISA, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, Feb 1992.

[6] P. Quinton, "Automatic synthesis of systolic arrays from recurrent uniform equations," in *11th Annual International Symposium on Computer Architecture, Ann Arbor*, pp. 208–214, June 1984.

[7] S. Rao, *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Standford University, U.S.A., Oct. 1985.

[8] J. Delosme and I. Ipsen, "Systolic array synthesis : computability and time cones," in *Parallel Algorithms and Architectures*, pp. 295–312, North-Holland, 1986.

[9] V. P. Roychowdhury, *Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms.* PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA, December 1988.

[10] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto, "On synthesizing systolic arrays from recurrence equations with linear dependencies," in *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, (New Delhi, India), pp. 488–503, Springer Verlag, LNCS 241, December 1986.

[11] S. V. Rajopadhye and R. M. Fujimoto, "Synthesizing systolic arrays from recurrence equations," *Parallel Computing*, vol. 14, pp. 163–189, June 1990.

[12] P. Quinton and V. Van Dongen, "The mapping of linear recurrence equations on regular arrays," *Journal of VLSI Signal Processing*, vol. 1, no. 2, pp. 95–113, 1989.

[13] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter, "Scheduling affine parameterized recurrences by means of variable dependent timing functions," in *Application Specific Array Processors* (S. Kung, E. Schwartzlander, J. Fortes, and K. Przytula, eds.), (Princeton University), pp. 100–110, IEEE Computer Society Press, Sept. 1990.

[14] S. V. Rajopadhye, L. Mui, and S. Kiaei, "Piecewise linear schedules for recurrence equations," in *VLSI Signal Processing, V* (K. Yao, R. Jain, W. Przytula, and J. Rabbaey, eds.), (Napa), pp. 375–384, IEEE Signal Processing Society, Oct 1992.

[15] P. Feautrier, "Some efficient solutions to the affine scheduling problem. I. one-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, Oct. 1992.

[16] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part II. multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec. 1992.

[17] W. Shang and A. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies," *IEEE Trans. on Computers*, vol. 40, no. 6, pp. 723–742, June 1991.

[18] A. Darte, L. Khachiyan, and Y. Robert, "Linear scheduling is nearly optimal," *Parallel Processing Letters*, vol. 1, no. 2, pp. 73–82, December 1991.

[19] A. Darte and Y. Robert, "Constructive methods for scheduling uniform loop nests," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 814–822, Aug 1994.

[20] A. Darte and Y. Robert, "Affine-by statement scheduling of uniform and affine loop nests over parametric domains," *Journal of Parallel and Distributed Computing*, vol. 29, no. 1, pp. 43–59, February 1995.

[21] A. Darte and F. Vivien, "Revisiting the decomposition of karp, miller and winograd," in *International Conference on Application–Specific Array Processors*, (Strasbourg, France), pp. 12–25, IEEE, July 1995.

[22] P. Gachet and B. Joinnault, *Conception d'algorithmes et d'architectures systoliques.* PhD thesis, Université de Rennes I, September 1987. (In French).

[23] Y. Saouter and P. Quinton, "Computability of recurrence equations," *Theoretical Computer Science*, vol. 114, 1993.

[24] S. V. Rajopadhye, "Synthesizing systolic arrays with control signals from recurrence equations," *Distributed Computing*, vol. 3, pp. 88–105, May 1989.

[25] F. C. Wong and J.-M. Delosme, "Broadcast removal in systolic algorithms," in *International Conference on Systolic Arrays*, (San Diego, CA), pp. 403–412, May 1988.

[26] Y. Yaacoby and P. R. Cappello, "Converting affine recurrence equations to quasi-uniform recurrence equations," in *AWOC 1988: Third International Workshop on Parallel Computation and VLSI Theory*, Springer Verlag, June 1988. See also, UCSB Technical Report TRCS87-18, February 1988.

[27] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankind, P. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An Integrated Solution to High-Speed Parallel Computing," in *Proceedings of Supercomputing '88*, pp. 330–339, IEEE Computer Society and ACM SIGARCH, nov 1988.

[28] M. Annaratone et al., "The Warp computer: architecture, implementation, and performance," *IEEE Transactions on Computers*, vol. C-36(12), pp. 1523–38, 1987.

[29] B. Baxter and B. Greer, "Apply: A parallel compiler on iWarp for image-processing applications,"

[30] P. S. Tseng, *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, 1988.

[31] H. Printz, H. Kung, T. Mummert, and P. Scherer, "Automatic mapping of large signal processing systems to a parallel machine," in *Proc. of SPIE, Real-Time Signal Processing XII*, (San Diego CA (USA)), Aug. 1989.

[32] D. Lavenier, B. Pottier, F. Raimbault, and S. Rubini, "Fine grain parallelism on a MIMD machine using FPGAs," in *IEEE Workshop on FPGAs for Custom Computing Machines*, (Napa Valley), pp. 2–8, Apr. 1993.

[33] M. Borah, R. Bajwa, S. Hannenhalli, and M. Irwin, "A SIMD Solution to the Sequence Comparison Problem on the MGAP," in *ASAP 94, the International Conference on Application Specific Array Processors*, pp. 336–345, IEEE Computer Society Press, 1994.

[34] R. Bajwa, R. Owens, and M. Irwin, "The MGAP's Programming Environment and the *C++ Language," in *ASAP'95* (P. Cappello, C. Mongenet, G.-R. Perrin, P. Quinton, and Y. Robert, eds.), pp. 121–124, IEEE Computer Society Press, July 1995.

[35] D. Foulser and R. Schreiber, "The Saxpy Matrix-1: A general purpose systolic computer," *Computer*, vol. 20, no. 7, pp. 35–43, July 1987.

[36] K. Johnson, A. Hurson, and B. Shirazi, "General-Purpose Systolic Arrays," *Computer*, pp. 20–31, Nov. 1993.

[37] ISATEC Soft - & Hardware GmbH, "The ISATEC Systola 1024 Parallel Computer." Technical Note, 1997. http://www.netzservice.de/Home/isatec.

[38] B. Schmidt, M. Schimmler, and H. Schröder, "Morphological Hough Transform on the Instruction Systolic Array," in *Euro-Par'97*, vol. LNCS 1300, pp. 788–806, Springer Verlag, 1997.

[39] D. Hammerstrom, "A VLSI architecture for high-performance, low-cost, on-chip learning," in *International Joint Conference on Neural Networks*, vol. 2, pp. 537–544, 1990.

[40] U. Ramacher et al., "Synapse-1: A high-speed general purpose parallel neurocomputer system," in *IPPS95*, pp. 774–773, IEEE Computer Society Press, Apr. 1995.

[41] J. Hirschberg, R. Hughey, and K. Karplus, "KESTREL: A Programmable Array for Sequence Analysis," in *ASAP'96* (I. C. S. Press, ed.), (Chicago, Illinois), pp. 25–34, Aug. 1996.

[42] P. Frison, D. Lavenier, H. Leverge, and P. Quinton, "A VLSI Programmable Systolic Architecture," in *International Conference on Systolic Array*, (Killarney, Irland), IEEE Computer Society Press, Jun 1989.

[43] P. Frison and D. Lavenier, "Experience in the Design of Parallel Processor Arrays," in *International Workshop on Algorithms and Parallel VLSI Architectures II* (P. Quinton and Y. Robert, eds.), (Bonas - France), pp. 233–242, Elsevier Science Publishers B.V., Jun 1991.

[44] R. Barzic and C. Bouville and F. Charot and G. Le Fol and P. Lemonnier and C. Wagner, "MOVIE: a Building Block for the Desing of Real Time Simulator of Moving Pictures Compression Algorithms," in *ASAP'95* (P. Cappello, C. Mongenet, G.-R. Perrin, P. Quinton, and Y. Robert, eds.), pp. 193–203, IEEE Computer Society Press, July 1995.

[45] D. Juvin, J. Basille, H. Essafi, and J. Latil, "Sympathi2 : a 1.5D Processor array for image applications," in *EUSIPCO Signal Processing IV : theories and application*, pp. 311–314, North Holland, 1988.

[46] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI Systemes*, vol. 4, no. 1, Mar. 1996.

[47] D. Buell, J. Arnold, and J. Walter, *Splash 2: FPGAs in a custom computing machine*. IEEE computer society press, 1996.

[48] B. Gunther, "SPACE 2 as a reconfigurable Stream Processor," Tech. Rep. CIS-97-008, School of Computer and Information Science, University of South Australia, 1997.

[49] P. Guerdoux-Jamet and D. Lavenier, "Systolic Filter for Fast DNA Similarity Search," in *ASAP'95*, (Strasbourg), July 1995.

[50] J. Peterson and P. Athanas, "High-Speed 2-D Convolution with a custom Computing Machine," *Journal of VLSI Signal Processing*, vol. 12, pp. 7–19, 1996.

[51] D. Pryor, M. Thistle, and N. Shirazi, "Text Searching on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines*, (Napa, California), 1993.

[52] T. Jebelean, "Implementing GCD Systolic Arrays on FPGA," in *International Workshop on Field-Programmable Logic and Applications*, vol. LNCS 849, (Prague, Czech Republic), 1994.

[53] J. Andersen, A. Nielsen, and O. Olsen, "A Systolic On-Line Non restoring Division Scheme," in *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, 1994.

[54] C. Ebeling, D. Cronquists, and P. Franlin, "Configurable computing: The catalyst for high performance architectures," in *IEEE International Conference on Application Specific, Systems, Architectures, and Processors*, (Zurich, Switzerland), July 1997.

[55] P. Guerdoux–Jamet, D. Lavenier, C. Wagner, and P. Quinton, "Design and Implementation of a Parallel Architecture for Biological Sequence Comparison," in *Euro-Par'96* (L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, eds.), no. 1123 in LNCS, (Lyon, France), pp. 11–24, Springer, Aug. 1996.

[56] F. Raimbault and D. Lavenier, "ReLaCS for Systolic Programming," in *ASAP'93*, Oct. 1993.

[57] P. Baglietto, M. Maresca, A. Migliaro, and M. Migliardi, "Parallel Implementation of the Full Search Block Matching Algorithm for Motion Estimation," in *ASAP'95* (P. Cappello, C. Mongenet, G.-R. Perrin, P. Quinton, and Y. Robert, eds.), pp. 182–192, IEEE Computer Society Press, July 1995.

[58] B. Christie, "Working demonstration of the MA7170 systolic correlator," in *International Conference on Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), (University of Oxford, UK), pp. 113–122, Adam Hilger, July 2-4 1986.

[59] N. Demassieux, F. Jutand, M. Saint-Paul, and M. Dana, "VLSI Architecture for a One Chip Video Median Filter," in *ICASSP 85*, pp. 1001–1004, 1985.

[60] P. Frison, P. Gachet, and P. Quinton, "Designing Systolic Arrays with DIASTOL," in *VLSI Signal Processing II* (S.-Y. Kung, R. E. Owen, and J. G. Nash, eds.), pp. 93–105, IEEE Press, Nov. 1986.

[61] C. Mauras, "Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones." Thèse de l'Université de Rennes 1, IFSIC, Dec. 1989.

[62] J. Rosseel and M. van Swaaij and F. Catthoor and H. De Man and H. Le Verge and P. Quinton, *Regular Array Synthesis for Image and Video Applications*, ch. 6, pp. 119–142. Kluwer Academic Publisher, 1993.

[63] M. VanSwaaij, "Automating high level control flow transformations for dsp memory management," in *IEEE workshop on VLSI Signal Processing* (IEEE, ed.), 1992.

[64] V. Van Dongen, "PRESAGE, a Tool for the Design of Low-Cost Systolic Circuits," in *IEEE International Symposium on Circuits and Systems*, (Espoo, Finland), June 1988.

[65] J. A. D. Baltus, "Efficient exploration of nonuniform space-time transformations for optimal systolic array synthesis," in *ASAP'93* (L. Dadda and B. Wah, eds.), (Venice), IEEE Computer Society Press, Oct. 1993.

[66] P. Held, P. Dewilde, E. Deprettere, and P. Wielage, *HiFi: from Parallel Algorithm to Fixed-Size VLSI Processor Array*, ch. 2, pp. 71–94. Kluwer Academic Publisher, 1993.