LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

Recent Developments in Systolic Design

by

Christian Lengauer
Jingling Xue

Recent Developments in Systolic Design

LFCS Report Series

ECS-LFCS-91-176

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

September 1991

Recent Developments in Systolic Design*

CHRISTIAN LENGAUER† JINGLING XUE†,‡
lengauer@dcs.ed.ac.uk xj@dcs.ed.ac.uk

LABORATORY FOR FOUNDATIONS OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF EDINBURGH
EDINBURGH EH9 3JZ
SCOTLAND

ECS–LFCS–91–176
SEPTEMBER 1991

Abstract

A systolic array is a network of processors that are locally and regularly connected. This special-purpose computing paradigm supports the parallel implementation of iterative algorithms in a variety of areas, e.g., numerical analysis, signal or image processing and graph theory.

The special appeal of systolic arrays is that they can be derived mechanically by provably correct and (in a sense) optimal synthesis methods. These methods transform algorithmic descriptions that do not specify concurrency or communication – usually functional or imperative programs – into functions that distribute the program's operations over time and space. This process is called systolic design. The distribution functions can then be refined further and translated into a description for fabrication of a VLSI chip or into a distributed program for execution on a programmable processor array.

Systolic design has received a lot of attention in the past decade. This paper is an overview and bibliography of recent results and current issues in the mechanical synthesis of systolic arrays.

---

*An abridged version of this paper was presented at the Int. Conf. on Parallel Computing Technologies (PaCT-91) in Novosibirsk, U.S.S.R., Sept. 1991.

†Supported by the Science and Engineering Research Council, Grant no. GR/G55457.

‡Supported by a University of Edinburgh Postgraduate Fellowship.

Contents

1 Introduction

With the advent of powerful VLSI design and fabrication techniques and of programmable processor networks, the reliable synthesis of multiple-instruction-multiple-data (MIMD) algorithms and their refinement into hardware and software has become increasingly attractive. The potential of VLSI for MIMD parallelism was first recognised by Kung and Leiserson [50], when they introduced the term *systolic array* to describe a processor network that was suitably restrained to meet the limitations of the then just emerging VLSI technology. The main restrictions are synchrony of the parallelism and locality of the interconnections between processors. The proceedings of recent yearly or bi-yearly conferences and workshops on regular array processors attest to the variety of areas in which systolic solutions have been proposed since then, e.g., numerical analysis, signal or image processing and graph theory. We refrain from tabulating individual examples here; there are too many and this is a review of systolic design, not of systolic algorithms. The proceedings of said conferences and workshops are referred to throughout the paper.

The special appeal of systolic arrays is that many can be derived by mechanical synthesis methods. These methods take a traditional (but very restricted) imperative or functional program and spread its individual operations over time and space. They are based on a geometric theory and deliver maximal parallelism, i.e., the shortest execution derivable from the source program, if the assumption is made that all individual operations take equal time. With the increasing possibilities promised by new computer architectures – increasing communication bandwidths and distances, decreasing communication cost, asynchronous parallelism in programmable processor arrays – the restrictions that characterise the systolic paradigm are losing their practical relevance. As this is being realised, the theory of systolic design is undergoing extensions to encompass less restricted regular processor arrays. Still, regularity is of prime importance; irregular optimisations or adaptations must be identified and implemented by hand.

We review the current status of systolic design. The paper is structured as follows:

Sect. 2. Initial methods could synthesise a class of (rather simple) systolic arrays autonomously with a guarantee of the correctness and time-minimality of the systolic execution. These are, essentially, systolic arrays whose source specifications can be expressed by one set of restricted nested loops (uniform recurrences).

Sect. 3. For a non-uniform source, human intervention is often necessary in the systolic design process, but there have been recent advances in the search for automatic schemes to turn more general forms of recurrences into uniform recurrences.

Sect. 4. Even when the syntactic requirements on the source can be satisfied, the systolic design process may still yield an unsatisfactory solution, for example, one with more processors or execution steps than are necessary to solve the problem. Systolic design methods guarantee time-minimality with respect to the source only. A different source may specify the same problem but lead to a faster systolic array.

Sect. 5. A uniform source guarantees linear distribution functions. Sometimes a systolic solution can be improved by breaking the linearity.

Sect. 6. A systolic design may be complicated by a limitation of the number or nature of processors or channel connections. Time-minimality can usually not be guaranteed in the presence of resource limitations.

Sect. 7. A realisation in hardware often requires control signals that tell cells what to do when. Methods for synthesising these signals exist and are being improved.

Sect. 8. A realisation in software requires the generation of computation and communication code for each cell and communication code for the injection and extraction of data to and from the array.

Sect. 9. Present programmable processor networks are not supporting parallelism of sufficiently low granularity to be ideally fit for systolic computations, but more suitable architectures are under development.

Sect. 10. Theoretical issues of systolic arrays and systolic design are also reviewed.

This survey is intended to be comprehensive, but it cannot be complete. The focus is on mechanical systolic design based on uniform recurrences. Elsewhere there is an overview that also considers other, earlier methods [29].

2 Uniform Systolic Design

2.1 The Source Description

The starting point of the mechanical systolic design methods that we review is a functional program [18, 71, 78] or an imperative program [40, 56], both without parallel commands and communication directives. Both adhere to the format:

$$\begin{array}{l}\textbf{for } i_1 \textbf{ from } lb_1 \textbf{ by } st_1 \textbf{ to } rb_1 \textbf{ do}\\ \quad\vdots\\ \textbf{for } i_r \textbf{ from } lb_r \textbf{ by } st_r \textbf{ to } rb_r \textbf{ do}\\ \quad i_1{:}\cdots{:}i_r\end{array}$$

The loops define a space Φ of points $(i_1, \cdots, i_r)$, called the *index space* ($\Phi \subset \mathbb{Z}^r$); the coordinates of the points are the loop indices. The bounds lb_k and rb_k are linear expressions in the loop indices i_1 to i_{k-1} $(0 < k \leq r)$ and in additional variables that specify the problem size; the bounds must make the index space a convex polyhedron [85]. The steps st_i are constants. We call $i_1{:}\cdots{:}i_r$ the *basic operation* of the program. It can be viewed as a procedure with r parameters whose body is of the form:

$$\begin{array}{ll} i_1{:}\cdots{:}i_r :: & \textbf{if } B_0(i_1, \cdots, i_r) \rightarrow S_1\\ & \quad\vdots\\ & [\!]\ B_t(i_1, \cdots, i_r) \rightarrow S_t\\ & \textbf{fi}\end{array}$$

$B_j \rightarrow S_j$ is called a *guarded command* [26]. The *guards* B_j are side-effect-free predicates that relate the loop indices; the *computations* S_j are functional or imperative but must be

without non-local references other than to the variables indexed by the loop indices. There is a formal correspondence between the functional and imperative style [14, 61].

In the functional style, a computation must contain a set of *uniform recurrence equations* (UREs) [71, 78], i.e., equations of the form:

$$v(I) = f_v(w(J), \cdots) \qquad \text{where} \quad I \in \Phi \quad \text{and} \quad J = I - \vartheta_v$$

The three dots in the argument list of f_v stand for an arbitrary but fixed number of similar arguments. The constant vectors ϑ_v are called *dependence vectors*; the (partial) order imposed on the computation by these vectors may not be violated in the systolisation. (For notational convenience, we assume that there is only one dependence vector per variable.) The dependence vectors form the columns of the *dependence matrix*. The graph whose vertices are the basic operations and whose edges are the dependences between the basic operations is called the *dependence graph*. This graph changes with the size of the problem. The graph whose vertices are the variable names and whose directed edges connect the arguments with the results of the UREs is called the *reduced dependence graph*. It is independent of the problem size.

In the imperative style, a computation may specify a reassignment, each non-local variable is subscribed by $r-1$ linear expressions in the loop indices, their coefficient matrix has rank $r-1$ and non-local variables in a fixed computation that agree in name must agree in subscripts. The missing index is made superfluous by the introduction of the re-assignment. The parallelisation is based on the concept of an *independence relation*, a semantic condition that ensures the correctness of the parallel composition of two basic operations and that is declared as follows [40]:

$$\textit{independence criterion} \implies i_1 : \cdots : i_r \text{ ind } j_1 : \cdots : j_r$$

The most general definition of semantic independence is rather complex [13]; generally more restrictive, simpler conditions are used. Uniform systolic design enforces the pipelining of all variables accesses. The according independence criterion is the absence of shared variables.

The product of two square matrices is the standard example in the literature for the illustration of concepts in systolic design.

Example: $n \times n$ Matrix Product $(\forall\, i,j : 0<i,j \leq n : c_{i,j} = (\sum\, k : 0<k \leq n : a_{i,k} \cdot b_{k,j}))$

Loops: **for** i **from** 0 **to** n **do**
for j **from** 0 **to** n **do**
for k **from** 0 **to** n **do**
$i{:}j{:}k$

Index Space: $\Phi = \{(i,j,k) \mid 0<i,j,k \leq n\}$

Basic Operation (imperative): $i{:}j{:}k ::$ **if** $k=0 \rightarrow c_{i,j} := 0$
[] $k \neq 0 \rightarrow c_{i,j} := c_{i,j} + a_{i,k} \cdot b_{k,j}$
fi

Independence Relation:

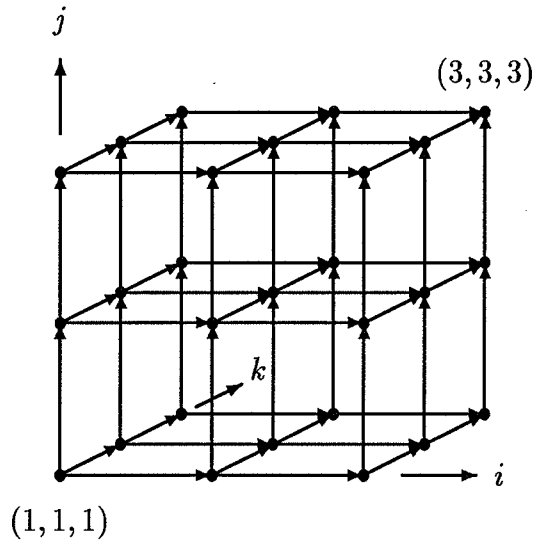
$$(i_0,j_0)\neq(i_1,j_1)\wedge(i_0,k_0)\neq(i_1,k_1)\wedge(k_0,j_0)\neq(k_1,j_1) \implies i_0{:}j_0{:}k_0 \text{ ind } i_1{:}j_1{:}k_1$$

Basic Operation (functional):

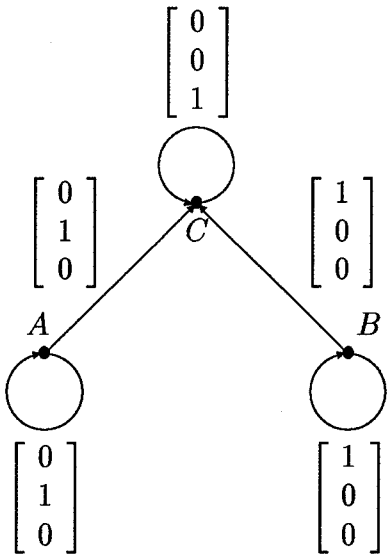
$$\begin{aligned}
i{:}j{:}k :: \quad & \textbf{if } j{=}0 \to A(i,j,k) = a_{i,k} \\
& [] \; j{\neq}0 \to A(i,j,k) = A(i,j{-}1,k) \\
& \textbf{fi} , \\
& \textbf{if } i{=}0 \to B(i,j,k) = b_{k,j} \\
& [] \; i{\neq}0 \to B(i,j,k) = B(i{-}1,j,k) \\
& \textbf{fi} , \\
& \textbf{if } k{=}0 \to C(i,j,k) = 0 \\
& [] \; k{\neq}0 \to C(i,j,k) = C(i,j,k{-}1){+}A(i,j{-}1,k){\cdot}B(i{-}1,j,k) \\
& \textbf{fi} , \\
& c_{i,j} = C(i,j,n)
\end{aligned}$$

Dependence Matrix: $[\vartheta_A,\vartheta_B,\vartheta_C] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Dependence Graph ($n{=}3$):



Reduced Dependence Graph:



(*End of Example*)

2.2 The Target Description

The systolic array is described by two functions; further functions are derived from them. Let $\mathbb{Z}$ denote the integers and $\mathbb{Q}$ the rationals:

$\mathsf{step} : \Phi \longrightarrow \mathbb{Z}, \quad \mathsf{step}(I) = \lambda I, \quad \lambda \in \mathbb{Z}^r.$

step specifies the temporal distribution. λ is the *scheduling vector.* I is computed at step λI.

$\mathsf{place} : \Phi \longrightarrow \mathbb{Z}^{r-1}, \quad \mathsf{place}(I) = \sigma I, \quad \sigma \in \mathbb{Z}^{(r-1)\times r}.$

place specifies the spatial distribution. σ is the *allocation matrix.* I is computed at location σI.

The combination of **step** and **place** is called the *space-time mapping.* The square matrix in which the coefficients of **step** occupy the first row and the coefficients of **place** occupy all other rows is called the *index transformation matrix.*

The mechanical derivation of a shortest execution is always possible in uniform systolic design; it employs techniques of integer programming in the functional style [78] and of program transformation in the imperative style [40]. One chooses a compatible distribution in space by a search [40, 65]. With this choice, one can also optimise other aspects of the array, e.g., the throughput or the number of processors of channels or storage registers per processor.

step and **place** each partition the index space; all domain elements with the same image form one partition. The partitions defined by **step** are slices (i.e., parallel hyperplanes [69]). The partitions defined by **place** are parallel lines that are defined by a common vector, the *iteration vector* u, satisfying $\sigma u = 0$. If the iteration vector also satisfies the condition $\lambda u > 0$, it specifies the iteration direction in the respective processors and can be viewed as the most concise characterisation of the choice of systolic array.

If **step** and **place** are given, the direction, speed and layout of all data travelling through the array are determined. Let V be the set of variables of the program:

$\mathsf{flow} : V \longrightarrow \mathbb{Q}^{r-1}, \quad \mathsf{flow}(v) = \sigma\vartheta_v / \lambda\vartheta_v, \quad \vartheta_v \in D.$

flow specifies the velocity with which elements of a stream travel at each step. Stream v is called *moving* if $\mathsf{flow}(v) \neq 0$ and *stationary* if $\mathsf{flow}(v) = 0$. ($\vartheta_v \in D$ means that ϑ_v is a column of matrix D.)

$\mathsf{input} : V \longrightarrow \Phi \longrightarrow \mathbb{Z}^{r-1}, \quad \mathsf{input}(v(I)) = \mathsf{place}(v(I)) - (\mathsf{step}(v(I)) - fs) \cdot \mathsf{flow}(v(I)).$

input specifies the layout of the data at the first execution step, *fs.* (This function has also been called **pattern** [40].) The layout at the k-th following step is $\mathsf{input}(v(I)) + k \cdot \mathsf{flow}(v(I))$.

Example: 3×3 Matrix Product

Partitions:

Fig. 1. $\mathsf{step}(i{:}j{:}k) = i + j + k$ is time-minimal for the previously presented set of UREs.

Systolic Arrays (at the first step; imperative source):

Figs. 2, 3. The dark cell is active. (*End of Example*)

2.3 Functional vs. Imperative

As is usually the case for functional vs. imperative programs, the functional style is based on a richer and cleaner theory. The systolisation in the functional style is independent of the problem size. Integer programming is NP-complete [34] but it is frequently asserted that, in the restricted setting of systolic arrays, it can often be solved efficiently.

Only very little work has addressed this problem. The minimisation of the step function with respect the computation of one specific variable (rather than all variables) can be formulated as a linear programming problem [87, 88]. The step function can also be minimised by combinatorial optimisation. A branch-and-bound method has been proposed and demonstrated to be effective on practical problems [95]. A method for minimising the number of processors in the systolic array has also been reported [96]. The solution space of projection vectors is bounded for a given step function; the minimising place function is obtained by enumeration.

To systolise a program in the imperative style, its loops must be expanded for a fixed problem size. Generalisation techniques of the kind described in [56] can then make the step function independent of the problem size. The imperative method can take other semantic aspects than independence into account, like idempotence and neutrality [40] (an idempotent operation has the same effect executed once or any number of times in succession; a neutral operation has no effect). Its parallelisation scheme can also generate non-systolic solutions [57]. There has been much, mostly informal work on the parallelisation of non-uniform loop programs; we do not review it here [92].

2.4 Implementations

Early rudimentary implementations of the functional style include DIASTOL [31, 32], ADVIS [66] and SYSTOL [68] among others [91]. A more elaborate effort is Crystal; it started out as a system for systolic design [18] and subsequently targeted other machine models like Hypercubes [19]. More recent systems are PRESAGE [91] and ALPHA [33, 60]; they can assist in non-uniform systolic design (Sect. 3). PRESAGE respects resource limitations (Sect. 6); so do Systars [70] and HIFI [5].

A different set of tools is not based on recurrence equations but on some subset of the functional language FP [6] for regular array design – the most prevalent is μFP [63]. These tools help the human interactively transform a simple functional specification into a program that represents an array. This approach is not based on the theory of uniform recurrences but on the theory of functional program transformations. This is particularly useful for reasoning about and comparing systolic arrays, as various interpretations and metrics can be applied [62].

The imperative method has also been implemented [40, 56] and used in the development of a complex new systolic array [59].

3 Syntactic Adaptations

Uniformity is a rather restrictive requirement on recurrences. Many loop algorithms are more easily phrased in a non-uniform way. Research is presently being conducted on the

transformation of non-uniform into uniform recurrences. In cases where the deviation from uniformity is not too serious, mechanical methods can be hoped for.

Recently, methods of transforming affine recurrences (AREs) into uniform recurrences have appeared [72, 73, 75, 93]. An *affine recurrence equation* differs from a uniform recurrence equation only in the format of the arguments' index vectors. In an affine recurrence, it is $J = \Delta_v I - \vartheta_v$, where the *affinity matrix* Δ_v is a constant $r \times r$ matrix; in a uniform recurrence, Δ_v is the identity matrix. Δ_v may be singular; if so, the dependence is non-injective. Only injective dependences enforce the full data pipelining required in systolic arrays.

Example: $n \times n$ Matrix Product

Loops, Index Space, Basic Operation (imperative): as in Sect. 2.1.

Independence Relation: $(i_0, j_0) \neq (i_1, j_1) \implies i_0{:}j_0{:}k_0$ ind $i_1{:}j_1{:}k_1$

Basic Operation (functional):

$$
\begin{aligned}
i{:}j{:}k :: \quad & A(i,0,k) = a_{i,k} \ , \\
& B(0,j,k) = b_{k,j} \ , \\
& \textbf{if } k{=}0 \rightarrow C(i,j,k) = 0 \\
& [] \ k{\neq}0 \rightarrow C(i,j,k) = C(i,j,k{-}1) + A(i,0,k) \cdot B(0,j,k) \\
& \textbf{fi} \ , \\
& c_{i,j} = C(i,j,n)
\end{aligned}
$$

Affinity Matrices: $\Delta_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $\Delta_B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $\Delta_C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Dependence Vectors: $\vartheta_A = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$, $\vartheta_B = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$, $\vartheta_C = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

(*End of Example*)

Uniformisation methods amount to the introduction of more pipelining by adding compatible dependences. In the matrix product, the added dependences are in the second index of A and the first index of B (see Sect. 2.1). Uniformisation methods can also handle data reflections in the systolic array, i.e., the case where **flow** is not constant for a variable.

One can obtain a linear step function directly from AREs with index vectors of one common length [72, 75, 101]. This may yield a faster execution than can be obtained via a uniformisation. For example, for the matrix product one obtains an execution that is three times as fast: $\textsf{step}(i{:}j{:}k) = k$. However, only place functions for UREs, not for AREs, enforce local communication (i.e., a bound on the communication length).

A further mild deviation from the ARE format is a set of nested loops with several basic operations (where scoping is indicated by indenting) [59]:

$$
\begin{array}{l}
\textbf{for } i_1 \textbf{ from } lb_1 \textbf{ by } st_1 \textbf{ to } rb_1 \textbf{ do} \\
\quad \textbf{for } i_2 \textbf{ from } lb_2 \textbf{ by } st_2 \textbf{ to } rb_2 \textbf{ do} \\
\quad\quad \textbf{for } i_3 \textbf{ from } lb_3 \textbf{ by } st_3 \textbf{ to } rb_3 \textbf{ do} \\
\quad\quad\quad i_1{:}i_2{:}i_3 \\
\quad\quad i_1{:}i_2
\end{array}
$$

This can be made uniform by letting the inner loop apply $i_1{:}i_2$ in an added step. In the functional setting, a source of this form corresponds to a set of AREs in which the length of the index vector varies between different variables; it is known how to derive a linear step function in this case [64].

4 Semantic Adaptations

The time-minimality of the systolic solution is only with respect to the source program. Different source programs with the same input/output behaviour may prescribe different dependences and therefore lead to systolic solutions of varying quality. This occurs, for example, in the presence of commutative or associative operations.

1. $n \times n$ Matrix Product:

   The cumulative additions ("inner product steps") [50] in a matrix product can be performed in any order, but one must choose a fixed accumulation order in both the imperative [40] and the functional source [31, 72]. In certain situations (e.g., band matrices [40]), the choice of order makes a difference to the execution time of the systolic solution.

2. Optimal String Parenthesisation by Dynamic Programming:

   $$(\forall\, i,j : 0<i,j\leq n : c_{i,j} = w_{i,j} + (\min\ k : i<k<j : c_{i,k}+c_{k,j}))$$

   The minimum must be accumulated from the middle to the two ends of the interval (i,j) in order to make a systolisation possible; this keeps the treatment of i and j symmetric [32, 39, 73].

There are other examples. If the expense in processors and communication is justified, a sequence of associative operations can be turned into a tree, in which each level of nodes can be performed in parallel, from the leaves to the root [59]. A small collection of semantic transformation techniques has been proposed recently [74].

The size of the search space makes a systematic identification of useful application orders a particular challenge which has not been addressed by systolic design so far. A desirable specification would be a minimally ordered source program, as is possible in GAMMA [7, 8] in the functional and Unity [17] in the imperative style. But these languages are too general to support an implemented scheme of systolic design. They leave too many implementation choices because they are targeted at a much wider range of architectures than systolic arrays. To support the mechanical development of systolic implementations, we need to address issues in the specification that need not be addressed in GAMMA or Unity programs. One requirement is that a dependence graph must be constructible from the specification.

5 Non-Linear Distributions

Uniform systolic design delivers linear distribution functions. In certain cases, non-linear distribution functions improve a systolic solution or make it at all possible. Non-linearity usually means piecewise linearity. We give two examples:

1. Processor-Time Minimal $n \times n$ Matrix Product:

   Cappello [16] presents a systolic array for the square matrix product that has $\lceil (3/4)n^2 \rceil$ processors and requires $3n-2$ execution steps. The best linear distribution requires n^2 processors. Cappello saves a quarter of the processors by overlapping the corner of the array that is inactive in the first few steps with the corner that is inactive in the last few steps. To keep communications neighbouring, the two-dimensional array must be made cylindrical, i.e., curved in the third dimension. $\lceil (3/4)n^2 \rceil$ processors is minimal: the step with the most parallelism requires that many.

2. The Algebraic Path Problem

   The algebraic path problem [80, 81], instances of which are the shortest paths problem and reflexive transitive closure, is solved by Gauss-Jordan elimination. It can be divided into three successive phases [81]:

   (a) the decomposition of the input matrix into two triangles,

   (b) the inversion of the two triangles, and

   (c) the multiplication of the inverted triangles.

   The step function of the according systolic solutions is piecewise linear with one linear piece per phase [41, 52]. The previously mentioned uniformisation methods have been tested on the algebraic path problem [72, 73].

   The algebraic path problem has been the most popular object of systolic design; new solutions are still emerging. Arrays have been proposed that are processor-time minimal for the standard source, with roughly $5n$ steps and $\lceil n^2/3 \rceil$ processors for an $n \times n$ matrix [12, 84]. Again, they were derived by overlapping portions of the array that are not active simultaneously.

Similar folding ideas have led to a general method of processor-time minimal systolic design [21].

6 Resource Limitations

Systolic design methods that return a time-minimal solution disregard resource limitations. Resource limitations fall into three classes.

6.1 Lack of Dimensions in the Processor Space

If the systolic design postulates more dimensions of processors than are available, a *projection* of the processor space becomes necessary. Dimensions of the processor space can be traded off to time, yielding a multi-dimensional step function, which can then be made one-dimensional again [98].

6.1.1 Projection of the Source Program

The simplest way of projecting is by absorbing loops in the source program inside the basic operation [59]. We demonstrate with the imperative source for the matrix product.

Example: $n \times n$ Matrix Product

Loops: **for** i **from** 0 **to** n **do**
for k **from** 0 **to** n **do**
$i{:}k$

Basic Operation (imperative): $i{:}k ::$ **for** j **from** 0 **to** n **do**
$i{:}j{:}k$

Independence Relation: $i_0 \neq i_1 \wedge k_0 \neq k_1 \implies i_0{:}k_0 \text{ ind } i_1{:}k_1$

$i{:}j{:}k$ is defined as previously. This specification leads to the same systolic arrays as for the matrix-vector product [40]. But, as can be seen by the hidden loop in the basic operation $i{:}k$, the streams with only one index contain vectors rather than scalars.

(*End of Example*)

6.1.2 Projection During the Systolic Design

There are a number of methods that accept arbitrarily many nested loops. Most work in this area has focussed on the synthesis of arrays with a one-dimensional processor layout – either approaching one-dimensional time directly or via multi-dimensional time.

In the first approach [55, 76, 98], one applies a $2 \times n$ index transformation to the source UREs. In the transformed UREs, one index represents time and the other space. This approach relies on heuristics-based techniques but can easily take any type of constraints in the specification into account.

In the second approach [46, 51, 78, 93], one applies an $n \times n$ index transformation matrix to the source UREs. In the transformed UREs, n–1 indices represent time and the remaining index represents space. Then, $(n-1)$-dimensional time is converted to one-dimensional time. This approach relies on integer programming, which makes it inflexible when the specification is constrained.

6.1.3 Projection of the Target Program

When a software representation is targeted, one can impose the projection on the distributed program rather than earlier in the design process. The projection then takes the form of a program transformation that aggregates formerly concurrent processes into one sequential process. Some languages provide process placement directives for this kind of aggregation (e.g., occam [42, 43]). This may involve potentially inefficient intra-processor communications.

While it is advisable to impose resource limitations as late in the programming process as possible, there are indications that the projection of systolic programs proceeds at too low a level: the regularity of channel communications is not properly recognised and gives rise to needless inductions in the description of the projection. Even the square matrix product example becomes unwieldy [58].

6.2 Lack of Processors in a Given Dimension

If a given dimension of the processor layout requires more processors than are available, a partitioning of the array is required. In *partitioning*, several cells of the ideal array are mapped to the same processor of the available array. There are two different approaches:

- The *locally parallel and globally sequential* strategy [51, 67], partitions the array into blocks such that the *number of cells per block* matches the available number of processors. The blocks are superimposed; cells that are at the same position in different blocks are mapped to the same processor and emulated in sequence. Opposite border processors in the partitioned array are connected by extra "wrap-around" channels. Extra memory is required at the boundary processors of the partitioned array to buffer data on the extra channels; the amount of memory required grows with the block size. This partitioning scheme does not work if the array to be partitioned contains counter flows.

- The *locally sequential and globally parallel* strategy [15, 21, 24, 102] partitions the array into blocks such that the *number of blocks* matches the available number of processors. All cells of that make up one block are mapped to the same processor. Again, cells that are mapped to the same processor are emulated in sequence. This approach requires extra memory, proportional to the number of blocks, at every processor of the partitioned array but does not exclude any combination of flows in the ideal array.

In order to simplify the code generation in a software realisation and the control signal generation in a hardware realisation, a partitioning must be regular. It should also minimise the number of processors and channels [24].

6.3 Lack of Channels

If channels postulated by the systolic design do not exist, a *re-routing* along existing channels becomes necessary. Simple cases of re-routing have been addressed by uniformisation methods [30, 72, 73, 94] (compare Sect. 3). AREs may specify an unbounded number of channels (i.e., broadcasts) or channels of unbounded length. UREs enforce pipelining and

local connections. The basic idea of re-routing methods is to transform the non-uniform, non-constant and broadcast data dependences in the ARE source program into uniform, constant and pipelining data dependences. This requires the introduction of additional data dependences. They can be:

- the null bases of the affinity matrices specified by the AREs [73, 72],
- canonical or non-canonical bases of the index space [94], or
- the extreme rays of the (pointed) cone generated by the original dependence vectors in the source program [72].

Uniformisations are not unique. Uniformisation methods provide a choice of many solutions without providing a measure of the quality of the resulting systolic array. The methods of [73, 94] require the affinity matrices to be singular. The method of [94] does not guarantee the existence of a step function but it guarantees the existence of multiple step functions as described in [78], one for each variable.

On programmable processor networks, system software may be able to perform the routing [20, 27]. On newer networks, the hardware performs this task [2, 86].

7 Hardware Realisation

If the systolic array is realised in hardware and requires a cell to perform different operations at different steps, the cell must be told when to perform what operation. It was first suggested a decade ago that this could be done by communicating *control signals* in addition to the variables needed for the computation [39]. Later, formal treatments of this idea emerged [18, 89].

These methods aim at a time-minimal array with a dimension one less than that of the index space. They have the drawback that the control signals are not derived not until the space-time mapping has been determined, make no allowances for resource limitations and do not apply for the generation of fixed-size or projected systolic arrays. Recently, these problems have been overcome by specifying the control signals by a system of UREs called *control UREs*, which has no notion of space and time. The space-time mapping technique is then simply extended to synthesise the data and control flow together.

The state of current VLSI technology imposes restrictions on the communication in the systolic array. This complicates the specification of the control UREs for systolic arrays of reduced dimension (Sect. 6.1.3). For example, in matrix product, no control signals are necessary in two-dimensional arrays, but control signals are generally indispensable in one-dimensional arrays. There are two types of space-time mapping: those that describe systolic arrays of dimension $n-1$ [100] and those that describe systolic arrays of reduced dimension [99]. The specification of systolic arrays of reduced dimension is complicated by the fact that their space-time mapping is not a bijection from $\mathbb{Q}^n$ and $\mathbb{Q}^n$ (it is not injective).

At present, there is no systematic method for specifying the control that governs the access of stationary streams. This problem may be best addressed in an architecture-dependent way.

8 Software Realisation

Systolic arrays can be viewed as synchronous programs: the index transformation yields recurrence equations over space-time [18] that can be coded in programming languages like ALPHA [33, 60], GAMMA [7, 8] or Unity [17]. The matrix product is less easily expressed in GAMMA than in the other languages, because GAMMA has only one data structure: the multiset.

The parallelisation of for-loops has been an active area of research for many years [54, 92]. Most of this work focuses more on the issue of concurrency than on the issue of communication because, predominantly, vector processors and other shared memory machines have been targeted.

Many commercial processor networks, like Warp [4] and transputer networks [35], have asynchronous parallelism and no shared memory. Recently, work on the synthesis of interprocess communication directives for such architectures has appeared – in the setting of systolic arrays, we call it *systolising compilation.* The system SDEF [27] takes a set of uniform recurrence equations and the space-time mapping derived from it and fills a distributed program skeleton in C with the appropriate computations and communications. The program code is for a fixed problem size, i.e., the loop limits must be constants.

A systolic program for the algebraic path problem was generated with an extension of another mechanical systolising compilation scheme [56]. It also takes the source program and the space-time mapping as input, but it generates abstract distributed program code that is parameterised with the problem size and that can be manipulated further, e.g., optimised or translated to a target language. The initial version of this scheme works only for place functions that are projections along one axis of the index space; they correspond to parallelisations of one or more loops in the source program. A more recent version works for all linear place functions [9, 10].

Ribas [79] and Ramanujam and Sadayappan [77] generate more concrete and efficient program code. They address resource limitations by projection and partitioning (Sect. 6), but put more restrictions on the source. That is, they handle a smaller class of systolic arrays.

Most of the work on systolising compilation uses matrix product as an example [10, 27, 56, 79].

9 Programmable Processor Networks

Following the taxonomy of Seitz [86], we classify networks with 2^n processors as *large-grain* if $n \approx 2$, *medium-grain* if $n \approx 8$ and *fine-grain* if $n \approx 14$.

9.1 Medium Grain

Past commercially available medium-grain processor networks were not ideally suited for systolic computations, mostly, because of their inflexible and too slow inter-processor communication. An example is the INMOS T800 transputer [44]. It is restricted to four channel connections and its communication speed is considerably slower than its computation speed.

Warp was ahead of its time in terms of communication speed. With arithmetic pipelining in its ten cells, it transcends the traditional systolic paradigm that forms the basis of mechanical systolic design methods. Therefore code generation for Warp has focussed on techniques outside the realm of traditional systolic design [53, 90], but systolic programs have also been generated for Warp [79].

Recently, there have been significant advances in the power of medium-grain processor networks.

The T9000 transputer [45] is the descendant of the T800. It has eight connections per processor (which makes complete neighbour connections in a two-dimensional systolic array possible) and comes with a powerful VLSI routing chip, the C104, that can fully interconnect 32 transputers.

*i*Warp [11] is the successor of Warp [4]. Like the T9000, it provides eight connections per processor and improved performance and communication schemes. With four processors per chip, arrays can contain up to 2^{10} processors.

9.2 Fine Grain

Fine-grain processor arrays have a higher number of processors with less computational power and storage capacity than medium-grain networks.

9.2.1 Bit-Level

In bit-level architectures, each processor holds one bit and is capable of simple logic operations.

The Computational Array Logic (CAL) [36, 49] is a two-dimensional bit-level processor array. A connection pattern between neighbouring processors can be specified and quickly (in below 100 μs) altered by software. A fast routing scheme aims to minimise delays in non-neighbour communications.

The Xilinx [97] has more complex cells than CAL and is a commercially maturer product. The Altera [3] has more sophisticated cells still, but uses EPROM rather than RAM technology, i.e., its routing pattern cannot be changed dynamically.

These networks can emulate bit-level systolic arrays; for examples, see any conference proceedings on systolic arrays. The arithmetic required for algorithms that use integer, or even real numbers must be explicitly configured on bit-level networks; this quickly requires an inordinate amount of cells.

9.2.2 Word-Level

Mosaic, under development at CalTech [86], will be a two-dimensional array of up to 2^{14} processors, each with a moderate amount of memory (64K) and with connections to its neighbours. An automatic routing scheme supports communication between any two processors; one specifies the position of the target processor relative to the source processor.

The Rewrite Rule Machine (RRM), under development at SRI [2], is planned to be a completely scalable massively parallel processor network, with hundreds of processors on a chip. Up to one hundred boards, each comprising up to one hundred chips, form the complete

network. Each chip operates in SIMD mode internally, while the chips operate in a MIMD mode collectively. The RRM is planned specifically for the execution of programs expressed as rewrite rules [1]. Recurrence equations can be viewed as rewrite rules.

10 Theoretical Issues

10.1 Computability

A set of uniform recurrence equations is called *explicitly defined* [48] or *computable* [72] if none of its variable instances depends on itself. The computability of uniform recurrence equations has been investigated widely [25, 48, 78, 83, 101]; it can be reduced to the halting problem for a Turing machine and is, therefore, undecidable [47]. The undecidability arises out of the possible unboundedness of the index space. For a fixed problem size (and, therefore, a finite index space), a set of UREs is computable, if its dependence graph is acyclic.

Karp et al. [48] point out the decidability of strict UREs. In *strict* UREs, each equation depends directly on each of its arguments. Strict UREs are computable, if the reduced dependence graph does not contain non-positive cycles.

The strong point of systolic design is that the existence of a *linear* step function for a set of uniform recurrence equations can be established automatically.

10.2 Systolic Automata

Systolic automata theory is an automata and complexity oriented discipline that has been dealing with a restricted class of problems that have a systolic solution in two-dimensional space-time.

Systolic automata are languages acceptors that are models of processor networks, usually in an array, trellis or tree-like form [37, 38]. (If the network is two-dimensional, it emulates space-time directly and has a one-dimensional time-minimal equivalent.)

Various basic problems concerning systolic architectures and computations have been studied on such models; examples are the computational power of various interconnection patterns (e.g., two-way vs. one-way), different ways of interacting with the external environment and different types of heterogeneity. Problems of classical language theory have also been investigated [22, 28].

Special attention has been devoted to the comparative study of the recognition power of classes of systolic automata with different underlying structure. Some restricted but useful design methods can be extracted from this theory [23] and have been applied in the improvement of a systolic array for string correction [82].

11 Acknowledgements

Thanks to Mike Barnett and Davide Sangiorgi for assistance.

12 References

[1] H. Aida, J. A. Goguen, and J. Meseguer. Compiling cuncurrent rewriting onto the rewrite rule machine. In S. Kaplan and M Okada, editors, *Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science. Springer-Verlag, 1990. Also: Technical Report SRI-CSL-90-03R, SRI International, Dec. 1990.

[2] H. Aida, S. Leinwand, and J. Meseguer. Architectural design of the rewrite rule ensemble. In J. Delgado-Frias and W. R. Moore, editors, *Proc. Int. Workshop on VLSI for Artificial Intelligence and Neural Networks*, 1990. Also: Technical Report SRI-CSL-90-17, SRI International, Dec. 1990.

[3] Altera Corporation. *Data Book.* Altera Corporation, 1990.

[4] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Trans. on Computers*, C-36(12):1523–1538, Dec. 1987.

[5] J. Annevelink and P. Dewilde. HIFI: A functional design system for VLSI processing arrays. In K. Bromley, S.-Y. Kung, and E. E. Swartzlander, editors, *Proc. Int. Conf. on Systolic Arrays*, pages 413–452. IEEE Computer Society, 1988.

[6] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *C. ACM*, 21(8):613–642, Aug. 1978.

[7] J.-P. Banâtre, A. Coutant, and D. Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2):133–144, Sept. 1988.

[8] J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, Nov. 1990.

[9] M. Barnett and C. Lengauer. The synthesis of systolic programs. In *Proc. Seminar on Research Directions in High-Level Parallel Programming Languages.* Springer-Verlag, 1991. To appear.

[10] M. Barnett and C. Lengauer. A systolizing compilation scheme. Technical Report ECS-LFCS-91-134, Department of Computer Science, University of Edinburgh, Jan. 1991. Abstract: *Proc. 1991 Int. Conf. on Parallel Processing, Vol. II*, Pennsylvania State University Press, 1991, 305–306.

[11] B. Baxter, G. Cox, T. Gross, H. T. Kung, D. O'Hallaron, C. Peterson, J. Webb, and P. Wiley. Building blocks for a new generation of application-specific computing systems. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 190–201. IEEE Computer Society Press, 1990.

[12] A. Benaini and Y. Robert. Spacetime-minimal systolic architectures for Gaussian elimination and the algebraic path problem. *Parallel Computing*, 15(1):211–226, 1990.

[13] E. Best and C. Lengauer. Semantic independence. *Science of Computer Programming*, 13(1):23–50, Dec. 1989.

[14] J. Bu and E. F. Deprettere. Converting sequential iterative algorithms to recurrent equations for automatic design of systolic arrays. In *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 88), Vol. IV: VLSI; Spectral Estimation*, pages 2025–2028. IEEE Press, 1988.

[15] J. Bu, E. F. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 591–602. IEEE Computer Society Press, 1990.

[16] P. R. Cappello. A spacetime-minimal systolic array for matrix product. In J. V. McCanny, J. McWhirter, and E. E. Swartzlander, editors, *Systolic Array Processors*, pages 347–356. Prentice-Hall, 1989. To appear in *IEEE Trans. on Parallel and Distributed Systems* titled: A processor-time-minimal systolic array for cubical mesh algorithms.

[17] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[18] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *J. Parallel and Distributed Computing*, 3(4):461–491, 1986.

[19] M. C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *J. Supercomputing*, 2:171–207, 1988.

[20] Lyndon J. Clarke and G. V. Wilson. Tiny: An efficient rouring harness for the INMOS transputer. *Concurrency: Practice and Experience*, 3(3), June 1991.

[21] P. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 5–18. IEEE Computer Society Press, 1990.

[22] K. Culik II, J. Gruska, and A. Salomaa. Systolic automata for VLSI on balanced trees. *Acta Informatica*, 18:335–344, 1983.

[23] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata, Part II. *Int. J. Computer Math.*, 16:3–22, 1984.

[24] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. Technical Report 91-10, Laboratoire LIP-IMAG, Ecole, Normale Supérieure de Lyon, Apr. 1991.

[25] J.-M. Delosme and I. C. F. Ipsen. Efficient systolic arrays for the solution of Toeplitz systems: An illustration of a methodology for the construction of systolic architectures for VLSI. In W. Moore, A. McCabe, and R. Urquart, editors, *Systolic Arrays*, pages 37–46. Adam Hilger, 1987.

[26] E. W. Dijkstra. *A Discipline of Programming.* Series in Automatic Computation. Prentice-Hall, 1976.

[27] B. R. Engstrom and P. R. Cappello. The SDEF systolic programming system. In S. K. Tewksbury B. W. Dickinson and S. C. Schwartz, editors, *Concurrent Computations*, chapter 15. Plenum Press, 1987.

[28] E. Fachini and M. Napoli. C-tree systolic automata. *Theoretical Computer Science*, 56:155–186, 1988.

[29] J. A. B. Fortes, K.-S. Fu, and B. W. Wah. Systematic design approaches for algorithmically specified systolic arrays. In V. M. Milutinović, editor, *Computer Architecture – Concepts and Systems*, chapter 11. North-Holland, 1988.

[30] J. A. B. Fortes and D. I. Moldovan. Data broadcasting in linearly scheduled array processors. In *Proc. 11th Ann. Int. Symp. on Computer Architecture*, pages 224–231. IEEE Computer Society Press, 1984.

[31] P. Frison, P. Gachet, and P. Quinton. Designing systolic arrays with DIASTOL. In S.-Y. Kung, R. E. Owen, and J. G. Nash, editors, *VLSI Signal Processing, II*, pages 93–105. IEEE Press, 1987.

[32] P. Gachet, B. Joinnault, and P. Quinton. Synthesizing systolic arrays using DIASTOL. In W. Moore, A. McCabe, and R. Urquart, editors, *Systolic Arrays*, pages 25–36. Adam Hilger, 1987.

[33] P. Gachet, C. Mauras, P. Quinton, and Y. Saouter. A language for the design of regular parallel algorithms. In F. André and J. P. Verjus, editors, *Hypercube and Distributed Computers*, pages 189–202. Elsevier (North-Holland), 1989.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman & Co., 1979.

[35] I. Graham and T. King. *The Transputer Handbook.* Prentice-Hall, 1990.

[36] J. P. Gray and T. A. Kean. Configurable hardware: A new paradigm for computation. In C. L. Seitz, editor, *Advanced Research in VLSI.* MIT Press, 1989.

[37] J. Gruska. Systolic automata: Power, characterization, nonhomogeneity. In M. P. Chytil and V. Koubek, editors, *Proc. Mathematical Foundations of Computer Science (MFCS '84)*, Lecture Notes in Computer Science 176, pages 32–49. Springer-Verlag, 1984.

[38] J. Gruska. Synthesis, structure and power of systolic computations. *Theoretical Computer Science*, 71:47–77, 1990.

[39] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proc. Caltech Conf. on VLSI*, pages 509–525, 1979.

[40] C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6):595–632, Nov. 1987.

[41] C.-H. Huang and C. Lengauer. An incremental, mechanical development of systolic solutions to the algebraic path problem. *Acta Informatica*, 27(2):94–124, Dec. 1989.

[42] INMOS Ltd. occam *Programming Manual.* Series in Computer Science. Prentice-Hall Int., 1984.

[43] INMOS Ltd. occam 2 *Reference Manual.* Series in Computer Science. Prentice-Hall Int., 1988.

[44] INMOS Ltd. *transputer Reference Manual.* Prentice-Hall, 1988.

[45] INMOS Ltd. *The T9000 transputer • Products Overview • Manual.* SGS-Thompson Microelectronics Group, first edition, 1991.

[46] H. V. Jagadish, S. K. Rao, and T. Kailath. Array architecture for iterative algorithms. *Proc. IEEE*, 75(9):1304–1320, Sept. 1975.

[47] B. Joinnault. *Conception d'algorithmes et d'architectures systoliques.* PhD thesis, l'Université de Rennes I, Sept. 1987.

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

[49] T. A. Kean and J. P. Gray. Configurable hardware: Two case studies of micro-grain computation. *J. VLSI Signal Processing*, 2(1):9–16, Sept. 1990.

[50] H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 8.3. Addison-Wesley, 1980.

[51] S.-Y. Kung. *VLSI Processor Arrays.* Prentice-Hall Int., 1988.

[52] S.-Y. Kung, S.-C. Lo, and P. S. Lewis. Optimal systolic design for the transitive closure and shortest path problems. *IEEE Trans. on Computers*, C-36(5):603–614, May 1987.

[53] M. Lam. *A Systolic Array Optimizing Compiler.* Kluwer Academic Publishers, 1988.

[54] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, Feb. 1974.

[55] P. Lee and Z. Kedem. Synthesizing linear-array algorithms from nested for loop algorithms. *IEEE Trans. on Computers*, C-37(12):1578–1598, Dec. 1988.

[56] C. Lengauer, M. Barnett, and D. G. Hudson. Towards systolizing compilation. *Distributed Computing*, 5(1):7–24, 1991.

[57] C. Lengauer and C.-H. Huang. A mechanically certified theorem about optimal concurrency of sorting networks. In *Proc. 13th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–317, 1986.

[58] C. Lengauer and J. W. Sanders. The projection of systolic programs. *Formal Aspects of Computing*, 2:273–293, 1990.

[59] C. Lengauer and J. Xue. A systolic array for pyramidal algorithms. *J. VLSI Signal Processing*, 3(3):239–259, 1991.

[60] H. Leverge, C. Mauras, and P. Quinton. A language-oriented approach to the design of systolic chips. In E. F. Deprettere and A.-J. van der Veen, editors, *International Workshop on Algorithms and Parallel VLSI Architectures, Vol. A: Tutorials*, pages 309–327. Elsevier (North-Holland), 1990. To appear in *J. VLSI Signal Processing.*

[61] B. Lisper. Single-assignment semantics for imperative programs. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE '89), Vol. II: Parallel Languages*, Lecture Notes in Computer Science 366, pages 321–334. Springer-Verlag, 1989.

[62] W. Luk. Analysing parameterised designs by non-standard interpretation. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 133–144. IEEE Computer Society Press, 1990.

[63] W. Luk, G. Jones, and M. Sheeran. Computer-based tools for regular array design. In J. McCanny, J. McWirther, and E. E. Swartzlander, editors, *Systolic Array Processors*, pages 589–598. Prentice Hall Int., 1989.

[64] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 100–110. IEEE Computer Society Press, 1990.

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

[66] D. I. Moldovan. ADVIS: A software package for the design of systolic arrays. *IEEE Trans. on Computer-Aided Design*, CAD-6(1):33–40, Jan. 1987.

[67] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Trans. on Computers*, C-35(1):1–12, Jan. 1986.

[68] C. Mongenet and G.-R. Perrin. Synthesis of systolic arrays for inductive problems. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE '89), Vol. I: Parallel Architectures*, Lecture Notes in Computer Science 365, pages 260–277. Springer-Verlag, 1989.

[69] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1988.

[70] E. T. L. Omtzigt. Systars: A CAD tool for the synthesis and analysis of VLSI systolic/wavefront arrays. In K. Bromley, S.-Y. Kung, and E. E. Swartzlander, editors, *Proc. Int. Conf. on Systolic Arrays*, pages 383–391. IEEE Computer Society, 1988.

[71] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Ann. Int. Symp. on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.

[72] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *J. VLSI Signal Processing*, 1(2):95–113, Oct. 1989.

[73] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, 1989.

[74] S. V. Rajopadhye. Algebraic transformations in systolic array synthesis: A case study. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, pages 361–370. North-Holland, 1990.

[75] S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14(2):163–189, June 1990.

[76] I. Ramakrishnan, D. Fussell, and A. Silberschatz. Mapping homogeneous graphs on linear arrays. *IEEE Trans. on Computers*, C-35(3):189–209, Mar. 1986.

[77] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Supercomputing '89*, pages 637–646. ACM Press, 1989.

[78] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, Mar. 1988.

[79] H. B. Ribas. *Automatic Generation of Systolic Programs from Nested Loops.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, June 1990. Technical Report CMU-CS-90-143.

[80] Y. Robert and D. Trystram. An orthogonal systolic array for the algebraic path problem. *Computing*, 39(3):187–199, 1987.

[81] G. Rote. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34(3):191–219, 1985.

[82] D. Sangiorgi and C. Lengauer. An improved systolic array for string correction. In *Proc. Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 211–225. Politecnico di Torino, 1991.

[83] Y. Saouter and P. Quinton. Computability of recurrence equations. Technical Report 1203, IRISA (INRIA-Rennes), Apr. 1990. To appear in *Theoretical Computer Science 97*, Aug. 1992.

[84] C. J. Scheiman and P. R. Cappello. A processor-time minimal systolic array for transitive closure. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 19–30. IEEE Computer Society Press, 1990.

[85] A. Schrijver. *Theory of Linear and Integer Programming.* Series in Discrete Mathematics. John Wiley & Sons, 1986.

[86] C. E. Seitz. Multicomputers. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, chapter 5, pages 131–200. Addison-Wesley, 1990.

[87] W. Shang and J. A. B. Fortes. On the optimality of linear schedules. *J. VLSI Signal Processing*, 1(2):209–220, Oct. 1989.

[88] W. Shang and W. A. B. Fortes. Time optimal linear schedules for algorithms with uniform dependences. *IEEE Trans. on Computers*, C-40:723–742, June 1991.

[89] J. Teich and L. Thiele. Control generation in the design of processor arrays. *J. VLSI Signal Processing*, 3(1/2):77–92, June 1991.

[90] P.-S. Tseng. *A Systolic Array Parallelizing Compiler.* Kluwer Academic Publishers, 1990.

[91] V. van Dongen and M. Petit. PRESAGE: A tool for the parallelization of nested loop programs. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, pages 341–359. North-Holland, 1990.

[92] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.

[93] Y. Wong and J. M. Delosme. Optimal systolic implementations of n-dimensional recurrences. In *Proc. IEEE Int. Conf. on Computer Design (ICCD 85)*, pages 618–621. IEEE Press, 1985. Also: Technical Report 8810, Department of Computer Science, Yale University, Apr. 1988.

[94] Y. Wong and J.-M. Delosme. Broadcast removal in systolic algorithms. In K. Bromley, S.-Y. Kung, and E. E. Swartzlander, editors, *Proc. Int. Conf. on Systolic Arrays*, pages 403–412. IEEE Computer Society, 1988.

[95] Y. Wong and J. M. Delosme. Optimization of computation time for systolic arrays. Technical Report YALEU/DCS/RR-651, Department of Computer Science, Yale University, May 1989.

[96] Y. Wong and J. M. Delosme. Optimization of processor count for systolic arrays. Technical Report YALEU/DCS/RR-697, Department of Computer Science, Yale University, May 1989.

[97] Xilinx, Inc. *The Programmable Gate Array Data Book.* Xilinx, Inc., 1991.

[98] J. Xue and C. Lengauer. On one-dimensional systolic arrays. In *Proc. ACM Int. Workshop on Formal Methods in VLSI Design.* Springer-Verlag, Jan. 1991. To appear.

[99] J. Xue and C. Lengauer. Specifying control signals for one-dimensional systolic arrays by uniform recurrence equations. In P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II.* Elsevier (North-Holland), 1991.

[100] J. Xue and C. Lengauer. The systematic derivation of control signals for systolic arrays. Technical Report ECS-LFCS-91-152, Department of Computer Science, University of Edinburgh, May 1991.

[101] Y. Yaacoby and P. R. Cappello. Scheduling a system of nonsingular affine recurrence equations onto a processor array. *J. VLSI Signal Processing*, 1(2):115–125, 1989.

[102] X. Zhong and S. Rajopadhye. Deriving fully efficient systolic arrays by quasi-linear allocation functions. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *Parallel Architectures and Languages Europe (PARLE '91), Vol. I: Parallel Architectures and Algorithms*, Lecture Notes in Computer Science 505, pages 219–235. Springer-Verlag, 1991.

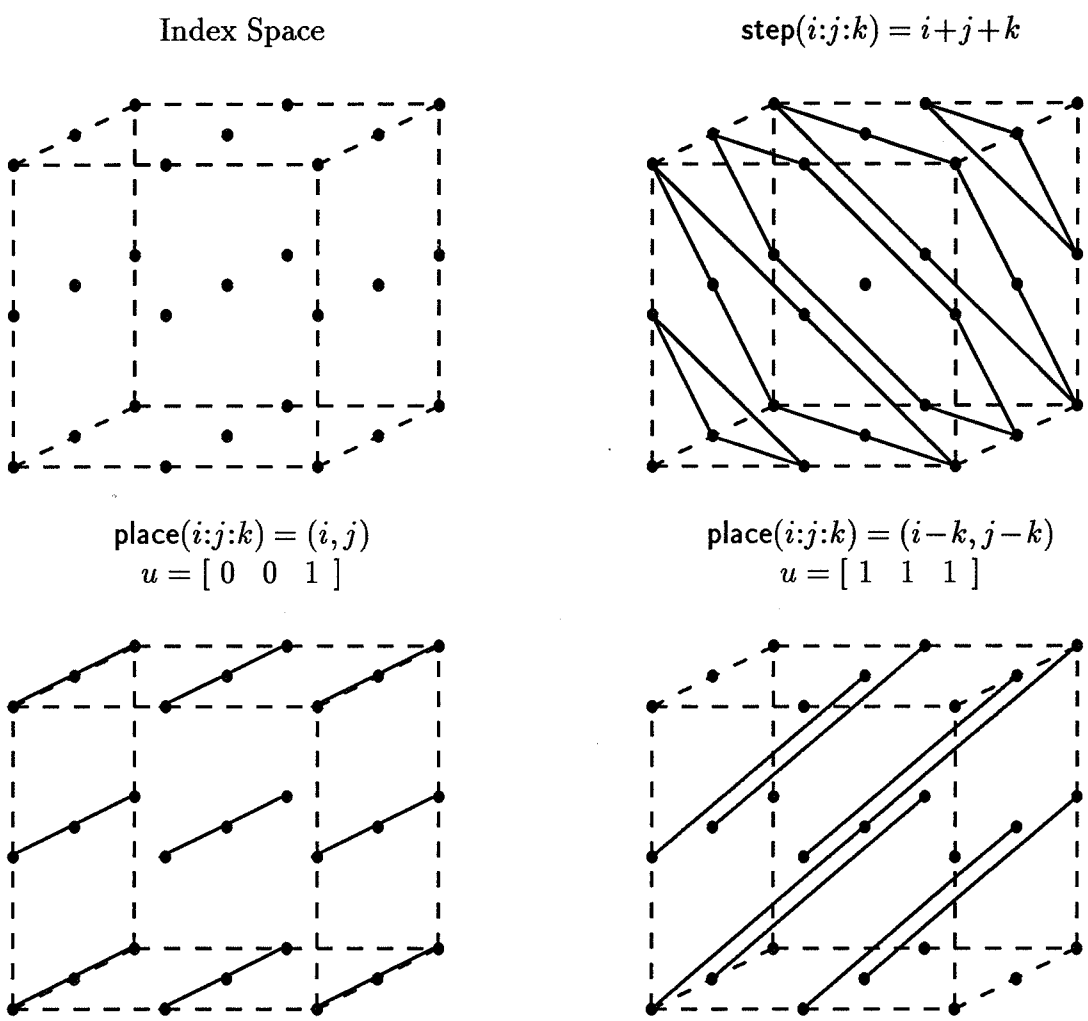


Figure 1: Square matrix product: partitions of the index space.

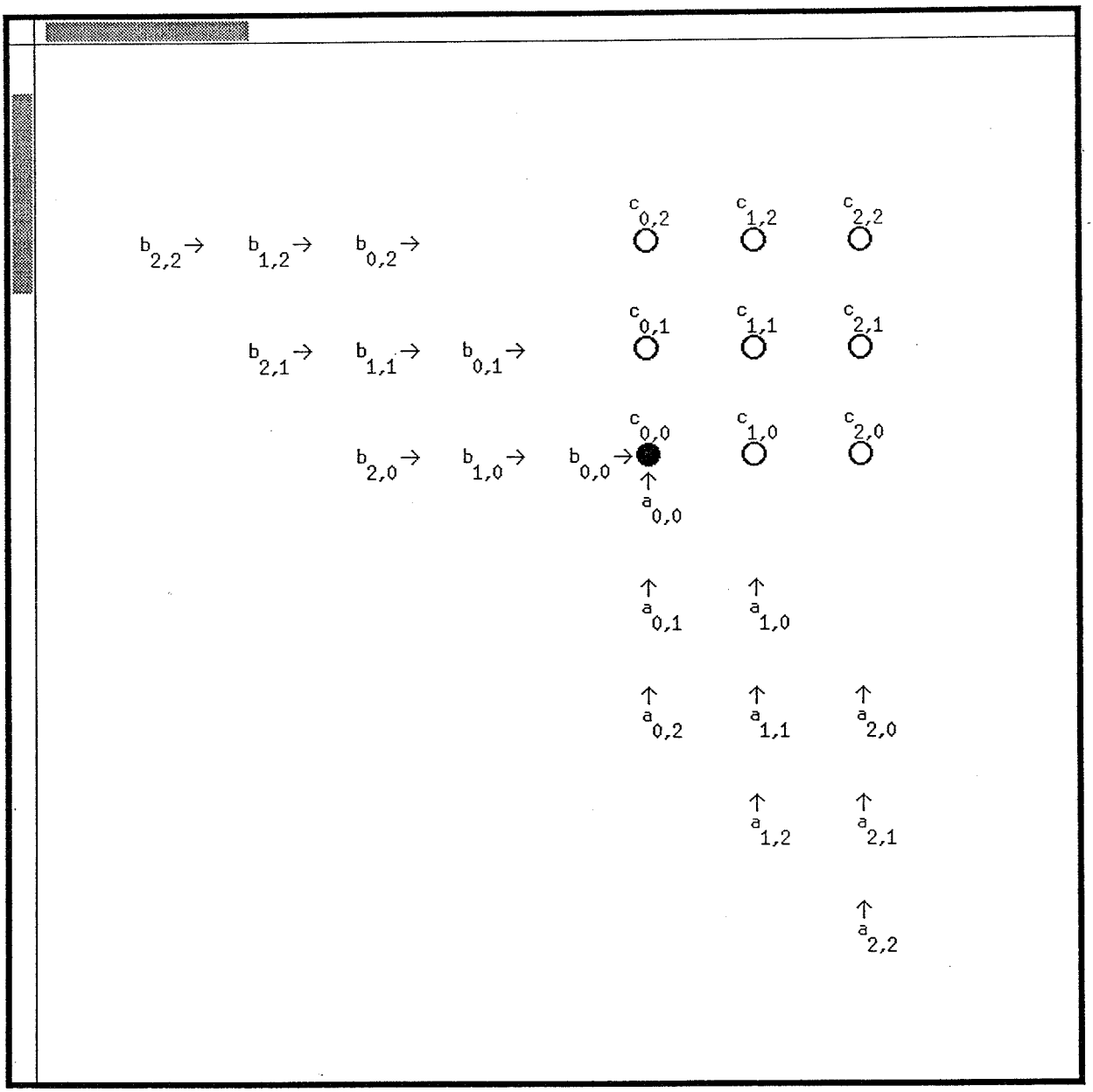


Figure 2: Square matrix product: the systolic array with iteration vector $u = [0\ 0\ 1]$.

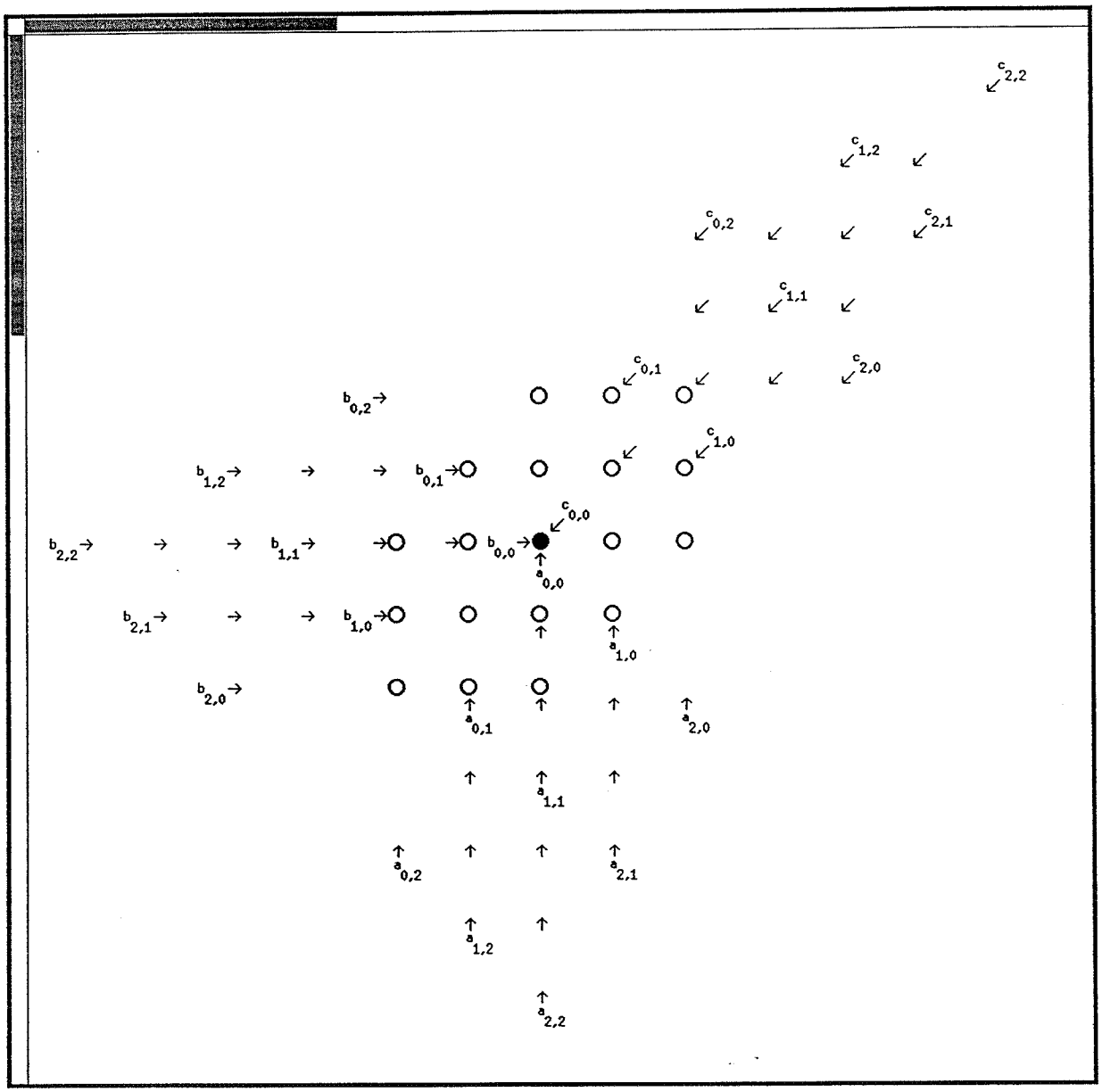


Figure 3: Square matrix product: the systolic array with iteration vector $u = [1\ 1\ 1]$.