Synthesizing Systolic Arrays with Control Signals from Recurrence Equations

Sanjay Rajopadhye

October 13, 1988
CIS-TR-86-12A*

Department of Computer and Information Science
University of Oregon

*This report is a revision of CIS-TR-86-12.

Synthesizing Systolic Arrays with Control Signals from Recurrence Equations

Sanjay V. Rajopadhye*
Computer Science Department
University of Oregon
Eugene, Or 97403

Abstract

We present a technique for synthesizing systolic arrays which have non-uniform data flow governed by control signals. The starting point for the synthesis is an *Affine Recurrence Equation*—a generalization of the simple recurrences encountered in mathematics. A large class of programs, including most (single and multiple) nested-loop programs can be described by such recurrences. In this paper we extend our earlier work [RFc] in two principal directions. Firstly, we present a class of transformations called *data pipelining* and show that they yield recurrences that have *linear conditional expressions* governing the computation. Secondly, we discuss the synthesis of systolic arrays that have non-uniform data flow governed by control signals. We show how to derive the control signals in such arrays by applying similar pipelining transformations to these *linear conditional expressions*. The approach is illustrated by deriving the Guibas-Kung-Thompson architecture for computing the cost of optimal string parenthesization.

1 Introduction

Systolic arrays are a class of parallel architectures consisting of regular interconnections of a very large number of simple processors, each one operating on a small part of the problem. They are typically designed to be used as back-end, special-purpose devices for computation-intensive processing. A number of such architectures have been proposed for solving problems such as matrix multiplication, L-U decomposition of matrices, solving a set of equations, convolution, dynamic programming, etc. (see [GKT,Kuna,Kunb] for an extensive bibliography).

*Supported by a University of Utah Graduate Research Fellowship, and NSF grant No. MIP-8802454

Most of the early systolic arrays were designed in an *ad hoc*, case-by-case manner. Recently there has been a great deal of effort on developing unifying theories for automatically synthesizing such arrays [CS,C,DIa,LM,LS,LW,MW,Mol,Qui,RFS,WD]. The approach is to analyze the program dependency graph and transform it to one that represents a systolic array. The problem of synthesis can thus be viewed as a special case of the *graph-mapping* problem where the objective is to transform a given graph to an equivalent one that satisfies certain constraints. For systolic array synthesis there are two major constraints, namely *nearest-neighbor communication* and *constant-delay interconnections.*

The initial specification for the synthesis effort is usually expressed as a *recurrence equation*—an equation which recursively defines the value of a function at all points in a domain, in terms of its value at other points in the domain. The domain is defined separately and is usually a convex hull in Z^n (Z denotes the set of integers). In most of the earlier work cited above, the recurrence was restricted to a special case, called *Uniform Recurrence Equations* (UREs). Here, the *difference*, $p - q$, between a point, p, and any point, q, that it depends on, is required to be a *constant vector*, b. For such recurrences, the dependency graph of the computation can be shown to be a *lattice* in Z^n. The problem of synthesizing a systolic array can then be solved by determining an appropriate *affine* transformation (i.e., one that can be expressed as a translation, rotation and scaling) of the original lattice. Such recurrences were defined in a seminal paper by Karp et al. [KMW] even before systolic arrays became popular, and much of the recent research is based on their foundations.

One principal reason that such an approach has been so successful is that the semantics of all systolic arrays can be formally described by recurrences that have *uniform* dependencies. This fact has been independently observed by a number of researchers [Cheb,MR,RK], notably those addressing the problem of systolic array *verification*. This implies that as long as the computation defined by a URE is well formed, there is a very direct mapping of the recurrence to a systolic array. We have therefore argued in a companion paper [RFc] (see also [RFc]) that forcing a designer to provide an *initial* specification that has uniform dependencies is too restrictive. We proposed a new class where the dependencies of a point p are *affine* functions of p. The recurrence equations characterizing such computations are called Affine Recurrence Equations (AREs*). We have also shown that simple systolic architectures (i.e., those that have uniform data flow) can be synthesized from AREs by a two-step process consisting of determining an *affine transformation* (i.e., a timing function and an allocation function) and a transformation called *data pipelining*. The first step is very similar to the techniques used for UREs. However, it merely yields a target architecture—one

*In some of our earlier work these recurrences are called Recurrence Equations with Linear Dependencies (RELDs), but the name ARE, first introduced by Delosme and Ipsen [DIb], is more appropriate.

whose data flow is neither spatially nor temporarily local. The second step, namely *data pipelining*, permits the dependencies to be localized, so that the architecture is systolic. A similar approach (called *broadcast removal*) has been described by Fortes and Moldovan [FM] although they do not describe how a *timing function* (which is called a *linear schedule* there) can be derived.

In this paper we present two main results. First, we develop a complete characterization of *data pipelining*, and present a taxonomy for it. We will show that the most general class of pipelining may be viewed as a *source-to-source* transformation that converts the ARE into a recurrence that has uniform dependencies at all points except those at certain boundaries. We call such recurrences Conditional Uniform Recurrence Equations (CUREs), because they have the property that the dependencies are *uniform* but the computation is governed by *conditional expressions*. Data pipelining is thus a constructive method for obtaining the CURE from the ARE. Our second result is a systematic method for deriving systolic arrays that have non-uniform data flow, governed by control signals. In such arrays, the data flow may change direction and/or speed within the array, and certain processors may perform specialized computations. The change in speed/direction or the specialized computation may either occur at all times, or at certain instants determined by control signals, and may be restricted to certain processors (such as those on a specific boundary), or may involve all processors in the array. We describe how such systolic arrays are synthesized from the CUREs obtained by pipelining the dependencies of the original ARE.

The rest of this paper is organized as follows. In the following section we formally define the various classes of recurrences that we shall use, and develop some notation. We also summarize how systolic arrays are synthesized from UREs, and briefly describe the main idea underlying *data pipelining*. Then, in Section 3 we present a taxonomy for the pipelining operations, and show how this yields a systematic procedure to derive a CURE from the original ARE. The taxonomy includes a generalized class of pipelining called *multistage pipelining*, where a number of dependencies may be involved in a single pipeline. Section 4 describes how to synthesize systolic arrays (with non-uniform data flow) from CUREs. To achieve this, we introduce a technique called *control pipelining* which is similar to *data pipelining* except that it is applied to the *conditional expressions* of the CURE. This yields a set of control dependencies which correspond to the control signals that govern the computation. The rest of the synthesis procedure is similar to that for UREs, except that there are additional constraints on the choice of timing and allocation functions, corresponding to the requirement that control signals must also have systolic interconnections. We illustrate the technique (in Section 5) by systematically deriving a well known systolic array for computing the cost of optimal string parenthesization (the so called "dynamic programming" array [GKT]). Finally we conclude by comparing our results with the approaches of other researchers.

2 Recurrence Equations: UREs, AREs and CUREs

We begin by introducing some notation. The following definitions are based on those by Karp et al. [KMW].

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

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

where $p \in D$;
$q_i \in D$ for $i = 1 \ldots k$;
and g is a single-valued function, strictly dependent on each of its arguments.

A *system* of recurrence equations is a set of m such equations, defining the functions $f_1, f_2, \ldots f_m$. In any equation defining say f_i, any of the f's (i.e., not restricted to f_i itself) may occur on the right hand side.

The domain D, of the recurrence is a subset of Z^n, and for the remainder of this paper we restrict our attention to the case where D is a convex hull. It should be immediately clear why such recurrences are an attractive starting point for synthesizing any class of *regular* architectures. The computation of the function f at any point in the domain involves applying the function g to exactly k arguments. The function g thus represents an atomic computation that is repreatedly evaluated with a number of different arguments, and thus precisely defines the functionality of the processor. The relationship between p and the q_i's determines where the k arguments are to come from. Based on this we have the following classes of recurrences.

Definition 2 A Recurrence Equation of the form defined above is called a *Uniform Recurrence Equation* iff $q_i = p + \mathrm{b_i}$, for $i = 1, \ldots, k$, where the $\mathrm{b_i}$'s are constant n-dimensional vectors.

Definition 3 A Recurrence Equation is said to have Affine Dependencies (called an *Affine Recurrence Equation*) iff $q_i = \mathrm{A_i}p + \mathrm{b_i}$, for $i = 1, \ldots, k$,
where $\mathrm{A_i}$'s are constant $n \times n$ matrices;
and $\mathrm{b_i}$'s are constant $n \times 1$ vectors;

Example 1: [due to Quinton] The following system of UREs, defined over a domain $D = \{[i,j] \mid 0 \leq i, 0 \leq j < k\}$ computes a stream of numbers,* $Y_0, Y_1, \ldots, Y_i, \ldots$ which is the result of

*We use upper case (and subscripts) to indicate input (and output) values, and lowercase for values associated with points in the domain.

convolving a stream, $X_0, X_1, \ldots, X_i, \ldots$ with a sequence of weights, $W_0, W_1, \ldots, W_{k-1}$.

$$\begin{aligned} y([i,j]^{\mathrm{T}}) &= y([i,j]^{\mathrm{T}}) + w([i,j]^{\mathrm{T}}) * x([i-1,j-1]^{\mathrm{T}}) \\ w([i,j]^{\mathrm{T}}) &= w([i-1,j]^{\mathrm{T}}) \\ x([i,j]^{\mathrm{T}}) &= x([i-1,j-1]^{\mathrm{T}}) \end{aligned} \tag{1}$$

Y_i will be computed at $[i,k]^{\mathrm{T}}$, provided the following boundary conditions are satisfied:

$$\begin{aligned} y([i,0]^{\mathrm{T}}) &= 0 \\ w([0,j]^{\mathrm{T}}) &= W_j \\ x([i,0]^{\mathrm{T}}) &= X_i \end{aligned}$$

Note that $[i,j]^{\mathrm{T}}$ is a point in the domain, and y, w and x are (correctly) defined as functions that take a (two-dimensional) vector as arguments, as defined above. There is however, no loss of generality and some notational convenience if we view them as functions that take two integer arguments, i and j, and we shall do so in the remainder of the paper. Also, observe that the above *system* of recurrences becomes a single URE, if we view the value computed at any point $[i,j]$ as a *tuple* of three terms—y, w and x. We may write this single URE as follows:

$$\begin{aligned} f(i,j) &= [y(i,j), w(i,j), x(i,j)] \\ &= [(\Pi_1 f(i,j-1) + \Pi_2 f(i-1,j) * \Pi_3 f(i-1,j-1)), \\ &\qquad \Pi_2 f(i-1,j),\ \ \Pi_3 f(i-1,j-1)] \end{aligned}$$

Here Π_i is the tuple-projection function that returns the i-th component of a tuple, and the boundary conditions are the same as above. ∎

Example 2: Using the method of dynamic programming, the cost $C_{i,j}$ of optimally parenthesizing the i through j elements of a string ($i < j$) may be defined recursively as follows:

$$C_{i,j} = \min_{i<k<j} (C_{i,k} + C_{k,j}) + h(i,j)$$

where $h(i,j)$ is the cost of the outermost parentheses; for strings of just two elements, which do not have any substrings, $C_{i,i+1}$, is $h(i,i+1)$. Hence the cost of parenthesizing a string of length n is $C_{1,n}$. If we replace the n-ary "min" operator by a binary min function and *iterate* over the index k, we have $C_{i,j} = f(i,j,1)$ where $f(i,j,k)$ is defined by the following ARE.

$$f(i,j,k) = \min \begin{pmatrix} f(i,j,k+1) \\ f(i,i+k,1) + f(i+k,j,1) \end{pmatrix} \tag{2}$$

The boundary conditions are $f(i,i+1,1) = h_{i,j}$, $f(i,j,j-i) = \infty$, and

$$f(i,j,1) = h_{i,j} + \min \begin{pmatrix} f(i,j,2) \\ f(i,i+1,1) + f(i+1,j,1) \end{pmatrix}$$

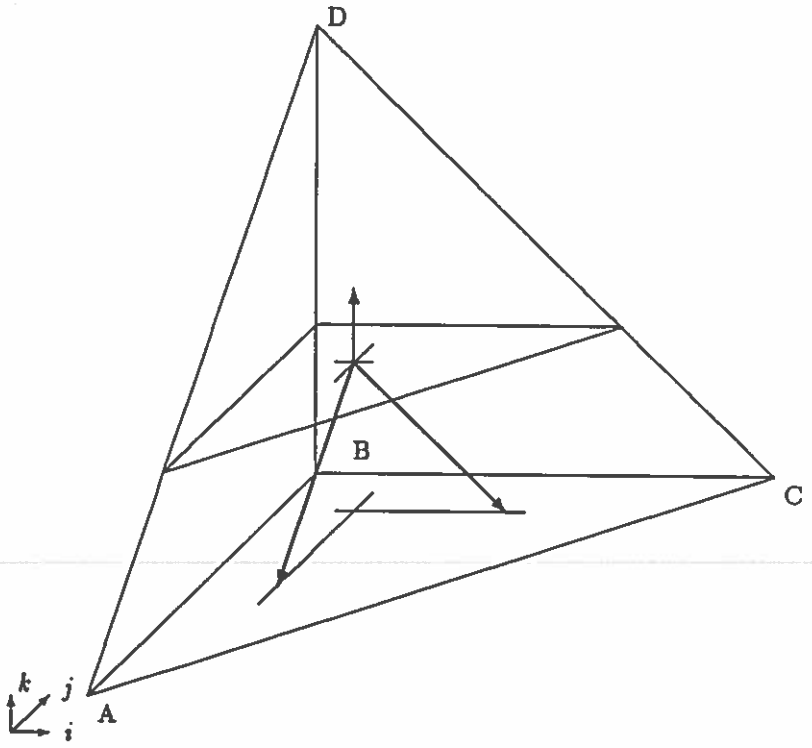


Figure 1: *Domain and Dependency Structure for the String Parenthesization ARE of Eqn 2*

The domain of the ARE is the tetrahedron ABCD, determined by the vertices $[1, 2, 1]$, $[1, n, 1]$, $[n-1, n, 1]$ and $[1, n, n-1]$ (see Fig. 1). Alternatively, it may be specified by the linear inequalities $1 \leq i \leq n-1$, $2 \leq j \leq n$ and $k \leq j - i$. Also note that at the $k = 1$ boundary, the computation is not simply a value available from the external world, but a new value computed at some point *inside* the domain. The dependencies of the ARE are as follows:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} b_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}; A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} b_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}; A_3 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} b_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \blacksquare$$

Definition 4 A *Conditional Uniform Recurrence Equation* over a domain D is defined to be an equation of the form:

$$f(p) = \begin{cases} g_1(f(q_{1,1}), f(q_{1,2}) \ldots f(q_{1,m_1})) & \text{if } \psi_1(p) \\ g_2(f(q_{2,1}), f(q_{2,2}) \ldots f(q_{2,m_2})) & \text{if } \psi_2(p) \\ \vdots & \\ g_k(f(q_{k,1}), f(q_{k,2}) \ldots f(q_{k,m_k})) & \text{if } \psi_k(p) \\ g_0(f(q_{0,1}), f(q_{0,2}) \ldots f(q_{0,m_0})) & \text{otherwise} \end{cases}$$

where $p \in D$;

each $q_{i,j}$ is of the form $p + \mathrm{b_{i,j}}$, where $\mathrm{b_{i,j}}$'s are constant n-dimensional vectors (the set of all the dependency vectors $\mathrm{b_{i,j}}$ is denoted by W);

each $\psi_i(p)$ is an *affine guard expression* given by the conjunction of a finite number of expressions of the form $\pi^{\mathrm{T}} p \geq \theta$, (where π^{T}'s are constant vectors in Z^n and θ's are constants in Z);

and g_i's are single valued functions which are strictly dependent on each of their arguments.

We assume that the computation is determinate, and that the guards are evaluated sequentially. Since the conditional expressions $[\pi_i^{\mathrm{T}} p = \theta_i^{\mathrm{T}}]_{i=1\ldots k}$ define a set of hyperplanes, they *implicitly* define the domain of the CURE.

2.1 Synthesizing Systolic Arrays from Uniform Recurrences—An Outline

As mentioned earlier, most of the earlier work on systolic array synthesis has concentrated on UREs. Since the dependency structure of the entire computation is completely specified by a small set of constant dependency vectors, it is only necessary to analyze these vectors in order to synthesize a systolic array. If we assume that the computation of g takes unit time once its arguments are available (a reasonable assumption since g is a *strict* function), then the problem of synthesizing a systolic array can be solved by mapping the original URE to a *space-time* domain by affine transformations. Such a mapping must satisfy the following constraints.

- The data dependencies of the original algorithm must be rendered spatially and temporally local since systolic arrays have nearest neighbor interconnections (spatial locality) and a finite memory in each processor.*

*Note that the finite memory mandates temporal locality, since the value used by any processor must have been produced by its neighbor only a finite number of "clock-ticks" ago.

- These transformed dependencies must be *uniform* over the whole space-time domain, since the processors in a systolic array are *identical* and have similar interconnections independent of their physical location in the array. The importance of this requirement has been demonstrated elsewhere [Raj].
- The mapping must be bijective i.e., two distinct points in the index-space should be mapped to two distinct points in the space-time domain. If it were not so, the computations from two distinct points in the problem domain would be scheduled on the same processor at the same time, giving rise to a *conflict.*
- The time component must preserve the dependencies of the original index-space, i.e., in order to schedule the computation at any point, all its arguments must first be evaluated.
- The *space* component of the transformed dependency vectors must correspond to nearest neighbor interconnections.

The time component of the mapping is called a *timing function*, and the space-component is the *allocation function.* It has been shown that first two constraints mentioned above can both be satisfied if we use *affine projections* as our mapping functions. Then the timing function has the form

$$t(p) = \lambda_t^{\mathrm{T}} p + \alpha_t$$

and is fully characterized by a vector λ_t^{T} and a scalar constant α_t. The allocation function (which maps every point p to an $(n-1)$-dimensional processor space) is similarly defined by

$$a(p) = \lambda_a p + \alpha_a$$

where λ_a is an $(n-1) \times n$ matrix and α_a is an $n-1$ vector. The complete mapping is thus defined by

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ t \end{bmatrix} = \lambda p + \alpha \equiv \begin{bmatrix} \lambda_a \\ \lambda_t^{\mathrm{T}} \end{bmatrix} p + \begin{bmatrix} \alpha_a \\ \alpha_t \end{bmatrix} \tag{3}$$

Then the third constraint is satisfied if λ is non-singular; the causality constraint is satisfied if for each dependency $\mathrm{b_i}$, $\lambda_t^T \mathrm{b_i} < 0$ and $\forall p \in D$, $\lambda_t^T p + \alpha_t > 0$; and the final constraint is satisfied by ensuring that for each dependency $\mathrm{b_i}$, $\lambda_a \mathrm{b_i} \in P$, where P is the set of permissible interconnections (i.e., $P = \{[0,0],[\pm 1,0],[0,\pm 1],[\pm 1,\pm 1]\}$ for the standard case). In general there is no unique solution to the above set of constraints, and a family of arrays are synthesized by finding solutions to the integer programming problem outlined above. It is also not critical which part of the problem is solved first—determining a timing function, or finding an allocation function.

2.2 Synthesizing Systolic Arrays from Affine Recurrences—Data Pipelining

When synthesizing systolic arrays from AREs we follow a similar approach. However, we have shown [RFc] that merely using affine transformations alone is not enough. It is necessary to explicitly pipeline the dependencies of the ARE. The key idea underlying data pipelining may be summarized as follows. If more than one point in the domain depends on some other point, one of them can *use* the value for its computation, and then *pass it on* to the other(s). Let p and p' be two points in D such that $\mathrm{A_i}p + \mathrm{b_i} = \mathrm{A_i}p' + \mathrm{b_i} = q$. Thus, computation of both $f(p)$ and $f(p')$ need $f(q)$ as their i-th arguments. Now, if we let $p - p' = v$, we may introduce a new function f' defined by $f'(p) = f'(p+v)$. It is simple to implement f' on any processor—it is simply an identity function that outputs the value it receives. However, by using f' we may transform the definition of f to be

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

where the i-th (affine) dependency $[\mathrm{A_i}, \mathrm{b_i}]$ has been replaced by a uniform dependency v. For the above scenario to work correctly, this transformation should be applicable throughout the entire domain. Also, the point that receives the value first, *must* be computed *before* the other point, thus imposing a *partial order* on these points. We shall discuss these ideas formally as follows.

Definition 5 The i-th *co-set* $C(p)$ of a point p in D is defined as the set of points in D, all of which depend (as their i-th dependency) on the same point as p, i.e.,

$$C(p) = \{q \mid \mathrm{A_i}q + \mathrm{b_i} = \mathrm{A_i}p + \mathrm{b_i}\}$$

Analogously, the i-th *inverse co-set* of a point p is defined to be the set of points in D that depend on p, i.e.,

$$C^{-1}(p) = \{q \mid \mathrm{A_i}q + \mathrm{b_i} = p\}$$

Lemma 1 *A point p' belongs to the i-th co-set of p, iff it is separated from p by a vector in the null space of* $\mathrm{A_i}$.

Proof: For any two points p and p' in D that depend on the same point q (as their i-th dependency), it is true by definition, that $A_i p + b_i = q = A_i p' + b_i$,
i.e., $A_i(p' - p) = 0$.
Hence $(p' - p)$ is a solution of $A_i x = 0$ and thus belongs to the null space of A_i.

Conversely, if v is a vector in the null space of A_i, and p is any point in D, consider the point $p' = p + v$. By definition of the ARE, p' depends on (as its i-th dependency) $A_i p' + b_i$, i.e., on $A_i(p + v) + b_i$. Since $A_i v = 0$, this is simply $(A_i p + b_i)$, which is the *same* point that p depends as its i-th dependency. ∎

We assume in the remainder of this paper that the dependency matrix A_i is singular. The reason for this is that the equation $A_i x = 0$ will then have several (actually infinite) nontrivial solutions v, and hence the co-set of any point in the domain will contain at least one other point, and the idea of data pipelining makes sense.

Once we have been able to identify, for any $p \in D$, the set, $C(p)$, of points that may potentially share their arguments, there are two questions that we must address. First, we must be able to "link" up the elements of $C(p)$ in a pipeline. Since we want to transform the affine dependency into a uniform one, we must identify a vector ρ, such that adding any multiple of it to p will yield an element of $C(p)$, and conversely, every element of $C(p)$ can be expressed as $p + k\rho$ (for $k \in Z$). This means that ρ must be a *basis* (see [Cas]) for the null space of A_i. However, since the null space of A_i may be (depending on its rank) multi-dimensional, we may have to deal with a *set* of basis vectors (which are in general, not unique). Moreover, the choice of ρ must be such that if there are any *other* constraints that the schedule is to satisfy, they must not be violated. The second crucial problem that we must solve is that of "initializing" the pipeline. Suppose that we have been able to find a (set of) appropriate basis vectors for the null space of A_i. All this will be of no use, if we cannot somehow make the required value available to at least one point in $C(p)$. Only then can this value be passed on to other points in $C(p)$. The investigation of this problem leads to our taxonomy of pipelining transformations. Before we formally address these two questions, we illustrate the ideas with a simple example.

Example 1: (contd) Consider the convolution problem described earlier. However, rather than starting from the URE of Eqn. 1, let us look at the *problem definition.* Convolution is specified by the equation $Y_i = \sum_{j=0}^{k-1} W_j * X_{i-j}$ and the decision to *iterate* over the j index yields the following equation, which is not (yet) an ARE. (as before, the upper case notation indicates that we refer to the elements of the input streams, and not values at points in the domain.)

$$Y_i = y(i,k); \text{ where } y(i,j) = y(i,j-1) + W_j * X_{i-j}$$

The domain of this computation ($D = \{[i,j] \mid 0 \leq i, 0 \leq j < k\}$) also follows directly from the decision to iterate over j. At each point $[i,j]^T$ the computation requires the value of W_j and X_{i-j}. These are input values that are not *computed* anywhere in the domain, and hence they must be obtained from the boundaries of the domain.* Since the *length* of the W stream is k, it makes sense to assign it to the $i = 0$ boundary (this is the only boundary of D of the correct length). Similarly, we assign the X stream to the $k = 0$ boundary.† We *now* have the following ARE for the computation.

$$y(i,j) = y(i,j-1) + w(0,j) * x(i-j,0) \tag{4}$$

and the boundary conditions $y(i,0) = 0$, $w(0,j) = W_j$ and $x(i,0) = X_i$. We see from Eqn. 4 that at any point $[i,j]^T \in D$ we need values from $[i,j-1]^T$, $[0,j]^T$ and $[i-j,0]^T$, and thus the three dependencies are as follows ($[A_1, b_1]$ for y (which is a uniform dependency), $[A_2, b_2]$ for w and $[A_3, b_3]$ for x).

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}; A_2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, b_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}; A_3 = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}, b_3 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Of these, the dependencies $[A_2, b_2]$ and $[A_3, b_3]$ need to be pipelined. In order to do so, we first determine the null space of A_2 and A_3. By simple linear algebra, we can show these to be $[a,0]^T$ and $[b,b]^T$ respectively (this means that all points given by $[i+a,j]^T$ require the *same* W value as $[i,j]^T$, and all points $[i+b,j+b]^T$ require the *same* X value as $[i,j]^T$). The next step is to determine the right basis vectors, ρ_2 and ρ_3, for the null space. We see that since both A_2 and A_3 have rank 1, their null spaces are one-dimensional, and their integer bases are unique modulo sign, namely $[\pm 1, 0]^T$ and $[\pm 1, \pm 1]^T$ respectively. So the problem is to determine the sign of the basis vectors. Consider the dependency $[A_2, b_2]$. Intuitively, we should "thread" the points in the co-set of $[i,j]^T$ in such a direction that they "lead to" $[0,j]^T$. This is done by ensuring that the dot product of the basis vector ρ_2 and $((A_2 p + b_2) - p)$ is positive. Thus $([0,j] - [i,j]) \cdot [a,0]^T > 0$, which implies that $a < 0$, and hence the correct basis, ρ_2 is $[-1,0]^T$. Similarly, for $[A_3, b_3]$, $([i-j,0] - [i,j]) \cdot [b,b]^T > 0$, which implies that $b < 0$, and hence the basis, ρ_3 is $[-1,-1]^T$. Using these two basis vectors as the new *uniform* dependencies, we obtain the URE of Eqn. 1. In order to do so, we have introduced two functions $w(i,j)$ and $x(i,j)$, and these are simply the identity functions with uniform dependencies $[-1,0]$ and $[-1,-1]$ respectively. At the boundaries of the domain these functions take the value W_j and X_{i-j} respectively. ∎

*The URE of Eqn. 1 thus *overspecifies* the problem, and may rule out some interesting solutions.

†There is another assignment for X, the $j = k$ boundary, and this too yields some interesting architectures.

From the above example it is clear that we may view data pipelining as a transformation for converting the original ARE into a URE. The timing and allocation functions may now be determined (using the standard techniques) and the target architecture may be synthesized. Although the three steps (pipelining, determination of a timing function and choosing an allocation function) are very closely related, and the choice of one affects the others, they are not *inherently sequential.* This fact has been well known in the context of uniform dependencies.* With affine recurrences, the pipelining operation may also be included in this list. We shall therefore use some of the constraints that apply on timing and allocation functions to guide the pipelining.

3 A General Theory of Pipelining Transformations

We shall now describe the pipelining transformations in detail. We first develop some basic results, and then present a taxonomy of such transformations. Initially we concentrate on the case when a single dependency is being pipelined in isolation (called *simple pipelining*). We shall then extend these results to the case where the pipeline involves more than one dependency (*multistage pipelining*). Our taxonomy is based on the observation that once we have identified the co-set of a point, pipelining involves the solution of two distinct, albeit related, problems.

- Determining a *basis* for the null space. Depending on whether the null space is one-dimensional or multi-dimensional, the pipeline is said to be *linear* or *extended.*
- Initializing the pipeline. Depending on how the pipeline is initialized, the transformations are classified as *direct* or *indirect.*

We shall now address these two problems. In general, if the rank of an $n \times n$ dependency matrix A is $n - k$, its null space consists of a k-dimensional subspace. Since the domain D consists of lattice points in Z^n, the null space is a sub-lattice of this. Formally (see [Cas]), a lattice is the set of all points of the form $x = u_1 a_1 + \cdots + u_n a_n$, where the u_i's are integers and a_i's are linearly independent real vectors called its basis. The basis is not unique, and in particular, if $v_{i,j}$ is an integer matrix with determinant ± 1, then $a'_i = \sum_j v_{i,j} a_j$ constitute an alternate basis(see [Cas]). Hence it is important to *choose* the basis, just as one chooses timing and allocation functions from the space of all such functions that satisfy the causality and locality constraints.

*For example, some authors choose to pick an allocation function first, while others determine a timing function first.

Definition 6 A vector ρ is said to be *consistent* with an ATF $[\lambda_t, \alpha_t]$ iff $\lambda_t^T \cdot \rho < 0$. A set $\Gamma = \{\rho_i\}$ of vectors is consistent with an ATF $[\lambda_t, \alpha_t]$ iff each of the ρ_i's is consistent with $[\lambda_t, \alpha_t]$. We say that ρ (or Γ) is λ_t-consistent.

Definition 7 A vector ρ is said to be *consistent* with an allocation function $[\lambda_a, \alpha_a]$ iff $\lambda_a \rho \in P$, the set of permissible interconnections. As before, a set $\Gamma = \{\rho_i\}$ of vectors is consistent with an AAF $[\lambda_a, \alpha_a]$ iff each of the ρ_i's is consistent with $[\lambda_a, \alpha_a]$. If ρ is consistent with both the timing and allocation functions, it is said to be λ-consistent.

Intuitively, we can say that if a vector ρ is consistent with an ATF $[\lambda_t, \alpha_t]$ then we can augment any ARE for which $[\lambda_t, \alpha_t]$ is a valid ATF by introducing a new dependency ρ, *and still retain* $[\lambda_t, \alpha_t]$ *as a valid ATF*. Conversely, if we intend to use a vector ρ as a new dependency, the timing function that we choose must be λ_t-consistent. The same argument holds for λ_a-consistency. We thus see that although the basis is potentially infinite, in practice we are concerned with a small number of vectors. This is analogous to the fact that in the traditional approach to systolic array synthesis, a large number of timing functions are possible, but only a few of them are of practical interest. Moreover, in a semi-infinite recurrence, the allocation function is predetermined by the problem specification (it must project the domain onto a *finite* processor space, and hence must be along the *ray* of the domain). As we shall see later, the problem of initializing the pipeline also imposes constraints on the direction of pipelining, especially for extended pipelines.

We know that if the basis of the null space of the dependency A is λ_t-consistent, the timing function imposes a partial order on all the points in $C(p)$. There is thus at least one point, $p_\perp$, that is the earliest scheduled point in $C(p)$, i.e., $\forall q \in C(p)$, $t(p_\perp) \leq t(q)$. The following lemma gives an important property of such points, namely that they *must* lie on or near a boundary of the domain.

Lemma 2 *If* $\{\rho_i \mid i = 1 \ldots k\}$ *is a* λ_t*-consistent basis of* $C(p)$*, then* $p_\perp + \rho_i \notin D$*, for* $i = 1 \ldots k$.

Proof: Let, if possible, $(p_\perp + \rho_i) \in D$. By definition, $t(p_\perp) \leq t(p_\perp + \rho_i)$, i.e., $\lambda_t^T \rho_i > 0$. But this is a contradiction, since ρ_i is λ_t-consistent. ∎

The above lemma implies that in order to solve the problem of initializing the pipeline, we must make $f(q)$ available to $p_\perp$, since these are the only points in $C(p)$ which do not depend on any other point in D.

3.1 Simple Linear Pipelining

Let us consider the case when *rank*(A) is $n-1$, so that the pipeline is linear. In this case ρ is unique (modulo sign), and the sign too may be determined from the requirement of λ_t-consistency (note that both ρ and $-\rho$ can never be λ_t-consistent simultaneously). As a result, $p_\perp$ is a unique function of p. We are now in a position to address the problem of initializing the pipeline. We must determine the conditions under which the value of $f(q) = f(\mathrm{A}p + \mathrm{b})$ can be made available at the point $p_\perp$. The simplest case is when q and $p_\perp$ are identical, in which case we have *direct* pipelining, where the pipeline is initialized at $p_\perp$ itself. A simple extension is the case when q is a constant distance from $p_\perp$. We define the two classes of linear pipelines as follows.

Definition 8 An affine dependency [A, b] can be pipelined with a *direct linear pipelining function* iff the rank of A is $n-1$, ρ is a λ_t-consistent basis vector for the null space of A, and $p_\perp = q$. The pipelining function is given by

$$f'(p) = \begin{cases} f(p) & \text{if } p = p_\perp \\ f'(p+\rho) & \text{otherwise} \end{cases} \tag{5}$$

Definition 9 An affine dependency [A, b] can be pipelined with an *indirect linear pipelining function* iff the rank of A is $n-1$, ρ is a λ_t-consistent basis vector for the null space of A, and there exists a λ_t-consistent vector ρ' such that $p_\perp \rho' = q$. The pipelining function is given by

$$f'(p) = \begin{cases} f(p+\rho') & \text{if } p = p_\perp \\ f'(p+\rho) & \text{otherwise} \end{cases} \tag{6}$$

Observe that since $p_\perp$ is a *function* of p, we may compute it by determining the intersection of a line given by $p + u\rho$ with a domain boundary. Since the domain is a convex hull, it is always possible to partition it by means of a *finite* number of hyperplanes into regions (each of which is a convex hull) such that a line $p + u\rho$ passing through *all* the points in the same region will intersect a single boundary of the domain, say $\pi^{\mathrm{T}} p = \theta$. By simple analytic geometry, this point (i.e., the intersection of $p + u\rho$ and $\pi^{\mathrm{T}} p = \theta$) is given by $p + (\frac{\theta - \pi^{\mathrm{T}} p}{\pi^{\mathrm{T}} \rho})$. Hence $p_\perp$, expressed as a function of p, has the following form.

$$p_\perp = \begin{cases} p + (\frac{\theta_1 - \pi_1^{\mathrm{T}} p}{\pi_1^{\mathrm{T}} \rho})\rho & \text{if } \phi_1(p) \\ p + (\frac{\theta_2 - \pi_2^{\mathrm{T}} p}{\pi_2^{\mathrm{T}} \rho})\rho & \text{if } \phi_2(p) \\ \vdots & \\ p + (\frac{\theta_i - \pi_i^{\mathrm{T}} p}{\pi_i^{\mathrm{T}} \rho})\rho & \text{if } \phi_i(p) \end{cases} \tag{7}$$

Here each $\phi_k(p)$ is a conjunction of expressions of the form $\mu^{\mathrm{T}} p \geq \nu$ which partition the domain into the regions mentioned above. Thus, the test for $p = p_\perp$ is simply a disjunction of a set of conjunctions of the form $(\phi_1(p) \wedge \pi_1^{\mathrm{T}} p = \theta_1) \vee (\phi_2(p) \wedge \pi_2^{\mathrm{T}} p = \theta_2) \vee \ldots \vee (\phi_i(p) \wedge \pi_i^{\mathrm{T}} p = \theta_i)$. The following theorem gives us necessary and sufficient conditions for the existence of a simple linear direct pipeline.

Theorem 1 *An affine dependency* [A, b] *can be pipelined along a simple linear direct pipeline if and only if the rank of* A *is* $n-1$, ρ *is a* λ*-consistent basis for the null space of* A, *and*

$$\mathrm{A}((\mathrm{A}p + \mathrm{b}) - p) = 0$$

The pipelining function, $f'(p)$ *defined by*

$$f'(p) = \begin{cases} f(p) & \text{if } \phi_1'(p) \\ \vdots & \\ f(p) & \text{if } \phi_i'(p) \\ f'(p+\rho) & \text{otherwise} \end{cases}$$

Proof: The result follows follows directly if we can show that $p_\perp = q$ iff $\mathrm{A}((\mathrm{A}p + \mathrm{b}) - p) = 0$. To show this, we see that if $q \in C(p)$, then by definition of co-set, all points in $C(p)$ depend on q, and because of λ_t-consistency, q must be the earliest scheduled point in $C(p)$, i.e., $p_\perp = q$. Conversely, if $p_\perp = q$ then $q \in C(p)$. But this is true iff $q - p$ is in the null space of A, i.e., iff $\mathrm{A}((\mathrm{A}p + \mathrm{b}) - p) = 0$.

Moreover, by the preceeding discussion, and the fact the disjunctions can be separated into separate cases, the pipelining function is as described above. ∎

Corollary 1 *The necessary and sufficient conditions for simple linear indirect pipelining are that* rank*(A) is* $n-1$, *A has a* λ*-consistent basis,* ρ *and*

$$\mathrm{A}((\mathrm{A}p + \mathrm{b}) - p) = \text{constant}$$

The pipelining function is given by

$$f'(p) = \begin{cases} f(p+\rho') & \text{if } \phi'_1(p) \\ \quad \vdots & \\ f(p+\rho') & \text{if } \phi'_i(p) \\ f'(p+\rho) & \text{otherwise} \end{cases} \tag{8}$$

Corollary 2 *The pipelining functions for simple linear pipelining (both direct and indirect) are CUREs*

3.2 Simple Extended Pipelining

Let us now return to the more general case when the *rank*(A) is $n-k$. Assume, for now, that $p_\perp$ is unique. We choose the first basis vector, say ρ_1 arbitrarily, as long as it is λ-consistent. This is the initial direction of the pipeline. Note that as in the case of linear pipelining, we may partition the domain into regions such that all pipelines in each region (along the direction ρ_1) intersect the same domain boundary. We may thus define the pipelining function as follows.

$$f'(p) = \begin{cases} f_1(p) & \text{if } \phi_1(p) \\ \quad \vdots & \\ f_m(p) & \text{if } \phi_m(p) \end{cases} \tag{9}$$

where each of the functions $f_i(p)$ is the pipelining function for one partition. A useful heuristic for the choice of ρ_1 is to minimize the number of such partitions. We shall now concentrate on the details of the f_i's.

For each f_i, we know that all the pipelines intersect a specific domain boundary, say, $\pi_1^T p = \theta_1$. Since $C(p)$ is a k-dimensional lattice, we know by Lemma 2 that $p_\perp$ belongs to the intersection, denoted by $C_1(p)$, of $C(p)$ and $\pi_1^T p = \theta_1$. However, $C_1(p)$ is a $k-1$-dimensional sub-lattice of $C(p)$, and has $k-1$ basis vectors. Note that any set of basis vectors for $C_1(p)$, together with ρ_1 constitute a basis for $C(p)$. Hence, we can make $f(q)$ available to every point in the domain (by pipelining it along ρ_1) if we can first pipeline it to each point in $C_1(p)$. We therefore choose our second basis vector ρ_2, such that it is a λ-consistent basis vector for $C_1(p)$, and pipeline along ρ_2 within $C_1(p)$. By the reasoning used in Lemma 2, $p_\perp$ must be on a *boundary* of $C_1(p)$, i.e., the intersection of

$C_1(p)$ with *another* bounding hyperplane of D. This boundary in not unique, but as before, we may partition $C_1(p)$ into regions with one boundary for each region. Again, a useful heuristic for the choice of ρ_2 is one that minimizes the number of such partitions. Let the boundary be given by $\pi_2^T p = \theta_2$, and let us denote the new intersection by $C_2(p)$. We now choose the third basis vector to pipeline in $C_2(p)$ and so on, until eventually, $C_{k-1}(p)$ is a linear pipeline, and we reach $p_\perp$. We may thus define each of the $f_k(p)$'s as follows.

$$f_k(p) = \begin{cases} f(p+\rho') & \text{if } \pi_1^T p = \theta_1 \wedge \pi_2^T p = \theta_2 \wedge \ldots \pi_{k-1}^T p = \theta_{k-1} \wedge \pi_k^T p = \theta_k \\ f_k(p+\rho_k) & \text{if } \pi_1^T p = \theta_1 \wedge \pi_2^T p = \theta_2 \wedge \ldots \pi_{k-1}^T p = \theta_{k-1} \\ \quad \vdots & \\ f_k(p+\rho_2) & \text{if } \pi_1^T p = \theta_1 \\ f_k(p+\rho_1) & \text{otherwise} \end{cases} \tag{10}$$

It is clear from the above discussion, that not only must the set of basis vectors be λ-consistent, but they must also be parallel to the boundaries of D. Thus, ρ_1 may be arbitrary (as long as it is λ-consistent), ρ_2 must be parallel to $\pi_1^T p = \theta_1$, ρ_3 must be parallel to $\pi_1^T p = \theta_1$ and $\pi_2^T p = \theta_2$, and similarly, ρ_k must be parallel to each of the boundaries involved in the above equation. This serves to prune the space of possible basis vectors. As in the case of linear pipelining, the above scheme will be useful only if we can initialize the pipeline at $p_\perp$. This can be done if q is close to (i.e., a *constant* vector away from) $p_\perp$, i.e., if

$$\mathrm{A}((\mathrm{A}p + \mathrm{b}) - p) = \text{constant}$$

Note that we have made one important assumption, namely that $p_\perp$ is unique. Let us look at the implications of this. Let $p_\perp$ not be unique. Then, the set of points $p_\perp$ constitute a sublattice of $C(p)$ (since they all lie on the same timing hyperplane) and in the pipelining process described above, one of the sublattices, say $C_i(p)$, will consist entirely of the points $p_\perp$. This sublattice will not have a λ_i-consistent basis vector. This means that we have a *set of points*, each requiring the value of $f(q)$, and each one scheduled at the same time instant. Clearly, it is impossible to pipeline $f(q)$ to each of them, so they *must* all depend directly on q. There is thus no way to avoid broadcasting, except by choosing a different, possibly suboptimal, timing function.

3.3 Multistage Pipelining

We have so far addressed the problem of pipelining an affine recurrence into a uniform one for a single dependency $[\mathrm{A_i}, \mathrm{b_i}]$, in isolation. In each of the schemes that were proposed, the primary

condition for initializing the pipeline was that the point $q = A_i p + b_i$ must be "close" to the null space of A, in the sense that $A_i((A_i p + b_i) - p)$ must be a constant. While this class of transformations is useful in a large number of cases, the above condition may not be always satisfied. We shall now present another technique called *multistage pipelining*, where it will be possible to pipeline a particular dependency, even though this condition is not satisfied. As with simple pipelining, multistage pipelining too, may be classified as linear or extended, direct or indirect. The pipelining is achieved by building the pipeline exactly as described above; thus if the value of $f(q)$ can be made available to $p_\perp$, then the pipelining is successful. In *simple* pipelining, the basic assumption was that this could be done, provided that the vector $p_\perp - q$ was a *constant* vector (independent of p). The question that we now address is what happens if this is not true.

Multistage pipelining is based on the key observation that there is an alternative way in which $p_\perp$ can receive the value of $f(q)$. Consider the j-th inverse co-set, $C_j^{-1}(q)$ of q (for $i \neq j$, i.e., corresponding to some *other* dependency $[A_j, b_j]$). Clearly, if this dependency is pipelineable, then every point p' in $C_j^{-1}(q)$ can get the value of $f(q)$ as its j-th argument. If this set is "close to" (i.e, a *constant* distance from) $p_\perp$, then we can successfully pipeline the i-th dependency too. All that we have to do is to obtain the value of $f(q)$ at $p_\perp$ *from the pipelining function for the j-th dependency.* The condition for multistage pipelining can thus be stated as follows.

$$A_j((A_i p + b_i) - p_\perp) = \text{constant}$$

where $[A_j, b_j]$ is a pipelineable dependency. The pipelining function $f_i(p)$ (for the indirect extended case) is defined by Eqn. 11 below.

$$f_i(p) = \begin{cases} f_j(p + \rho') & \text{if } \pi_1^T p = \theta_1 \wedge \pi_2^T p = \theta_2 \wedge \ldots \pi_{k-1}^T p = \theta_{k-1} \wedge \pi_k^T p = \theta_k \\ f_i(p + \rho_k) & \text{if } \pi_1^T p = \theta_1 \wedge \pi_2^T p = \theta_2 \wedge \ldots \pi_{k-1}^T p = \theta_{k-1} \\ \quad \vdots & \\ f_i(p + \rho_2) & \text{if } \pi_1^T p = \theta_1 \\ f_i(p + \rho_1) & \text{otherwise} \end{cases} \tag{11}$$

Note that this function is identical to Eqn 10, except that at $p_\perp$ (i.e., in the first line of the definition), its value is $f_j(p_\perp + \rho')$ rather than $f(p_\perp + \rho')$ where f_j is the pipelining function for $[A_j, b_j]$. For notational simplicity, we have shown the case when all the pipelines intersect a single domain boundary, and there is no need to partition the domain. Thus the subscripts here refer to the dependency that is being pipelined, unlike in Eqn 10 where they indicate a specific partition of the domain.

4 Synthesizing Systolic Arrays from CUREs

Since all the pipelining functions that we have described in the previous section are CURE's, we now have a constructive procedure to transform an ARE into a CURE. These CURE's have no computational expense (at all points in the domain, the *value* that they return is identically equal to their inputs), except for the linear conditional expressions (LCEs). Moreover, the new dependencies that have been introduced are consistent with the timing and allocation functions. It would seem that the target architecture is induced automatically once the data pipelining functions are determined. There is however, one essential difference between a URE and a CURE. In a URE, the timing and allocation functions determine which point is mapped to which processor at which time, and the definition of the recurrence determines that the processor functionality is g. In a CURE (see Defn. 4), the processor functionality is somewhat more complicated. Clearly, the processor must be capable of computing each of the functions, $g_0, g_1, g_2, \ldots, g_k$, and the computation is no longer strict, since the arguments of only one of the g's are used at any time instant. In addition, each processor must compute the LCE's that determine which g_i is applicable. Let us see what this computation would entail.

Since convex hulls are closed under affine transformations, the image of D in the processor-time space is also a convex hull. In particular, the image of a hyperplane $\pi^{\mathrm{T}} p = \theta$ is another hyperplane, $\pi'^{\mathrm{T}} p' = \theta'$, where p' is a point in the processor-time space. We denote the $n-1$ dimensional vector representing the processor id by $\overline{x}$, and since $p' = [\overline{x}, t]^{\mathrm{T}} = \lambda p + \alpha$ and λ is non-singular, $\pi'^{\mathrm{T}} = \pi^{\mathrm{T}} \lambda^{-1}$ and $\theta' = \theta + \pi^{\mathrm{T}} \cdot \alpha$. Thus each LCE in the CURE is the conjunction of a number of linear expressions of the form $\mu^{\mathrm{T}} [\overline{x}, t]^{\mathrm{T}} \mathrm{op}\ \nu$, where "op" is a comparison operator. Computing each of these involves a dot-product (of two vectors) and an integer comparison. More significantly, it also requires the values of $\overline{x}$ and t, *i.e.*, the processor must be aware of its location in the array and also the current time. An array of such processors can hardly be considered a *systolic array*, where each processor is expected to be small and simple and all processors must be identical. We call such architectures quasi-systolic. More formally, we have the following definition.

Definition 10 An architecture is said to be quasi-systolic if it is made up of a number of processors, each one capable of computing a small number of functions, and all its data dependencies are regular, uniform and tesselating. However, the control that determines which specific function is to be used by any processor at any time is not based on any local function.

We thus see that the pipelining transformations yield a quasi-systolic array. However, we are interested in architectures that have no global control, i.e., pure systolic arrays. In order to obtain

such architectures from the quasi-systolic arrays, we use a transformation called *control pipelining*. Control pipelining is based on the simple observation that as in data pipelining, there is no need to *recompute* the control information if we are able to *share* it. Let us first consider the case when all the LCE's are conjunctions of equalities, and thus have the form, $\pi_1^T p = \theta_1 \wedge \ldots \wedge \pi_m^T p = \theta_m$. We see that all the points on the hyperplane $\pi_i^T p = \theta_i$ will satisfy the i-th conjunction of the LCE, and hence belong to what is called its i-th *control co-set*. As in the case of data pipelining, we need to determine a λ-consistent basis for this ($n-1$-dimensional) sublattice. The value that is pipelined along this pipeline is simply the information that the linear condition $\pi_i^T p = \theta_i$ is satisfied, i.e., a boolean value. Moreover, since $[\lambda_a, \alpha_a]$ maps the domain D on to an $(n-1)$-dimensional processor space, the intersection of the pipeline with another domain boundary *must* lie on the edge of the processor array. Hence, unlike data pipelining, there is no need for *extended* pipelining, and we may easily initialize the pipeline at the appropriate boundary. These results are described formally by the following theorem.

Theorem 2 *Given that C is a CURE and the linear conditional expressions in C are conjunctions of some of $\pi_1^T p = \theta_1$, $\pi_2^T p = \theta_2$, ..., $\pi_m^T p = \theta_m$, and $[\lambda, \alpha]$ generates a valid quasi-systolic array that computes it. The CURE C' which is identical to C but has each $\pi_i^T p = \theta_i$ replaced by $f_i(p)$ defined below, is equivalent to C and $[\lambda, \alpha]$ generates a pure systolic array for it, if for $i = 1, \ldots, m$, $\exists \sigma_i$ such that*

$$\begin{aligned} \pi^T \sigma_i &= 0 \\ \lambda_t \cdot \sigma_i &< 0 \\ \textit{and} \quad \lambda_a \cdot \sigma_i &\in P \end{aligned}$$

The f_i's are defined as $f_i(p) = f_i(p+\sigma_i)$, and are initialized to 1 at the boundary of the processor array.

The proof is obvious from the above discussion, since the first condition on σ_i ensures that it is a basis vector for the hyperplane $\pi_i^T p = \theta_i$, and the other two ensure λ-consistency. We also have a straightforward extension to the case when some of the expressions are inequalities. In that case each processor, $\overline{x}$ contains a one-bit register which is initialized to either 1 or 0, based on whether $\pi_i^T(\lambda^{-1}[\overline{x}, 0]^T) + \alpha) - \theta_i$ is positive or not. The value in this register is used instead of the conditional expression $\pi_i^T p \geq \theta_i$. The function $f_i(p)$ corresponding to the control pipeline for the equality case, $\pi_i^T p = \theta_i$ is used to *toggle* the value in the register.

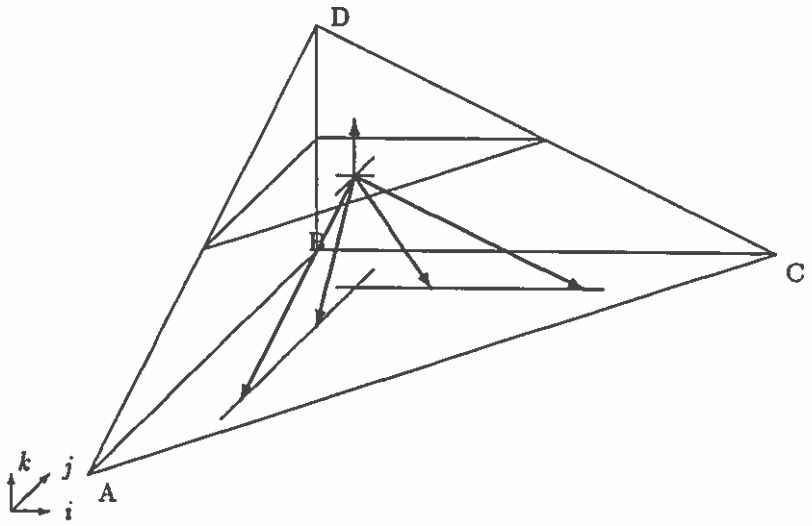
5 Example: Optimal String Parenthesization

Let us now return to the problem of Example 2 (Section 2). As described there, Eqn. 2 defines an ARE to compute the cost of optimally parenthesizing a string, using a simple dynamic programming algorithm. In this section we shall describe how data pipelining and control pipelining can be used to derive the well known Guibas-Kung-Thompson systolic array for this problem. We first note that this ARE does not admit an affine timing function. A formal proof of this is described in [Raj], and is based on the argument that the longest dependency path in the ARE is $O(n^2)$, i.e., the *algorithm itself is a quadratic one*, and hence cannot be implemented on a systolic array. It is therefore necessary to reformulate the problem as an ARE that has a valid ATF, and we shall use the ARE shown in Eqn. 12 below, as a starting point for the synthesis. We must emphasize at this point, that our theory does not claim to answer the question of *how* this ARE is obtained. Indeed, this problem involves issues in automatic programming, and techniques such as the "fold-unfold" transformations of Burstall and Darlington [BD] could be profitably used. Also note that we have included all the boundary conditions as part of the recurrence itself. The domain for this ARE is the convex hull bounded by $k > 0$, $j - i \geq 2k$, $i > 0$ and $j > 0$ (see Fig. 2).

$$c(1,n) = f(1,n,1)$$

where $f(i,j,k)$ is defined as

$$f(i,j,k) = \begin{cases} h_{i,j} & \text{if } j-i=1 \\ h_{i,j} + \min \begin{pmatrix} f(i,i+k,1)+f(i+k,j,1) \\ f(i,j,k+1) \\ f(i,j-k,1)+f(j-k,j,1) \end{pmatrix} & \text{if } k=1 \\ \infty & \text{if } 2*k > j-i \\ \min \begin{pmatrix} f(i,i+k,1)+f(i+k,j,1) \\ f(i,j,k+1) \\ f(i,j-k,1)+f(j-k,j,1) \end{pmatrix} & \text{otherwise} \end{cases} \tag{12}$$



$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} b_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}; A_2 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} b_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}; A_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} b_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix};$$

$$A_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix} b_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}; A_5 = \begin{bmatrix} 0 & 1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} b_5 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix};$$

Figure 2: *Domain and Dependency Structure for the String Parenthesization ARE of Eqn 12*

5.1 Timing Function

The first step in the synthesis procedure is to determine an ATF, $[\lambda, \alpha]$ (denoted by $[[\mathtt{a}, \mathtt{b}, \mathtt{c}], \alpha]$) for this ARE. Techniques for determining timing functions for AREs are discussed elsewhere (see [Raj]). It can be shown that $[[\mathtt{a}, \mathtt{b}, \mathtt{c}]^T, \alpha]$ is a valid ATF if it satisfies the following inequalities (for all $[i, j, k]$

in the domain).

$$\begin{aligned} \mathtt{b}(j-i-k) &> -\mathtt{c}(k-1) \\ \mathtt{a}k &< \mathtt{c}(k-1) \\ \mathtt{c} &< 0 \\ \mathtt{b}k &> \mathtt{c}(k-1) \\ \mathtt{a}(j-i-k) &< \mathtt{c}(k-1) \end{aligned}$$

Since a, b and c are restricted to be integers, and $j - i \geq 2k$, these can be reduced to $\mathtt{c} < 0$; $\mathtt{b} \geq -\mathtt{c}$ and $\mathtt{a} \leq \mathtt{c}$. The optimal ATF thus corresponds to the smallest integer (absolute-valued) solution to the above inequalities, and is given by $\lambda_{opt} = [-1, 1, -1]^{\mathrm{T}}$ (with $\alpha_{opt} = 1$), i.e.,

$$t(i, j, k) \equiv j - i - k + 1$$

Thus, the time at which the computation terminates, i.e., $t(1, n, 1)$ is $n - 1$.

5.2 Pipelining the Data Dependencies

The next step in the synthesis procedure is to pipeline the affine dependencies. For this the null space of each of A_1, A_2, A_4 and A_5 must first be computed. It is easy to show that the rank of each of these matrices is 2, and hence their null spaces are all one-dimensional, specified by a single basis vector each, say ρ_1, ρ_2, ρ_4 and ρ_5, respectively. It is also a matter of straightforward linear algebra to solve the appropriate systems of equations and derive that ρ_1, ρ_2, ρ_4 and ρ_5 are respectively, $[0, m_1, 0]^{\mathrm{T}}$, $[m_2, 0, m_2]^{\mathrm{T}}$, $[0, m_4, -m_4]^{\mathrm{T}}$ and $[m_5, 0, 0]^{\mathrm{T}}$.

However, the dot-product $\rho_2 \cdot \lambda$ is $[m_2, 0, -m_2]^{\mathrm{T}} \cdot [-1, 1, -1]^{\mathrm{T}}$ which is zero (similarly $\rho_4 \cdot \lambda$ is also zero). This means that it is *impossible* to obtain a basis for the null spaces of either A_2 or A_4 that is λ_t-consistent . It is therefore necessary to choose another timing function for which the basis of the null spaces of A_i and A_i can be λ_t-consistent. It is seen that $[-2, 2, -1]$ satisfies this requirement, and hence a satisfactory ATF is the following

$$t(i, j, k) \equiv 2(j - i) - k + 1$$

In fact, it can be shown that $[[-2, 2, -1]^{\mathrm{T}}, -1]$ is the optimal ATF that is also λ-consistent. Thus, $t(1, n, 1)$ is $2n - 2$, and the algorithm is half the speed of the optimal one. However, this is the only

way the algorithm may be implemented on a systolic array. Now, λ_t-consistent basis vectors for each of the dependencies can be derived, and are $[0,-1,0]^T$, $[1,0,-1]^T$, $[0,-1,-1]^T$ and $[1,0,0]^T$.

The next step is to pipeline each of the dependencies of the ARE, either by using direct pipelining or through multistage pipelining. To recapitulate, for any affine dependency $[A_i, b_i]$, the condition for simple pipelining is that for any point $p = [i,j,k]^T \in D$, the expression $A_i((A_i p + b_i) - p)$ must be constant.

For the four dependencies that need to be pipelined, $A_i p + b_i$ are $[i, i+k, 1]^T$, $[i+k, j, 1]^T$, $[i, j-k, 1]^T$ and $[j-k, j, 1]^T$ respectively, and hence the values of $(A_i p + b_i) - p$ are $[0, i+k-j, 1-k]^T$, $[k, 0, 1-k]^T$, $[0, -k, 1-k]^T$ and $[j-k-i, 0, 1-k]^T$, respectively. Then, the values of $A_i((A_i p + b_i) - p)$ for each of the four dependencies may be easily computed to be $[0, 1-k, 0]^T$ for A_1, $[1,0,0]^T$ for A_2 $[0,-1,0]^T$ for A_4 and $[1-k, 0, 0]^T$ for A_5. Thus, it is clear that only $[A_2, b_2]$ and $[A_4, b_4]$ can be pipelined by a simple linear indirect pipeline. By straightforward computational geometry we can determine that for all points in the domain, both $C_2(p)$ and $C_4(p)$ intersect the $k = 1$ boundary, and $\rho_{2\perp}$ and $\rho_{4\perp}$ are $[i+k-1, j, 1]^T$ and $[i, j-k+1, 1]^T$ respectively; thus the terminal dependencies are $\rho_2' = [1,0,0]^T$ and $\rho_4' = [0,-1,0]^T$, respectively (note that they are λ_t-consistent). Thus the pipelining functions f_2 and f_4 are as follows.

$$f_2(i,j,k) = \begin{cases} f(i+1,j,k) & \text{if } k = 1 \\ f_2(i+1,j,k-1) & \text{otherwise} \end{cases}$$

and

$$f_4(i,j,k) = \begin{cases} f(i,j-1,k) & \text{if } k = 1 \\ f_4(i,j-1,k-1) & \text{otherwise} \end{cases}$$

It is also clear from the above discussion that $[A_1, b_1]$ and $[A_5, b_5]$ are not amenable to simple pipelining; if any pipelining is to be achieved, it must be multistage. In order to test for this, it is again straightforward to determine that $\rho_{1\perp}$ and $\rho_{5\perp}$ both lie on the $j - i = 2k$ boundary and are $[i, i+2k, k]^T$ and $[j-2k, j, k]^T$ respectively, and hence $A_i p + b_i - \rho_{i\perp}$ (for $i = 1, 5$) are $[0, -k, 1-k]^T$ and $[k, 0, 1-k]^T$ (denoted henceforth by Δ_1 and Δ_5) respectively. Since $[A_2, b_2]$ and $[A_4, b_4]$ are the only dependencies that have been pipelined so far, the $[A_1, b_1]$ dependency can be multistage pipelined iff either $A_2 \cdot \Delta_1 =$ constant or $A_4 \cdot \Delta_1 =$ constant. Similarly, for the dependency $[A_5, b_5]$, Δ_5 must satisfy either $A_2 \cdot \Delta_5 =$ constant or $A_4 \cdot \Delta_5 =$ constant. It is a matter of straightforward algebra to ascertain that indeed, $A_4 \cdot \Delta_1 = [0,-1,0]^T$ and $A_2 \cdot \Delta_5 = [1,0,0]^T$. Thus, it is possible to pipeline $[A_1, b_1]$ by first pipelining along the dependency ρ_1, and then switching to a new dependency ρ_4 (and its corresponding function f_4). Similarly, $[A_5, b_5]$ can be completely pipelined by first pipelining along the dependency ρ_5, and then switching to a new dependency ρ_2

(and its corresponding function f_2). The original ARE is thus equivalent to the following system of CUREs.

$$c(1,n) = f(1,n,1)$$

where $f(i,j,k)$ is defined as

$$f(i,j,k) = \begin{cases} h_{i,j} & \text{if } j-i=1 \\ h_{i,j} + \min \begin{pmatrix} f_1(i,i+k,1) + f_2(i+k,j,1) \\ f(i,j,k+1) \\ f_4(i,j-k,1) + f_5(j-k,j,1) \end{pmatrix} & \text{if } k=1 \\ \infty & \text{if } 2k > j-i \\ \min \begin{pmatrix} f_1(i,i+k,1) + f_2(i+k,j,1) \\ f(i,j,k+1) \\ f_4(i,j-k,1) + f_5(j-k,j,1) \end{pmatrix} & \text{otherwise} \end{cases}$$

where

$$f_2(i,j,k) = \begin{cases} f(i+1,j,k) & \text{if } k=1 \\ f_2(i+1,j,k-1) & \text{otherwise} \end{cases} \quad f_4(i,j,k) = \begin{cases} f(i,j-1,k) & \text{if } k=1 \\ f_4(i,j-1,k-1) & \text{otherwise} \end{cases}$$

and

$$f_1(i,j,k) = \begin{cases} f_4(i,j,k) & \text{if } 2k=j-i \\ f_1(i,j-1,k) & \text{otherwise} \end{cases} \quad f_5(i,j,k) = \begin{cases} f_2(i,j,k) & \text{if } 2k=j-i \\ f_5(i+1,j,k) & \text{otherwise} \end{cases}$$

5.3 Determining the Control Dependencies and Allocation Function

The final step in the synthesis procedure is to choose an allocation function that satisfies the constraints of locality of interconnections, and if necessary, choose appropriate λ-consistent control dependencies σ_i. The allocation function must also map these control dependencies to neighboring processors. From the above CURE, it is clear that the guard expressions that need to be evaluated are the following.

$$\begin{aligned} k &= 1 \\ j-i &= 2k \\ j-i &= 1 \end{aligned}$$

The data dependencies of the CURE and the associated delays (derived from $\lambda \cdot \rho_i$) are as follows.

	$[0,-1,0]$	for f_1	with a delay of 2 units
	$[1,0,-1]$	for f_2	with a delay of 1 unit
	$[0,-1,-1]$	for f_4	with a delay of 1 unit
	$[1,0,0]$	for f_5	with a delay of 2 units
and	$[0,0,1]$	for f	with a delay of 1 unit

The terminal dependencies are $[1,0,0]$ for f_2, and $[0,-1,0]$ for f_4. It is also clear that the value $h_{i,j}$ is a constant value, to be input from the external world, and that all the points in D need its value when $k = 1$. Thus, the most obvious choice for an allocation function is thus a simple vertical projection, i.e.,

$$[x, y] = a(i, j, k) \equiv [i, j]$$

With this allocation function, the five dependencies above are mapped to $[0,-1]$, $[1,0]$, $[0,-1]$, $[1,0]$, and $[0,0]$, respectively. This means that each processor $[x, y]$ gets two values (corresponding to f_2 and f_5) from processor $[x+1, y]$ over lines of delay and 2 time units, respectively; it similarly receives two values (f_1 and f_4) from processor $[x, y-1]$ over lines of delay 1 and 2, respectively. The value corresponding to f remains in the processor (in an accumulator register) and is updated on every cycle. Thus the only remaining problem is to choose appropriate control dependencies for the three control planes $k = 1$, $j - i = 2k$ and $j - i = 1$, specified by $\pi_1 = [0,0,1]$ $\pi_2 = [-1,1,-2]$ and $\pi_3 = [-1,1,-1]$, respectively. However, the third one intersects the domain at only one line $[i, i+1, 1]$ and this line is mapped by the allocation function to the processors $[i, i+1]$.* As a result there is no need for a control signal. All the processors $[i, i+1]$ merely output the value $h_{i,i+1}$ at time instant $t = 1$. The important control dependencies are thus those corresponding to π_1 and π_2. These correspond to control signals σ_1 and σ_2 if $\pi_1 \cdot \sigma_1 = 0$ and $\pi_2 \cdot \sigma_2 = 0$. It is very straightforward to deduce that σ_1 should be $[c_i, c_j, 0]$ where c_i and c_j are arbitrary integers. The conditions of λ-consistency yield one constraint, namely $2(c_j - c_i)$ must be negative, and the constraint of locality of interconnections yields another constraint, namely that the vector $[c_i, c_j]$ must be one of the six permissible interconnection vectors $\{(\pm 1, 0), (0, \pm 1), (\pm 1, \pm 1)\}$. This yields only two possible values for σ_1, $[1,0]$ and $[0,-1]$ and any one of them can be chosen (say the former). This corresponds to a vertical control signal that travels with a delay of two time units. For σ_2, the analysis is similar; $\sigma_2 \cdot \pi_2 = 0$ yields $\sigma_2 = [c_i, c_i + 2c_k, c_k]$, λ-consistency yields $3c_k < 0$. However, nearest-neighbor interconnection cannot be achieved, since the smallest (absolute) value

*In fact this line is the only part of the domain that is mapped to this subset of the processors.

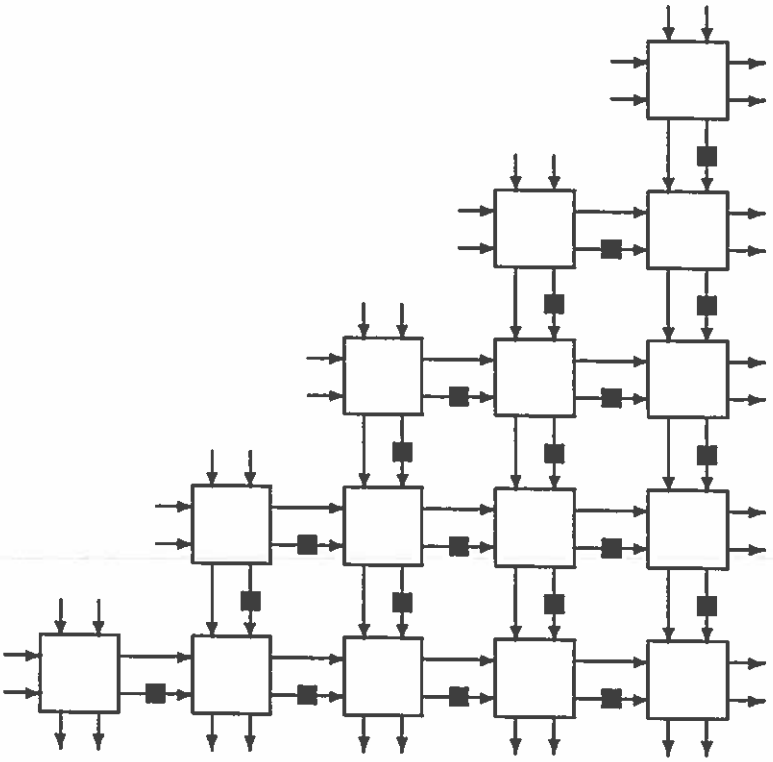


Figure 3: Final Architecture for String Parenthesization

for c_k that satisfies λ-consistency is -1, and that is not nearest-neighbor (any larger absolute value for c_k will correspond to an even more distant interconnection). If this last constraint is relaxed $\sigma_2 = [0, -2, -1]$ is a valid choice. This corresponds to a (horizontal) control signal that connects every alternate processor and travels at a speed of two processors every three time units.

The final architecture that this yields is shown in Figure 3 and is identical to the Guibas et al. [GKT] one, which is a well known systolic array and is considered by many to be a classic example of the intuitive *eureka* steps involved in systolic array design. One important point that needs to be mentioned here is that although the techniques presented in these two chapters have been developed in the context of *pure* systolic arrays (i.e., those that permit only nearest neighbor interconnections) the same techniques are directly applicable when the architecture is not strictly systolic. As we have just seen, merely relaxing a few constraints achieves this result.

6 Related Work and Discussion

Recently the problem of pipelining of dependencies of an algorithmic specification has become the focus of increased research activity. We shall now present a historical perspective, and attempt to relate the results presented here to the contributions of other researchers. The earliest work on this problem was by Fortes and Moldovan in their "broadcast removal" paper [FM]. They showed that the broadcast in an array that had affine dependencies could be eliminated if one pipelined along the null space of the dependency. Although many of their techniques are valid in the general case, their definition of broadcast was somewhat restricted. They defined broadcast as the condition where a number of points depend on a single one, *and* they are all scheduled at the same time. Thus, for example, the convolution example presented in Sec. 2 does not have any broadcasts, and hence does not need any pipelining. Hence, although their work was pioneering, the significance of their results has been overlooked. Moreover, they did not address how the array (with broadcast) was derived in the first place.

In [RPF] we addressed the problem of determining affine timing functions for AREs and also described null-space pipelining (a detailed version of that paper appears as [RFc]). Multistage and control pipelining that we have described in the current paper were first recorded as [RFa] (subsequently published as [RFb]), and some early work on the taxonomy was also presented in [Raj]. The string parenthesization array has been the an interesting architecture that has been widely investigated in the literature. We shall now summarize those results.

Guerra and Melhem have presented a method to derive the systolic array for the string parenthesization problem [GM]. Given a specification that is in *canonic form* (their definition of canonic form consists of statements that have uniform dependencies but may have conditional expressions—analogous to CUREs), the standard mapping techniques may be used. Their paper addresses the problem of transforming the specification into a canonic form. Their initial specification, is more general than UREs, but different from AREs. The dependencies between a point p and a point q must be such that $p - q$ must be constant in all but one dimensions, and is arbitrary in the one dimension. This is a very awkward formalism to work with, and although the results are applicable to the string parenthesization problem, it does not generalize well.

Chen's Crystal system [Chea] takes a different approach, espousing a *general-purpose* parallel program development methodology. Although the example used in her paper is again string parenthesization, the techniques that are presented are applicable to arbitrary algorithms. However, the price for the generality is that simple techniques that we use for restricted cases (such as AREs) are

not fully utilized, but a degree of "overkill" is required. As stated in the paper, "The set of synthesis methods ...is by no means a complete set," and since the system general-purpose, "...new theorems and new insights ...[can] be integrated readily within the existing design framework." But once the library of available transformations grows, it will become extremely difficult to manage, since an exponential number of possible matches (of the template in the library, against the current program) must be tried out. This makes such a system very inefficient. We discuss a few specific problems with the approach of Crystal as follows.

The initial specification describes a set of processes over an index space, the ensemble of such processes can be depicted by a directed acyclic graph (DAG) and all the transformations are crucially linked to the DAG. Hence, determining, say, the fan-out of a node—a problem that is "solved" in a single line in the paper (see page 465)—involves determining the out-degree of a graph, an $O(n)$ problem. During the course of the synthesis, a number of such traversals of the DAG must be made. An immediate result of this is that infinite computations, such as many signal-processing applications cannot be handled (note for instance, that in the Appendix, one of the first things that the user specifies is the problem size: n = 6). Another step that Crystal must perform is to infer that the fan-out of the DAG is $O(n)$. How it does this, given just an specific instance of a family of DAGs is not explained. This problem is impossible to determine in general, without explicit user input. Moreover, each input program to the system is a separate problem, and architectures for a *family* of problems cannot be synthesized, except by going through the system for each *instance*. The approaches based on recurrences can be easily parameterized (see for example [Qui]).

Let us consider Proposition 4.2 in Chen's paper, which is used to derive an $O(n)$ algorithm (equivalent to the ARE of Eqn. 12) from a quadratic one that is similar to Eqn. 2. The proposition states that an associative (binary) operator applied to a sequence of operands may be replaced by a composition of ternary operations. Clearly, similar claims are valid for 4-ary or 5-ary operations, and since the problem size is just 6, a naive user might be tempted to use one of these transformations hoping to get a fast implementation. However, it can be seen that none of these will work. Thus, in order to use the system, the user needs to have knowledge of what the final architecture is to be. At this stage, this is not really a critical shortcoming of Crystal; after all our theory too, does not explain how the linear time algorithm is obtained. Any system that automatically provides the book-keeping functions while a user tries out a number of transformations will serve a very useful role in the design process.

The problem is that Crystal continues to use the power of such a general program transformation framework even after the linear time algorithm has been derived. The results that we have presented

indicate that constructive techniques are available for AREs,. For example, since it has been shown that systolic arrays must correspond to recurrences with uniform dependencies that have affine timing functions (see [RK]), and hence any algorithm that has super-linear speed is an overkill. Thus, there is no need to have a operator that has more fan-in than a ternary one. The same argument indicates that there is no need to consider broadcast removal schemes that are super-linear (such as a tree). Thus, a user directive to Crystal or heuristic decision made by Crystal may be avoided. Similarly, Lemma 2 that was presented in Section 3 indicates that of the linear schemes, the ones that initialize the pipe in the middle are impossible, and it is totally unnecessary to attempt such transformations.

Huang and Lengauer too, describe a system [HLa] that can transform a class of programs into systolic arrays. Although the example that they use is the algebraic path problem [HLb] it has enough irregular data flow to be of interest. Their system too suffers from the drawbacks of Crystal, by using a general theorem-prover based reasoning system, when constructive methods are available.

The system that comes closest to the results presented here is possibly the DIASTOL system of Quinton et al. In [GJQ] they describe how the string parenthesization array may be derived using their methodology. Their results were derived independently of ours and are along the same lines. The initial specification is a system of multi-linear recurrence equations (MLRE) which are not formally defined, but seem similar to AREs. A sequence of transformations is applied, eventually yielding a system of UREs (strictly speaking, they are not UREs, but have conditional expressions and are hence CUREs). Timing and allocation functions are then selected to yield the classic Guibas et al. array, as well as an improved one that has only about half the processors.* The paper does not present a *constructive theory*, but only its application to the specific problem, and thus leaves a number of issues unresolved. We describe a few of them here.

The first transformation that is used is called "middle serialization," and is similar to Chen's Proposition 4.2, in that it yields a linear time algorithm from a quadratic one. As in Crystal, such a transformation cannot be automated using simple linear algebra (as the remainder of DIASTOL seems to use) and requires extensive user interaction. The paper however, does not clearly state this, and the reader is left with the impression that all the transformations have a constructive theory underlying them.

*This improved architecture and also another similar one can be derived from the system of CUREs of Sec. 5 by the allocation functions $a(i,j,k) \equiv [i,k]$ and $a(i,j,k) \equiv [j,k]$.

The next transformation is the pipelining of the dependencies, of which two are pipelined (using *simple pipelining*) and two pipelines end at some other domain boundary which is *not* close to the desired destination. For these pipelines they use a transformation called routing (analogous to multistage pipelining) which "switches" the pipeline to a different direction. The process is explained in one line, "In order to route $c_{i,k}$ from the point $[i,k,i]$ [where it is computed] to $[i,2k-i,k]$ it can be seen that $k-i$ translations along $[0,-1,-1]$ are sufficient." There is no explanation of why/how this particular direction was chosen and how this may be automated. The observation that the second stage of the pipeline is identical to the pipeline for another dependency, is made later and is not related to the decision to route along the chosen points. As we have shown, this is a crucial part of the decision to "switch." Otherwise, why not pipeline from $[i,k,i]$ *first* along $[0,-1,0]$ for $j-i-2k$ steps (thus reaching $[i,i+j-k,i]$), and *then* along $[0,-1,-1]$ for $k-i$ steps, thus reaching $[i,j,k]$ as desired? Or any ofa number of such variants? Moreover, the paper does not discuss control pipelining at all. Clearly, the ideas are similar to our techniques, but the mathematical theory underlying the transformations had not been fully developed then (1987).

Recently, these problems have attracted considerable attention. Yaacoby and Cappello have addressed the pipelining problem [YCa,YCb] and also the problem of scheduling of AREs [YCc]. In particular, they give necessary and sufficient conditions for the existence of ATFs for a system of AREs. Note that our earlier results [RFc] gave only sufficient conditions for the existence of ATFs for AREs; by using the Yaacoby-Cappello results we can directly inform the user when an ATF does not exist for the given specification. Then it is then up to the user to specify an alternate ARE. Even more recently Roychowdhury et al. [RTRK] have presented a theory of pipelining that seems more general than ours. They consider affine dependencies, as well as affine "targets," which may permit transformations such as arbitrary rearrangements of sequences of associative operators, etc.

6.1 Efficiency Considerations and Conclusions

Before discussing efficiency conditions, we must mention the principal limitation of our method. Our initial specification is a single ARE, and not a system of AREs, and hence we are unable to address some of the more general problems. Rao et al. have introduced [Rao] Regular Iterative Algorithms (called RIAs, see also [JRK,RK]) are similar to UREs (except, that the computation at any point is not considered to be atomic and different variables at the same index point may be computed at different times on different processors). RIAs permit conditional computations, where the value of a variable at any index point may be computed by one of a finite number of

functions (like the functions g_i in CUREs), depending on some (affine) conditional expressions (like the $\psi_i(p)$ in CUREs). However, all the functions g_i are required to have the *same* dependencies, and the conditional computation is thus a restricted form of that available in CUREs. In spite of this there are a number of issues that are interesting. Rao et al. have shown that with RIAs (i.e., a *system* of recurrences with uniform dependencies), a schedule that is an affine function only of the index point is not adequate—it merely provides a coarse timing function, and the scheduling of the different functions (labeled by say integer subscripts) of the system may cause conflicts. The true timing function must be an affine function of the index point *plus* a linear function of the subscripts (subject to some renaming). The question that arises is, what happens when we construct a *system* of CUREs by pipelining a single ARE? For the initial ARE, it is perfectly adequate to consider only coarse timing functions. However, on pipelining this ARE we get a system of CUREs, and for arbitrary CUREs, coarse timing functions are inadequate. As a result, it seems that the restriction that the pipeline must be $lambda_i$-consistent may be overly pessimistic and may needlessly rule out certain transformations. Note that the main intent of this paper was to present the pipelining transformations. We believe that the theory described here can be adapted to the case of systems of AREs, if the coarse as well as the fine components of the timing functions are determined first.

We now discuss the efficiency of our algorithms. It is clear that for the case when the null space of the dependency is linear, the pipelining process is deterministic, and the pipeline, if one exists, is unique. For the higher dimensional case there is quite some leeway in the choice, but we have given a useful heuristic to restrict the choice. We also observe that all the examples that we have encountered so far have linear pipelines. The only case that is of some concern is control pipelining, where the basis is clearly $n-1$ dimensional. However this needs only a simple linear pipeline (since the domain boundary must correspond to a boundary of the processor array, and a value of 1 can be easily fed there). Hence almost any arbitrary direction will do, and it does not significantly affect the final array—the At and the AT^2 complexity remains the same, and the processor functionality is affected only in the direction of the control signal. The approaches based on recurrences thus provide a compact (finite) representation of the entire computation graph, independently of the size of the problem. This is the crucial power of the methodology. The computation of the pipelines, timing functions etc, depend on matrix operations and a search, which may seem more complicated than graph-traversal type program transformation tools, but the asymptotic complexity is certainly in favor of the recurrence based approaches.

Synthesizing a systolic array from an ARE involves three steps, namely determining an *affine timing function* (ATF), determining an allocation function, and *data pipelining* of the dependencies. These three subproblems may be solved in any order, as long as certain mutual constraints are

satisfied. However, each of these steps requires the solution of an integer constraint satisfaction problem and in general, there is no unique solution. The synthesis problem thus involves a search, and the order in which one attempts to solve the constraints can critically affect the efficiency of a synthesizer based on the theory. This problem is interesting in its own right, and is beyond the scope of this paper.

We have presented a technique for systematically deriving systolic architectures from a general class of recurrence equations. Our principal contributions have been to propose two major steps for the synthesis process (in addition to determining a timing and allocation function). In the first step the dependencies are pipelined, so that results that are required by more than one points in the domain can profitably be shared by being *pipelined* among all the points that require them. We have presented a theory for such pipelining transformations. We have shown that in the general case, this results in a recurrence equation that has uniform dependencies, but must perform a non-strict computation (governed by conditional expressions). Our second result has been the development of a technique whereby the computation of these conditional expressions may be optimized, thus yielding systolic arrays that have control signals governing the data-flow.

References

[BD] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Cas] J. W. S. Cassels. *An Introduction to the Geometry of Numbers.* Springer Verlag, 1959.

[Chea] Marina C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 3(6), December 1986.

[Cheb] Marina C. Chen. A parallel language and its compilation to multiprocessor machines for VLSI. In *Principles of Programming Languages*, ACM, 1986.

[Chec] Marina C. Chen. *Space-Time Algorithms: Semantics and Methodology.* PhD thesis, California Institute of Technology, Pasadena, Ca, May 1983.

[CS] Peter R. Cappello and Kenneth Steiglitz. Unifying VLSI designs with linear transformations of space-time. *Advances in Computing Research*, :23–65, 1984.

[DIa] Jean-Marc Delosme and Ilse C. F. Ipsen. An illustration of a methodology for the construction of efficient systolic architectures in VLSI. In *International Symposium on VLSI Technology, Systems and Applications*, pages 268–273, Taipei, Taiwan, 1985.

[DIb] Jean-Marc Delosme and Ilse C. F. Ipsen. Systolic array synthesis: computability and time cones. In Maurice Cosnard et al., editors, *Parallel Algorithms and Architectures Conference*, pages 295–312, 1986.

[FM] J. A. B. Fortes and D. Moldovan. Data broadcasting in linearly scheduled array processors. In *Proceedings, 11th Annual Symposium on Computer Architecture*, pages 224–231, 1984.

[GJQ] P. Gachet, B. Joinnault, and Patrice Quinton. Synthesizing systolic arrays using diastol. In A. Moore, W. McCabe and R. Urquhart, editors, *Systolic Arrays*, Adam Hilger, Oxford, England, 1987.

[GKT] 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*, pages 509–525, January 1979.

[GM] C. Guerra and R. Melhem. Synthesizing non-uniform systolic designs. In *Proceedings of the International Conference on Parallel Processing*, pages 765–771, IEEE, August 1986.

[HLa] Chua-Huang Huang and Christian Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6):595–632, November 1987.

[HLb] Chua-Huang Huang and Christian Lengauer. Mechanically derived systolic solutions to the algebraic path problem. In W. E. Proebster and H. Reiner, editors, *Proceedings, VLSI and Computers*, pages 307–310, IEEE Computer Society Press, 1987.

[JRK] H. V. Jagadish, Sailesh Rao, and Thomas Kailath. Array architectures for iterative algorithms. *Proceedings of the IEEE*, 75(9):1304–1321, September 1987.

[KMW] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[Kuna] H. T. Kung. Let's design algorithms for VLSI. In *Proc. Caltech Conference on VLSI*, January 1979.

[Kunb] H. T. Kung. Why systolic architectures. *Computer*, 15(1):37–46, January 1982.

[LM] M. S. Lam and J. A. Mostow. A transformational model of VLSI systolic design. *IEEE Computer*, 18:42–52, February 1985.

[LS] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1:41–68, 1983.

[LW] G. J. Li and B. W. Wah. Design of optimal systolic arrays. *IEEE Transactions on Computers*, C-35(1):66–77, 1985.

[Mol] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1):113–120, January 1983.

[MR] R. G. Melhem and Werner C. Rheinboldt. A mathematical model for the verification of systolic networks. *SIAM Journal of Computing*, 13(3):541–565, August 1984.

[MW] W. L. Miranker and A. Winkler. Space-time representation of computational structures. *Computing*, 32:93–114, 1984.

[Qui] Patrice Quinton. *The Systematic Design of Systolic Arrays.* Technical Report 216, Institut National de Recherche en Informatique et en Automatique INRIA, July 1983.

[Raj] S. V. Rajopadhye. *Synthesis, Otimization and Verification of Systolic Architectures.* PhD thesis, University of Utah, Salt Lake City, Utah 84112, December 1986.

[Rao] Sailesh Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, Information Systems Lab., Stanford, Ca, October 1985.

[RFa] S. V. Rajopadhye and R. M. Fujimoto. *Systolic Array Synthesis by Static Analysis of Program Dependencies.* Technical Report UUCS-86-0011, University of Utah, Computer Science Department, August 1986.

[RFb] S. V. Rajopadhye and R. M. Fujimoto. Systolic array synthesis by static analysis of program dependencies. In *Proceedings, Parallel Architectures and Languages, Europe*, Springer Verlag LNCS No 258, Eindhoven, the Netherlands, June 1987.

[RFc] Sanjay V. Rajopadhye and Richard M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, :Accepted for Publication, 1988.

[RFS] I. V. Ramakrishnan, D. S. Fussell, and A. Silberschatz. Mapping homogeneous graphs on linear arrays. *IEEE Transactions on Computers*, C-35:189–209, March 1985.

[RK] Sailesh Rao and Thomas Kailath. What is a systolic algorithm. In *Proceedings, Highly Parallel Signal Processing Architectures*, pages 34–48, SPIE, Los Angeles, Ca, Jan 1986.

[RPF] 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*, Springer Verlag, LNCS No 241, New Delhi, India, December 1986.

[RTRK] Vawani Roychowdhury, Lothar Thiele, Sailesh K Rao, and Thomas Kailath. On the localization of algorithms for VLSI processor arrays. Unpublished Manuscript.

[WD] U. C. Weiser and A. L. Davis. A wavefront notational tool for VLSI array design. In *VLSI Systems and Computations*, pages 226–234, Carnegie Mellon University, October 1981.

[YCa] Yoav Yaacoby and Peter R. Cappello. *Bounded Broadcast in Systolic Arrays.* Technical Report TRCS88-13, University of California at Santa Barbara, April 1988.

[YCb] Yoav Yaacoby and Peter 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.

[YCc] Yoav Yaacoby and Peter R. Cappello. *Scheduling a System of Affine Recurrence Equations onto a Systolic Array.* Technical Report TRCS87-19, University of California at Santa Barbara, Computer Science Department, Santa Barbara, Ca, February 1988.