Chapter 5

Advanced Systolic Design

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

Abstract

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

5.1 Introduction

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

Systolic arrays immediately caught on, since they involved a fascinating interplay between algorithm and architecture design. When researchers started in-

DOCKET
ALARM
Find authenticated court documents without watermarks at docketalarm.com.

vestigating automatic synthesis, a third stream joined this confluence, namely the analysis, manipulation and transformation of programs. Some of the early designs are classics. The Guibas et al. array for optimal string parenthesization [2] is one of our all time favorites.

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

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

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

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

In the first part we use a formalism called systems of recurrence equations (SREs) to describe both the initial specification as well as the final array. The entire process of systolic design is viewed as the application of a series of transformations

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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

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

5.2 Systolic Design by Recurrence Transformations

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

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

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

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

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

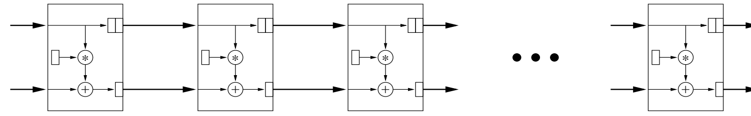

Figure 5.1 The systolic array for Eqns. (1–3).

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

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

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

5.2.1 Recurrence Equations : Definitions and Notation

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

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

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

where

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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

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

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

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

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

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

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

Domains

An important part of the SRE formalism is the notion of domain, i.e., the set of indices where a particular computation is defined. The domain of the *variables* of an SRE are usually specified explicitly. The domains that we use are *polyhedra*

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.