Algebraic Transformations in Systolic Array Synthesis: A Case Study

Sanjay V. Rajopadhye
Computer Science Department
University of Oregon
Eugene, OR 97403
sanjay@cs.uoregon.edu

CIS-TR-89-12
September 6, 1989

Abstract

We describe the use of a guided, transformational approach to systolic array synthesis, using algebraic properties of the operators, such as associativity and commutativity of operands. While there are well known, constructive methods for systolic array synthesis, they do not typically permit the kinds of transformations that we illustrate. We thus expand the design space, to the point of deriving implementations where the standard techniques fail to find any. Obviously, our method needs user guidance, but we develop a characterization that enables us to avoid arbitrary program restructuring.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

1 Introduction

It has now become widely accepted that with the growing complexity of VLSI circuits, it is imperative that the design be based on a formal basis that enables the user to reason about the system in an abstract manner. Within such a framework (such as the HOL system [Gor87]), a design specification may be expressed as a theorem regarding the expected behavior of the design in a formal logical theory. An implementation can be *derived* from it by a sequence of *correctness preserving* transformations, i.e., a sequence of deductions in the theory. Such a formal approach is essential not only for the derivation of such *a priori* correct designs, but also for the *a posteriori* verification of completed designs as was done for the VIPER microprocessors [Coh87]. The advantage is that both approaches may be pursued in the *same* framework. In automating this process, we are faced with conflicting requirements. If we want the problem of determining the equivalence of any two formulæ in the theory to be decidable, we must sacrifice the expressive power of the theory. Since this is too restrictive, one usually sacrifices decidabilitiy for expressive power. As a result, a large number of transformations are applicable at any stage in the design, and it is necessary to use heuristics or user input to guide the synthesis process. In many instances however, there are more direct methods for deriving the final implementation. In software design, for example, such formal methods may be used to derive a provably correct program, which is then compiled to produce the final executable code. Although compilers are (hopefully) based on a formal semantics of the language and the translation to object code may be viewed as a sequence of transformations, this sequence is deterministic. There is thus, a well defined point beyond which no user input is required. By the same token, when we design CAD tools, we must be able to use *constructive* techniques whenever thay are available, i.e., the formal, theorem-proving approach must be used to design the *specification* itself, and should then interface to a "silicon compiler."

In the this paper we present a case study of an attempt to achieve this marriage within a unified framework. We are interested in hardware design (in particular, systolic array design) where the initial specifications is expressed as a *program* in a (declarative) language

[RF88]. We shall describe the derivation of a family of systolic arrays for ARMA (Auto Regressive Moving Average) filtering. During the design, we shall employ two distinct classes of transformations for obtaining the final implementations. The first set corresponds to a method that is now well known among the researchers in systolic array synthesis (such as [Raj89]), and the second one uses algebraic properties of the operators of the computation. The first set is based on a *constructive* theory, and may hence be automated, while the latter requires user input. Our intention here is to clearly delineate the bounds of the *constructive* methods and present a few transformations that are currently beyond them. This is of necessity a preliminary document that reports on work in progress, since our catalog of transformations is by no means complete.

As is well known, systolic arrays are a class of parallel architectures that consist of simple processors connected locally in a regular, tessellating pattern. Based on the early work of a number of researchers such as Cappello [CS84], Fortes [For83], Quinton [Qui83], and Rao [Rao85] (and many others), a theory for deriving such arrays automatically from algorithmic specifications has emerged. However, the main restriction in these methods has been that the *initial* algorithm was required to have what are called *uniform* dependencies. As a result, a large part of the design effort was spent in obtaining the problem specification itself. There was therefore, a move towards the development of techniques so that systolic arrays could be derived from a more general class of algorithms. Such techniques have been developed by Fortes and Moldovan [FM84], Quinton and Van Dongen [QV89], Rajopadhye [RF88,Raj89], Roychowdhury et al. [RTRK88], Yaacoby and Cappello [YC88] and others. They typically focus on initial specifications that have *affine* dependencies, and a common characteristic of these thechniques is that each of them attempts to perform a localization of the data dependencies of the initial algorithm. In this paper we refer to such localization techniques as *pipelining transformations*. Such a step is *essential*, since it has been shown that all systolic arrays can be *described* by algorithms that have uniform dependencies. The transformation from AREs to UREs is thus based on a well developed, constructive theory and we shall merely illustrate its use in this paper. Our thrust will be on transformations that are needed when the constructive theory is inapplicable. The rest of this paper is organized as follows. In the following section, we present an overview of the synthesis

techniques. Then in Sec 3 we describe the problem definition, and develop an ARE that we shall use as our initial specification. In Secs 4, 5 and 6 we present three classes of transformations and the systolic implementations that can be derived for the problem. Finally, we conclude by indicating directions for future work.

2 Overview of the Constructive Methodology

In some sense, all the approaches to systolic array synthesis are based on an analysis of the data dependencies of an algorithmic specification. In a typical scenario for synthesizing systolic arrays one starts with an initial algorithm that consists of the computation of a function at all points in an *index-space* (*viz* the integer lattice points in a subset of Euclidean space). A mathematical description of such an algorithm is given by the following definition.

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 $p \in D$; D is a convex hull subset of Z^n,
$q_i \in D$ 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.

A Recurrence Equation of the form defined above is called a *Uniform Recurrence Equation* (URE) if $q_i = p + \mathrm{b_i}$, for $i = 1, \ldots, k$, where the $\mathrm{b_i}$'s are constant n-dimensional vectors. It is said to be an *Affine Recurrence Equation* (ARE) if for $i = 1, \ldots, k$, $q_i = \mathrm{A_i}p + \mathrm{b_i}$, where the $\mathrm{A_i}$'s are constant $n \times n$ matrices, and $\mathrm{b_i}$'s are constant $n \times 1$ vectors.

Note that since g is a strict single-valued *function*, a system of recurrence equations is equivalent to a purely applicative program. Also, AREs properly include UREs since

the the latter are simply AREs with A_i matrices being the identity matrix. The function g is used to implicitly define the computation performed by a single processor, and thus defines the granularity of the computation.

The early work on systolic array synthesis due to Cappello and Steiglitz [CS84] and Miranker and Winkler [MW84] viewed the computation as a lattice in an index-space. The synthesis problem then consists of affine projections of this lattice on a *space-time* domain. Fortes and Moldovan used techniques of linear transformations on the dependency vectors of nested-loop algorithms [For83,Mol83]. Another similar approach was presented by Quinton [Qui83] where the problem is expressed as a URE. Li and Wah [LW85] address the question of optimality of the derived arrays. The notation used in each of these papers was completely different, and each of them introduced many subtleties. As a result, the underlying assumption of constant vector data-dependencies was not immediately clear. Rao's dissertation [Rao85] provided a unifying theme among all these approaches. He investigated the relationship between systolic arrays and a class of algorithms called Regular Iterative Algorithms (RIAs, which are very similar to UREs in that they have constant dependencies). The main idea in the synthesis of systolic arrays is that once we have an algorithm expressed as a recurrence, we assume that the function g can be computed in constant time by a processor. The synthesis then simply consists of *mapping* the *index-space* of the original recurrence to a *space-time* domain, i.e., assigning a *place* and a *time* to each point in the domain of the recurrence. It has been shown (by a number of researchers, see [RF88], [Rao85] for example) that such a mapping must necessarily be an *affine transformation* of the *index space* (i.e., independent of the function g).

As mentioned above, many researchers have investigated the problem when the initial recurrence is an ARE, and the field has now converged to a point where AREs (and systems of AREs) have been accepted as the de-facto initial specification. There is now a well developed theory for scheduling such recurrences (i.e., determining a *timing function* for them). While there exist AREs that do not admit an *affine* timing functions, this class of schedules are precisely what is needed for systolic array design. Even after one obtains an affine schedule for the ARE, the arrays that can be derived are not systolic, but have

non-local and time-dependent interconnections. Thus, an essential step is to localize the dependencies of the algorithm. It has been shown that this can be achieved by a technique called *data pipelining*, which can be viewed as a source-to-source transformation on the input algorithm characterized by the null space of the dependency matrix. While, the existence of a rank-deficient dependency matrix is a necessary condition for data pipelining, it is not sufficient. It is also important to ensure that the pipeline can be correctly *initialized*, i.e., the required data is made available to the first processor in the pipeline, and there are constructive techniques that enable us to derive the transformed recurrences automatically. We shall next illustrate these techniques and the conditions that determine when they are applicable, and then discuss the class of transformations that are required when the constructive methods are *not* applicable.

3 Problem Definition and Naive Derivation

The Auto-Regressive, Moving-Average filter problem may be stated (in the time domain) as follows. Given an infinite sequence of input values, $X_1, X_2, \ldots X_i \ldots$ and two finite sets of weights, $A_1, A_2, \ldots A_k$ and $B_1, B_2, \ldots B_k$ of equal length, k (the assumption about equal lengths can be made without loss of generality, since the smaller set can always be padded with enough zeroes to make them the same length), determine a sequence of output values, $Y_1, Y_2, \ldots Y_i \ldots$ defined by the following equation.

$$Y_i = \sum_{j=1}^{k} A_j * X_{i-j} + \sum_{j=1}^{k} B_j * Y_{i-j} \tag{1}$$

In order to obtain an ARE for this computation, we first replace the summations in the above definition by iterations, since the summations imply the use of hardware with unbounded fan-in. As we shall soon see, there is no *unique* way in which to do this, and in fact, the algebraic properties will be used to derive other provably equivalent AREs. We shall follow the convention that upper case variables refer to input/output values of

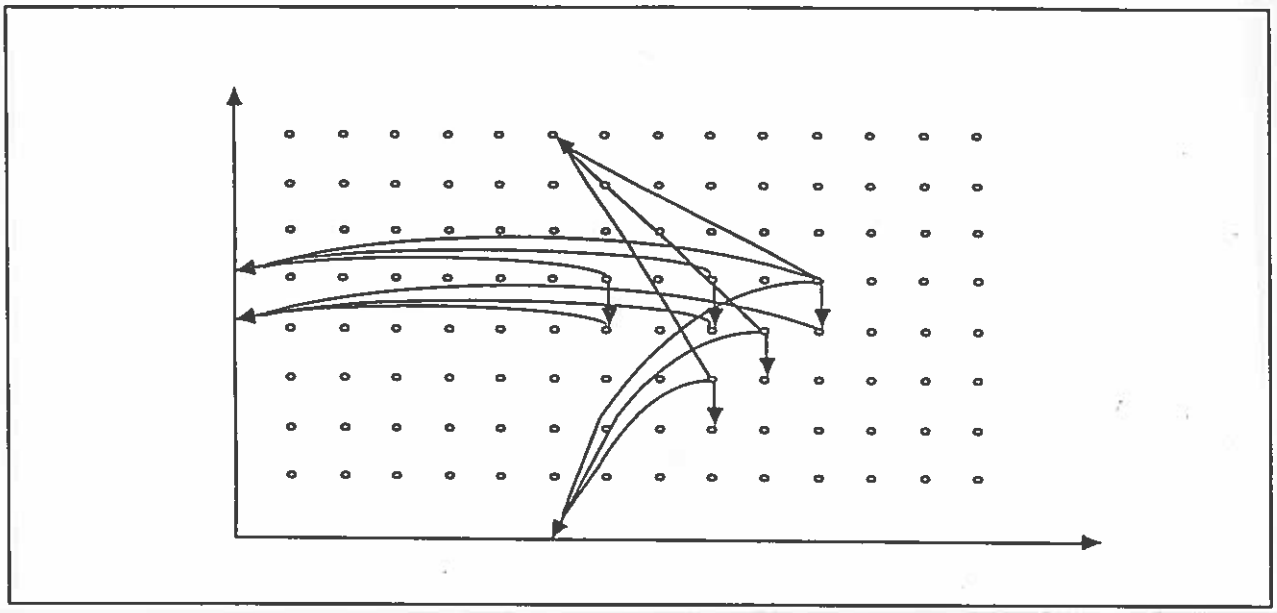

Figure 1: *Domain and Dependency Structure of the ARE of Eqn 3. For the sake of clarity, the dependencies of only a few points in the domain are shown.*

the program (and correspond directly to variables in the specification), while lower case variables are internal variables that are used by the program. We shall therefore attempt to introduce these variables in a systematic manner. In a naive approach, this is done by index j itself as the iteration dimension, yielding the following algorithm.

$$y(i,j) = y(i,j-1) + A_j * X_{i-j} + B_j * Y_{i-j} \tag{2}$$

where the boundary conditions are $Y_i = y(i,k)$ and $y(i,0) = 0$. The domain of this computation is $D = \{[i,j] \mid 0 < i, 0 < j < k\}$ and follows from the decision to use j as the iteration dimension. At each point $[i,j] \in D$, the computation requires the values of A_j, B_j, X_{i-j} and Y_{i-j}, in addition to $y(i-1,j)$. Of these, Y_{i-j} (and $y(i-1,j)$) value *must* be obtained from exactly the point where it is computed, i.e., $[i-j,k]$ ($[i-1,j]$). The other three are input values that are not *computed* by the algorithm, and *must* therefore be obtained from outside the domain, in particular, from points that are *adjacent* to the domain boundary. Let us now explore the design choices that these facts present. Since

the *length* of the A and B streams is k, we should assign them to the $i = 1$ boundary (this is the only boundary of D of the correct length). Thus we will have A_j and B_j available as* $a(0, j)$. Since the domain has two boundaries (namely $j = 1$ and $j = k$) that can have a direct correspondence with the X stream, we have two choices. Let us, for now, choose the $j = 1$ boundary, and let X_m be available as $x(m, 0)$. Eqn 2 can thus be reduced to the following, which *now* has affine dependencies, and is thus an ARE.

$$y(i, j) = y(i, j - 1) + a(0, j) * x(i - j, 0) + b(0, j) * y(i - j, k) \tag{3}$$

The boundary conditions are $y(i, 0) = 0$, $a(0, j) = A_j$, $b(0, j) = B_j$ and $x(i, 0) = X_i$. The domain and the dependency structure for the ARE is shown in Fig 1.

We observe that the computation at any point $[i, j]$ requires five arguments — an a value, a b value, an x and two y values, from the points $[0, j]$ (for *both* a and b), $[i - j, 0]$, $[i - j, k]$ and $[i - 1, j - 1]$ respectively. Of these five points, only the last one is local (i.e., a constant vector away from $[i, j]$). Hence the first step is to localize the four dependencies by data pipelining. We see that for any point, $[i, j]$, all points of the form $[i', j]$ require the *same* a (and b) arguments. Conversely, all the points that require the same a (and b) arguments as $[i, j]$ lie on the horizontal line passing through $[i, j]$. This line intersects the boundary of the domain at $[1, j]$, and one can reach this point from any point on the line by repeatedly adding the vector $-1, 0]$ (since $[-1, 0]$ is an integral *basis* for this line). Also, the point that $[1, j]$ depends on is $[0, j]$ (since A_j has been assigned as $a(0, j)$) which is simply a constant vector from it. We therefore introduce an auxiliary function that simply passes on the a (and b) values to its right neighbor, introducing a new *pipelining dependency*, $[-1, 0]$. The function is initialized at the $j = 1$ boundary from $[0, j]$, using the *terminal dependency* $[-1, 0]$. We have thus pipelined the affine dependency for $a(0, j)$ (and also $b(0, j)$).

*Note that this choice is also not unique: the points on the $i = 0$ line could be assigned *any* permutation of the A and B streams. However, as we shall see later, this does not yield any realistic design options — one of the permutations will be exactly what is required.

Similarly for the $x(i-j,0)$ argument, the points that need the same value as $[i,j]$ have the form $[i+l,j+l]$, and form a line of slope 1, which intersects the domain boundary at $[i-j+1,1]$ and $[i-j+k,k]$. Since we have decided to assign X_m to be adjacent to the $j=1$ boundary, we need to consider the former point. The value required at this point is X_{i-j}, which has been assigned to the point $[i-j,0]$, and hence this pipeline too, can be initialized with the terminal dependency, $[-1,-1]$. Based on the above discussion, we have the following system of AREs which is provably equivalent to Eqn 3, and the corresponding domain and dependency graph is shown in Fig 2.

$$y(i,j) = y(i-1,j) + f_a(i,j) * f_x(i,j) + f_b(i,j) * y(i-j,k) \tag{4}$$

$$f_a(i,j) = \begin{cases} a(i-1,j) & \text{if } i=1 \\ f_a(i-1,j) & \text{otherwise} \end{cases} \tag{5}$$

$$f_x(i,j) = \begin{cases} x(i-1,j-1) & \text{if } j=1 \\ f_x(i-1,j-1) & \text{otherwise} \end{cases} \tag{6}$$

$$f_b(i,j) = \begin{cases} b(i-1,j) & \text{if } i=1 \\ f_b(i-1,j) & \text{otherwise} \end{cases} \tag{7}$$

The procedure to pipeline the data dependencies of an ARE thus consists of the steps outlined below. In the interests of clarity, we shall use an informal manner of presentation, although it must be emphasized that these transformations are constructive and can be automated.

- Identify the points that can potentially share their arguments. This can be done by determining an integer basis for the null space of the dependency matrix. The basis vectors may not be unique, so the following constraints are used to guide the choice.
- Determine the intersection of this null space with the domain boundary. Choose the intersection point that is a constant vector fromn the desired point. The choice of this point may guide the selection of the basis vector.
- Determine the terminal dependency.

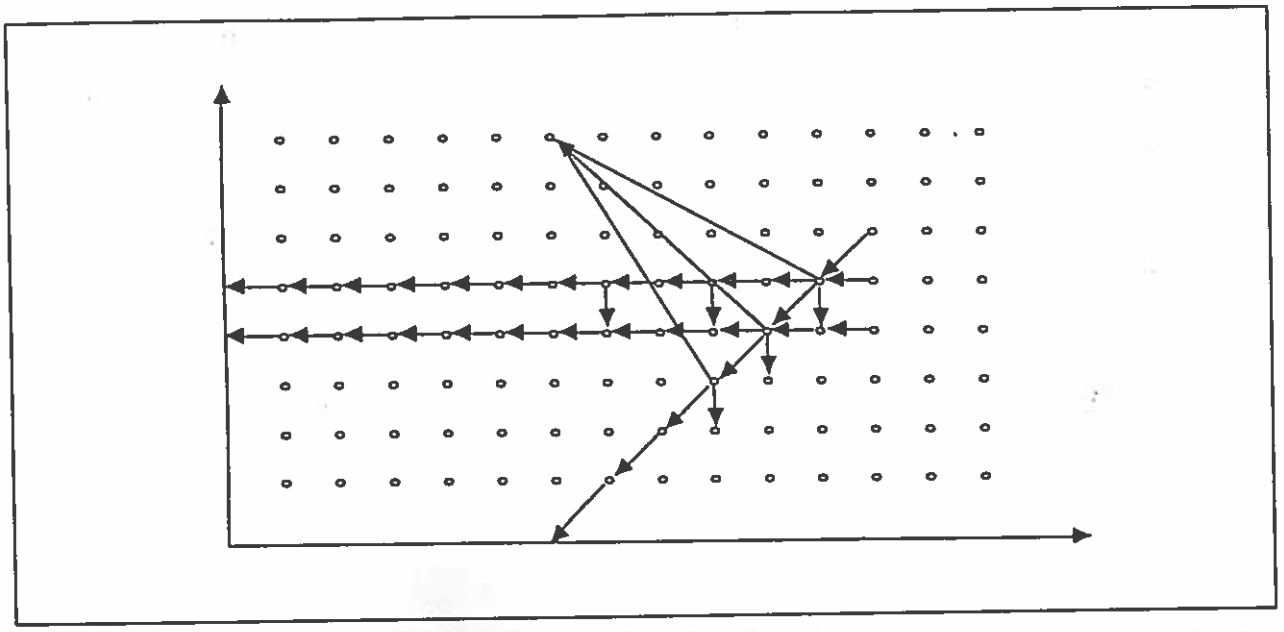

Figure 2: *Domain and Dependency Structure of the system of AREs of Eqns 4-7, showing how all but one of the dependencies can be pipelined.*

Let us see what happens when we try to pipeline the the Y dependency. The main point to note here is that Y is not an input, but a computed value and thus we do not have the freedom of manipulating its location. Thus, points that need the Y values will have to get them from where they are computed. These points (just like the $x(i-j,0)$ dependency, form the straight line of slope 1, which as we know intersects the domain at $[i-j+1,1]$ and $[i-j+k,k]$, neither of which is a *constant* vector from $[i-j,k]$ (see Fig 2). As a result it is not possible to pipeline this dependency, and a CAD system based on the standard design methods will not be able to proceed further. We must now step away from these methods and apply other transformations that are based on the algebraic properties of the computations.

4 Reversing Chains of Dependencies

The principal property of the computations that we shall use is the associativity (and occasionally commutativity) of some of the operations. One of the simplest, and possibly the most useful transformation that this permits us to perform is a reversal of some of the dependencies. It is based on the following theorem, which is a generalization of a similar result for UREs presented by Rajopadhye [Raj86] (Th. 2.2).

Lemma 1 *Let* $f(p) = g(f(\mathrm{A}_1 p + \mathrm{b}_1), f(\mathrm{A}_2 p + \mathrm{b}_2), \ldots f(\mathrm{A}_i p + \mathrm{b}_i), \ldots f(\mathrm{A}_k p + \mathrm{b}_k))$ *be an ARE where the i-th dependency* $[\mathrm{A}_i, \mathrm{b}_i]$ *is uniform (i.e.,* A_i *is the identity matrix). If g satisfies the following property:*

$$g(x_1, x_2, \ldots, x_i, \ldots, x_k) = g_1(x_i, g_2(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$$

where g_1 *is an associative and commutative binary function, and* g_2 *is an arbitrary* $(k-1)$*-ary strict function, then the ARE is equivalent to the following one.*

$$f(p) = g(f(\mathrm{A}_1 p + \mathrm{b}_1), f(\mathrm{A}_2 p + \mathrm{b}_2), \ldots f(p - \mathrm{b}_i), \ldots f(\mathrm{A}_k p + \mathrm{b}_k))$$

Proof: The proof is along the same lines as that used in [Raj86] and is omitted for brevity. ∎

Corollary 1 *If the input value for the i-th dependency is an* identity *of* g_1*, it is not necessary that* g_1 *be commutative.*

Intuitively, the above theorem enables us to *reverse* the direction of some of the dependencies in the original ARE (provided they are originally uniform or have already been pipelined). The conditions for this are that the argument corresponding to this dependency must be used "exactly once" in an associative and commutative binary function. In Eqn 3 this is true (with g_1 being addition, $g_2(a, b, x, y) = ax + by$), and we therefore have

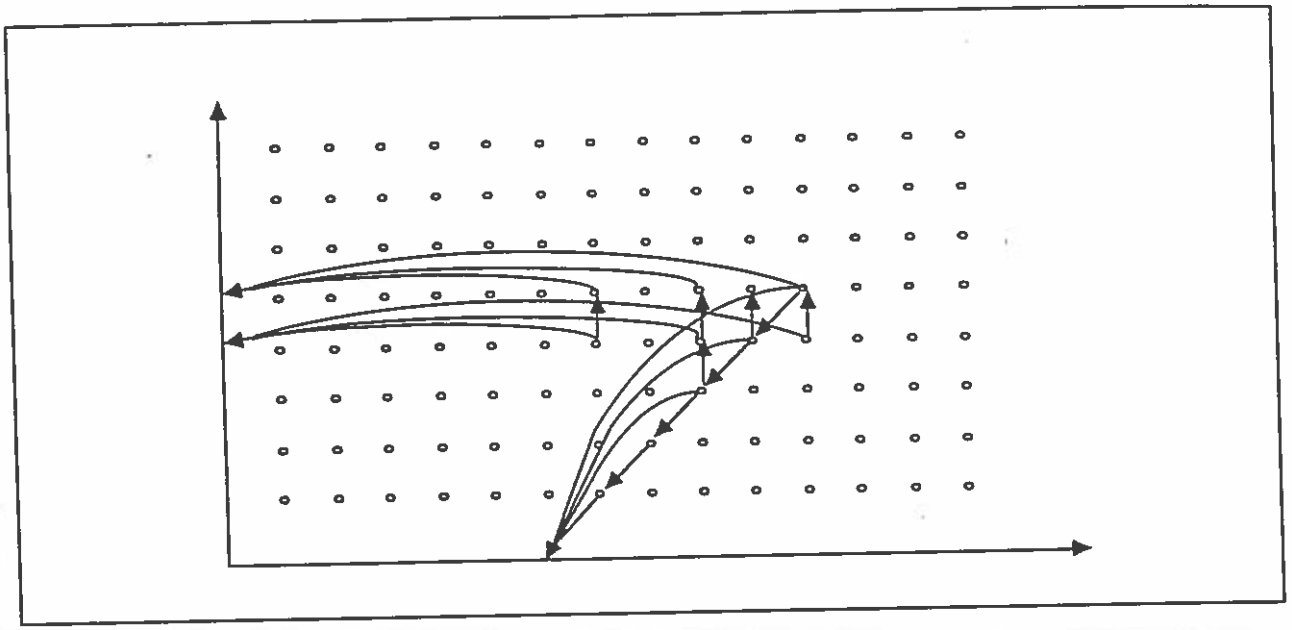

Figure 3: *Domain and Dependency Structure of the ARE with reversed dependency (Eqn 8)*

the following ARE that is equivalent to the original. Its domain and dependency structure is shown in Fig 3.

$$y(i,j) = y(i,j+1) + a(0,j) * x(i-j,0) + b(0,j) * y(i-j,1) \tag{8}$$

The boundary conditions must now be changed to $y(i,k+1) = 0$, and the results are now available as $Y_i = y(i,1)$ at the $j = 1$ boundary (other boundary conditions remain the same as before).

4.1 Deriving the Final Architecture

For this ARE, the a, b and x dependencies may be pipelined exactly as before. Moreover, we may now also pipeline the Y_{i-j} dependency as follows. As before, the set of points that may share the same value consists of the 45° line, which intersects the domain boundary at $[i-j+1,1]$ and $[i-j+k,k]$. This time, however, the required value is computed at

$[i-j,1]$ which yields a pipeline with $[-1,0]$ as the terminal dependency. We are therefore able to derive the following system of UREs.

$$y(i,j) = y(i,j-1) + f_a(i,j) * f_x(i,j) + f_b(i,j) * f_y(i,j) \quad (9)$$

$$f_a(i,j) = \begin{cases} a(i-1,j) & \text{if } i = 1 \\ f_a(i-1,j) & \text{otherwise} \end{cases} \quad (10)$$

$$f_x(i,j) = \begin{cases} x(i-1,j-1) & \text{if } j = 1 \\ f_x(i-1,j-1) & \text{otherwise} \end{cases} \quad (11)$$

$$f_b(i,j) = \begin{cases} b(i-1,j) & \text{if } i = 1 \\ f_b(i-1,j) & \text{otherwise} \end{cases} \quad (12)$$

$$f_y(i,j) = \begin{cases} y(i-1,j) & \text{if } j = 1 \\ f_y(i-1,j-1) & \text{otherwise} \end{cases} \quad (13)$$

We may now proceed with the constructive methods to derive a systolic implementation for this system of UREs. This is achieved by selecting appropriate timing and allocation functions. Of these, the allocation function is really straightforward. Since our domain is infinite, with $[1,0]$ defining its *ray*, the allocation function *must* be orthogonal to it, i.e., $a(i,j) = j$. This is the only allocation function that yields a finite array. In order to derive a timing function (in particular, the optimal one) we proceed as follows. Let $t(i,j) = li + mj + n$ denote the timing function. In order to satisfy causality of the computation, the following must hold.

$$\begin{aligned} li+mj+n &> l(i-1)+mj+n &\Rightarrow\quad l &> 0 \\ li+mj+n &> li+m(j+1)+n &\Rightarrow\quad m &< 0 \\ li+mj+n &> l(i-1)+m(j-1)+n &\Rightarrow\quad l+m &> 0 \end{aligned}$$

Solving these constraints is an integer programming problem, and the optimal timing function corresponds to the smallest integer solution, which yields $l = 2$ and $m = -1$. We must choose n such that $t(i,j) > 0 \forall [i,j] \in D$, which yields $n = 0$. Thus the optimal timing function is $t(i,j) = 2i - j$. The timing and allocation function now yield a systolic

architecture. Because of space constraints we shall not show the final implementations, and neither will we so much so much detail for the remaining arrays that we derive.

5 Selective Reversal

We may observe from Eqn 3 that, at each point in the domain, there are *two* add operations. This gives us some additional flexibility in manipulating the ARE, permitting us to re-order only a part of the ARE. The general approach is as follows. We first identify that calculating $g(x_1, \ldots, x_n)$ requires two applications of an associative and commutative operator, (i.e., three operands are involved), one of them being a single, uniform dependency, and the other two using *disjoint* sets of arguments. Next, we determine the straight line for which the uniform dependency is a basis vector. We then determine a geometric (renaming) transformation that reverses the points on this line, and apply it to *only one of the operands*. Let us now discuss each of these steps in more detail.

Let $f(p) = g(f(\mathrm{A}_1 p + \mathrm{b}_1), f(\mathrm{A}_2 p + \mathrm{b}_2), \ldots f(\mathrm{A}_i p + \mathrm{b}_i), \ldots f(\mathrm{A}_k p + \mathrm{b}_k))$ be an ARE. Assume that one of the dependencies (without loss of generality, the first one) is uniform (A_1 is the identity matrix). Assume that g can be expressed as follows:

$$g(x_1, \ldots, x_k) = g_1(x_1, g_1(g_2(x_2, \ldots, x_i), g_3(x_{i+1}, \ldots, x_k)))$$

where g_1 is associative and commutative. Consider the "output" points of the domain (points such that $f(p)$ is an output value), and the line parallel to b_1 passing through them. If the ARE is well formed, this line *must* intersect another domain boundary, and thus form a line segment. We consider the transformation, $\mathcal{T}$ that reverses this line segment, i.e., $\mathcal{R}(p)$ maps p to its mirror image p'. We then have the following proposition.

Remark 1 *Given the conditions above, the following ARE is equivalent to the original.*

$$\begin{aligned} f(p) \;=\; & g(f(\mathrm{A}_1 p + \mathrm{b}_1), f(\mathrm{A}_2 p + \mathrm{b}_2), \ldots f(\mathrm{A}_i p + \mathrm{b}_i), \\ & f(\mathrm{A}_{i+1}\mathcal{R}(p) + b_{i+1}), \ldots f(\mathrm{A}_k \mathcal{R}(p) + b_k)) \end{aligned}$$

Proof: Omited for brevity. The intuition underlying the proof is that because of associativity and commutativity of the operation, we may reorder the computations arbitrarily. In particular, we are permitted to pair the g_3 part from point $\mathcal{R}(p)$ together with the g_2 part of p. ∎

Let us see how this transformation may be applied to our filtering ARE (Eqn 3). We see that the first dependency is uniform $[-1, 0]$, g_1 is addition, g_2 is ax and g_3 is by, and they use disjoint sets of arguments, as required. The line parallel to $[-1, 0]$ intersects the domain at $[i, 1]$ and $[i, k]$ and thus yields a vertical line segment. The geometric transformation that reverses the points on this line is given by $\mathcal{R}([i, j]) = [i, k-j+1]$, i.e., simply a replacement of j by $k - j + 1$. This yields the following equivalent ARE for the computation.

$$y(i,j) = y(i,j-1) + a(0,j) * x(i-j,0) + b(0,k-j+1) * y(i+j-k-1,k) \qquad (14)$$

We now see that two dependencies have been changed. To pipeline the $y(i+j-k-1, k)$ dependency, we see that the points that require the same value form the line of slope -1, and this line intersects the domain boundary at $[i+j-k, k]$ and $[i+j-1, 1]$. Of these, the former is adjacent to $[i+j-k-1, k]$ and we may therefore initialize the pipeline correctly. However, there is now a problem with the $b(0, k-j+1)$ dependency. Its pipeline consists of the points parallel to the i-axis, and intersects the domain boundary at $[1, j]$. However, the value required here is B_{k-j+1} which has been assigned to $b(0, k-j+1)$. However, remember that B is an input, and we had a number of choices when we assigned it to the boundary. Hence we may simply reverse the B vector, and obtain a correctly initialized pipeline. We will thus have the following system of UREs, whose dependency structure is shown in Fig 4. Following the standard synthesis methods, final array may be obtained for it. For complete details of the derivation, the reader is referred to [RMK89]

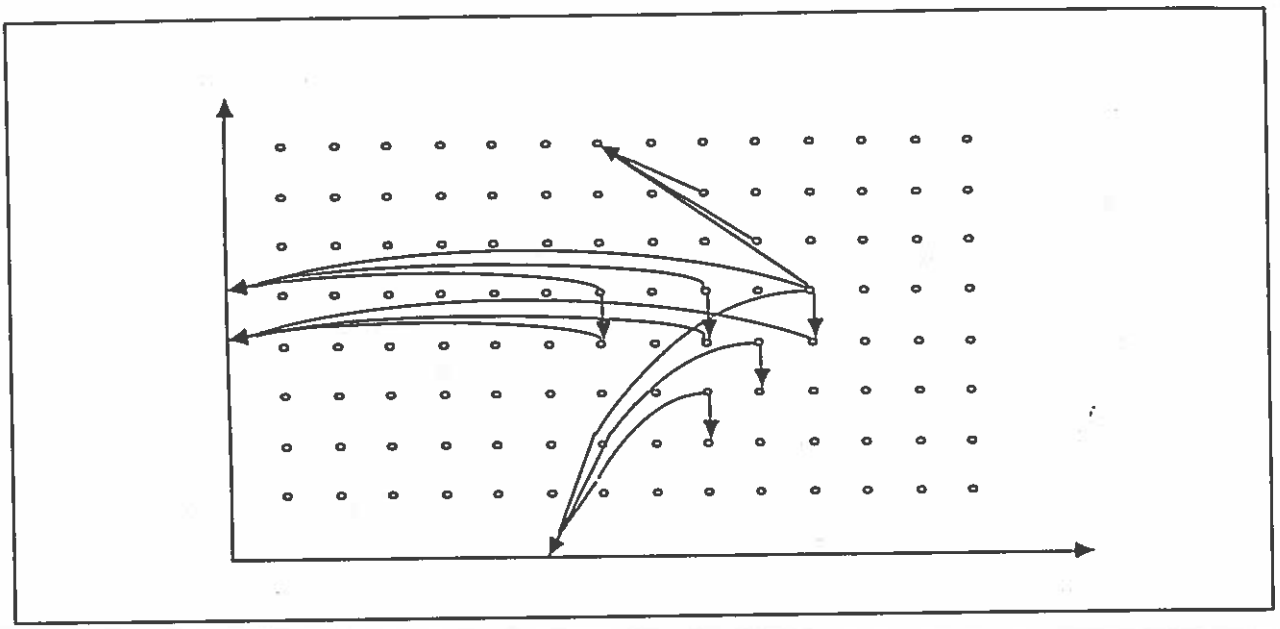

Figure 4: *Dependency Structure for the System of AREs of Eqns 15-19. The x-pipeline will have a 45° slope and the y-pipeline will have a −45° slope.*

$$y(i,j) = y(i,j-1) + f_a(i,j) * f_x(i,j) + f_b(i,j) * f_y(i,j) \tag{15}$$

$$f_a(i,j) = \begin{cases} a(i-1,j) & \text{if } i = 1 \\ f_a(i-1,j) & \text{otherwise} \end{cases} \tag{16}$$

$$f_x(i,j) = \begin{cases} x(i-1,j-1) & \text{if } j = 1 \\ f_x(i-1,j-1) & \text{otherwise} \end{cases} \tag{17}$$

$$f_b(i,j) = \begin{cases} a(i-1,j) & \text{if } i = 1 \\ f_b(i-1,j) & \text{otherwise} \end{cases} \tag{18}$$

$$f_y(i,j) = \begin{cases} y(i-1,j) & \text{if } j = k \\ f_y(i-1,j+1) & \text{otherwise} \end{cases} \tag{19}$$

6 A Folding Transformation

We shall now present another transformation that uses similar algebraic properties of the computations. We proceed as before, and detect whether an associative-commutative function g_1 is applied at the topmost level (in g) and that one of its operands is an argument that is propagated along a dependency (wlog, the first one, $\mathrm{A}_1, \mathrm{b}_1$) that is already uniform. The second operand of g_1 should be a function g_2 of the remaining arguments. Thus,

$$g(x_1, \ldots, x_n) = g_1(x_1, g_2(x_2, \ldots, x_n))$$

We also determine the *dependency chain* corresponding to the uniform dependency (i.e., the line segment that has the dependency as a basis vector). Now, instead of reversing the points on this chain, we shall fold the chain onto itself, thus obtaining a smaller, denser domain. Once again, the transformation is a geometric one that maps, in this case, two points, say p and p' in the original domain to a single point, s, in the target domain. One of these, say, p, is identical to s, and the other one is a function of p, b_1 and the original domain. Let us denote this function by $\mathcal{F}(p)$. We then have the following remark.

Remark 2 *Given the conditions above, the following ARE is equivalent to the original.*

$$\begin{aligned} f(p) \;=\; & g_1(f(p + \mathrm{b}_1), g_1(g_2(f(\mathrm{A}_2 p + \mathrm{b}_2), \ldots f(\mathrm{A}_k p + \mathrm{b}_k)), \\ & g_2(f(\mathrm{A}_1 \mathcal{F}(p) + b_1), \ldots f(\mathrm{A}_k \mathcal{F}(p) + b_k)))) \end{aligned}$$

Let us now apply this transformation to the ARE of Eqn 8. We see that the transformation must once again be applied to the vertical line segment $[i, 1]$ to $[1, k]$. We may easily determine that the function $\mathcal{F}([i, j]) = [i, k - j + 1]$. Hence, the following ARE is equivalent to the original specification.

$$\begin{aligned} y(i,j) \;=\; & y(i, j+1) + a(0,j) * x(i-j, 0) + b(0,j) * y(i-j, 1) + \\ & a'(0, k-j+1) * x(i+j-k-1, 0) + b'(0, k-j+1) * \\ & y(i+j-k-1, 1) \end{aligned} \tag{20}$$

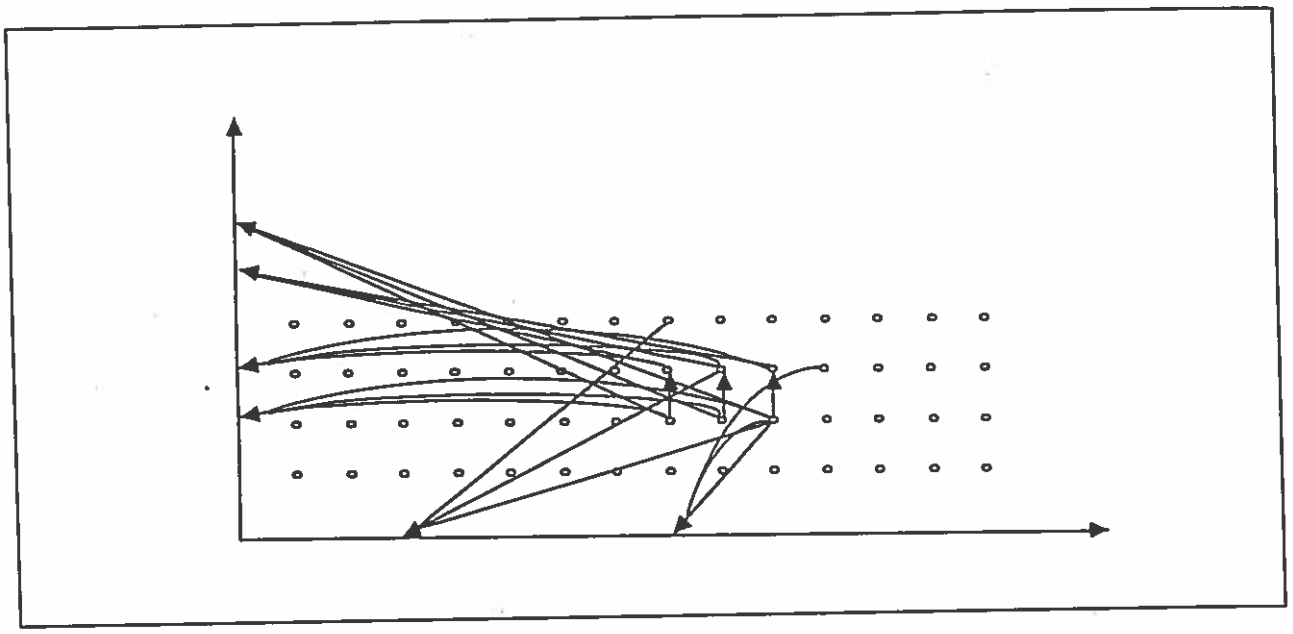

Figure 5: *The dependency structure of the folded ARE: Eqn 20*

However, the domain is reduced to half (see Fig 5), and thus $[0, k-j+1]$ is no longer on the boundary. In fact, it is necessary to also fold the input arrays, and so there should be two a-values (as well as two b-values) input at each boundary point. The above equation reflects this, and the boundary conditions are now $a(0,j) = A_j$, $a'(0,j) = A_{k-j+1}$, $b(0,j) = B_j$, $b'(0,j) = B_{k-j+1}$, $y(i,k/2) = 0$ and $x(i,0) = X - i$. and the output, Y_i is computed as $y(i,0)$. The domain and dependency structure of this ARE is shown in Fig 5, and once again, a new array is obtained for the system of recurrences. This derivation uses another *constructive* technique called multistage pipelining [Raj89], and the final array has the x-values recycled into the array from one end. As before, once the system of UREs is derived, the final array is obtained *automatically*.

7 Conclusions and Future Work

We have presented a case study of the application of a number of semantics preserving transformations to the systematic design of systolic arrays. The transformations all use

only a simple algebraic property — associativity and commutativity of the operators. In spite of this, the transformations that we have do not form a complete set by any means: many other transformations are possible. They are more thoroughly explored in [RMK89], where a number of new systolic implementations are derived by the use of these and similar transformations.

We note that the transformation used in Sec 4 is actually a special case of the selective reversal transformation (Sec 5). In fact, this may be verified by applying the selective transformations, successively to the ARE of Eqn 3. The first application should be exactly as in Sec 5, and the second one to the ax product term. This will cause the x-pipeline to consist of the line with slope -1, and this intersects the domain boundary at $[i-j, k]$. It is therefore necessary to reassign the x input to the $j = k$ boundary in order to correctly initialize this pipeline. This is easy to since X is an input stream. It should be easy to verify that this yields an ARE that is merely an α-substitution for Eqn 8. Another interesting observation is that although the transformations are based on algebraic properties of the operators in g, they are eventually expressed as simple geometric manipulations of the *domain* of the original ARE.

Once we have identified the algebraic properties that serve as the basis of our transformations, there are two crucial issues that must be addressed if such transformations are to be included in a CAD tool framework. The first question is whether these are general transformations, or they are simply a "bag of tricks" that work for the example that was used to illustrate them. We feel that since associativity and commutativity are fairly universal algebraic properties that may be found in many algebraic varieties, this is not the case.

The second issue deals with automation. It should be intuitively clear that since the operator is associative and commutative, *any* permutation of the *sequence* should yield a valid computation (as well as other rearrangements that do not correspond to just a permutation, such as trees, etc.). However, having such a large catalog of transformations will make any system either very inefficient, or else, put too much burden on the user.

It is therefore imperative that the set of applicable transformations be pruned to those that are useful in the context of systolic array design. But we know that the *constructive methods* for designing systolic arrays require AREs as their initial specifications. Hence the transformations that we provide in our catalog must maintain an affine dependency structure. We see that both $\mathcal{R}$ and $\mathcal{F}$ do this. The primary reason is that $\mathcal{R}$ and $\mathcal{F}$ are *affine functions*. Since the transformations can be reduced to purely geometric manipulations of the *domain of computation*, the new dependency will be of the form $A_i(T(p)) + b_i$ if $\mathcal{T}$ is the transformation. For this to be affine, it is sufffficient for $\mathcal{T}$ itself to be an affine function. Hence we have a powerful characterization of the transformations that we should permit in our catalog. It rules out structures like broadcast trees, etc., which is precisely what we desire.

The open problems that remain to be addressed are firstly to develop a fairly complete catalog of such transformations. Moreover, we would like to investigate properties that are not simply associativity and commutativity, but other general algebraic properties of the programs. One specific example of this is the case when there are two "top-level" operators in the program. We wil therefore not be able to perform the kind of *arbitrary* restructuring that was possible here. We are investigating the conditions that such functions must satisfy so that *afffine* restructuring is possible.

7.1 Acknowledgments

The author wishes to thank Moataz Mohamed for working out the details of the final architectures, Ben Manuto forpainstakingly drawing the figures in this paper, Dave Keldsen for long editorial, and Sayfe Kiaei for introducing a problem which turned out to have many interesting ramifications in spite of the fact that it has been extensively studied.

References

[Che86] Marina C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 3(6):461–491, December 1986.

[Coh87] Avra Cohn. A proof of correctness of the VIPER microprocessors: The first level. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Calgary, Alberta, Canada, January 1987. Kluwer Academic Publishers.

[CS84] Peter R. Cappello and Kenneth Steiglitz. Unifying VLSI designs with linear transformations of space-time. *Advances in Computing Research*, 2:23–65, 1984.

[FM84] Jose 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.

[For83] Jose A. B. Fortes. *Algorithm Transformations for Parallel Processing and VLSI Architecture Design.* PhD thesis, University of Southern California, Los Angeles, Ca, 1983.

[Gor87] Michael J. C. Gordon. HOL: A proof generating system for higher order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Calgary, Alberta, Canada, January 1987. Kluwer Academic Publishers.

[LW85] G. J. Li and B. W. Wah. Design of optimal systolic arrays. *IEEE Transactions on Computers*, C-35(1):66–77, 1985.

[Mol83] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1):113–120, January 1983.

[MW84] W. L. Miranker and A. Winkler. Space-time representation of computational structures. *Computing*, 32:93–114, 1984.

[Qui83] Patrice Quinton. The systematic design of systolic arrays. Technical Report 216, Institut National de Recherche en Informatique et en Automatique INRIA, July 1983.

[QV89] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, 1989. To appear.

[Raj86] Sanjay V. Rajopadhye. *Synthesis, Otimization and Verification of Systolic Architectures.* PhD thesis, University of Utah, Salt Lake City, Utah 84112, December 1986.

[Raj89] Sanjay V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, pages 88–105, May 1989.

[Rao85] Sailesh Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, Information Systems Lab., Stanford, Ca, October 1985.

[RF88] Sanjay V. Rajopadhye and Richard M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, page Accepted for Publication, 1988.

[RMK89] Sanjay V. Rajopadhye, Moataz Mohamed, and Sayfe Kiaei. New systolic arrays for ARMA filters. Technical Report CIS-TR-89-??, University of Oregon, Computer Science Department, 120 Deschutes Hall, Eugene, Or 92403-1202, August 1989. submitted.

[RTRK88] Vawani Roychowdhury, Lothar Thiele, Sailesh K Rao, and Thomas Kailath. On the localization of algorithms for VLSI processor arrays. In Robert W. Brodersen and Howard S. Moscovitz, editors, *VLSI Signal Processing, III*, pages 459–470, Monterey, Ca, November 1988. IEEE Accoustics, Speech and Signal Processing Society, IEEE Press. A detailed version is submitted to IEEE Transactions on Computers.

[YC88] 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. See also, UCSB Technical Report TRCS87-18, February 1988.