

SASCOM: An Automatic CAD Tool for the Generation of Systolic Arrays

Ivan Chung Wing Wong

**CIS-TR-90-08
April 6, 1990**

Abstract

SASCOM, an automated CAD tool for the design of systolic architecture is presented. It is built to automate the process of designing systolic architectures. Starting from a problem specified as systems of Affine Recurrence Equations (ARE), SASCOM can automatically generate all systolic implementations for the problem. The mechanics behind its ability to generate systolic architectures include (1) data pipelining of non-uniform dependencies, (2) generation of all valid and distinct space-time transformation functions, and (3) derivation of the final architecture specifications from the transformation functions. The fundamental theory and algorithms to perform each of these tasks are described in detail. The system currently runs on the Macintosh-II under the MacScheme programming environment. It has a front end editor and a back end graphical simulator. The interface to these two parts will be briefly described.

Also included in the thesis is a new theory on the number of systolic architectures derivable for a problem specification. By analyzing a few constraints which characterize systolic arrays, an upper bound on the number of distinct systolic architectures is derived. This upper bound is found to be depended only on the dimension of the input recurrence. The result of this theory is incorporated into SASCOM to make it a more complete automatic CAD tool as compared to other existing systems.

Department of Computer and Information Science
University of Oregon

**SASCOM: AN AUTOMATIC CAD TOOL FOR
THE GENERATION OF SYSTOLIC ARRAYS**

**Ivan Chung Wing Wong
Computer Science Department
University of Oregon
Eugene, Or 97403**

Abstract

SASCOM, an automated CAD tool for the design of systolic architecture is presented. It is built to automate the process of designing systolic architectures. Starting from a problem specified as systems of Affine Recurrence Equations (ARE), SASCOM can automatically generate all systolic implementations for the problem. The mechanics behind its ability to generate systolic architectures include (1) data pipelining of non-uniform dependencies, (2) generation of all valid and distinct space-time transformation functions and (3) derivation of the final architecture specifications from the transformation functions. The fundamental theory and algorithms to perform each of these tasks are described in detail. The system currently runs on the Macintosh-II under the MacScheme programming environment. It has a front end editor and a back end graphical simulator. The interface to these two parts will be briefly described.

Also included in the thesis is a new theory on the number of systolic architectures derivable for a problem specification. By analyzing a few constraints which characterize systolic arrays, an upper bound on the number of distinct systolic architectures is derived. This upper bound is found to be depended only on the dimension of the input recurrence. The result of this theory is incorporated into SASCOM to make it a more complete automatic CAD tool as compared to other existing systems.

TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION.....	1
2. SURVEY OF EXISTING SYSTEMS.....	5
2.1 SDEF.....	6
2.2 Crystal.....	7
2.2 ADVIS.....	7
2.3 STEP and PLACE.....	8
3. SYNTHESIS OF SYSTOLIC ARRAYS: A THEORY.....	9
3.1 Specifying a Problem as an ARE.....	9
3.2 Pipelining Non-Uniform Dependencies.....	11
3.3 Finding Timing Functions.....	13
3.4 Finding Allocation Functions.....	14
4. ALLOCATION FUNCTION AND ITS IMPLICATIONS:	
NUMBER OF ALL SYSTOLIC ARCHITECTURES.....	16
4.1 Timing Function.....	16
4.2 Allocation Function.....	17
4.3 Dense Array Constraint.....	21
4.4 Nearest-Neighbor Interconnection Constraint.....	22
4.5 The Total Number of Systolic Architectures.....	24
5. THE SOFTWARE: SASCOM.....	28
5.1 Systolic Array Generation.....	29
5.2 Algorithm 1: Pipelining the Affine Dependencies.....	31
5.3 Algorithm 2: Finding an Optimal Timing Function.....	33
5.4 Algorithm 3: Finding All the Possible Allocation Functions.....	34
5.5 Algorithm 4: Building the Final Architecture Specification.....	36
5.6 An Example: LU-Decomposition.....	42
5.7 Conclusion.....	50

Chapter	Page
APPENDIX	
A. DERIVATION OF THE BOUND ON THE NUMBER OF SYSTOLIC ARCHITECTURES.....	
	52
B. BNF OF THE INPUT SPECIFICATION LANGUAGE.....	
	57
C. TARGET ARCHITECTURE SPECIFICATION.....	
	58
REFERENCES.....	
	61

LIST OF FIGURES

Figure	Page
1.1. A Diagrammatic View of Systolic Computing.....	1
1.2. Overall Structure of SASCOM.....	3
4.1. Two Arrays That are Isomorphic to Each Other.....	19
5.1. The Data flow Diagram of the Systolic Array Generator.....	29
5.2. The Dialog Box Requesting the User for the Equation of a Boundary Plane.....	46
5.3 The Dialog Box Showing the Choices of Valid and Distinct Architectures Generated for the Problem.....	49

CHAPTER 1

INTRODUCTION

A systolic array is defined in the most general sense as a network of (identical) simple processors that circulate data in a regular fashion [8,9]. This type of architecture derives its computation power by organizing a large number of processing units to achieve the effects of pipelining in one or more directions. A diagrammatic view of the systolic concept is shown in figure 1.1.

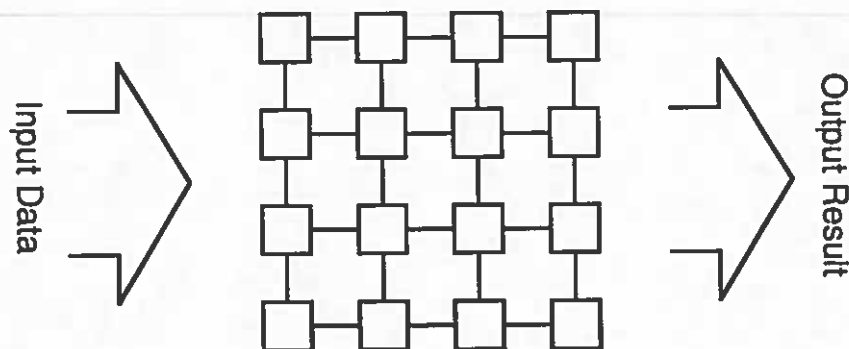


Figure 1.1 A diagrammatic view of systolic computing.

As shown in the figure, a stream of data is pipelined into a network of processors. Once inside the network, the data stream is kept flowing rhythmically and is massaged by each processing element along the way. Thus the array behaves much like an automobile assembly line. While an automobile assembly line is linear, systolic systems can be two-dimensional; thus allowing a higher degree of parallelism.

Powerful as it is, this computing paradigm has had somewhat limited success partly because of the tedious process involved in designing one. Many researchers have reported systematic design methodologies to derive systolic architectures [2,3,4,10,13,15,16,17]. The shift from using ad hoc methods to some systematic design methodologies has facilitated

the design process. However, manually applying these methodologies is still quite tedious. A CAD tool that can fully automate this design process is needed. SASCOM is built to meet such a need. Starting from a problem specified as an Affine Recurrence Equation (ARE), SASCOM can generate systolic implementations by performing a three step process -- (1) data pipelining, (2) linear mapping to space-time (3) generating architecture descriptions.

There are three major issues involved in designing SASCOM:

(1) Choice of Input specification language

Two criteria are used in choosing an input specification language for SASCOM: (i) it should be powerful enough to be able to specify all problems for which systolic implementations can be derived; (2) It should be a high level language to provide an abstraction from the hardware to algorithm level. Systems of Affine Recurrence Equations (ARE) are adopted since it satisfies both criteria.

(2) Choice of systematic design methodology

The choice of a design methodology and the input specification language are closely related. Rajopadhye [16] and other researchers has laid out a systematic approach to generate valid transformation functions from an ARE specification. This approach can be characterized by the use of data pipelining to transform affine dependencies into uniform dependencies; followed by the use of linear mapping functions to map computations from the problem domain to the space-time domain. This approach is adopted by SASCOM. In addition, a theory on the upper bound on the number of possible systolic implementations is developed. This bound depends only on the number of dimensions of the input recurrence. This result is incorporated into SASCOM giving it the power to generate all systolic implementations for any given problem.

(3) Optimal systolic implementation

The cost of a systolic architecture can be roughly divided into two parts: (1) execution time and (2) hardware requirements. Researchers have been able to treat these two types of cost independently and derive more than one optimality criteria [7,10,21,22,23] for each of

them. SASCOM uses execution time as the primary criterion (i.e. the number of clock cycles needed for the completion of the algorithm). Using a linear programming algorithm, SASCOM derives architectures with optimal execution time. The optimality criterion for hardware requirements is more involved. In general, hardware cost can be characterized by the processor counts and the hardware complexity of individual processors. However, there is no consensus as to how the latter cost should be measured. SASCOM's approach is to generate all systolic implementations with their associated processor counts as cost measures. Presented with all available systolic implementations, a user can then choose an optimal one based on his/her own optimality criterion for hardware complexity. This approach is practical since the number of implementation is not very large.

The front end of SASCOM is an editor that permits the user to edit and execute input specifications at the algorithm level. The back-end of the system is a graphical simulator to simulate the final architecture. A diagrammatic view of SASCOM is shown in figure 1.2.

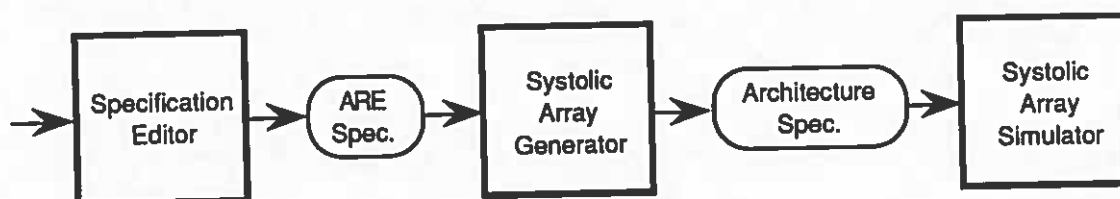


Figure1.2 Overall structure of SASCOM

As shown in the figure, an ARE input specification is fed into the systolic array generator. The result is an architecture specification which can be input to the graphical systolic array simulator. The systolic array simulator can also be run independently by manually supplying an architecture specification. SASCOM therefore serves as an integrated software package for the design of systolic arrays starting from editing and verifying problem specification to generation and simulation of systolic architectures.

The following chapters of this thesis will cover all major components of SASCOM. We will start the discussion in chapter two by looking at some other CAD systems that perform similar tasks. Chapter three will present the theoretical framework behind SASCOM. The topics

include data pipelining and mapping functions. Chapter three will discuss the upper bounds on the number of possible systolic architectures for any given problem. This bound is found to depend only on the number of dimension of the input recurrence. Based on this theory, SASCOM is able to find all systolic architectures for any given problem. Finally, in chapter four, we will present the overall structure of SASCOM, the algorithms used and some development notes.

Finally, I would like to thank several people who made this project possible. SASCOM is the result of the collective effort of a group of four. With Ben Manuto working on the simulator and Ed Kilham on the input editor, I could focus all my attention on developing the systolic architecture generator. And of course, without the help, the numerous discussions and supervision of Professor Sanjay Rajopadhye, the project would never have been completed.

CHAPTER 2

SURVEY OF EXISTING SYSTEMS

There are five major aspects which set SASCOM apart from other systems in the research community:

(1) Input problems are specified as a system of Affine Recurrence Equations (ARE). As pointed out by Rajopadhye [16], ARE's provide a higher level notation to specify problems than Uniform Recurrence Equations (URE's) [6,15].

(2) The process of data pipelining is almost completely automatic. With data pipelining, an ARE specification can be turned into an equivalent system of URE's. This enhances the generation of linear transformation mapping functions.

(3) SASCOM uses a linear programming algorithm to generate optimal timing functions.

(4) All valid and distinct allocation functions (processor space mapping functions) are generated for purely systolic interconnections. In chapter three, we will present a bound on the number of systolic implementations for any given problem. Based on this model, the cost of all systolic implementations of a problem can be pre-computed before an optimal architecture is selected.

(5) A hardware specification of the target architecture is generated as an output for simulation.

In the remainder of this chapter, we will look at four other existing systems. All these systems aim at assisting the systolic array design process. A survey of these systems can justify the need for a more complete CAD tool.

2.1 SDEF

The SDEF system due to Engstrom and Cappello [5] is a software tool that consists of an editor, a translator, a domain type database, and a systolic array simulator. It is used as a tool to express and execute systolic algorithms. The input to the system is an SDEF specification which specifies four properties of systolic architecture: the index set (S), domain dependencies (D), space-time embedding (E) and process function (F). A systolic algorithm specified in this SDEF language can be translated by the translator into C programs and finally run by a simulator.

The four properties (SDEF) identified by Engstrom and Cappello roughly correspond to the four terms we used in our system -- problem domain, data dependencies, timing-allocation functions and processor executable code. There is almost a one-to-one correspondence between the SDEF specification and our way of specifying final systolic architectures. Such a coincidence is by no means arbitrary. As Engstrom and Cappello have pointed out, by breaking down the specification into four parts, a systolic algorithm can be specified in sufficient detail effectively. Engstrom and Cappello have also discussed the difference between the SDEF specification language and other languages such as W2 [1]. Unlike W2, SDEF specifies processor interconnections apart from the program's computation constructs. Our system also adopts such a scheme. While W2's choice reflects the general-purpose, message passing capabilities of their systems, SDEF and SASCOM reflects the traditional view that a systolic architecture is problem specific and is identified by its own unique processor interconnection pattern.

The difference between the SDEF and our system is the emphasis. While SDEF aims to provide a clean way to specify systolic algorithms, our system is built as an automated CAD tool to accept mathematical specifications and *generate* all possible systolic architectures based on the constraints we set.

2.2 Crystal

The Crystal programming environment by Chen [3] provides a general purpose, executable language to specify parallel algorithms. Systolic algorithms, being a type of parallel algorithms, can be specified as recurrence equations in Crystal. Chen identified a three-step process to transform an input specification to an architecture specification -- (1) reducing fan-in and fan-out degrees, (2) performing space-time mapping and (3) optimizing control signals. Crystal does not perform these transformations automatically.

A distinguished feature of Crystal is the use of space-time recursion equations (STREQ) to specify target systolic architectures. A STREQ is a recurrence equation obtained by renaming the indexes of the original specification from the problem to the space-time domain by means of space-time mapping functions. Such an architecture specification is desirable since it maintains a direct correspondence between the input problem specification and the target architecture specification. SASCOM on the hand maintains a distinction between a higher level problem specification (ARE) and a lower level architecture specification (hardware). The lower level architecture specification is designed to be compatible with the simulator which can be used as a stand-alone application.

The main difference between Crystal and SASCOM is again the emphasis. Parallel algorithms can be specified and executed in the Crystal system. But unlike SASCOM, it does not perform any transformation automatically to generate systolic architectures.

2.3 ADVIS

The ADVIS system by Moldovan [12] is one of the pioneer software systems for the design of systolic arrays. The mathematical foundation of the system is based on the transformation of algorithms from sequential forms to suitable parallel forms [13]. Systolic architectures are derived by means of mapping from the problem domain to the space-time domain using linear transformation functions. These transformation functions, P and S are semantically the same as our timing and allocation functions.

A major breakthrough of ADVIS is its ability to automatically generate transformation functions and hence architecture mappings by solving linear diophantine equations. Using this approach, ADVIS is capable of generating valid P and S transformation mappings. ADVIS however does not offer a performance index to optimize the P functions. Moreover, it does not distinguish between identical systolic arrays that could be generated by distinct S functions.

2.4 STEP and PLACE

Kothari, Gannett and Oh in their paper [7] have mentioned a system to design systolic architectures. Their prototype is able to generate STEP and PLACE mapping functions (corresponding to timing and allocation functions respectively) for a given problem specification. Based on a constraint similar to our nearest-neighbor interconnection constraint, they were able to come up with an upper bound on the number of possible systolic architectures for a given problem. They however do not provide a model to differentiate between non-distinct architectures.

It should also be noted that none of the above systems automatically performs data pipelining. By allowing users to specify problems in ARE's, SASCOM provides a high level specification language for systolic algorithms. However, since not all problems specified in ARE can derive systolic architectures, data pipelining is needed to transform the ARE specification into an equivalent URE specification. Another important feature of SASCOM is its ability to generate all systolic architectures for a given problem. Such a feature is lacking in the above three systems.

CHAPTER 3

SYNTHESIS OF SYSTOLIC ARRAYS: A THEORY

Till now, we have been casually using terms such as "pipelining" and "mapping functions". In this chapter, we will clearly define these terms. In addition, we will lay out the theoretical foundations of the design methodology adopted by SASCOM. This methodology is based mainly on the Affine Recurrence Equations (ARE) model described by Rajopadhye [16].

Before we go into details, let's first briefly outline the design methodology:

- i) Specify the problem in form of a system of ARE's.
- ii) Pipeline the non-uniform dependencies in the problem specification.
- iii) Find a valid timing and allocation function pair which defines a systolic architecture.
- iv) Derive the final architecture by performing mappings from the problem domain to space-time domain.

3.1 Specifying a Problem as an ARE

The first step in designing a systolic implementation for a problem is to define the problem in a specification language. The choice of a suitable specification language is large. Quinton [9] has shown that a special subclass of Uniform Recurrence Equations (URE's) can specify many problems for which systolic implementations can be derived¹. Formally, a URE over a domain D is defined to be an equation of the form:

$$f(p) = g(f(p - w_1), f(p - w_2), \dots, f(p - w_k)) \quad (\text{eq. 3.1})$$

¹ Similar results have been presented by other researchers. Moreover, Rao [17] has shown that every systolic array can be described by a system of URE's.

where $p \in D$;

w_i 's for $i = 1..k$ are constant n -dimensional vectors; and

g is a single valued function dependent on each of its arguments.

The result of Quinton's theory suggests that URE's would be the good choice for specification language. However, Rajopadhye has pointed out the usefulness of allowing problems to be specified in a less restricted type of recurrence equations. This type of recurrence equation is called an Affine Recurrence Equation (ARE) and is defined as follows:

$$f(p) = g(f(A_1p + b_1), f(A_2p + b_2), \dots, f(A_kp + b_k))$$

where

p is a point in an n -dimensional problem domain;

A_i 's are $n \times n$ constant matrices;

b_i 's are constant n -dimensional vectors; and

g is a single valued function dependent on each of its arguments.

Notice that the depending computations are linear ($A_i p + b_i$) instead of uniform ($p - w_i$), hence the term "affine". For example, the multiplication of two square matrices A and B can be specified by the following system of ARE's.

$$f(i,j,k) = \begin{cases} 0 & \text{if } k = 0 \\ f(i,j,k-1) + A(i,k,k-1) * B(k,j,k-1) & \text{otherwise} \end{cases} ;$$

$$A(i,j,k) = \begin{cases} A_{ij} & \text{if } k = 0 \\ A(i,j,k-1) & \text{otherwise} \end{cases}$$

and

$$B(i,j,k) = \begin{cases} B_{ij} & \text{if } k = 0 \\ B(i,j,k-1) & \text{otherwise} \end{cases}$$

The power of using ARE's (instead of URE's) to specify problems lies in its flexibility. However, not every problem specified as a system of ARE yields systolic architectures. In fact, the next step in deriving systolic architectures from an ARE specification is to pipeline all non-uniform dependencies into uniform dependencies. Only those ARE's whose

dependencies can be localized can be mapped to systolic architectures. The following section describes how pipelining is performed.

3.2 Pipelining Non-Uniform Dependencies

A problem with non-uniform dependencies means that a computation depends on some other computations which are at a non-uniform (linear) distance away. Such a linear dependency when mapped to the processor domain (by linear mapping functions) also generates a non-uniform (linear) processor interconnection. However, by definition of systolic array, such a connection is not permitted. Hence, in order to derive architectures for problems with linear dependencies, the non-uniform dependencies have to be converted into uniform dependencies. The following pipelining theorem [16] helps us to do just that.

Theorem 3.1

A particular dependency $[A, b]$ of an ARE can be made uniform if the dependency matrix A has a non-trivial zero ρ such that

$$\exists n | \forall p \in D, \quad p + k\rho = Ap + b ;$$

Intuitively, pipelining can be explained as follows. Suppose a computation at point p requires the result of a computation at point q of a "linear" distance away; i.e. $q = Ap + b$. If we can identify all points (say p' 's) that need the same value (q), then we can somehow pass it along these points (hence the "pipelining"). Hence if p and p' both needs the value of q , then

$$\begin{aligned} Ap + b &= Ap' + b. && \text{(eq. 3.2)} \\ \Rightarrow A(p - p') &= 0 \end{aligned}$$

$(p - p')$ is therefore a solution to $Ax = 0$ and thus belongs to the nullspace of A . Conversely, all solutions to $Ax = 0$ satisfy the condition given in (eq. 3.2) and therefore

3.3 Finding Timing Functions

After all non-uniform dependencies in the original ARE are pipelined, we have a new system of URE's. This system of URE can now be used to derive systolic architectures. Intuitively, we can look at the problem of deriving systolic architectures as follows. A problem specified as a system of ARE's defines a dependency structure in the problem domain. We can remove the non-uniform dependencies in the ARE by means of pipelining. The uniform dependencies then form a lattice of dependencies in the problem domain. To generate a systolic architecture, we map this dependency structure in the problem domain to a space-time domain; where the space domain denotes the processor locations and the time domain defines the time at which the computation is performed by each processor. Two mapping functions, one for mapping to the time domain and the other for mapping to the space domain, are therefore required. These two functions are known as the timing and allocation functions respectively. This timing and allocation function pair, together with the computation defined by the given URE, can therefore fully specify a systolic architecture. In what follows, we will first look at the concept of an Affine Timing Function. Many researchers have developed similar theories. We will follow the notation used by Quinton [15].

An affine timing function is a function mapping a point in the problem domain to a positive scalar value. It can be represented in the following form:

$$t(p) = \lambda_t p - a.$$

where λ_t is an $n \times 1$ vector and a is a scalar constant.

Intuitively, a timing function maps a computation in the problem domain to the time domain. In other words, it identifies the time at which the specified computation is performed. Since the dependency structure of the problem specifies a precedence relationship, the timing function has to guarantee that a computation is performed only after its arguments have been computed. Hence, if there are k dependencies d_1, d_2, \dots, d_k in the specified problem, i.e. computation at point p is dependent on points $q_i = p + d_i$ for $i = 1..k$ then

require the same value q as p does. Notice that the rank of A has to be less than n (the dimension of the problem domain) in order for pipelining to be possible. When the rank of A is less than $n-1$ we get an "extended pipeline." Moreover, under certain conditions, it is possible to involve multiple data dependencies in a single pipeline (called multi-stage pipelining [16]). SASCOM does not implement this and other recent techniques developed by some researchers [19,24,25], but only addresses the case when the rank of A of $n-1$. This null space of A is then a straight line, and has a unique integer basis ρ (up to sign).

Now there is one more thing to consider. After we have identified all the points $p + k\rho$ that need the same argument as p , we need to know how these points get their first value, or in other words, how the pipeline is initialized. Let's call this point that gets the first value, p_t .

Since p_t is the earliest scheduled point in the nullspace of A , it has to be a point on some domain boundary (say, $\theta p = \pi$). A proof of this is given by Rajopadhye []. In addition, it has to be a constant distance away from the original point on which p depends. Hence,

$$Ap + b - p_t = \rho_{\perp} \quad \text{and} \quad \rho_{\perp} \text{ is a constant vector.}$$

Putting all this together, if the original ARE is:

$$f(p) = g(f(A_1p + b_1), f(A_2p + b_2), \dots, f(A_jp + b_j), \dots, f(A_kp + b_k)),$$

after $A_jp + b_j$ is pipelined, the specification becomes:

$$f(p) = g(f(A_1p + b_1), f(A_2p + b_2), \dots, f_j(p), \dots, f(A_kp + b_k)) \text{ and}$$

$$f_j(p) = \begin{cases} f(p + \rho_{\perp}) & \text{if } \theta_j p = \pi_j \\ f(p + \rho) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \lambda_i p &> \lambda_i (p + d_i) \\ \Rightarrow \lambda_i d_i &< 0 \quad \text{for } i = 1 \dots k; \forall p \in D \end{aligned}$$

In addition, the timing function has to map computations to positive time instants. This constraint can be expressed as constraints on the vertices and ray (we assume that the domain has at most one ray) of the domain.

Given that v_1, v_2, \dots, v_m are m domain vertices and r is ray in the domain,

$$\begin{aligned} \lambda_i v_i &\geq a \quad \text{for } i = 1 \dots m, \text{ and} \\ \lambda_i r &\geq 0. \end{aligned}$$

Any timing function satisfying these three constraints is a valid timing function. Finding an optimal timing function from this set of valid timing functions is a linear programming problem, which will be discussed in chapter 5.

3.4 Finding Allocation Functions

While the timing function maps a computation to the time domain, an allocation function maps it to the processor domain. Formally, given an n -dimensional problem domain, an allocation function is a linear function which maps points from the Z^n problem space to the Z^{n-1} processor space;

$$\text{i.e.} \quad A(p) = \lambda_a p$$

where λ_a is an $n \times n-1$ integral matrix.

A valid allocation function must guarantee that two computations scheduled to be performed at the same time are mapped to two different processors. This constraint is represented as follows:

$$T(p) = T(q) \wedge A(p) = A(q) \Rightarrow p = q$$

In this chapter, we have briefly presented the fundamental framework of a methodology to synthesize systolic architectures. Starting from an ARE specification of a problem, we first pipeline the non-uniform dependencies and turn the original ARE into a URE specification. We then find valid timing and allocation functions to map the computations in the problem domain to a time-processor domain which fully specifies a systolic architecture.

From the discussion on timing and allocation functions, it is clear that they are not unique. The constraints presented in this chapter only define a search space of valid timing and allocation functions. Finding a systolic architecture for a problem therefore becomes a search problem. However, as we will see in the next chapter, by focusing on a few additional constraints that characterize systolic architectures, we can limit the search space of systolic architectures to a fairly small set. By choosing suitable optimality criteria, we can find an optimal architecture for a specified problem.

CHAPTER 4

ALLOCATION FUNCTION AND ITS IMPLICATIONS: NUMBER OF ALL SYSTOLIC ARCHITECTURES

We have seen in the last chapter a methodology to design systolic architectures from systems of affine recurrences. Using a timing and an allocation function, we can map computations from the problem to the space-time domain to generate a final architecture. Such a timing and allocation function pair therefore uniquely determines a systolic architecture. In this chapter, we develop a lower bound on the number of valid allocation functions. This lower bound is found to be a small number and depends only on the dimension of the input recurrence equation. For two and three dimensional recurrences, the bounds are 4 and 13 respectively. Such small bounds imply that it is feasible to pre-compute all possible systolic architectures before we select an optimal one. We will start our discussion by going back to our old friends from last chapter, the timing and allocation function.

4.1 Timing Function

A timing function of an architecture defines the time at which each processor performs the computation corresponding to each index point defined by the input recurrence. Formally, a timing function, T , maps the points in the problem domain to positive integer points such that if p depends on q i.e. $p - q = c$ where c is a positive constant vector, then $T(p) > T(q)$; or equivalently, if d_1, d_2, \dots, d_k are dependencies in the problem domain, then $T(d_i) < 0$ for all $i = 1 \dots k$. Although there are in general infinite number of timing functions for a given problem domain as we have seen in the last chapter, at least one optimal timing function could be found consistent with a given allocation function. This optimal timing function can be found using a linear programming algorithm given in the next chapter.

4.2 Allocation Function

Given a problem domain in Z^n , an allocation function is an affine function mapping points from Z^n to Z^{n-1} . It defines the mapping of computations from the problem domain to processors in the architecture domain. Furthermore, uniform dependencies in the problem domain are mapped to processor connections in the processor domain. Hence if d_1, d_2, \dots, d_k are k dependencies in the problem domain and A is an allocation function, then

$$A[d_1 | d_2 | \dots | d_k] = [c_1 | c_2 | \dots | c_k] = C \quad (\text{eq. 4.1})$$

where A is an $(n-1) \times n$ matrix, $[d_1 | d_2 | \dots | d_k]$ an $n \times k$ matrix and $[c_1 | c_2 | \dots | c_k]$ an $(n-1) \times k$ matrix. Each of the columns, c_j in C is a vector denoting the direction of a processor interconnection¹. The matrix C then fully specifies the connection pattern of the final architecture, and we call it the *connection pattern matrix*.

Given a valid allocation function, we can choose an optimal timing function and hence derive a target array. Thus the problem of finding all systolic arrays for a problem can be reduced to finding the number of valid allocation functions. Finding an allocation function for a particular application is, in general, a constraint-satisfaction problem. There are basically three constraints that the allocation function has to satisfy -- non-conflict with the timing function, the dense array and the nearest-neighbor interconnection. Based on these three constraints, we can find the entire set of valid allocation functions. We note that not all valid allocation functions define distinct architectures. We must therefore derive a condition which enables us to identify architectures that are isomorphic to each other.

Non-conflict with The timing function constraint

This constraint requires that the allocation function does not conflict with the timing function: if the timing function requires two computations to be performed at the same time,

¹ Chen [3] calls each c_j a basis connection vector.

the allocation function must guarantee that they are performed at different processors. Hence if $T(x)$ is a timing function and $A(x)$ an allocation function, then the following condition holds:

$$T(p) = T(q) \wedge A(p) = A(q) \Rightarrow p = q$$

The dense array constraint

Intuitively, the dense array constraint requires that every processor in the final architecture maps to at least one existing point in the problem space. Such a constraint is necessary to guarantee that there do not exist processors in the final architecture that do not perform any computation ("hole").

Formally, if $A(x)$ is a valid timing function, then

$\forall p \text{ in } Z^{r-1}, \exists q \text{ in } Z^n$, such that

$$A(q) = p.$$

The nearest-neighbor interconnection constraint

This constraint requires the allocation function to map all the data dependencies of the URE to nearest-neighbor interconnections in the final architecture. These interconnections are identified by a set of permissible interconnections. Many researchers have employed different set of permissible interconnections in defining systolic arrays. In this thesis, we will look at several of them to see the effects of each of them on the bound of the number of systolic arrays.

The above three constraints limit the search space for valid allocation functions. The following section defines a condition, based on which we can determine whether two allocation functions yield distinct architectures or not.

Condition for Distinct Architectures

Intuitively, two systolic architectures are the same if the logical attributes of the processors and the processor interconnections are the same. By logical attributes, we mean attributes such as the number of accumulators, input and output buffers for each processor, the function computed by each processor, etc. Attributes specific to hardware implementations such as whether it has hardwired control or not are not considered here.

As an example, consider the two arrays shown in figure 4.1. Let's assume that they have been derived from a URE with two dependencies, d_1 and d_2 by the allocation functions A_1 and A_2 respectively. Their interconnection pattern matrices are given by $C_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and $C_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ respectively. The functions computed by the processors in both arrays are the same since they are determined by the same problem specification. Since one of the arrays is obtained merely by "skewing" the other one, it is obvious that they are identical.

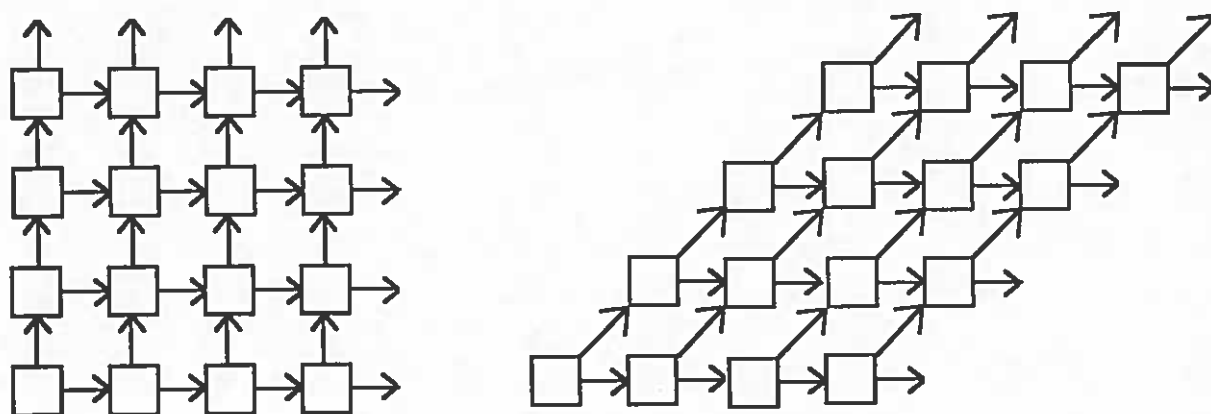


Figure 4.1. Two arrays that are isomorphic to each other.

We thus see that two architectures could be the same even if the processor interconnection patterns given by their interconnection pattern matrices are not the same and they have different allocation functions. We shall now express our definition formally:

Definition 4.1

Two architectures for a given problem specification given by their allocation functions A_1 and A_2 are the same if their connection pattern matrices C_1 and C_2 are linear transformations of each other.

In the example above, we see that the two connection pattern matrices are linear transformation of each other (since $\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$). They therefore represent the same architecture. The condition for distinct architecture can also be described in terms of allocation functions for architectures. This is given by the following theorem.

Theorem 4.1

Two architectures deduced by their allocation functions A_1 and A_2 are the same if A_1 and A_2 are linear transformations of each other.

Proof

Let d_1, d_2, \dots, d_k are the dependencies in the problem domain, then

$$A_1 [d_1 | d_2 | \dots | d_k] = C_1 \text{ and}$$

$$A_2 [d_1 | d_2 | \dots | d_k] = C_2$$

If A_1 and A_2 are linear transformation of each other, there exists non-singular matrix T , such that

$$T \cdot A_1 = A_2$$

$$\Rightarrow T \cdot A_1 \cdot [d_1 | d_2 | \dots | d_k] = A_2 \cdot [d_1 | d_2 | \dots | d_k] = C_2$$

$$\Rightarrow T \cdot C_1 = C_2,$$

Hence, C_1 and C_2 are linear transformation of each other. By definition 4.1, A_1 and A_2 yield the same architectures.

Next, we will look at the dense array and nearest-neighbor interconnection constraint in more detail. Note that although we can define problems in any number of dimensions, we normally fabricate systolic architectures in 2-dimension. Therefore, in the following discussion, we will limit our examples to mappings from Z^2 and Z^3 problem domains to Z^1 and Z^2 processor domains respectively.

4.3 Dense Array Constraint

Let $[i_1, i_2, \dots, i_n]^T$ be a point in the problem domain. Then an allocation function can be represented as:

$$A(p) = \lambda_a \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{bmatrix}$$

where λ_a is an $n-1 \times n$ integer matrix.

The dense array constraint requires that each point in the processor domain maps to at least one point in the problem domain. Hence, the equation:

$$A(p) = b$$

has integer solution(s) p for all $b \in Z^{n-1}$, the processor domain.

The following theorem from Schrijver [20] gives a necessary and sufficient condition for the above constraint to be true.

Theorem 4.2

Let A be an integral $m \times n$ -matrix of full row rank. Then the system $Ax = b$ has an integral solution x for each integral vector b if and only if the *g.c.d.* of the subdeterminants of A of

order m is 1 (such a matrix is also known as a unimodular matrix).

Our dense array constraint can therefore be fully specified by the above theorem.

For 2-dimensional recursion, an allocation function is given by the following:

$$A(p) = [a \ b] \begin{bmatrix} i \\ j \end{bmatrix} = X \quad (\text{eq. 4.2})$$

Applying theorem 4.2, we get

$$g.c.d(a, b) = 1.$$

An allocation function in 3-dimension is given by:

$$A(p) = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} a_1i + b_1j + c_1k \\ a_2i + b_2j + c_2k \end{bmatrix} \quad (\text{eq. 4.3})$$

Applying again theorem 1, we get

$$g.c.d.((a_1b_2 - a_2b_1), (a_1c_2 - a_2c_1), (b_1c_2 - b_2c_1)) = 1.$$

4.4 Nearest-neighbor Interconnection Constraint

Locality of processor interconnection has been one of the most important properties of systolic architectures. Such a property enhances the modularity and therefore the extensibility of the architecture. We now show how the nearest-neighbor interconnection constraint also limits the search space of possible architectures for a particular problem. The nearest-neighbor interconnection constraint requires an allocation function to map each dependency to a permissible nearest-neighbor interconnection. Let A be the allocation function, d_1, d_2, \dots, d_k be the k dependencies of the problem and P be the set of permissible interconnections. Then,

$$\begin{cases} A(d_1) \in P \\ A(d_2) \in P \\ \vdots \\ A(d_k) \in P. \end{cases} \quad (\text{eq. 4.4})$$

Using the connection pattern matrix notation, (eq. 3.4) may thus be stated as follows.

$$C = A[d_1 | d_2 | \dots | d_k] = [c_1 | c_2 | \dots | c_k] \quad (\text{eq. 4.5})$$

where $c_i \in P$ for $i = 1 \dots k$.

We see that (eq. 4.5) represents K systems (each one of $n-1$ equations) of diophantine equations. Since P is a finite set, there are only a finite number of ways to form C . Therefore, there exists only a finite number of systems of equations like (eq. 4.5). By solving each one of these systems for A , we can therefore find all the valid allocation functions. However, if this is done naively, the number of solutions will be too large. We must therefore use other constraints to remove duplicates and choose only valid allocation functions that yield distinct systolic arrays. There are two properties of these system of equations that are important for the derivation of the total number of systolic architectures.

(i) Another look at the condition for distinct architecture

According theorem 4.1, not all connection pattern matrices generate distinct architectures. Given a finite P , those that are linear transformation of each other will not generate distinct architectures.

(ii) Number of dependencies and dimension of the problem

If $k < n$, the system of equations, (eq. 4.5) is under-determined. There are potentially infinite number of solutions and hence infinite number of architectures. If $k = n$, the system of equations is a determined one. There is at most one solution to each system of equations of the form of (eq. 4.5) and therefore at most one architecture defined by (eq. 4.5). If $k > n$, (eq. 4.5) becomes an over-determined system of equations. There is again at most one solution and at most one architecture defined by each system of equations of the form of (eq. 4.5).

The $k < n$ case is not very interesting as can be seen by considering point p in the problem domain. Any point that either depends on p or that p depends on is obtained by a linear combination of the dependencies. Hence, the dependencies span (i.e. constitute the basis for) a lattice space that contains the component of dependency graph of the entire computation that is connected to p . Since $k < n$, this lattice space must be a subspace of the problem space. We can therefore partition the problem domain into a number of independent subdomains; i.e. into independent subproblems. The number of dimensions of each of these subdomains is then equal to the number of dependencies of each of them ($k = n$), and this reduces to the $k = n$ case.

Since for $k \geq n$, there is at most one solution to a given system of equations of the form of (eq. 4.5), each distinct system of equations determines at most one unique allocation function. Hence, the number of allocation functions is bound by the number of distinct systems of equations that can be formulated, given d_1, d_2, \dots, d_k and P , the set of permissible interconnections. This reduces to the number of distinct² connection pattern matrices that can be found.

4.5 The Total Number of Systolic Architectures

We now look at two specific P 's which describe the permissible interconnections for linear and 2-dimensional arrays. These two P 's are the most commonly used sets of permissible interconnections for systolic architectures.

a. 2-dimensions

For linear arrays, the commonly used set of permissible interconnections is $P = \{0, 1, -1\}$. Given this P and k dependencies (note that $k \geq 2$), there are 3^k possible number of C 's that can be formed. Consider the following two cases:

² i.e. distinct modulo the constraint imposed by the distinct architecture condition.

(i) If $k=2$, the number of possible C 's is $3^2 = 9$. But one of these permutation is $[0 \ 0]$ for which we get:

$$A[d_1 \mid d_2] = [0 \ 0] \quad (\text{eq. 4.6})$$

This yields an architecture which maps all points in the domain to a single processor, and is not a realistic solution.

Now let's look at the connections defined by $[0 \ 1]$ and $[0 \ -1]$. According to the distinct architecture constraint, since $[0 \ 1]$ and $[0 \ -1]$ are linear transformation of each other (since $[0 \ 1] = -1 \times [0 \ -1]$), they therefore describe the same architecture. Similarly, $[1 \ 0]$ and $[-1 \ 0]$ define the same array as do the pairs, $[1 \ 1]$ and $[-1 \ -1]$; and also $[1 \ -1]$ and $[-1 \ 1]$. Hence, by eliminating the uniprocessor ($C = [0 \ 0]$) architecture and grouping the non-unique architectures, we obtain 4 valid, unique architectures.

(ii) If $k > 2$, there are 3^k C 's and therefore 3^k systems of equations of the form:

$$A[d_1 \mid d_2 \mid \dots \mid d_k] = [c_1 \mid c_2 \mid \dots \mid c_k] \quad (\text{eq. 4.7})$$

But notice that any such system is overdetermined, and may be solved in two steps:

(a) Form a determined system of equations by choosing any two dependencies d_i and d_j where $i \neq j$ and i and j are in $\{1, \dots, k\}$; i.e. the system

$$A[d_i \mid d_j] = [c_i \mid c_j] \quad (\text{eq. 4.8})$$

(b) The solution, A obtained from (eq. 4.8) is then back substituted into (eq. 4.7) and tested for consistency with the rest of the equations.

The number of solutions is again limited by (eq. 4.8) which is the same as the $k=2$ case.

Therefore, the bound on the number of distinct allocation functions is again 4. Therefore, we have the following theorem:

Theorem 4.3

The bound on the number of distinct systolic architectures is 4 for 2-dimensional input problem.

b. 3-dimension

For planar arrays, we use the standard set of permissible interconnections as shown below:

$$P = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\}$$

This includes rectangular mesh, together with one diagonal, and thus expresses all purely systolic arrays (including the hexagonal ones). With this P and k ($k \geq 3$) dependencies, d_1, d_2, \dots, d_k , there are 7^k ways to form the connection pattern matrix. Although this is a large number, we can show that many of these connection pattern matrices generate redundant solutions.

Theorem 4.4

There are at most 13 distinct and valid systolic implementations for 3-dimensional input problem.

The proof is a generalization of the 2-dimensional case and is given in appendix A.

There are two other P s that are also commonly used by researchers:

$$P_1 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}, \text{ and}$$

$$P_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right\}$$

P_1 and P_2 define the connections for the eight-neighbors and mesh array configuration respectively. The bounds on the number of systolic architectures for these two P 's are 25 and 9 respectively. The derivation of these results are also given in appendix A.

We have shown that given a problem specification, the number of all possible systolic architectures that can be derived is bounded. For the most common cases in practice, these bounds are fairly low: for linear arrays, the total possible number of systolic architectures is equal to 4; and for planar arrays, the number is 13.

CHAPTER 5

THE SOFTWARE: SASCOM

We have seen in the last two chapters the theoretical framework for the automatic generation of systolic arrays. The theory is embodied in a software system that accepts problems specified as affine recurrence equations and produces systolic architectures for the input problem. The system is composed of three parts: (1) an input editor, (2) a systolic architecture generator; and (3) a systolic array simulator. Figure 1.1 is repeated below to show a diagrammatic view of SASCOM.



Figure 1.1 Overall structure of SASCOM.

The Editor

The front end of the system is a text editor to accept ARE specifications. The reader is referred to appendix B for the BNF of the specification language. A problem specified in this language is first parsed to an internal representation, which is then passed on to the systolic array generator for the derivation of systolic architectures.

The Systolic Array Generator

This is the core part for systolic array generation. It performs pipelining, generation of valid mapping functions and derivation of final architecture. The result of these operations is an architecture specification for the simulator.

The Simulator

The simulator provides a graphical user-interface to perform step by step tracing, inspection of state of individual processors and debugging of processor specification. It may also be used as a stand-alone application.

While the editor and simulator are important parts of the system, the main emphasis of this thesis is on automatically generating systolic architectures. We shall therefore focus the discussion in this chapter on the algorithms used by the systolic array generator.

5.1 Systolic Array Generation

The process of systolic array generation can be broken down into four major steps. Each of these steps are performed by a separate module of the generator. A diagrammatic view of these modules is shown in figure 5.1.

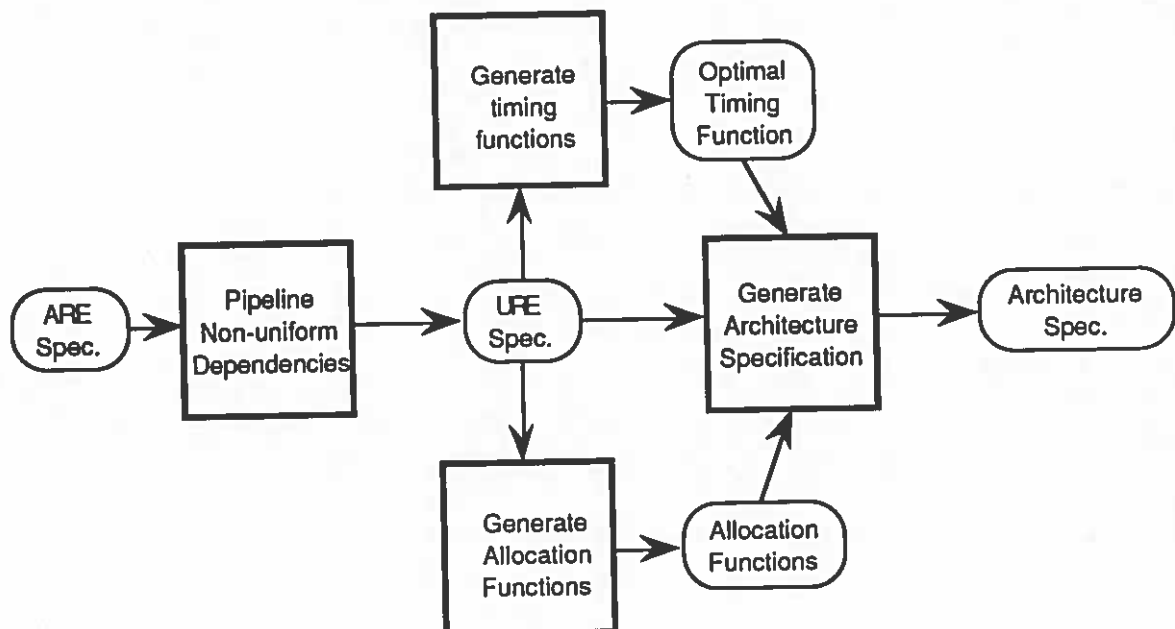


Figure 5.1 The data flow diagram of the systolic array generator.

Figure 5.1 shows the top level flow of the systolic array generator, which is an implementation of the synthesis methodology presented in chapter 2. The four steps involved in systolic array generation are listed in the following:

1. Pipeline the affine dependencies and transform the original ARE into a system of uniform recurrence equations (URE).
2. Find the optimal timing function(s).
3. Find all possible allocation functions for the problem¹.
4. Build the final architecture specification.

We have made four design decisions in our system:

(1) User's input to the system is kept at a minimum level since it is built to automate the entire design process.

(2) All allocation functions are derived and the cost of each resulting architecture computed before generating the architecture specifications. Such an arrangement is possible because of the reasonably small bounds on the number of distinct allocation functions described in the previous chapter.

(3) The architecture specification language generated by the generator is closer to a hardware than a mathematical description. Such a choice maintains a distinction between a higher level problem specification (ARE) and a lower level architecture specification (hardware).

¹ Step 2 and 3 may be performed independently of each other.

(4) SASCOM currently accepts only finite problems, and this limits the applicability of the system, particularly in real-time and DSP applications².

In the rest of this chapter, we present the algorithms used by SASCOM to perform each of four steps shown in figure 5.1. Then we will demonstrate the mechanism of these algorithms by tracing through one complete example -- LU-decomposition of a square matrix.

The algorithms will be presented in the following format -- first, the theory behind the algorithm (if there is one) is explained, then the inputs and outputs to the algorithm; finally, the algorithm is presented.

5.2 Algorithm 1: Pipelining the Affine Dependencies

The theory of pipelining has been presented in chapter 2.

Inputs:

The original affine recurrence equation.

Output:

The transformed uniform recurrence equations (URE).

Algorithm:

1. Given an affine (non-uniform) dependency $d = Ap + b$ in an n -dimensional problem space, find the nullspace of A ; where A is a constant $n \times n$ matrix and b is a constant n -dimensional vector.

² It is theoretically possible to perform the derivations for such recurrences too, though it involves reasoning about parameterized systems of ARE's. In the first version of SASCOM, this generality was deliberately sacrificed for the sake of simplicity.

1.1 The nullspace of d is found by solving the equation $A p = \overline{0}$ for p .

1.2 The value of the free variable(s) in p is (are) bound to be the smallest positive integer.

1.3 The sign of the free variables(s) in p is (are) found by applying the following heuristic:

Compare the results of taking two dot-products: (1) d with p with its free variable bound to -1 and (2) d with p with its free variable bound to 1. The one having the larger product gives the correct sign of the free variable³.

2. Find the ρ_{\perp} , the distance between the point of intersection of the newly pipelined dependency with a boundary plane of the problem domain. ρ_{\perp} must be a constant vector in order for pipelining to work.

2.1 The user is prompted for the equation of a boundary plane with which she thinks the new dependency might intersect.

2.2 The intersection point (p_i) of that boundary plane and the new dependency is then calculated. If the intersection point is not a constant vector away from the original depending point; i.e. ($\rho_{\perp} = d - p_i$) is not a constant vector, step 2.1 is performed again.

3. A uniform recurrence equation is formed by replacing the original affine dependencies with their corresponding pipelined dependencies and by adding in new functions formed by the ρ_{\perp} .

³ There is a well developed theory for determining the so called pipelining transformations. In the most common cases however, the null space is linear and the above heuristic yields satisfactory results. The generality provided by the theory was sacrificed for simplicity of implementation of the system.

5.3 Algorithm 2: Finding an Optimal Timing Function

Given a list of n data dependencies, d_1, d_2, \dots, d_n , a valid timing function $T(p)$ has to satisfy the following constraints:

$$\begin{cases} T(d_1) \leq 1 \\ T(d_2) \leq 1 \\ \vdots \\ T(d_n) \leq 1 \end{cases}$$

This set of constraints forms a convex hull embodying a space of valid timing functions.

The optimal timing functions are the timing functions from the set of valid timing functions which gives the least processing time for the given problem. Given a timing function and the set of domain vertices $\{v_1, v_2, \dots, v_n\}$, the processing time is calculated as:

$$\begin{aligned} s &= \max \{ T(v_i) - T(v_j) \mid i, j \in \{1, \dots, n\} \} \\ &= \max \{ T(v_i - v_j) \mid i, j \in \{1, \dots, n\} \} \end{aligned}$$

This gives us an optimizing criterion to choose the optimal timing functions. Notice that this criterion is a linear condition. Therefore, the linear programming algorithm can be used to find an optimal timing function.

Inputs:

The data dependencies and the vertices of the problem domain.

Outputs:

An optimal timing function.

Algorithm:

1. Find the vertices of the convex hull defined by the following set of constraints:

$$\begin{cases} T(d_1) \leq 1 \\ T(d_2) \leq 1 \\ \vdots \\ T(d_n) \leq 1 \end{cases} \quad (\text{ineq. 5.1})$$

Since we are not considering problem domains with rays, the linear programming theory dictates that at least one of the vertices is an optimal timing function.

1.1. The vertices of the convex hull are found by solving sets of equations formed by all combinations of k inequalities (k is the number of dimension of the problem domain) from (ineq. 5.1).

2. Of all the T 's (vertices) found in the above step, apply the optimizing criterion, $s = \max \{ T(v_i - v_j) \mid i, j \text{ in } \{1, \dots, n\} \}$ on each of them and find the one T with the least s . This T is an optimal timing function.

5.4 Algorithm 3: Finding All the Possible Allocation Functions

Recall from last chapter that the nearest-interconnection constraint dictates that:

$$A(d_1, d_2, \dots, d_n) = C \quad (\text{eq. 5.2})$$

where A is the allocation function, $d_1 \dots d_n$ are the data dependencies and C is one of the valid connection pattern matrices. The entire set of valid allocation functions for a given problem is found by solving systems of equations like (eq. 5.2) with all valid C 's.

Inputs:

The data dependencies, set of permissible interconnections and an optimal timing function from step 2.

Outputs:

The entire set of valid and unique allocation functions.

Algorithm:

1. Build all systems of nearest-neighbor interconnection constraints.

1.1. Given n data dependencies, form all valid connection pattern matrices.

1.1.1 Choose from the set of permissible interconnections all valid combinations of n interconnections.

1.2. For each connection pattern matrix found in step 1, form a valid nearest-interconnection constraint of the form of (eq. 5.2).

2. Solve all systems of nearest-interconnection constraints for integer solutions. Rule out those systems of equations which do not have any integer solution.

3. Rule out the invalid allocation functions.

3.1 Apply the linearly independent rows constraint to rule out the invalid allocation functions.

3.2 Rule out the architectures that are not dense by applying the dense array constraint to the allocation functions.

3.3 Apply the distinct architecture constraint to rule out the non-unique architectures.

4. Generate consistent optimal timing functions for each valid allocation function. Given a timing and an allocation function, the two functions are consistent if the timing function is not orthogonal to the normal of the allocation function.

4.1 For each of the valid allocation functions, test to see if the optimal timing function found in step 2 is linearly dependent on the row(s) of the allocation function. If it is, alter the timing function until it is consistent with the allocation function.

5.5 Algorithm 4: Building the Final Architecture Specification

The final architecture is derived by mapping the computations from the problem domain to the processor-time domain. Three inputs are required -- (1) the pipelined URE defining the computations performed in each processor; (2) the timing function specifying the time at which each processor performs its designated computation and (3) the allocation function mapping the computations and data dependencies to processors and processor interconnections respectively.

After being pipelined, the original ARE specification becomes a system of conditional URE's, of which each is of the form:

$$f(p) = \begin{cases} g_1 & \text{if } c_1(p) \\ g_2 & \text{if } c_2(p) \\ \vdots & \vdots \\ g_k & \text{if } c_k(p) \\ g_{k+1} & \text{otherwise} \end{cases} \quad (\text{eq. 5.3})$$

where

g_i 's are single value functions of the form ($b_{i,j}$'s are constant n-dimensional vectors):

$$g_i = (h_i(f(p + b_{i,1}), f(p + b_{i,2}), \dots, f(p + b_{i,j})));$$

each $c_i(p)$ is a conjunction of simple equalities and inequalities of the form :

$$\theta_{i,1}p = \pi_{i,1} \wedge \dots \wedge \theta_{i,r}p = \pi_{i,r} \wedge \theta_{i,r+1}p \geq \pi_{i,r+1} \wedge \dots \wedge \theta_{i,r+s}p \geq \pi_{i,r+s}$$

where

$\theta_{i,j}$'s are constant vectors in Z^n ,

$\pi_{i,j}$'s are scalar constants, and,

r, s are non-negative integers.

$f(p)$ defines a variable to be computed in each of the processor. The value of $f(p)$ is computed by evaluating one of the g_i 's depending on whether the guard condition $c_i(p)$ is true or not. The truth of $c_i(p)$ is established when all of its conjuncts (linear equalities and inequalities involving the index points) are true.

Let $\theta p = \pi$ be a conjunct of a guard expression, $c_m(p)$ where $1 \leq m \leq k$. Since the timing and allocation functions may be viewed as simple (linear) renamings of the index space, this condition is equivalent to:

$$\theta \left[\frac{A}{T} \right]^{-1} [x_1, \dots, x_{n-1}, t]^T = \pi$$

where

$[x_1, \dots, x_{n-1}, t]$ represents the coordinates in the space-time domain and $\left[\frac{A}{T} \right]^{-1}$ is the inverse of the space-time mapping function.

This reduces to

$$\theta' [x_1, \dots, x_{n-1}, t] = \pi$$

where

$$\theta' = [\theta_1, \theta_2, \dots, \theta_n]^T = \theta \left[\frac{A}{T} \right]^{-1} \quad (\text{eq. 5.4})$$

if $\theta_n \neq 0$, the linear equality is dependent on time and is therefore called a *time-dependent condition*. Next we present the concept of control signals.

¹ It should have been $\theta \left[\frac{A}{T} \right]^{-1} [x_1, \dots, x_{n-1}, t+a]^T = \pi$ where $-a$ is the constant term in the timing function. It simply affects certain initialization and is left out for simplicity.

Control Signals

The purpose of control signals in systolic architectures is to inform individual processors the truth of certain time-dependent condition. Given a processor location (x_1, \dots, x_{n-1}) , the time t at which a condition is true can be calculated from (eq. 5.4). Such a time-dependent condition can be implemented by pipelining a Boolean signal so that its arrival to processor (x_1, \dots, x_{n-1}) at time t signifies the truth of the condition. The theory of data pipelining presented in chapter 3 can also be used to generate control signal pipelines as can be seen by the following argument:

Let q and q' be two points in the problem domain that both satisfy the condition, $\theta p = \pi$.

Then

$$\begin{aligned}\theta q &= \pi = \theta q' \\ \Rightarrow \theta (q - q') &= 0\end{aligned}$$

Hence, the nullspace of θ characterizes all the points, p in the problem domain such that $\theta p = \pi$. The basis vector(s) of this nullspace define(s) the direction(s) of the pipeline. There are in general many ways to choose the basis vector(s) and thus the direction(s) of a pipeline. Since control pipelining is performed after data pipelining, SASCOM always tries to choose the direction of the control pipeline to be the direction of one of the existing data dependencies. Each control pipeline is initialized at the boundary processors. Given the location $[x_1, \dots, x_{n-1}]$ of one such processor, the pipeline is initialized at time $t = - \{ [\theta_1, \dots, \theta_{n-1}] \cdot [x_1, \dots, x_{n-1}] + \pi_m \} / \theta_n$.

We have presented a method to pipeline control signals for conditions in the form of one simple equality. To handle a conjunction of equalities, a control signal is generated for each conjunct. The truth of the composite condition is established when all the control signals arrive at a processor at the same time. SASCOM does not currently handle conditions involving inequalities, although the same method can easily be generalized to handle inequalities. This can be done by interpreting the arrival of a control signal as signifying the time at which the processor changes the value of a certain flip-flop. Thus it denotes a range of space-time values rather than a single space-time point.

Note that if $\theta_n = 0$, (eq. 4.5) reduces to

$$[\theta_1, \dots, \theta_{n-1}] \cdot [x_1, \dots, x_{n-1}] = \pi,$$

and the equality is independent on time. We call such a condition a *time-independent condition*. A time-independent condition (say $\theta p = \pi$) in the form of an equality defines a hyperplane in the processor domain. The points (processors) on this hyperplane differ from the rest of the points in the processor domain in that they are the only processors that perform the computation g guarded by the condition, $\theta p = \pi$ in the URE. This condition therefore partitions two types of processors in the processor domain -- one computing g , the other one not. For a time-independent condition in the form of an inequality, the partition is a half-space in the processor space. In general, the different processor types of an architecture can be found by considering all intersections of the spaces formed by all the time-independent guard expressions in the URE.

Next, we present the algorithm to derive the final architecture specification.

Inputs:

The transformed (pipelined) conditional URE problem specification, a timing function and an allocation function.

Outputs:

An architecture specification for the systolic array simulator. The specification includes (1) a data structure (*parity*) which specifies the internal structure of each processor and the processor interconnection, (2) the code executed by each processor and (3) the input expressions defining the sequence of input data going into the processor array.

Algorithm:**1. Find all processor locations in the architecture domain.**

1.1. Determine the mappings of the domain vertices to corner processor locations (i.e. vertices) in the architecture domain by using the allocation function.

1.2. Apply a polygon-fill algorithm to find all integer coordinate points inside the convex hull generated by the vertices. These correspond to the processor locations.

2. Determine the control signals needed.

2.1. A control signal is need for every guard expression that is not time independent.

2.2. Pipeline the conditional expression by finding the nullspace of θ . The direction (basis vector) of the pipeline is set to be the direction of one of the existing data pipelines.

2.3 The pipeline is initialized at the boundary processors at time $t = - \{ [\theta_1, \dots, \theta_{n-1}] \cdot [x_1, \dots, x_{n-1}] + \pi_m \} / \theta_n$.

3. Find the variables and their attributes in the final architecture:

Make one pass through the problem specification, and assign for each function a variable name and determine the attributes (direction of connection, propagation delay, etc.) of its arguments (dependencies). Let d be a dependency of an argument to compute a certain variable, then its direction of connection is given by $A(d)$ and the propagation delay is $T(d)$ where A and T are the allocation and timing functions. Since control pipelining has already been performed (in step 2), the flip-flops and wires needed for the control variables are also generated by this step.

4. Build the data structure (pary) for simulation. Pary is implemented as a two dimensional scheme array.

4.1. The different processor types are classified according to the discussion given at the beginning of this section. A hardware specification for each processor type is generated.

4.2. Copies of the hardware specification from step 4.1 are placed in the two dimensional array in such a way that location (i,j) of the array contains the hardware specification for processor (i,j) of the architecture.

5. Derive the code for each type of processors.

6. Determine the input expressions for the architecture. The system uses two ways to get input data to the architecture:

(1) If the input guard condition maps to a single processor for a linear array or a line of processors for a planar array, the inputs are fed into the processor array in form of a stream of data.

(2) Otherwise, the allocation function is such that the data must be preloaded, and for this purpose, a dummy accumulator is set up in the processor.

6.1. Each input expression of type (1) inputs is obtained simply by mapping its input guard condition, $\theta p = \pi$ to the space-time domain:

$$\theta \left[\frac{A}{T} \right]^{-1} [x_1, \dots, x_{n-1}, t]^T = \pi$$

where $[x_1, \dots, x_{n-1}, t]$ are the coordinates in the space-time domain.

6.2. For type (2) inputs, since the input data are already stored in an accumulator inside a processor, there is no need for an input expression. However, control signals are used to

initiate the use of the input data. These control signals are pipelined in the standard way defined earlier.

5.6 An Example: LU-Decomposition

We now present an example to illustrate the various steps. The problem of LU-Decomposition is to decompose a square matrix into an upper and lower matrices:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

Problem specification

The problem can be specified in our software by the following system of ARE:

```
(program
  (function
    (f i j k)
    (Input
      (matrix-a (1 2 3) (2 8 11) (3 22 35)))
    (output
      ((= i k) (matrix-u i j))
      ((= j k) (matrix-l i j)))
    (bounds
      (<= k j) (<= k i) (<= i 3) (<= j 3) (<= k 3) (> k 0))
    (cond ((= k 0) (Input matrix-a i j))
          ((= k j) (/ (f i j (- k 1)) (f k j (- k 1))))
          (else (- (f i j (- k 1)) (* (f i k k) (f k j (- k 1))))))))
```

Key words are in bold typeface. Notice that the language is scheme-like (prefix notation). Because of this, SASCOM can be closely tied with the scheme programming environment we chose to use. The system above is equivalent to the following equations in infix notation:

$$f(i,j,k) = \begin{cases} a_{ij} & \text{if } k = 0 \\ \frac{f(i,j,k-1)}{f(k,j,k-1)} & \text{if } k = j \\ f(i,j,k) - f(i,k,k) * f(k,j,k-1) & \text{otherwise} \end{cases}$$

Appendix B contains a BNF specification of the input language. In this language, the inputs, outputs, domain boundary and the bodies of the recursive functions are all fully specified.

Pipelining

Notice that $(f\ k\ j\ (-\ k\ 1))$ and $(f\ i\ k\ k)$ are two affine dependencies which need to be pipelined. These dependencies are parsed into their matrix representations:

$$A_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } b_1 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \text{ for } (f\ k\ j\ (-\ k\ 1));$$

$$A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } b_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ for } (f\ i\ k\ k).$$

To pipeline the two dependencies, their nullspaces are calculated by solving the equations:

$$\begin{aligned} A_1 p_1 &= \overline{0} & \text{and} & & & & (\text{eq. 5.8}) \\ A_2 p_2 &= \overline{0} \end{aligned}$$

for p_1 and p_2 .

The nullspaces for ρ_1 and ρ_2 are $\begin{bmatrix} k_1 \\ 0 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ k_2 \\ 0 \end{bmatrix}$ respectively, where k_1 and k_2 are two integer constants. The magnitudes of the two variables k_1 and k_2 are then bound to 1, and their signs are chosen as follows:

1) $(k_1, 0, 0)^T$ with k_1 first bound to 1 is multiplied with $(k, j, k-1)^T$.

i.e. $(1, 0, 0)^T * (k, j, k-1)^T = (k, 0, 0)^T$

2) Then the same operation is done with k_1 bound to -1.

i.e. $(-1, 0, 0)^T * (k, j, k-1)^T = (-k, 0, 0)^T$

3) The two products are then compared. The one with the smaller product has the correct binding for k_1 . Since k is positive in the problem domain, $(-k, 0, 0)^T < (k, 0, 0)^T$. k_1 is therefore bound to -1.

With the same consideration, k_2 in $(0, k_2, 0)^T$ is bound to -1.

Solving for ρ_{\perp}

Next, ρ_{\perp} 's for both dependencies are solved in the following way.

First, the user is prompted for a domain boundary plane where each of the two new dependencies is expected to intersect. Figure 5.3 shows the user-interface of the system.

Say, the equations input are $(= i k)$ for $(-1 \ 0 \ 0)$ and $(= j k)$ for $(0 \ -1 \ 0)$. The intersection of the dependencies with their corresponding input planes are solved.

i.e. (k, j, k) for $(-1, 0, 0)$ and $(= i k)$;
 (i, k, k) for $(0, -1, 0)$ and $(= j k)$.

ρ_{\perp} 's for the two dependencies are then found by subtracting from the original dependencies from these intersection points.

i.e. $\rho_{\perp 1} = (k, j, k-1) - (k, j, k) = (0, 0, -1);$
 $\rho_{\perp 2} = (i, k, k) - (i, k, k) = (0, 0, 0).$

With the new pipelined dependencies, the original ARE specification is transformed into a pipelined system of conditional URE 's where f_1 and f_2 are the pipelining functions.

```
(f i j k)
  (cond ((= k 0) (input matrix-a i j))
        ((= k j) (/ (f i j (- k 1)) (f2 i j k)))
        (else (- (f i j (- k 1)) (* (f1 i j k) (f2 i j k))))))

(f1 i j k)
  (cond ((= j k) (f i j k))
        (else (f1 i (- j 1) k))))

(f2 i j k)
  (cond ((= i k) (f i j (- k 1))
        (else (f2 (- i 1) j k))))
```

Notice the addition of two new functions, f_1 and f_2 .

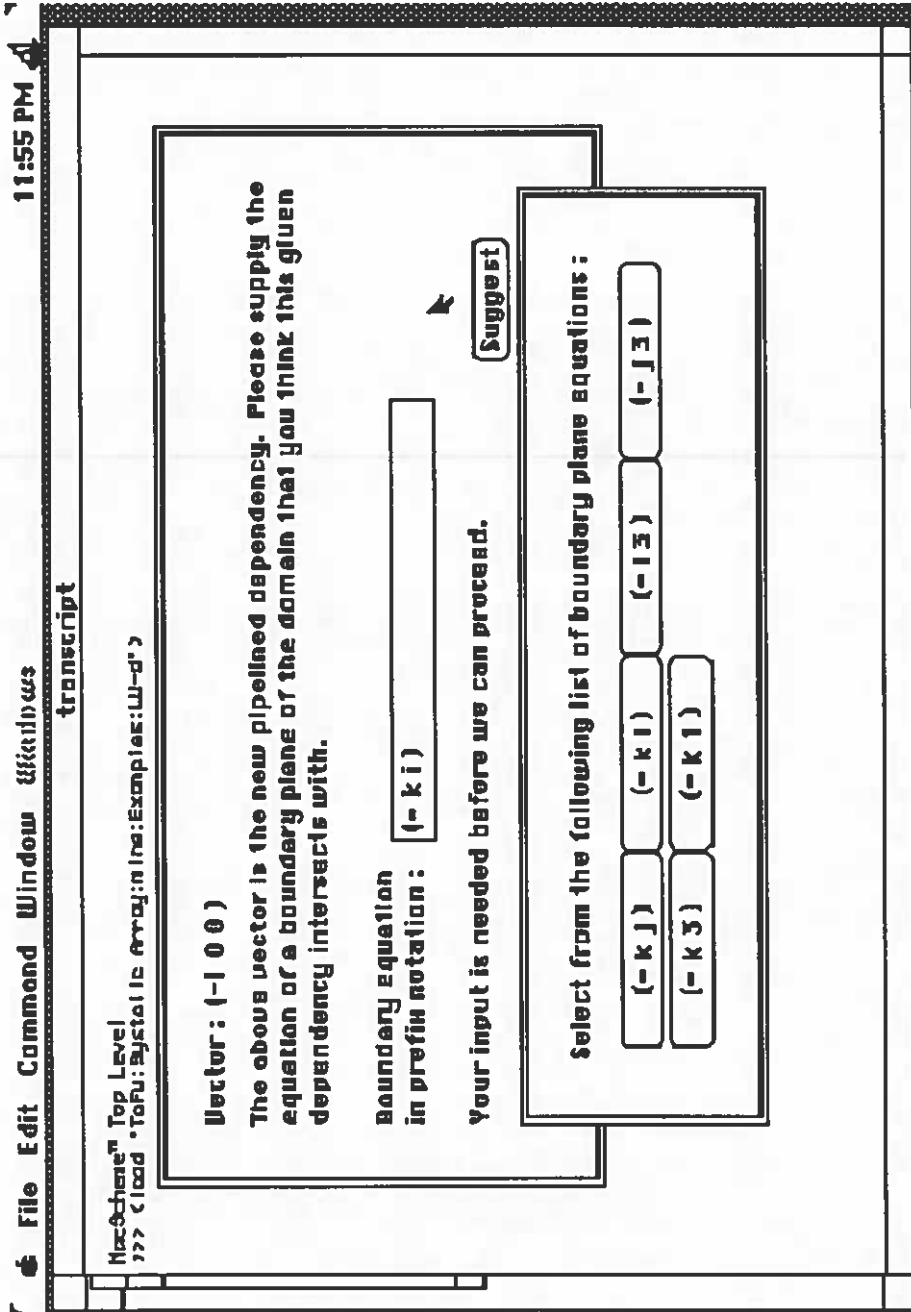


Fig. 5.2 The dialog box requesting the user for an equation of a boundary plane.

Finding an Optimal Timing Function

The constraint for valid timing functions is given by the following system of inequalities:

$$\begin{cases} T([-1,0,0]) \leq 1 \\ T([0,-1,0]) \leq 1 \\ T([0,0,-1]) \leq 1 \end{cases}$$

Finding the optimal timing function involves first finding the vertices of the hyperspace formed by the above set of inequalities. This can be done by running the diophantine equation solver on the following system of equations:

$$\begin{cases} T([-1,0,0]) = 1 \\ T([0,-1,0]) = 1 \\ T([0,0,-1]) = 1 \end{cases}$$

This system has only one vertex, (1,1,1), which becomes the optimal timing function.

Finding All Allocation Functions

With the new pipelined dependencies added to the original uniformed dependency, the new dependencies structure is:

$$[d_1 \mid d_2 \mid d_3] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

A valid allocation function can be found as described before. It turns out that 13 valid and distinct allocation functions can be found:
















$$\begin{aligned} & \begin{bmatrix} 0 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \\ & \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}. \end{aligned}$$

Generating the Final Architecture

After all the allocation and timing functions are generated, the cost of each architecture (area times processing time) is calculated. This together with the general shape of the architecture are presented to the user, using the dialog box interface shown in figure 5.4.

The user can choose from this set of architectures an architecture he wants to simulate. The allocation and timing function pair corresponding to the architecture chosen by the user is then used to generate the final architecture specification. An example of an architecture specification generated is given in appendix C.

Mac98 Home Top Level

<input type="radio"/> AC:18 	<input type="radio"/> AC:36 	<input type="radio"/> AC:48 	<input type="radio"/> AC:36 	<input type="radio"/> AC:48 	<input type="radio"/> AC:36 
<input type="radio"/> AC:36 	<input type="radio"/> AC:36 	<input type="radio"/> AC:36 	<input type="radio"/> AC:18 	<input type="radio"/> AC:48 	<input type="radio"/> AC:36 
<input type="radio"/> AC:36 	<input type="radio"/> AC:18 	<input checked="" type="radio"/> AC:18 			

The above are all possible architectures for the given problem.
Please select any one of them for simulation.

Fig. 5.3 The dialog box showing the choices of valid and distinct architectures generated for the problem.

Conclusion

Systolic arrays are a type of special-purpose parallel architecture. Their popularity has been limited partly because of the tedious process in designing one. SASCOM is a CAD tool built to automate the design process. Based on a systematic design methodology, SASCOM accepts ARE's as input and generates target systolic architecture specifications.

Systolic array generation in SASCOM can be summed up by three processes -- data pipelining, generating transformation functions and deriving the target architecture specification. Data pipelining is required to transform the non-uniform dependencies in the input ARE to uniform dependencies. The result is a conditional URE specification that performs the same computation as the input ARE.

The derivation of an optimal timing function from a URE is well understood. Since, the optimality criterion is linear, the standard linear programming algorithm can be used to obtain an optimal timing function. The generation of allocation function is more involved. There is no consensus in the research community on a single optimality criterion for allocation function. However, by analyzing the properties of the set of permissible interconnection vectors, the upper bound on the number of distinct systolic architectures is found to be small. This makes it practical to generate all systolic architectures and their associated cost before an optimal architecture is selected.

The process of generating a target architecture specification is rather tedious. It involves analysis on the guard expressions of the pipelined URE specification to generate control signals and determine the different types of processors of an architecture. While there are heuristics to perform these analysis, a general theory is lacking.

SASCOM is written in MacScheme+TS and is currently running on the Macintosh-II system. To date, SASCOM is completely operational and performs the three processes listed above with satisfactory result. There are however some issues that SASCOM has not dealt with and require more research. The following are a list of them:

(1) SASCOM does not deal with infinite problem domains. Such problems require analysis involving parameterized equations. SASCOM's mathematical analyzer is not capable of handling them.

(2) In SASCOM, data pipelining is performed before an optimal timing function is found¹. Recently, Quinton [14] has shown that it is possible to derive an optimal timing function before data pipelining. This is a better arrangement since the timing function found imposes an extra constraint on the data pipelines so that their directions can be determined automatically and there is no need to prompt the user.

(3) Control signals pipelining leaves a lot of freedom in choosing the direction of the control pipeline. SASCOM heuristically chooses the direction of the control pipeline to be the direction of one of the existing data dependencies. While such a method guarantees that the control pipeline would not be in conflict with the timing function, it might not generate an optimal pipeline. In fact, it is an open question as to how to choose an optimal control pipeline.

(4) The target architecture specification generated for simulation is a hardware-level specification. It is designed to be compatible with the input language to the simulator which can be used as a stand-alone application. This low-level specification language gives the simulator the flexibility to specify non-systolic architectures. However, when used as the output specification language for the generator, it creates a conceptual gap between the input and output specification (ARE's vs. hardware specifications). Chen [3] has discussed a language called Space-Time Recurrence Equations (STREQ). An STREQ is a recurrence equation obtained by renaming the indexes of the original specification from the problem to the space-time domain by means of space-time mapping functions. There are two advantages in using this language as the target architecture specification language: (i) it is a subclass of ARE and thus eliminates the conceptual gap between the input and output specification; and (ii) it characterizes all and only systolic architectures.

¹ During the time when the SASCOM project first started, there was no other alternative.

APPENDIX A

DERIVATION OF THE BOUND ON THE NUMBER
OF SYSTOLIC ARCHITECTURES

We now apply the constraints stated in chapter 3 to derive the number of valid architectures possible for three different P 's:

$$(1). P_1 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\};$$

$$(2). P_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}, \text{ and}$$

$$(3). P_3 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right\}$$

$$(1) P_1 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\}$$

To find the number of unique connection pattern matrix, we can apply the following consideration.

First, we can group the column vectors in P into four groups:

1. $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$
2. $\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}$
3. $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}$
4. $\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix}$

Then, consider the ways we can combine 3 vectors from these 4 groups:

1) 3 vectors from each group (we can represent this as (1,1,1), (2,2,2) (3,3,3) and (4,4,4)). These combinations cannot form any valid connection pattern matrix since the two rows of the matrices formed this way are not linearly independent of each other.

2) 1 or 2 vector(s) from group 1 combined with 2 or 1 vector(s) from one group out of groups 2, 3 and 4 ((1,1,2), (1,2,2), (1,1,3), (1,3,3), (1,1,4), (1,4,4)) will not generate any valid connection pattern matrix because such a matrix either has a row with all zeros or has both of its rows being equal.

3) 1 vector from groups 2, 3, or 4 and 2 vectors from one group (different from the first one) out of groups 2, 3 and 4 ((2,3,3), (2,4,4), (3,2,2), (3,4,4), (4,2,2), (4,3,3)) generate 6 valid and unique connection pattern matrices (applying the distinct architecture constraint).

4) 1 vector from group 1 and 2 vectors from two groups out of groups 2, 3 and 4 ((1,2,3), (1,2,4), (1,3,4)) generate 3 valid and unique connection pattern matrices (applying the distinct architecture constraint).

5) 3 vectors, each from one group from groups 2, 3 and 4 ((2,3,4)) generate another 4 valid and unique connection pattern matrices (applying the distinct architecture constraint).

There are therefore a total of $3 + 6 + 4 = 13$ unique connection pattern matrices. These matrices are given below as reference:

From groups: (2,3,3), (2,4,4), (3,2,2), (3,4,4), (4,2,2), (4,3,3)

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}, \\ \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

From groups: (1,2,3), (1,2,4), (1,3,4)

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

From groups: (2,3,4)

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}.$$

$$(2) P_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}.$$

We can group these nine vectors into five groups:

1. $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$
2. $\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}$
3. $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}$
4. $\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix}$
5. $\begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix}$

The derivation for the number of valid C's is essentially the same as in (i). Consider the following cases:

1) 3 vectors from each group (we can represent this as (1,1,1), (2,2,2), (3,3,3), (4,4,4) and (5,5,5)). These combinations cannot form any valid connection pattern matrix since the two rows of the matrices formed this way are not linearly independent of each other.

2) 1 or 2 vector(s) from group 1 combined with 2 or 1 vector(s) from one group out of groups 2, 3 and 4 ((1,1,2), (1,2,2), (1,1,3), (1,3,3), (1,1,4), (1,4,4), (1,5,5), (1,1,5)), will not generate any valid connection pattern matrix because such a matrix either has a row with all zeros or has both of its rows being equal.

3) 1 vector from groups 2, 3, or 4 and 2 vectors from one group (different from the first one) out of groups 2, 3 and 4 ((2,3,3), (2,4,4), (2,5,5), (3,2,2), (3,4,4), (3,5,5), (4,2,2), (4,3,3),

(4,5,5), (5,2,2), (5,3,3), (5,4,4)) generate 6 valid and unique connection pattern matrices (applying the distinct architecture constraint).

4) 1 vector from group 1 and 2 vectors from two groups out of groups 2, 3 and 4 ((1,2,3), (1,2,4), (1,3,4), (1,2,5), (1,3,5), (1,4,5)) generate 3 valid and unique connection pattern matrices (applying the distinct architecture constraint).

5) 3 vectors, each from one group from groups 2, 3 and 4 ((2,3,4), (2,3,5), (3,4,5)) generate another 16 valid and unique connection pattern matrices (applying the distinct architecture constraint).

There are therefore a total of $6 + 3 + 16 = 25$ unique architectures possible.

$$(3) P_3 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right\}$$

We can group the five vectors into 3 groups:

1. $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$

2. $\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}$

3. $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}$

Again, by applying similar consideration:

1) 3 vectors from each group (we can represent this as (1,1,1), (2,2,2) and (3,3,3)). These combinations cannot form any valid connection pattern matrix since the two rows of the matrices formed this way are not linearly independent of each other.

2) 1 or 2 vector(s) from group 1 combined with 2 or 1 vector(s) from one group out of groups 2, 3 and 4 ((1,1,2), (1,2,2), (1,1,3), (1,3,3)) will not generate any valid connection

pattern matrix because such a matrix either has a row with all zeros or has both of its rows being equal.

3) 1 vector from groups 2, 3, or 4 and 2 vectors from one group (different from the first one) out of groups 2, 3 and 4 ((2,3,3), and (3,2,2)) generate 6 valid and unique connection pattern matrices (applying the distinct architecture constraint).

4) 1 vector from group 1 and 2 vectors from two groups out of groups 2, 3 and 4 ((1,2,3)) generate 3 valid and unique connection pattern matrices (applying the distinct architecture constraint).

There are therefore $3 + 6 = 9$ valid architectures possible for P_3 .

APPENDIX B

BNF OF THE INPUT SPECIFICATION LANGUAGE

The following the BNF of the input specification language.

```

<program> ::= (program <domain> <fun-def>*)
<domain> ::= (bounds <guard-expr>*)
<guard-expr> ::= (<relational-op> <linear-expr>
<number>)
| (<relational-op> <linear-expr>
<linear-expr>)
<relational-op> ::= < | <= | = | >= | >
<linear-expr> ::= (+ <term>*) | <term>
<term> ::= (* <number> <index>)
| (* <number> <size param>) | <index>
| <size-param>
<fun-def> ::= (<fun-name> <formals> <fun-body>)
<fun-name> ::= <id>
<formals> ::= ( <index>+ )
<index> ::= i | j | k
<fun-body> ::= (cond <clause>*)
<clause> ::= (<guard-conjunction> <clause-body>)
<guard-conjunction> ::= (and <guard-expr>*) | <guard-expr> |
else
<clause-body> ::= <input-body> | <guarded-output> <body>
|
<input-body> ::= (input <input-name> <linear-expr>*)
<guarded-output> ::= (output <output-name> <linear-expr>*)
| (if <guard-expr> (output <output-name>
<linear-expr>*))
<body> ::= <any Scheme expression>

<input-name> ::= <id>
<size-param> ::= [1, m, n]
<number> ::= [0-9]*
<id> ::= [a-z, A-Z] [a-z, A-Z, 0-9]*

```

APPENDIX C

TARGET ARCHITECTURE SPECIFICATION

The following is an example of an architecture specification generated by the systolic array generator.

The following architecture is selected:

```
((0 1 0) (0 0 1)) (1 1 1))
```

The following is the processor array data-structure:

```
##((ptype-14 ((acc12 float 1) (aco13 float 1))
  ((c8 bool 1 data-input)
   (c5 bool 1 data-input)
   (y9 float 1 data-input)
   (x10 float 1)
   (y11 float 1 data-input))
  ((c8 bool (1) (c8 2 1))
   (c5 bool (1) (c5 2 1))
   (y9 float (1) (data-output))
   (x10 float (1) (x10 2 1))
   (y11 float (1) (data-output))))
0
0
)
#((ptype-15 ((acc12 float 1) (aco13 float 1))
  ((c8 bool 1)
   (c5 bool 1)
   (y9 float 1 data-input)
   (x10 float 1)
   (y11 float 1 data-input))
  ((c8 bool (1) (c8 3 1))
   (c5 bool (1) (c5 3 1))
   (y9 float (1) (y9 2 2))
   (x10 float (1) (x10 3 1))
   (y11 float (1 1) (y11 2 2))))
(ptype-14 ((acc12 float 1) (aco13 float 1))
  ((c8 bool 1 data-input)
   (c5 bool 1 data-input)
   (y9 float 1)
   (x10 float 1)
   (y11 float 1))
  ((c8 bool (1) (c8 3 2))
   (c5 bool (1) (c5 3 2))
   (y9 float (1) (data-output))
   (x10 float (1) (x10 3 2))
   (y11 float (1) (data-output))))
```

```

)
)
0
)
#((ptype-15 ((acc12 float 1) (aco13 float 1))
  ((c8 bool 1)
   (c5 bool 1)
   (y9 float 1 data-input)
   (x10 float 1)
   (y11 float 1 data-input))
  ((c8 bool (1) (disconnected))
   (c5 bool (1) (disconnected))
   (y9 float (1) (y9 3 2))
   (x10 float (1) (disconnected))
   (y11 float (1 1) (y11 3 2))))
(ptype-15 ((acc12 float 1) (aco13 float 1))
  ((c8 bool 1)
   (c5 bool 1)
   (y9 float 1)
   (x10 float 1)
   (y11 float 1))
  ((c8 bool (1) (disconnected))
   (c5 bool (1) (disconnected))
   (y9 float (1) (y9 3 3))
   (x10 float (1) (disconnected))
   (y11 float (1 1) (y11 3 3))))
(ptype-14 ((acc12 float 1) (aco13 float 1))
  ((c8 bool 1 data-input)
   (c5 bool 1 data-input)
   (y9 float 1)
   (x10 float 1)
   (y11 float 1))
  ((c8 bool (1) (disconnected))
   (c5 bool (1) (disconnected))
   (y9 float (1) (data-output))
   (x10 float (1) (disconnected))
   (y11 float (1) (data-output))))
)
)

```

The following are codes for each type of processor

```

(problem-16
  ((ptype-15
    (((acc12) (aco13))
     ((c8) (c5) (y9) (x10) (y11))
     ((c8) (c5) (y9) (x10) (y11)))
    ((begin (set! y9.out (- y9.in (* x10.in acc12)))
            (set! y11.out (- y9.in (* x10.in acc12)))
            (set! x10.out x10.in)
            (cond ((equal? c5.in 1) (set! acc12 y11.in)) (else))
            (if (equal? c8.in 1) (set! aco13 y9.out))
            (set! c8.out c8.in)
            (set! c5.out c5.in))))
  (ptype-14
    (((acc12) (aco13))
     ((c8) (c5) (y9) (x10) (y11))
     ((c8) (c5) (y9) (x10) (y11)))
    ((begin (set! y9.out (/ y9.in acc12))

```



```

(set! y11.out (/ y9.in acc12))
(set! x10.out y9.in)
(cond ((equal? c5.in 1) (set! acc12 y11.in)) (else))
(if (equal? c8.in 1) (set! acc13 y9.out))
(set! c8.out c8.in)
(set! c5.out c5.in))))))

```

The following are all input expressions

```

((y11 matrix-a
  ((1 1)
   (lambda (i j t) (+ (* 1 t) -1 (* -1 i))))
   (lambda (i j t) (* 1 i))))
  ((2 1)
   (lambda (i j t) (+ (* 1 t) -1 (* -1 i))))
   (lambda (i j t) (* 1 i))))
  ((3 1)
   (lambda (i j t) (+ (* 1 t) -1 (* -1 i))))
   (lambda (i j t) (* 1 i))))))
(y9 matrix-a
  ((1 1)
   (lambda (i j t) (+ (* 1 t) -1 (* -1 i))))
   (lambda (i j t) (* 1 i))))
  ((2 1)
   (lambda (i j t) (+ (* 1 t) -1 (* -1 i))))
   (lambda (i j t) (* 1 i))))
  ((3 1)
   (lambda (i j t) (+ (* 1 t) -1 (* -1 i))))
   (lambda (i j t) (* 1 i))))))
(c5 matrix-18
  ((3 3) (lambda (i j t) 1) (lambda (i j t) (+ t -9))))
  ((2 2) (lambda (i j t) 1) (lambda (i j t) (+ t -6))))
  ((1 1) (lambda (i j t) 1) (lambda (i j t) (+ t -3))))))
(c8 matrix-17
  ((3 3) (lambda (i j t) 1) (lambda (i j t) (+ t -8))))
  ((2 2) (lambda (i j t) 1) (lambda (i j t) (+ t -5))))
  ((1 1) (lambda (i j t) 1) (lambda (i j t) (+ t -2))))))

```

REFERENCES

1. Bruegge, B., Chang, C., Cohen, P., Gross, T., Lam, M., Lieu, P., Noaman, A., and Yam, D. Programming Warp. *Proc. COMPCON*, Spring 1987.
2. Cappello, P.R. and Steiglitz, K. Unifying VLSI Array Designs with geometric transformations. *Proc. IEEE International Conference on Parallel Processing*, 1983.
3. Chen, M. A Design Methodology for Synthesizing Parallel Algorithms and Architectures. *Journal of Parallel and Distributed Computing*, pp. 461-491, 1986.
4. Delosme, J. M. and Ipsen, I.C.F. An Illustration of a Methodology for the Construction of Efficient Systolic Architectures in VLSI. *Int Symp. on VLSI Technology, Systems and Applications*. Taipei, Taiwan, pp268-273, 1985.
5. Engstrom, B.R. and Cappello, P.R. The SDEF programming System. *Journal of Parallel and Distributed Computing* 7, 201-231 (1989).
6. Karp, R.M., Miller, R.E. and Winograd, S. The Organization of Computations for Uniform Recurrence Equations. *JACM* 14, 3, Jul. 1967, pp. 563-590.
7. Kothari, S.C., Gannett, E. and Oh, H. Optimal Designs of Linear Flow Systolic Architectures. *ICPP* 89.
8. Kung, H.T. and Leiserson, C.E. Algorithms for Processor Arrays. *In Introduction to VLSI Systems*. Addison-Wesley, Menlo Park, CA 1980.
9. Kung, H.T. Why Systolic Architectures. *Computer*, vol. 15 pp. 37-46, Jan 1982.
10. Li, G.J. and Wah, B.W. The Design of Optimal Systolic Arrays. *IEEE Trans. Comput.* C-34:66-67, 1985.
11. Miranker, W. L. and Winkler, A. Space-time Representation of Computational Structures. *Computing* 32:93-144, 1984.
12. Moldovan, D.I. ADVIS: a Software Package for the Design of Systolic Arrays. *IEEE Transaction on Computer-Aided Design*, vol. CAD-6, No. 1, Jan. 1987.

13. Moldovan, D.I. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, vol. 71, pp. 113-120, Jan. 1983.
14. Quinton, P., and Dongen, V.V. The Mapping of Linear Recurrence Equations on Regular Arrays. *Journal of VLSI Signal Processing*, 1, 95-113, 1989.
15. Quinton, P. Automatic Synthesis of Systolic Arrays from Uniform Recurrence Equations. *Proc. 11th Int'l. Symp. Computer Architectures*.
16. Rajopadhye, S.V. Synthesis, Verification and Optimization of Systolic Arrays. *Ph. D. Dissertation*, University of Utah, Dec. 1986.
17. Rao, S. and Kailath, T. Regular Iterative Algorithms and their Implementation on Processor Arrays. *Proc. IEEE*, Vol. 76, No. 3, March 1988.
18. Rote, G. On the Connection Between Hexagonal and Unidirectional Rectangular Systolic Arrays. *Technical Report*, Technische Universitat Graz, Institut fur Mathematik, Kopernikusgasse 24, A-8010 Graz, Austria.
19. Roychowdhury, V.P. and Rao, S.K., Thicle, L. and Kailath, T. On the Localization of Algorithms for VLSI Processor Arrays. *VLSI Signal Processing III*, Monterey, Ca, Nov. 1989, pp. 459-470.
20. Schrijver, A. Theory of Linear and Integer Programming. *Wiley-Interscience*, 1986.
21. Shang, W. and Fortes, A.B. On the Optimality of Linear Schedules. *Journal of VLSI Signal Processing*, 1, pp. 209-220, 1989.
22. Wong, Y. and Delosme, J.M. Optimization of Computation Time. *Research Report YALEU/DCS/RR-651*, May 1989.
23. Wong, Y. and Delosme, J.M. Optimization of Processor Count for Systolic Arrays. *Technical Report, YALEU/DCS/RR-607*, May 1989.
24. Wong, Y. and Delosme, J.M. Transformation of Broadcasts into Propagation in Systolic Algorithms. *Technical Report, YALEU/DS/RR-701*, May 1989.

25. Yaacoby, Y. and Cappello, P.R. Scheduling a System of Affine Recurrence Equations onto a Systolic Array. *Tech. Report TRCS 87-19*, University of California at Santa Barbara, Comput. Sci. Dept., Santa Barbara, Ca, Feb. 1988.