The University of Edinburgh

Department of Computer Science

The Formal Synthesis of Control Signals for Systolic Arrays

by

Jingling Xue

CST-90-92

(also published as ECS-LFCS-92-203)

April, 1992

James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.

The Formal Synthesis of Control Signals for Systolic Arrays

Jingling Xue

PhD
University of Edinburgh
1992
(Graduation date July 1992)

Index of Symbols

(The page reference indicates where the symbol is first used)

V	a variable	C	a control variable
E	an evolution control variable	A	an $m \times n$ matrix
P	a convex polytope in $\mathbf{Z}^n$	F	a facet of P
X	a set in $\mathbf{Z}^n$	H	a half-space in $\mathbf{Z}^n$
$x, y, \vartheta, z_1, z_2, \cdots, z_m$	vectors in $\mathbf{Z}^n$	i	an integer > 0)

V^{in}, 115

V^{out}, 115

X', 51

$[x : y]$, 12

$[x \boxdot y]$, 12

Ω_C, 112

C_H, 50

$\Cap$, 9

Γ_V, 100

P_H, 60

Φ, 16

Ψ, 31, 95

Ψ_V, 31, 95

$\Psi_V^{\perp}$, 31, 95

Ψ_V^{d}, 31, 95

Ψ_V^{p}, 31, 95

Ψ_V^{r}, 95

Ψ_V^{s}, 31, 95

H_C, 50

in_V, 17

out_V, 18

$\mathrm{aff}(X)$, 12

$\mathfrak{B}$, 60

$\mathfrak{F}$, 126

$\mathfrak{L}$, 132

$\mathfrak{S}$, 50

$\mathfrak{S}_{\mathrm{src}}$, 57

$\boxdot$, 12

$\mathcal{D}$, 19

$\mathrm{half}(\mathfrak{S}, P)$, 53

$\mathcal{I}_V$, 31

$\mathcal{O}_V$, 32

$\mathcal{P}$, 24, 85

$\mathcal{P}(X)$, 51

$\mathcal{S}$, 24, 85

$\mathcal{V}$, 17

Ⓕ, 118
$\textcircled{I}_V$, 88
$\textcircled{I}'_V$, 96
Ⓛ, 118
$\textcircled{O}_V$, 88
$\textcircled{P}_{\mathrm{src}}$, 49
Ⓣ, 44
condeq(V), 48
condeq(Φ), 48
conv(X), 12
δ_V, 94
deq(Φ), 16
deq(Φ_V), 16
dim(X), 12
facets(P), 13
fst(X, ϑ), 19
flow, 24
in(X, ϑ), 19
inject, 120
input, 88
lst(X, ϑ), 19
lin(X), 12
span($z_1, z_2, \cdots, z_m$), 10
paths(X, ϑ), 98
output, 88
pattern, 25
place, 24
planes(X, A), 94
pi, 88, 147
po, 88, 147
rank(A), 10
ray(x, ϑ), 20
rays(X, ϑ), 20
sign, 10
sig(V), 47
soak(E, i), 124
step, 24
sup(P, F), 13
$\mathfrak{S}^{\mathrm{c}}$, 50
$\overline{x}$, 25
$\varphi(\overline{x})$, 116
$p(X, \vartheta, x)$, 98
t_{fst}, 24, 89
t_{lst}, 24, 89
$x.i$, 88
$x \overset{\vartheta}{\nrightarrow} y$, 20
$x \overset{\vartheta}{\rightarrow} y$, 20

Index of Terms

(The page reference indicates where the term is first defined)

Ⓟ-partition, 48
π-model, 86
τ-model, 89
k-parallelepiped, 13

adapted index space, 145
address variable, 159
allocation
  function, 24
  matrix, 24
  vector, 85
AREs, 16

basic
  conditional, 17, 179
  domain predicate, 179
  expression, 179
border cell, 23, 87, 89
broadcast dependence, 19

CCUREs, 4, 46
cell program, 114
channel, 23
characteristic function, 10
communication constraint domain, 112
computation
  control flow, 4
  equation, 18
  point, 19
  state, 147
  UREs, 4
conditional, 17, 19, 179
control
  channel, 14
  UREs, 4, 41
  variable, 42
convex
  class partition, 48
  domain partition, 48

data
  channel, 14
  dependence graph, 18
  dependence matrix, 19
  dependence vector, 17
  pipelining, 35
  variable, 42
domain
  of a variable, 16
  of an equation, 16
  predicate, 17, 179
draining point, 31

equality, 17, 179
evolution control variable, 114
expression, 179
extended
    data dependence graph, 31
    domain of a variable, 31
    index space, 31, 95
    source UREs, 32, 96

first
    computation point, 19
    step number, 89

guard, 8, 46
guarded command, 8

host program, 114

image under a space-time mapping (overbar), 25
index
    mapping, 17
    space, 16
    vector, 17
inequality, 17, 179
initialisation control variable, 117
inner-product UREs, 148
input
    cell, 23, 87, 89
    data, 17
    equation, 17
intermediate data, 18
internal cell, 23, 87, 89
isomorphic, 10

last
    computation point, 19
    step number, 89
latency, 25
linear part, 17
loading variable, 159

moving, 24
multi-projection, 84
multi-step algorithm, 168

non-domain predicate, 179
non-injectivity of a space-time mapping, 101
normal form of domain predicates, 17, 179

output
    cell, 23, 87, 89
    data, 18
    equation, 18

parameterised systolic array, 37
partitioning, 37
path, 98
PCUREs, 4, 46
pipelined, 42
    UREs, 46
pipelining
    dependence, 19
    equation, 20
    point, 31
    variable, 20
    vector, 35
place function, 24
point, 17

processor space, 24
projection, 38
    vector, 25
propagation
    control flow, 4
    state, 147
    UREs, 4

recovery variable, 159
redundant, 65
relaying point, 95

scheduling
    matrix, 102
    space, 24
    vector, 24
separation set, 50
set of defining half-spaces, 50
soaking point, 31
source UREs, 3
space-time
    diagram, 97
    mapping, 24
    matrix, 25, 85, 102
    point, 97
    UREs, 28
stationary, 24
step function, 24
systolic
    algorithm, 3
    array, 23
    array model, 23
    design, 1

temporal hyperplane, 25
termination control variable, 118
timing function, 24
type of a point, 44

unconditional, 19
undefined point, 31, 98
uniformisation, 35
UREs, 18

validity of a space-time mapping, 26, 90, 103, 134, 139
    for a variable, 117

Abstract

The distinguishing features characteristic of systolic arrays are synchrony of computations, local and regular connections between processors and massive decentralised parallelism. The potential of the systolic array lies in its suitability for VLSI fabrication and its practicality for a variety of application areas such as signal or image processing and numeric analysis. With the increasing possibilities promised by advances in VLSI technology and computer architecture, more and more complex problems are now solvable by systolic arrays.

This thesis describes a systematic method for the synthesis of control signals for systolic arrays that are realised in hardware. Control signals ensure that the right computations are executed at the right processors at the right time. The proposed method applies for iterative algorithms defined over a domain that can be expressed by a convex set of integer coordinates. Algorithms that can be implemented as systolic arrays can be expressed this way; a large subclass can be phrased as affine (or uniform) recurrence equations in the functional style and as nested loops in the imperative style. The synthesis of control signals from a program specification is a process of program transformation and construction. The basic idea is to replace the domain predicates in the initial program specification which constitute the abstract specification of control signals by a system of uniform recurrence equations by means of data pipelining. Then, systolic arrays with a description of both data and control signals can be obtained by a direct application of the standard space-time mapping technique.

Acknowledgements

I would like to thank first of all my supervisor, Christian Lengauer, for his help, encouragement and advice in the development of this thesis. His concern for clarity and mathematical elegance has greatly influenced my work. Also, I would like to thank Patrice Quinton and Björn Lisper for helpful discussions over the electronic network. Finally, I would like to thank my wife Lili for her support and patience in making this thesis possible.

My research work was supported financially by an Overseas Research Studentship, a University of Edinburgh Studentship, and the Science and Engineering Research Council, grant no. GR/G55457.

This is a revised version of my PhD thesis. The comments and suggestions put forward by my examiners, Patrice Quinton and Mike Fourman, have been most helpful.

Declaration

I declare that this thesis has been composed by myself and that the work reported herein is my own, and the ideas and results that I do not attribute to others are due to myself.

All papers that I have published as a registered PhD student are listed below.

- C. Lengauer and J. Xue. A systolic array for pyramidal algorithms. *J. VLSI Signal Processing*, 3(3):239–259, 1991. Early version: Technical Report ECS-LFCS-90-114, Department of Computer Science, University of Edinburgh, June 1990.

- 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. Early version: Technical Report ECS-LFCS-90-116, Department of Computer Science, University of Edinburgh, July 1990.

- 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.

- 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*, chapter 17, pages 181–187. Elsevier (North-Holland), 1992. Early version: Technical Report ECS-LFCS-91-156, Department of Computer Science, University of Edinburgh, June 1990.

- C. Lengauer and J. Xue. Adapting a sequential algorithm for a systolic design. In G. M. Megson, editor, *Transformational Approaches to Systolic Design*. Chapman-Hall, 1991. To appear.

- C. Lengauer and J. Xue. Recent developments in systolic design. Technical Report ECS-LFCS-91-176, Department of Computer Science, University of Edinburgh, Sept. 1991.

- J. Xue. Specifying control signals for systolic arrays by uniform recurrence equations. Parallel Processing Letters, 1(2):83–93, 1992.

26 March 1992

J Xue

Jingling Xue

Table of Contents

1. Introduction **1**

1.1 Outline of the Thesis 5

1.2 Notation and Terminology 7

1.2.1 Logic 7

1.2.2 Programming Languages 8

1.2.3 Sets 9

1.2.4 Functions 9

1.2.5 Linear Algebra 10

1.2.6 Abstract Algebra 10

1.2.7 Convex Analysis 11

1.2.8 Arithmetic 14

1.2.9 Symbols for Depicting Systolic Arrays 14

2. Data Flow Synthesis for (n–1)-Dimensional Systolic Arrays **15**

2.1 Introductory Remarks 15

2.2 Affine and Uniform Recurrence Equations 16

2.3 The Systolic Array Model 23

2.4 The Space-Time Mapping 24

2.5 Input and Output Extension 30

2.6 Uniformisation 35

2.7 Conclusion 37

3. Control Flow Synthesis for (n–1)-Dimensional Systolic Arrays 40

3.1 Introductory Remarks 40

3.2 The Requirements on Control Flow 42

3.3 The Synthesis of Control Flow 43

3.3.1 The Computation Control Flow 50

3.3.2 The Propagation Control Flow 60

3.4 The Optimisation of Control Flow 63

3.5 The Extension of the Space-Time Mapping 65

3.6 Related Work 67

3.7 Examples 68

3.7.1 Dynamic Programming 69

3.7.2 LU-Decomposition 76

3.8 Conclusion 81

4. Data Flow Synthesis for One-Dimensional Systolic Arrays 84

4.1 Introductory Remarks 84

4.2 One-Dimensional Systolic Array Models 85

4.3 The Mapping Conditions 90

4.4 The Extension of the Index Space 93

4.5 The Space-Time Diagram 97

4.6 A Synthesis Procedure 102

4.7 Related Work 106

4.8 Conclusion 108

5. Control Flow Synthesis for One-Dimensional Systolic Arrays 110

5.1 Introductory Remarks 110

5.2 The Synthesis of Propagation Control Flow 114

5.3 The PCUREs for Three-Dimensional UREs 117

5.3.1 The Evolution Control Flow 117

5.3.2 The Initialisation Control Flow 123

5.3.3 The Termination Control Flow 129

5.3.4 The Space-Time Mapping 133

5.4 The PCUREs for n-Dimensional UREs 136

5.5 Related Work 141

5.6 Examples 141

5.6.1 Matrix Product 141

5.6.2 Dynamic Programming 143

5.7 Conclusion 144

6. The Elimination of Propagation Control Flow 147

6.1 Introductory Remarks 147

6.2 Example: 1-D Convolution 149

6.3 Systolic Arrays of $n-1$ Dimensions 153

6.4 Systolic Arrays of One Dimension 156

6.5 Conclusion 157

7. The Loading, Recovery and Access of Stationary Data 158

7.1 Introductory Remarks 158

7.2 The Loading and Recovery of Stationary Data 160

7.3 The Access of Stationary Data 168

7.4 Conclusion 175

8. Conclusion 176

A. The Normalisation of Domain Predicates 178

List of Tables

5–1 The host program for three-dimensional UREs (to be refined). . . . 120

5–2 The cell program for three-dimensional UREs. 121

5–3 The specification of the initialisation control variables. 125

5–4 The host program for three-dimensional UREs (the final version). 131

5–5 The host program for n-dimensional UREs ($n>2$). 139

5–6 The cell program for n-dimensional UREs ($n>2$). 140

7–1 Several schemes for the loading and recovery of stationary variable C for matrix product. 163

9–1 The abstract syntax of the domain predicates. 179

List of Figures

1–1 The synthesis of control signals. 4

1–2 Illustration of some concepts in $\mathbf{R}^2$ in convex analysis. 13

2–1 The data dependence graph of the AREs for matrix product 21

2–2 The data dependence graph of the UREs for matrix product 22

2–3 S. Y. Kung's systolic array for matrix product. 28

2–4 The geometrial interpretation of S. Y. Kung's systolic array for matrix product. 29

2–5 The data dependence graph of the space-time UREs that specify S. Y. Kung's systolic array. 30

2–6 Kung-Leiserson's systolic array for matrix product. 33

2–7 The extended data dependence graph of the UREs for matrix product. 34

3–1 The construction of the quotient set $\Phi/\textcircled{T}$. 45

3–2 Domain predicate replacement. 54

3–3 The optimisation of pipelined UREs. 64

3–4 The data dependence graph for dynamic programming 70

3–5 Four systolic arrays for dynamic programming. 74

3–6 The data dependence graph for LU-decomposition 77

3–7 Two systolic arrays for LU-decomposition. 80

3–8 An iterative design process for the systhesis of systolic arrays. . . 82

4–1 Two one-dimensional systolic array models. 86

4–2 A systolic array in which the neighbouring communication cannot be enforced. 91

4–3 The extended index space for matrix product. 95

4–4 The space-time diagrams for one-dimensional arrays. 99

5–1 The space-time diagram for LU-decomposition. 113

5–2 The notation for specifying the cell program. 115

5–3 The evolution of the evolution control variable 119

5–4 The evolution control variable for LU-decomposition. 122

5–5 The interplay between the evolution control variable and the initialisation control variables. 124

5–6 The construction of the initialisation control variables. 126

5–7 The initialisation control variables for LU-decomposition. 129

5–8 The interplay between the evolution control variable and the termination control variables. 130

5–9 The termination control variables for LU-decomposition. 132

5–10 A hierarchical construction of the PCUREs for n-dimensional UREs. 137

5–11 Adaptation of the index space for the construction of the PCUREs. 145

6–1 The elimination of the propagation control signals for 1-D convolution. 152

7–1 The construction of loading and recovery variables ($n=2$). 160

7–2 The data dependence graph of the UREs with the loading and recovery of stationary variable C for matrix product. 165

7–3 A two-dimensional array for matrix product that handles the loading and recovery of stationary variable C. 166

7–4 The data dependence graph of the UREs with the loading, recovery and access of stationary variable C for matrix product. 172

7–5 A one-dimensional array for matrix product that handles the loading, recovery and access of stationary variable C. 173

7–6 The data dependence graph of the UREs with an address variable for dynamic programming. 175

9–1 The distribution of the points satisfying predicate $2\lfloor i/2 \rfloor - j = 0$. . 184

1

Introduction

The potential of VLSI for MIMD parallelism was first recognised by H. T. Kung and C. E. Leiserson [41], when they introduced the term *systolic array* to describe a processor network that was suitably restrained to meet the requirements of VLSI technology. The characteristic features of a systolic array are synchronous and decentralised parallelism and local and regular interconnections between processors (or cells). 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 and image processing, pattern recognition and combinatorial theory. This suggests that systolic computation has an important, enduring rôle to play in concurrent computation.

The special appeal of systolic arrays is that they can be derived systematically, and often 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 programs or imperative programs – into functions that distribute the computations prescribed by the program over space and time. This process is called *systolic design.* The challenge, as stated by H. T. Kung [39], is to ensure that *the right data arrive at the right cells at the right time.* The distribution functions essentially describe the velocities and distribution of data, i.e., the flow of data over space and time. They 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.

There have been a plethora of synthesis methods proposed in the literature.

These methods focus mainly on the synthesis of data flow. They are based on geometry, linear algebra and convex analysis. The basic idea to describe a systolic array by two distribution functions: a timing function that specifies the temporal distribution of the computations and an allocation function that specifies their spatial distribution such that concurrent computations are allocated to different processors. The combination of the timing function and allocation function is called a *space-time mapping* or *space-time transformation.* As has been proved by S. K. Rao [70], a systolic array is a program resulting from an application of the space-time mapping to the initial program specification. Of course, not every space-time mapping describes a systolic array. A valid space-time mapping must preserve the behaviour of the source program in some sense and must respect the constraints imposed on resources such as channel connections in systolic arrays. As pointed out by P. Quinton [62], the space-time mapping also needs to meet conditions that allow a full mechanisation of the underlying synthesis method.

The timing and allocation function constitute a complete description for fabrication of a VLSI chip if every processor in the systolic array performs the same computation at all time steps. The description is incomplete, however, if some processor performs different computations at different time steps. In this case, a control mechanism is called for that instructs the processors in the systolic array when to perform which computation. L. J. Guibas, H. T. Kung and C. D. Thompson [30] suggested a decade ago that this control mechanism may be implemented by communicating control signals throughout the systolic array in much the same way as the data are communicated. Like in the synthesis of data flow, the challenge in the synthesis of control flow is to ensure that *the right control signals arrive at the right cells at the right time.* So far, the systematic synthesis of control flow has not received adequate attention. In most systolic solutions that have been proposed in the literature, the derivation of the control flow is conducted after that of the data flow and in an informal and problem-specific manner.

With the advent of powerful VLSI design and fabrication techniques and of programmable processor networks, the development of formal methods for the synthesis of control signals has become increasingly important. First, complex applications, like the algebraic path problem [9,19,73], that were not considered for a systolic design in the past are now solvable by systolic arrays. A manual

derivation of control signals for these applications is intractable and error-prone. Second, increasing control complexity makes it necessary to evaluate and further optimise the quality of the systolic array with respect to not only the data flow but also the control flow. This is made possible by methods that allow the systematic synthesis of control signals. Third, the synthesis of systolic arrays from algorithmic descriptions is a process of program transformation. At any stage, many transformations are applicable. The choice of different transformations leads to systolic arrays with differing quality. One should be able to assess a transformation by its impact on the quality of the systolic array in terms of not only the data flow but also the control flow. Methods that allow the systematic synthesis of control signals accomplish this by making the specification of control signals explicit in the early stages of the systolic design.

This treatise is concerned with the synthesis of control signals for systolic arrays that are realised in hardware. The proposed method applies for iterative algorithms defined over a domain that can be expressed by a convex set of integer coordinates (or points) called the *index space*. Algorithms that are amenable to systolic design can be expressed in this way. They are often informally referred to as *systolic algorithms*; many can be expressed in the form of affine recurrence equations (AREs) in the functional style [15,20,52,63,66,70] and nested loops in the imperative style [26,31,46,57]. A recurrence equation defines recursively the computation of a target variable indexed by one point in terms of variables indexed by other points. The set of all index vectors associated with the target variable of a recurrence equation is specified by a predicate called a *domain predicate*. A domain predicate specifies the domain of an equation at which the target variable of the equation is defined. Informally, control signals must ensure that the computation of a recurrence equation takes place within the domain specified by the domain predicate of the equation.

A subclass of AREs, called *uniform recurrence equations* (UREs), which was proposed by R. M. Karp and R. E. Miller and S. Winograd for representing numerical algorithms for a parallel implementation [36], has been used extensively for the specification of systolic algorithms in systolic design. We shall present our method based on input that is in the form of UREs. We refer to these initial UREs as the

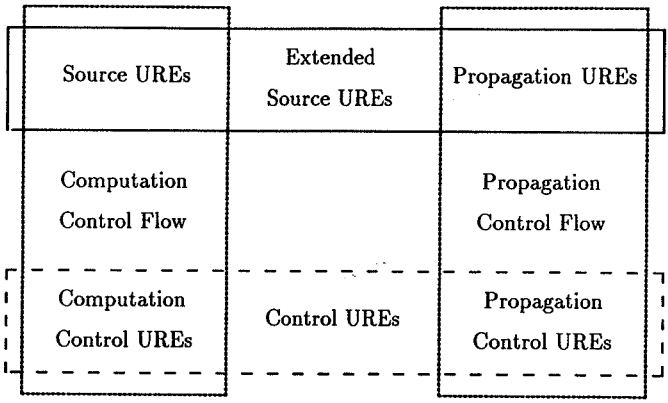


Figure 1–1: The synthesis of control signals.

source UREs. A generalisation of the method to certain other types of recurrence equations is straightforward and will be alluded to where appropriate.

To conform to current VLSI fabrication technology, the exchange of input and output data with the external environment is restricted to the border cells of a systolic array. The source UREs may have to be extended to include the specification of the propagation of input data from border cells to internal cells and of output data from internal cells to border cells. The UREs added to the source UREs in the extension are called *propagation UREs.* The extended specification, which consists of the source UREs and propagation UREs, is called the *extended source UREs.* The domain for which the extended source UREs are defined is called the *extended index space.* (The source UREs might be called *computation UREs* since they prescribe the computations to be performed by systolic arrays.)

The synthesis of control signals is a process of program transformation and construction. The domain predicates of the extended source UREs constitute the initial specification of control signals. We obtain systolic arrays with a description of both data flow and control flow in two successive steps:

1. Transform the domain predicates of the extended source UREs to a system of UREs called the *control UREs* (Fig. 1–1). The control UREs can be divided into two systems of UREs: the *computation control UREs* (CCUREs),

which specify the control signals for the appropriate computations prescribed by the source UREs, and the *propagation control UREs* (PCUREs), which specify the control signals for the propagation of input and output data prescribed by the propagation UREs. Then replace the domain predicates in the source UREs by predicates in computation control variables and those in the propagation UREs by predicates in propagation control variables.

2. Obtain systolic arrays with a description of both data and control flow from the previously transformed UREs by applying different space-time mappings.

In the first step, the challenge lies in the specification of control signals in terms of the control UREs. Once this is accomplished, the replacement of domain predicates that arises due to the introduction of the control UREs is straightforward. In the second step, the issues that need to be addressed are (1) the definition of systolic array models, (2) the corresponding mapping conditions that ensure the validity of the space-time mapping and (3) the procedures by which valid space-time mappings, i.e., systolic arrays, are systematically generated. These three issues will not be emphasised in the thesis; they have been studied extensively in the realm of systolic design. Because the specification of control signals in the first step depends on issues addressed in the second step, we shall conduct our presentation in the reverse of the order in which systolic arrays are derived. That is, we first describe the systolic array model under consideration, then the mapping conditions for the validity of the space-time mapping, then some procedures for the generation of systolic arrays, and finally the specification of control signals.

Next, we give an outline of this thesis. In addition, each chapter contains a section of introductory remarks to prepare the reader for its technical context.

1.1 Outline of the Thesis

In Chap. 2, we review the basic technique of the synthesis of data flow for $(n-1)$-dimensional systolic arrays from n-dimensional UREs. We start with the definition of n-dimensional AREs and the systolic array model that has been used extensively in systolic design. Next, we describe the standard space-time mapping technique, which delivers systolic arrays of $n-1$ dimensions with maximal parallelism, i.e,

a shortest execution (or latency) derivable from the n-dimensional source UREs. This is followed by a brief description of uniformisation techniques that transform AREs to UREs. This chapter concludes with a brief review of several current research issues in systolic design.

In Chap. 3, we present our method for the specification of control signals for $(n-1)$-dimensional systolic arrays from n-dimensional UREs. The specification of control signals relies on a partition of the extended index space such that all the computations associated with one block in the partition are specified by a common set of control signals. We provide necessary and sufficient conditions for the correctness of control flow and a mechanisable procedure that constructs the specification of control signals from the source UREs. We also describe some optimisation techniques for the purpose of eliminating redundant control hardware in the systolic array.

In Chap. 4, we review the synthesis of data flow for one-dimensional systolic arrays and extend and improve previous results. We first define the two most frequently adopted one-dimensional systolic array models. We then investigate the corresponding mapping conditions for the validity of the space-time mapping. In addition, we provide a range of equivalent mapping conditions, which increase our understanding of various properties of one-dimensional systolic arrays and provide insight into the construction of the control UREs for one-dimensional systolic arrays in Chap. 5. Finally, we describe a procedure for the systematic generation of one-dimensional systolic arrays.

In Chap. 5, we focus on the specification of control signals for one-dimensional systolic arrays. We begin by demonstrating that the control UREs that are constructed in Chap. 3 for $(n-1)$-dimensional systolic arrays may result in very inefficient one-dimensional systolic arrays. The problem lies in the specification of the PCUREs and is caused by the non-injectivity of the space-time mapping (over the extended index space) for one-dimensional systolic arrays. We solve this problem by providing an alternative construction of the PCUREs. To complete our method for the specification of control signals for systolic arrays, we show that the PCUREs constructed in this chapter and the CCUREs constructed in the previous chapter apply for systolic arrays of any r dimensions with $0<r<n$.

In Chap. 6, we are concerned with the elimination of control signals for systolic

arrays. We consider a special class of UREs that do not require the CCUREs. We show how algebraic properties of some operators in the source UREs, like the satisfaction of the Unit Law and the Zero Law (Sect. 1.2), can be exploited to eliminate the PCUREs. We provide necessary and sufficient conditions for the elimination of the PCUREs. Although it is difficult to give a general treatment for arbitrary UREs, the underlying idea of the optimisation of control signals can be generalised straightforwardly to other systolic algorithms.

In Chap. 7, we describe a scheme for loading, recovering and accessing stationary variables. It is sometimes beneficial to make certain variables stationary in order to improve some aspects of the systolic array, e.g., to reduce the latency or size of the array. Unfortunately, the space-time mapping does not provide any help in handling stationary variables. We present a systematic method that modifies the source UREs such that the specification for loading, recovering and accessing stationary variables is included. Then, specifying the control signals for systolic arrays synthesised from this new system of UREs is just a matter of constructing the control UREs based on the methods described in Chaps. 3 and 5.

Finally, in Chap. 8, we give an overview of the results presented in the thesis and comment on some open problems and suggestions for future research.

1.2 Notation and Terminology

1.2.1 Logic

The logic connectives are $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\Longrightarrow$ (implies), $\Longleftrightarrow$ (if and only if), $\forall$ (for all) and $\exists$ (there exists). We use true and false to denote the propositional constants. The notation iff is also used for $\Longleftrightarrow$.

The notation for a quantified expression is $(quant : range : term)$, where *quant* specifies a quantifier and a list of dummy variables for the quantification, *range* specifies the range of the dummies and *term* is some function or predicate on the dummies [21]. Any binary, commutative, associative operator that has an identity element may be used as a quantifier (for the definition of identity element of an operator, see Sect. 1.2.6). Examples of quantifiers (and the corresponding

interpretation in the case of an empty range) include: $\forall$ (true), $\exists$ (false), max $(-\infty)$ and min $(+\infty)$; the values true, false, $-\infty$ and $+\infty$ are the identity elements of the operators $\forall$, $\exists$, max and min, respectively.

Some proofs follow the notation of [21], in which a proof step has the following layout:

$$
\begin{array}{ll}
 & \ldots \\
 & \neg(P \wedge \neg P) \wedge (Q \vee \neg Q) \\
\Longleftrightarrow & \{\text{De Morgan's law; excluded-middle law}\} \\
 & (\neg P \vee \neg\neg P) \wedge \mathsf{true} \\
 & \ldots
\end{array}
$$

in place of $\Longleftrightarrow$ to the left of the brace may be $\Longrightarrow$, $=$, $\subseteq$, etc. The brace may contain a number of hints separated by conjunctive semicolons.

The symbol $\square$ marks the end of theorems, definitions, examples, and so forth.

1.2.2 Programming Languages

A conditional command, i.e., **if**-statement is written as follows [21]:

$$
\begin{array}{llll}
\textbf{if} & B_1 & \rightarrow & S_1 \\
[] & B_2 & \rightarrow & S_2 \\
 & & \vdots & \\
[] & B_n & \rightarrow & S_n \\
\textbf{fi} & & &
\end{array}
$$

or, if it is short and simple enough, on one line as:

$$\textbf{if } B_1 \rightarrow S_1 \ [] \ B_2 \rightarrow S_2 \ \cdots \ [] \ B_n \rightarrow S_n \ \textbf{fi}$$

B_i is a Boolean expression and S_i a command. $B_i \rightarrow S_i$ is a *guarded command*; B_i acts as a *guard*; it has to be validated before S_i can be executed. One statement S_i whose guard B_i is true is executed; if no guard is true, the **if**-statement aborts. If **else** appears in the place of B_n, it stands for $(\forall\, i : 0 < i < n : \neg B_i)$. **skip** denotes the empty statement. All **if**-statements that appear in the thesis are deterministic.

A **for**-loop has the usual meaning and is written as follows:

for i **from** lb **to** rb **do**

S_1

S_2

$\vdots$

S_n

We indicate scoping by indenting: $S_1, S_2, \cdots, S_n$ are the statements executed at each loop step. All **for**-loops that appear in the thesis have unit stride.

1.2.3 Sets

The notation $\{x \mid \mathcal{P}(x)\}$ denotes the set of elements x that satisfy the condition $\mathcal{P}(x)$. The set whose members are all the objects appearing in the list $x_1, x_2, \cdots, x_m$ and no others is denoted by $\{x_1, x_2, \cdots, x_m\}$. The empty set is denoted by $\varnothing$. If two sets have no element in common, they are called *disjoint*. The number of elements in a set S is denoted by $|S|$. Let A and B denote sets. We say that A is a *subset* of B (or that A *is contained in* B) iff every member of A is also a member of B. Our notation for "A is a subset of B" is $A \subseteq B$. When $A \subseteq B$ and $A \neq B$, we say that A is a *proper subset* of B and we write $A \subset B$. If A and B are sets, the *relative complement* of B in A, denoted by $A \backslash B$, is the set $\{x \mid x \in A \wedge x \notin B\}$. Let A be a subset of a given set S. The *complement* of A in S is the subset $\{x \mid x \in S \backslash A\}$. It is denoted by $\complement_S A$. For the set-theoretic notions of *union* and *intersection*, we use the symbols $\cup$ and $\cap$. For sets $S_1, S_2, \cdots, S_m$, where the elements of every set S_i are themselves sets, we define $(\Cap\, i : 0 < i \leqslant m : S_i) = \{(\bigcap\, i : 0 < i \leqslant m : X_i) \mid (\forall\, i : 0 < i \leqslant m : X_i \in S_i)\}$.

$\mathbf{Z}$, $\mathbf{Q}$ and $\mathbf{R}$ denote the set of integers, rationals and reals, respectively. Let S be $\mathbf{Z}$, $\mathbf{Q}$ or $\mathbf{R}$. S^+ denotes the positive subset, S_0^+ the non-negative subset (i.e., $S^+ \cup \{0\}$) and S^n the n-fold Cartesian product of S.

1.2.4 Functions

$f : D \rightarrow R$ indicates that f is a function with *domain* D and *range* (or *codomain*) R. The set $\{f(x) \mid x \in S\}$ for a set $S \subseteq D$ is called the *image* of the set S under

f and is denoted by $f(S)$. The *composite* $f \circ g$ of two functions is the function obtained by applying them in succession – first g, then f – provided the domain of f is the image of g. The notation $f : D \rightarrowtail R$ denotes that function f is injective from domain D to range R. Two sets A and B are called *isomorphic* if there exists a bijection $f : A \to B$. Let $f : S \to \{a, b\}$ be a total function. If $T \subseteq S$ and $f(T) \cap f(S \backslash T) = \varnothing$, then f is called the *characteristic function* of T in S and is denoted by $\chi_T : S \to \{a, b\}$. The function sign is a mapping from $\mathbf{Q}$ to the set $\{-1, 0, 1\}$: $\mathsf{sign}(x) = \textbf{if } x < 0 \to -1 \ [] \ x = 0 \to 0 \ [] \ x > 0 \to 1 \ \textbf{fi}$.

1.2.5 Linear Algebra

Let V be a vector space over a field F. A set of vectors $x_1, x_2, \cdots, x_m$ in V is called *linearly dependent* if there are coefficients $\lambda_1, \lambda_2, \cdots, \lambda_m$ in F, but not all zero, such that $(\sum\ i : 0 < i \leqslant m : \lambda_i x_i) = 0$. Otherwise, it is called *linearly independent.* A set B of vectors in V is a *basis* for a subspace L of V if B is a maximal linearly independent subset of L. Every linear subspace L has a basis, and all bases of L have the same number of elements. This common number of elements is called the *dimension* of L. The vector space *spanned* by the vectors $x_1, x_2, \cdots, x_m$ in V is the set $\mathrm{span}(x_1, x_2, \cdots, x_m) = \{(\sum\ i : 0 < i \leqslant m : \lambda_i x_i) \mid (\forall\ i : 0 < i \leqslant m : \lambda_i \in F)\}$. For an $m \times n$ matrix A over a field F, the set of solutions to $Ax = 0$ is a vector space called the *null space* of A. The *rank* of the matrix A, denoted by $\mathrm{rank}(A)$, is the number of maximal linearly independent column (or row) vectors of A. Often we do not distinguish whether a vector is a row or column vector and assume that this is deducible from the context. When we write xy for $x, y \in V$, for example, we mean that x is a row vector and y is a column vector with the same number of components.

1.2.6 Abstract Algebra

For any two sets A and B, a subset $R \subseteq A \times B$ is called a *binary relation* between A and B. $(a, b) \in R$ is often written as aRb. Let X be a set. A *partition* P of a set X consists of a set of disjoint subsets (called *P-blocks*) of X whose union is the set X. A relation R between X and X is called an *equivalence relation* on X when it has the following three properties: (1) *Reflexivity:* xRx for all $x \in X$.

(2) *Symmetry:* xRy implies yRx for all $x, y \in X$. (3) *Transitivity:* xRy and yRz imply xRz for all $x, y, z \in X$. Given an equivalence relation E on a set X, the *equivalence class* under E of any element $x \in X$ is the set $C_E(x)$ of all the elements y of X that bear the relation E to x: $C_E(x) = \{y \mid y \in X \wedge yEx\}$. Any subset C of X that has the form $C = C_E(x)$, for some x, is called an *equivalence class* of E (or an *E-class*). The set of all possible E-classes is $X/E = \{C \mid C \subseteq E \wedge C = C_E(x) \wedge x \in X\}$ and is called the *quotient set* of X by E. When restricting E to a subset $S \subseteq X$, the quotient set of S by E becomes: $S/E = \{S\} \Cap X/E$, i.e., $S/E = \{S \cap C \mid C \in X/E\}$. Let P and Q be partitions of a set X. P is *finer* than Q if $(\forall\, x, y : x, y \in X : (\exists\, S : S \in P : x, y \in S) \Longrightarrow (\exists\, T : T \in Q : x, y \in T))$. Let E and F be equivalence relations on a set X. E is *finer* than F if X/E is finer than X/F, i.e., if $(\forall\, x, y : x, y \in X : xEy \Longrightarrow xFy)$.

Let S be a non-empty set. A *binary operation* on S is any function $* : S \times S \rightarrow S$. An element $u \in S$ is called a *unit element* for operation $*$ if

$$(\forall\, s : s \in S : u * s = s = s * u) \qquad \text{(Unit Law)}$$

A unit element is often called an *identity element* (or *neutral element*). We call u a *left* (*right*) *unit* for $*$ if $(\forall\, s : s \in S : u * s = s)$ $((\forall\, s : s \in S : s * u = s))$. An element $z \in S$ is called a *zero element* for operation $*$ if

$$(\forall\, s : s \in S : z * s = z = s * z) \qquad \text{(Zero Law)}$$

A zero element is also called an *annihilator*. We call z a *left* (*right*) *zero* if $(\forall\, s : s \in S : z * s = z)$ $((\forall\, s : s \in S : s * z = z))$.

1.2.7 Convex Analysis

If $S \subset \mathbf{R}^n$ and $x \in \mathbf{R}$, the set $x + S = \{x + y \mid y \in S\}$ is called a *translate* of S. For any two sets A and B, $A + B = \{a + b \mid a \in A \wedge b \in B\}$. A translate of a subspace of $\mathbf{R}^n$ is called an *affine set*. Two affine sets are *parallel* if one is a translate of the other. (This definition of parallelism is more restrictive than the usual one, in that it does not include the idea of a line being parallel to a plane. One has to speak of a line which is parallel to another line within a given plane.) All non-empty parallel affine sets have a unique subspace. The *dimension* of an affine set is the dimension of the corresponding parallel subspace. The *dimension* of a set S is

the dimension of the smallest affine set containing it, and is denoted by $\dim(S)$. By convention, $\dim(\varnothing) = -1$. A set $S \subset \mathbf{R}^n$ is *full-dimensional* if $\dim(S) = n$. An affine set of dimension 1 is a *line*. An affine set of dimension $n-1$ is a *hyperplane*. Let $\{x \mid \pi x = \delta\}$ be a hyperplane, where $\pi \in \mathbf{R}^n \setminus \{0\}$, $x \in \mathbf{R}^n$, and $\delta \in \mathbf{R}$. The sets $\{x \mid \pi x \leqslant \delta\}$ and $\{x \mid \pi x \geqslant \delta\}$ are called *closed half-spaces*. The sets $\{x \mid \pi x < \delta\}$ and $\{x \mid \pi x < \delta\}$ are called *open half-spaces*. We may speak unambiguously of the open and closed half-spaces associated with a given hyperplane. We denote the hyperplane $\{x \mid \pi x = \delta\}$ by $[\pi : \delta]$. We use $\boxdot$ to denote a fixed but arbitrary relational operator in $\{<, \leqslant, >, \geqslant\}$. The notation $[\pi \boxdot \delta]$ stands for the half-space $\{x \mid \pi x \boxdot \delta\}$.

The *affine hull* of a set S is the intersection of all affine sets that contain S. It is denoted by $\mathrm{aff}(S)$. The unique linear subspace parallel to the affine hull of a set S is denoted by $\mathrm{lin}(S)$. A subset S of $\mathbf{R}^n$ is a *convex set* if $(1-\lambda)x + \lambda y \in S$ for every $x \in S$, $y \in S$ and $0 < \lambda < 1$ in $\mathbf{R}$ (it is an affine set if the range of λ is extended to $\mathbf{R}$). The *convex hull* of a set S is the intersection of all convex sets that contain S. It is denoted by $\mathrm{conv}(S)$.

The hyperplane $[\pi : \delta]$ is said to *bound* the set S if either $(\forall\, x : x \in S : \pi x \leqslant \delta)$ or $(\forall\, x : x \in S : \pi x \geqslant \delta)$. A hyperplane H is said to support a set S at a point $x \in S$ if $x \in H$ and H bounds S. A *supporting half-space* to S is a half-space containing S and bounded by a supporting hyperplane to S.

Let A and B be two non-empty sets in $\mathbf{R}^n$. A hyperplane H is said to *separate* A and B if (1) A is contained in one closed half-space and B is contained in the opposite closed half-space and (2) H is not a supporting hyperplane for both A and B. The standard definition of separation only requires that A and B be contained in the opposite closed half-spaces (i.e., (2) is not required). For the purpose of this thesis, our definition of separation deliberately excludes the case when H supports both A and B. A hyperplane H is said to separate A and B *strictly* if A and B are contained in the two opposite open half-spaces.

Fig. 1–2 gives a pictorial illustration of some concepts defined previously.

A *polyhedral convex set* (or, *convex polyhedron*) is the intersection of finitely many closed half-spaces. A *polytope* (or, *convex polytope*) is the convex hull of finitely many points, i.e., a bounded convex polyhedron. Let S be a convex polyhedron in $\mathbf{R}^n$. A subset F of S is called a *face* of S if either $F = \varnothing$ or $F = S$ or

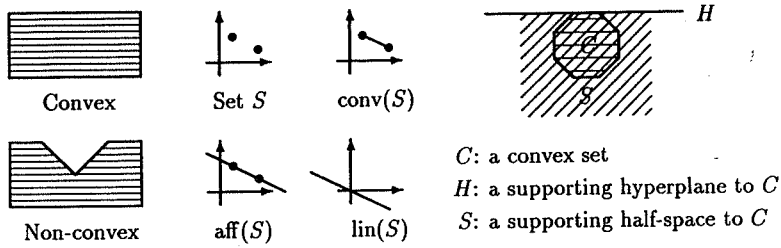


Figure 1–2: Illustration of some concepts in $\mathbf{R}^2$ in convex analysis.

if there exists a supporting hyperplane H to S such that $F = S \cap H$. The faces $\varnothing$ and S are called *improper*. All other faces are called *proper*. If the dimension of F is k, then F is called a k-face of S. It is customary to refer to the 0-faces of S as *vertices*, the 1-faces as *edges* and the $(n-1)$-faces as *facets*. The notation facets(S) stands for the set of all facets of S. For a facet F of S, sup(S, F) denotes the supporting half-space to S that contains F.

A subset S of $\mathbf{R}^n$ is called a *cone* if it is closed under positive multiplication, i.e., $\lambda x \in S$ for all $x \in S$ and $\lambda \in \mathbf{R}^+$. A *convex cone* is a cone that is a convex set. The convex cone *finitely generated* by the vectors $x_1, x_2, \cdots, x_m$ is the set $\text{cone}(x_1, x_2, \cdots, x_m) = \{(\sum i : 0 < i \leqslant m : \lambda_i x_i) \mid (\forall\, i : 0 < i \leqslant m : \lambda_i \in \mathbf{R}_0^+)\}$, i.e, it is the smallest convex cone containing $x_1, x_2, \cdots, x_m$. A cone that is a proper subset of a line is called a *ray* (or *half-line*). A 1-face of a convex polyhedron that is a ray is called an *extreme* ray.

Let $x_1, x_2, \cdots, x_m$ be linearly independent vectors in $\mathbf{R}^n$ and $o \in \mathbf{R}^n$ be the reference point or (the origin). Let L_i be the line segment from the origin o to the point $\alpha_i x_i$ for some non-zero α_i in $\mathbf{R}$. Then $(\sum i : 0 < i \leqslant m : L_i)$ is called a *k-parallelepiped*. It is called a *hypercube* if the vectors $x_1, x_2, \cdots, x_m$ are mutually orthogonal, i.e., if $(\forall\, i, j : 0 < i \leqslant m \wedge 0 < j \leqslant m : i \neq j \Longrightarrow x_i x_j = 0)$.

Let A be an $m \times n$ real matrix and $a \in \mathbf{R}^m$. The function from $\mathbf{R}^n$ to $\mathbf{R}^m$ that maps every element $x \in \mathbf{R}^n$ to the element $Ax \in \mathbf{R}^m$ is called a *linear transformation*. The function from $\mathbf{R}^n$ to $\mathbf{R}^m$ that maps every element $x \in \mathbf{R}^n$ to the element $Ax + a \in \mathbf{R}^m$ is called an *affine transformation*.

All concepts introduced so far in $\mathbf{R}^n$ have parallels in $\mathbf{Q}^n$ ($\mathbf{Z}^n$). Let $S = \{x \mid \mathcal{P}(x)\}$ be any of the sets defined previously in $\mathbf{R}^n$, the corresponding set in $\mathbf{Q}^n$ ($\mathbf{Z}^n$)

is the set of all the rational (integral) points in S, and is given by $\{x \mid \mathcal{P}(x) \wedge x \in \mathbf{Q}^n\}$ ($\{x \mid \mathcal{P}(x) \wedge x \in \mathbf{Z}^n\}$). For example, a convex polyhedron in $\mathbf{Z}^n$ (or an integral convex polyhedron) is the set of all integral points in the intersection of finitely many closed half-spaces.

See [72,75] for further details of convex analysis and the theory of polyhedra.

1.2.8 Arithmetic

As is customary, we write $\lfloor x \rfloor$ for the *floor* of $x \in R$, namely for the greatest integer not greater than x, and we write $\lceil x \rceil$ for the *ceiling* of x, i.e., for $-\lfloor -x \rfloor$, the smallest integer not smaller than x. We write $|x|$ for the absolute value of $x \in R$, i.e., $|x| = \textbf{if } x \geqslant 0 \rightarrow x \; [] \textbf{ else} \rightarrow -x \textbf{ fi}$. The operator **div** denotes integer division. The operator **mod** denotes the modulo operation. For $x, y \in \mathbf{Z}$, $x \mid y$ iff x divides y, i.e., iff $y \textbf{ mod } x = 0$.

1.2.9 Symbols for Depicting Systolic Arrays

Unless otherwise stated, we use the following symbols in the depiction of systolic arrays. Boxes represent processors, lines represent the communication channels between processors. Solid lines are *data channels*; they carry the data flow. Dashed lines are *control channels*; they carry the control flow. The delay buffers associated with data and control channels are represented by fat dots and circles, respectively. The direction of data or control signals that move along a channel is represented by an arrow tip at one end of the channel.

2

Data Flow Synthesis for (n–1)-Dimensional Systolic Arrays

2.1 Introductory Remarks

The synthesis of systolic arrays from a program specification proceeds in two successive steps:

1. The specification is refined and transformed to a systolisable source, and

2. The systolisable source is mapped to a systolic array.

Based on the work of [36], researchers have shown that a program in the form of UREs [62,70] (or in the form of nested loops [57]) serves well as the systolisable source. This has spurred research in systolic design in two major directions. One is to develop methods for the mapping of UREs to systolic arrays. The main design tool is a space-time mapping that delivers systolic arrays by means of index transformations of the UREs. The other focuses on the transformation of program specifications of a more general form to UREs. The main result achieved so far is a uniformisation technique for transforming AREs to UREs.

This chapter provides the technical background for the thesis. We review the basic techniques used in the previous two steps, i.e., the basic techniques

underlying the synthesis of data flow for systolic arrays. We only describe in detail the techniques that are necessary for the understanding of the results of this thesis. Other related issues are discussed in the conclusion.

The rest of this chapter is organised as follows. Sect. 2.2 presents the definition of affine and uniform recurrence equations and some related concepts. Sect. 2.3 defines the most commonly used systolic array model. Sect. 2.4 describes the standard space-time mapping technique for the synthesis of the data flow with respect to the systolic array model. In addition, it defines the validity of the space-time mapping and the mapping conditions that ensure this validity. Sect. 2.5 describes a technique for ensuring that input and output data are always handled at the border cells of the systolic array, as required by the systolic array model defined in Sect. 2.3. Sect. 2.6 presents three uniformisation methods for the transformation of AREs to UREs. Sect. 2.7 concludes the chapter by outlining some other related issues and commenting on some current research directions.

2.2 Affine and Uniform Recurrence Equations

Definition 2.1 A *system of affine recurrence equations* is a finite collection of equations each of which has the form [63]:

$$I \in D \;\rightarrow\; V(I) = f(W(\rho_W(I)), \cdots) \tag{2.1}$$

where

- $D \subset \mathbf{Z}^n$ is a union of disjoint convex polytopes in $\mathbf{Z}^n$ and is called the *domain of the equation.* All defining equations with the same target variable must be distinct. This simplifies the presentation with no loss of generality. Domains of equations that have the same target variable must be disjoint. This ensures that the system of AREs is well-formed. The *index space*, denoted Φ, is the convex hull of the union of all the domains of equations. The *domain of variable* V, denoted Φ_V, is the convex hull of the union of the domains of all equations with target variable V. deq(Φ) (deq stands for "domain of equation") denotes the set of all domains of equations. deq(V) denotes the set of the domains of all equations with target variable V.

- I is called an *index vector* or *point*. The predicate $I \in D$ is called a *domain predicate*. Each domain predicate can be put into disjunct normal form, where each disjunct consists of a conjunction of predicates called *conditionals* that are generally linear (or affine) functions of the index vector I. Conditionals that are in one of the following forms

$$\pi I = \delta,\ \pi I \neq \delta,\ \pi I < \delta,\ \pi I \leqslant \delta,\ \pi I > \delta,\ \pi I \geqslant \delta \tag{2.2}$$

  are called *basic conditionals*, where $\pi \in \mathbf{Z}^n$ is a *coefficient vector* and $\delta \in \mathbf{Z}$ is a *constant*. $\pi I = \delta$ is called an *equality*. $\pi I \neq \delta$ is called an *inequality*. (Note that $\pi I = \delta \iff \pi I \leqslant \delta \wedge \pi I \geqslant \delta$ and $\pi I \neq \delta \iff \pi I < \delta \vee \pi I > \delta$.) A domain predicate is said to be in *normal form* if its constituent conditionals are basic conditionals.

- V and W are variable names belonging to a finite set $\mathcal{V}$.

- f is a function that has time complexity $O(1)$.

- ρ_W is an affine mapping from $\mathbf{Z}^n$ to $\mathbf{Z}^\ell$ $(0 \leqslant \ell \leqslant n)$ called an *index mapping*:

$$\rho_W(I) \ = \ \Delta_W I - \vartheta_W \tag{2.3}$$

  where $\Delta_W \in \mathbf{Z}^\ell \times \mathbf{Z}^n$ and $\vartheta_W \in \mathbf{Z}^\ell$. Δ_W is called the *linear part* of ρ_W. If $\ell = n$, n-vector $I - \rho_W(I)$ is called a *data dependence (vector)*.

- The "$\cdots$" stands for an arbitrary but fixed number of similar arguments.

- The symbol $\twoheadrightarrow$ separates the domain predicate from the corresponding defining equation. (Note that $\rightarrow$ is used in [63]. We choose $\twoheadrightarrow$ to avoid notational confusion with the use of $\rightarrow$ in the **if**-statement (Sect. 1.2).) □

The elements of a variable V are partitioned into three subsets:

- *Input data* are only read. They are identified by adding the following equations, called *input equations*, to the source program:

$$I \in D_V \ \twoheadrightarrow \ V(I) = v_{in_V(I)} \tag{2.4}$$

  where in_V is a mapping from $\mathbf{Z}^n$ to $\mathbf{Z}^{h_{in_V}}$ $(0 \leqslant h_{in_V} \leqslant n)$. That is, input element $V(I)$ has the value $v_{in_V(I)}$.

- *Output data* are only assigned. They are identified by adding the following equations, called *output equations*, to the source program:

$$I \in E_V \quad \twoheadrightarrow \quad v_{out_V(I)} = V(I) \tag{2.5}$$

where out_V is a mapping from $\mathbf{Z}^n$ to $\mathbf{Z}^{h_{out_V}}$ $(0 \leqslant h_{out_V} \leqslant n)$. That is, output element $V(I)$ has the value $v_{out_V(I)}$.

- *Intermediate data* are read and assigned. Equations that are neither input nor output equations are called *computation equations*. In other words, the computation equations are thosed introduced by Def. 2.1.

Remark. Input and output equations are not taken into account in the definitions of Φ_V, Φ, deq(V) and deq(Φ) (Def. 2.1). They only serve to declare which elements of variables are input and output data. It is the computation equations that specify the computations to be performed in the systolic array. □

Following [15,64], we sometimes write the set of all defining equations with the same target variable V in the following abbreviated form:

$$V(I) \;=\; \begin{cases} I \in D_1 & \twoheadrightarrow \quad f_1(W_1(\rho_1(I)), \ldots) \\ I \in D_2 & \twoheadrightarrow \quad f_2(W_2(\rho_2(I)), \ldots) \\ & \vdots \\ I \in D_p & \twoheadrightarrow \quad f_p(W_p(\rho_p(I)), \ldots) \end{cases} \tag{2.6}$$

When specifying a system of AREs, we shall adopt one of the previous two styles of notation, whichever is more convenient.

The concept of a data dependence graph plays an important rôle in systolic design. A *data dependence graph* has one node for each point of the index space and a directed arc from node J to node I iff a variable indexed by J is an argument in the equation for a variable indexed by I.

UREs are a subclass of AREs. They differ in the format of the index mapping. In UREs, the index mapping ρ_W of (2.3) is of the form:

$$\rho_W(I) \;=\; I - \vartheta_W \tag{2.7}$$

That is, the linear part of the index mapping is the identity matrix. The data dependence vector $I - \rho_W(I)$ in the AREs becomes ϑ_W in the UREs. ϑ_W is a constant, i.e., is independent of the index vector I. This exposes the unformity and

regularity of the computations prescribed by UREs, as opposed to AREs that are not UREs. Data dependence vector $I-\rho_W(I)$ in the AREs depends on the index vector I if the linear part Δ_W is not the identity matrix. If Δ_W is singular, the data dependence is not injective. Non-injective dependences are called *broadcast dependences.* Injective dependences are called *pipelining dependences.* Broadcast dependences allow the same variable to appear as arguments in the defining equations of an unbounded number of target variables. This represents the broadcast of a datum from one processor to an unbounded number of processors. In addition, irregular and non-constant data dependences result in irregular and non-constant channel connections. The advantage of UREs is that they enforce pipelining and regular and local channel connections.

By convention, a system of UREs is called *unconditional* if all its computation equations are defined for the index space, i.e., if $I \in \Phi$ is the domain predicate for all its computation equations. Otherwise, it is *conditional.*

We now introduce some concepts that are related to UREs. The *data dependence matrix* $\mathcal{D}$ is a matrix whose column vectors are the data dependence vectors [70]. For notational convenience, we assume that, for each variable name V, there is only one associated data dependence vector, denoted ϑ_V. When we write $\vartheta \in \mathcal{D}$, we mean that ϑ is a data dependence vector, that is, a column of $\mathcal{D}$.

The points of the index space are called the *computation points.* If I is inside and $I-\vartheta_V$ is outside the domain Φ_V of variable V, I is called a *first computation point* of V. If I is inside and $I+\vartheta_V$ is outside the domain Φ_V of variable V, I is called a *last computation point* of V. The set of first computation points, $\mathrm{fst}(\Phi_V, \vartheta_V)$, and the set of last computation points, $\mathrm{lst}(\Phi_V, \vartheta_V)$, of V are given by

$$\begin{aligned} \mathrm{fst}(\Phi_V, \vartheta_V) &= \{I \mid I \in \Phi_V \wedge I-\vartheta_V \notin \Phi_V\} \\ \mathrm{lst}(\Phi_V, \vartheta_V) &= \{I \mid I \in \Phi_V \wedge I+\vartheta_V \notin \Phi_V\} \end{aligned} \tag{2.8}$$

(A computation point of V can be both a first and a last computation point of V.) Pick a pair of first and last computation points that are in a line parallel to ϑ_V. Then, following the dependence between the two points, the first computation point must be computed before the last computation point.

The set $\mathrm{in}(\Phi_V, \vartheta_V)$ is the translate of $\mathrm{fst}(\Phi_V, \vartheta_V)$ by $-\vartheta_V$:

$$\mathrm{in}(\Phi_V, \vartheta_V) = \mathrm{fst}(\Phi_V, \vartheta_V) - \vartheta_V \tag{2.9}$$

Since $\mathrm{fst}(\Phi_V, \vartheta_V)$ and $\mathrm{in}(\Phi_V, \vartheta_V)$ are translates of each other, we shall use either of them in our analysis, as is convenient. By convention, the input data are supplied at the points of $\mathrm{in}(\Phi_V, \vartheta_V)$ and used for the first time at the points of $\mathrm{fst}(\Phi_V, \vartheta_V)$. The output data are assumed to be available at the points of $\mathrm{lst}(\Phi_V, \vartheta_V)$. If this is not the case, they can be made so by adding pipelining equations [70]; a *pipelining equation* is of the form:

$$I \in D \;\twoheadrightarrow\; V(I) = V(I - \vartheta_V) \tag{2.10}$$

A variable is called a *pipelining variable* if all its computation equations are pipelining equations.

Having explicitly defined the domains at which input data are supplied and the domains at which output data are defined, we can make the input equations of (2.4) and the output equations of (2.5) in the source UREs more explicit. Replacing D_V of (2.4) by $\mathrm{in}(\Phi_V, \vartheta_V)$ yields the input equations:

$$(\forall\, V : V \in \mathcal{V} : I \in \mathrm{in}(\Phi_V, \vartheta_V) \;\twoheadrightarrow\; V(I) = v_{in_V(I)}) \tag{2.11}$$

Replacing E_V of (2.5) by $\mathrm{lst}(\Phi_V, \vartheta_V)$ yields the output equations:

$$(\forall\, V : V \in \mathcal{V} : I \in \mathrm{lst}(\Phi_V, \vartheta_V) \;\twoheadrightarrow\; v_{out_V(I)} = V(I)) \tag{2.12}$$

where $v_{in_V(I)}$ and $v_{out_V(I)}$ are input and output data of variable V, respectively.

Let us introduce some notation. For $I, \vartheta \in \mathbf{Q}^n$, $\mathrm{ray}(I, \vartheta)$ denotes the set of the elements $I, I+\vartheta, I+2\vartheta, \cdots$ in $\mathbf{Q}^n$, i.e.,

$$\mathrm{ray}(I, \vartheta) \;=\; \{J \mid J = I + m\vartheta \wedge J \in \mathbf{Q}^n \wedge m \in \mathbf{Z}_0^+\}$$

We write $\mathrm{ray}(I, \pm\vartheta)$ for $\mathrm{ray}(I, \vartheta) \cup \mathrm{ray}(I, -\vartheta)$. For convenience, we write $I \xrightarrow{\vartheta} J$ if $J \in \mathrm{ray}(I, \vartheta)$ and $I \overset{\vartheta}{\nrightarrow} J$ if $J \notin \mathrm{ray}(I, \vartheta)$. For $D \subset \mathbf{Q}^n$, we define

$$\mathrm{rays}(D, \vartheta) \;=\; (\bigcup\, I : I \in D : \mathrm{ray}(I, \vartheta))$$

We write $\mathrm{rays}(D, \pm\vartheta)$ for $\mathrm{rays}(D, \vartheta) \cup \mathrm{rays}(D, -\vartheta)$.

Let us use the multiplication of two square matrices as an example to illustrate some concepts introduced previously. This example will be used for illustration throughout the thesis.

Example 2.1 $m \times m$ Matrix Product

Specification: $(\forall\, i, j : 0 < i \leqslant m \wedge 0 < j \leqslant m : c_{i,j} = (\sum\, k : 0 < k \leqslant m : a_{i,k} b_{k,j}))$

AREs:

$$A(i,j,k) = \left\{ \; 0<i\leqslant m \wedge 0=j \wedge 0<k\leqslant m \;\; \twoheadrightarrow \;\; a_{i,k} \right.$$

$$B(i,j,k) = \left\{ \; 0=i \wedge 0<j\leqslant m \wedge 0<k\leqslant m \;\; \twoheadrightarrow \;\; b_{k,j} \right.$$

$$C(i,j,k) = \begin{cases} 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0=k & \twoheadrightarrow \; 0 \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow \; C(i,j,k-1) \\ & \qquad +A(i,0,k)B(0,j,k) \; \bullet \end{cases} \tag{2.13}$$

$$c_{i,j} = \left\{ \; 0<i\leqslant m \wedge 0<j\leqslant m \wedge k=m \;\; \twoheadrightarrow \;\; C(i,j,k) \right.$$

Index Space: $\Phi = \{(i,j,k) \mid 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m\}$

Domains of Variables: $\Phi_A=\varnothing \quad \Phi_B=\varnothing \quad \Phi_C=\Phi$

Index Mappings:

$$\triangle_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \triangle_B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \triangle_C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\vartheta_A = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \vartheta_B = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \vartheta_C = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Data Dependence Graph ($m=4$)*:*

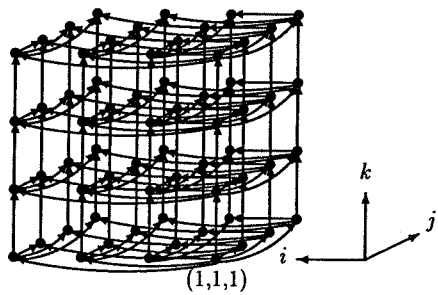


Figure 2–1: The data dependence graph of the AREs (2.13) ($m=4$).

Remark. When we label a computation equation by a number of graphical symbols (here, $\bullet$), we mean that the equation is only defined for the points (i.e., nodes) of the corresponding data dependence graph depicted by those symbols. □

UREs:

$$A(i,j,k) = \begin{cases} 0<i\leqslant m \wedge 0=j \wedge 0<k\leqslant m & \twoheadrightarrow a_{i,k} \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow A(i,j-1,k) \bullet \end{cases}$$

$$B(i,j,k) = \begin{cases} 0=i \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow b_{k,j} \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow B(i-1,j,k) \bullet \end{cases}$$

$$C(i,j,k) = \begin{cases} 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0=k & \twoheadrightarrow 0 \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow C(i,j,k-1) \\ & \quad +A(i,j-1,k)B(i-1,j,k) \bullet \end{cases} \tag{2.14}$$

$$c_{i,j} = \left\{ 0<i\leqslant m \wedge 0<j\leqslant m \wedge k=m \twoheadrightarrow C(i,j,k) \right.$$

Index Space: $\Phi = \{(i,j,k) \mid 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m\}$

Domains of Variables: $\Phi_A=\Phi_B=\Phi_C=\Phi$

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

First Computation Points: $\mathrm{fst}(\Phi,\vartheta_A) = \{(i,1,k) \mid 0<i\leqslant m \wedge 0<k\leqslant m\}$

$\mathrm{fst}(\Phi,\vartheta_B) = \{(1,j,k) \mid 0<j\leqslant m \wedge 0<k\leqslant m\}$

$\mathrm{fst}(\Phi,\vartheta_C) = \{(i,j,1) \mid 0<i\leqslant m \wedge 0<j\leqslant m\}$

Last Computation Points: $\mathrm{lst}(\Phi,\vartheta_A) = \{(i,m,k) \mid 0<i\leqslant m \wedge 0<k\leqslant m\}$

$\mathrm{lst}(\Phi,\vartheta_B) = \{(m,j,k) \mid 0<j\leqslant m \wedge 0<k\leqslant m\}$

$\mathrm{lst}(\Phi,\vartheta_C) = \{(i,j,m) \mid 0<i\leqslant m \wedge 0<j\leqslant m\}$

Data Dependence Graph ($m=4$)*:*

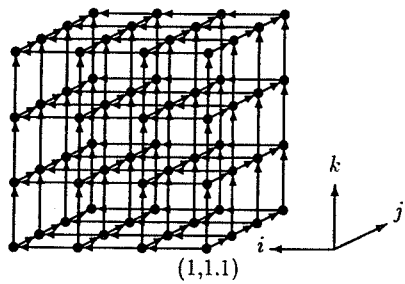


Figure 2–2: The data dependence graph of the UREs (2.14) ($m=4$). □

2.3 The Systolic Array Model

There is one systolic array model that has been the basis of many synthesis methods [20,31,57,62,70].

Definition 2.2 A *systolic array* is a network of cells that are placed at the grid points of an r-dimensional set $\mathcal{L} \in \mathbf{Z}^r$ $(0<r<n)$ with the following properties:

Prop. 1. (Synchrony of Computation) The array is driven by a global clock that ticks in unit time. A cell is active at every clock cycle (i.e., at every step).

Prop. 2. (Uniqueness of Channel Connections) For every variable V, postulate the existence of a unique, directed connection from the cell at location p to the cell at location $p+d_V$, for some constant vector $d_V \in \mathbf{Z}^r$. This postulate is either true for all $p \in \mathcal{L}$ or false for all $p \in \mathcal{L}$. A directed connection is also called a *channel*; it is an *input channel* to the cell at its destination and an *output channel* to the cell at its source. If $p \in \mathcal{L}$ and $p-d_V \notin \mathcal{L}$, cell p is called an *input cell* of variable V. If $p \in \mathcal{L}$ and $p+d_V \notin \mathcal{L}$, cell p is called an *output cell* of variable V. Both input and output cells of V are also called *border cells* of variable V; they are connected to the external environment. The cells that are not border cells of V are called *internal cells* of V.

Prop. 3. (Linearity of Velocity) All channels represented by d_V, for a fixed variable V, are associated with the same number of delay buffers. Thus, all elements of V move with the same constant velocity. □

Prop. 1 assumes that the computation at a point in the index space takes unit time. The sentence "a cell is active at every clock cycle" in Prop. 1 is interpreted as follows. If a cell receives a value on an input channel at a given time step, then it will receive a value on the same input channel and send a value on the corresponding output channel at the next time step. As a special case, if the values on all the input channels of a cell at a given time step are undefined (or $\perp$), then the cell is conventionally regarded as inactive. However, it is sometimes convenient to regard such a cell as active. In this case, the cell at the given step performs an undefined computation; it will send the undefined value, $\perp$, at the next time

step on all its output channels. By Prop. 2, only border cells are connected with the external environment. Depending on the length of a channel, a border cell may be a processor inside the array – a few processors away from the real border of the array. Prop. 2 requires that a cell always receives (sends) the elements of a variable via a fixed input (output) channel. It also enforces the regularity and locality of channel connections. Prop. 2 and Prop. 3 ensure that all elements of the same variable move with a fixed velocity.

2.4 The Space-Time Mapping

The space-time mapping maps every computation (or point) in the source UREs to a time step and a processor allocation. It is a linear (or affine) transformation of the index space. The advantage of linear (or affine) index transformations is that they enforce easily the linearity of variables' velocities and the regularity of channel connections.

Definition 2.3 A space-time mapping consists of two components: step and place. Useful functions defined in terms of step and place are flow and pattern.

step : $\mathbf{Z}^n \rightarrow \mathbf{Z}$, step$(I) = \lambda I$, $\lambda \in \mathbf{Z}^n$. step specifies the temporal distribution. λ is the *scheduling vector*. Point $I \in \Phi$ is computed at step λI. t_{fst} and t_{lst} are the first and last step number. $\mathcal{S} = \{t \mid t_{\text{fst}} \leqslant t \leqslant t_{\text{lst}}\}$ is the *scheduling space*. step is called the *step function* or *timing function*.

place : $\mathbf{Z}^n \rightarrow \mathbf{Z}^r$, place$(I) = \sigma I$, $\sigma \in \mathbf{Z}^{r \times n}$. place specifies the spatial distribution. σ is the *allocation matrix*. $\mathcal{P} = \{\sigma I \mid I \in \Phi\}$ is the *processor space*; it has r dimensions. Point $I \in \Phi$ is computed at cell σI. place is called the *place function* or *allocation function*.

flow : $\mathcal{V} \rightarrow \mathbf{Q}^r$, flow$(V) = \sigma\vartheta_V / \lambda\vartheta_V$, $\vartheta_V \in \mathcal{D}$. flow specifies the velocity with which elements of a variable travel at each step. Variable V is called *moving* if flow$(V) \neq 0$ and *stationary* if flow$(V) = 0$. $\sigma\vartheta_V$ represents the direction and length of a connecting channel at a cell for variable V; the number of delay buffers associated with that channel is $\lambda\vartheta_V - 1$.

$\mathsf{pattern} : \mathcal{V} \rightarrow \Phi_{\mathsf{fst}} \rightarrow \mathbf{Z}^{r \times n}$, $\mathsf{pattern}(V(I)) = \mathsf{place}(I) - (\mathsf{step}(I) - t_{\mathsf{fst}})\mathsf{flow}(V)$, $\Phi_{\mathsf{fst}} = (\bigcup\ V : V \in \mathcal{V} : \mathrm{fst}(\Phi_V, \vartheta_V))$. pattern specifies the location of variables in the processor space at the first step. □

The *space-time matrix*, Π, is the matrix formed by the scheduling vector in the first row and the allocation matrix in the remaining rows:

$$\Pi = \begin{bmatrix} \lambda \\ \sigma \end{bmatrix} = \begin{bmatrix} \lambda \\ \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_r \end{bmatrix} = \begin{bmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_n \\ \sigma_{1,1} & \sigma_{1,2} & \cdots & \sigma_{1,n} \\ \sigma_{2,1} & \sigma_{2,2} & \cdots & \sigma_{2,n} \\ & & \vdots & \\ \sigma_{r,1} & \sigma_{r,2} & \cdots & \sigma_{r,n} \end{bmatrix} \tag{2.15}$$

A space-time matrix uniquely determines a space-time mapping. For convenience, we sometimes refer to the space-time matrix as the space-time mapping. We shall denote the image of some x under a given space-time mapping with an overbar: $\overline{x}$. Here, x may be a point, a set of points, a data dependence vector, and so on. For example, the image $\overline{\vartheta}_V$ of data dependence vector ϑ_V is given by

$$\overline{\vartheta}_V = \Pi\vartheta = \begin{bmatrix} \lambda\vartheta_V \\ \sigma\vartheta_V \end{bmatrix} = \begin{bmatrix} \lambda\vartheta_V \\ \sigma_1\vartheta_V \\ \sigma_2\vartheta_V \\ \vdots \\ \sigma_r\vartheta_V \end{bmatrix} \tag{2.16}$$

This chapter considers only space-time mappings with $r = n-1$. They describe systolic arrays of $n-1$ dimensions. The motivation for employing an $(n-1)$-dimensional array to implement a system of n-dimensional UREs is to obtain a time-minimal systolic array, an array with the smallest number of execution steps (called *latency*) derivable from the source UREs.

The space-time mapping can be interpreted geometrically. All points that are scheduled concurrently belong to a hyperplane called a *temporal hyperplane*. The set of points that are scheduled at step t is in the temporal hyperplane $[\lambda : t]$. The processor space is obtained by a projection, namely, by eliminating one dimension from the index space along a chosen direction called the *projection vector*, and denoted by u. (This interpretation assumes that $r = n-1$.) Here, $u \in \mathbf{Z}^n$ is a

normalised vector that satisfies $\sigma u = 0$ [70]. The points that are in a line parallel to the projection vector are assigned the same processor location. The set of points that are allocated to cell p is in the line $\text{ray}(I, \pm u)$, where $\sigma I = p$. In what follows, we shall use either the projection vector or the allocation matrix, as is convenient.

Data dependence vectors are projected onto channels. For variable V, dependence vector ϑ_V is projected onto channel $\overline{\vartheta}_V$. The first component of $\overline{\vartheta}_V$ decremented by 1 represents the number of delay buffers associated with the channel and the vector formed by the remaining components represents the length and direction of the channel.

The following mapping rules characterise the validity of a space-time mapping.

Definition 2.4 A space-time mapping is *valid* for the source UREs in the systolic array model (Def. 2.2) iff

Precedence Rule: The step assigned to a target variable is larger than the steps assigned to its arguments. (This ensures that the data dependences prescribed in the source UREs are preserved.)

Computation Rule: Concurrent computations are mapped to different processors.

Communication Rule: At most one element of a variable is injected to a fixed input cell at any time step.

Delay Rule: The number of buffers associated with a communication channel is a non-negative integer. □

The following mapping conditions ensure the validity of a space-time mapping [63,70].

Theorem 2.1 *A space-time mapping* Π *is valid if*

- $(\forall\ V : V \in \mathcal{V} : \lambda\vartheta_V > 0)$ (*Precedence Constraint*)
- $\text{rank}(\Pi) = n$, *i.e.*, $\lambda u \neq 0$ (*Computation and Communication Constraint*)

It is straightforward to see that the precedence constraint is equivalent to the precedence rule. We sometimes write $\lambda\vartheta_V \geqslant 1$ instead of $\lambda\vartheta_V > 0$ in the precedence constraint if we want to emphasise the fact that the evaluation of a point takes unit

time. The computation and communication constraint implies the computation and communication rule. (The converse is not true. For example, any space-time mapping ensures the computation and and communication rule if the index space contains only one point.) This can be understood in two different ways. First, the non-singularity of Π (i.e., $\text{rank}(\Pi)=n$) ensures that the space-time mapping is a bijection from $\mathbf{Q}^n$ to $\mathbf{Q}^n$. Therefore, points that are mapped to the same step must differ in their processor coordinates. Second, $\lambda u \neq 0$ means that the projection vector is not orthogonal to the scheduling vector. Hence, any line parallel to the projection vector can only intersect at most one point in any temporal hyperplane. To see that the delay rule is satisfied, recall that data dependence vectors are projected onto channels. The number of delay buffers associated with a channel for variable V is the non-negative integer $\lambda\vartheta_V - 1$.

We may view the synthesis of systolic arrays as a process of program transformations [15,28,45]. It amounts to finding a suitable linear index transformation, i.e., a space-time mapping of the index space. In the transformed UREs, one index represents time and the remaining indices represent processor coordinates. Finding different systolic arrays means just finding different valid space-time mappings.

We use matrix product to illustrate both the geometrical and transformational view of the synthesis of systolic arrays.

Example 2.2 $m \times m$ Matrix Product

The following space-time mapping describes S. Y. Kung's two-dimensional arrays for matrix product [70] (Fig. 2–3):

$$\Pi = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad \Pi(i,j,k) = (t,x,y)$$

The array consists of m^2 cells and runs in $3m-2$ steps.

First, we consider the geometrical view. The processor space is obtained by projecting the index space along the projection vector $u=(0,0,1)$. The points that are scheduled at step t are in the hyperplane $[\lambda : t]$. The points that are mapped to cell p are in the line $\text{ray}(I, \pm u)$, where $\sigma I = p$. Elements of A travel along channel $\sigma\vartheta_A = (0,1)$ with a speed of one unit per step. Elements of B travel

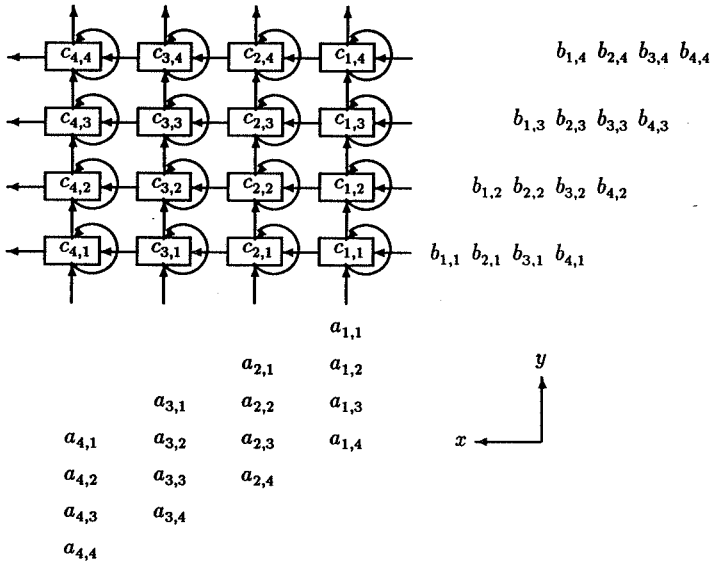


Figure 2–3: S. Y. Kung's systolic array for the UREs (2.14) of matrix product with data distribution at the first step ($m=4$).

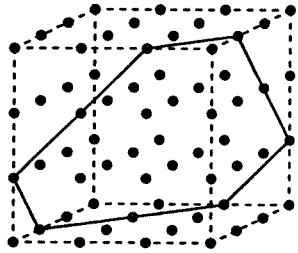
along channel $\sigma\vartheta_B=(1,0)$ with a speed of one unit per step. Elements of C are stationary because $\sigma\vartheta_C=0$.

Second, we consider the transformational view. This array is described precisely by the following UREs called the *space-time UREs* obtained from the source UREs and the space-time mapping by replacing (1) the indices i, j, and k of all the variables by t, x and y, respectively, (2) all the data dependence vectors by their images under the space-time mapping as defined in (2.16), and (3) the indices i, j and k in the domain predicates of the source UREs by the expressions in the corresponding components of vector $\Pi^{-1}(t,x,y)$.

The Space-Time UREs:

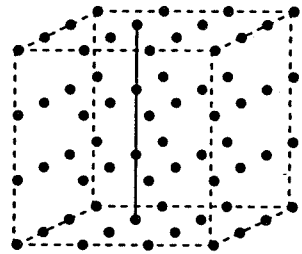
$$A(t,x,y)=\begin{cases} 0<x\leqslant m \wedge 0=y \wedge 0<t-x-y\leqslant m & \twoheadrightarrow\ a_{x,t-x-y} \\ 0<x\leqslant m \wedge 0<y\leqslant m \wedge 0<t-x-y\leqslant m & \twoheadrightarrow\ A(t-1,x,y-1) \end{cases}$$

$$B(t,x,y)=\begin{cases} 0=x \wedge 0<y\leqslant m \wedge 0<t-x-y\leqslant m & \twoheadrightarrow\ b_{t-x-y,y} \\ 0<x\leqslant m \wedge 0<y\leqslant m \wedge 0<t-x-y\leqslant m & \twoheadrightarrow\ B(t-1,x-1,y) \end{cases}$$



The scheduling vector $\lambda=(1,1,1)$

(a)



The projection vector $u=(0,0,1)$

(b)

Figure 2–4: The geometrial interpretation of S. Y. Kung's systolic array (Fig. 2–3). (a) The temporal distribution. All temporal hyperplanes are parallel with the normal λ; one is depicted. (b) The spatial distribution. Points in any line parallel to the projection vector u are projected to the same cell; one is depicted.

$$C(t,x,y) = \begin{cases} 0<x\leqslant m \wedge 0<y\leqslant m \wedge 0=t-x-y & \twoheadrightarrow\ 0 \\ 0<x\leqslant m \wedge 0<y\leqslant m \wedge 0<t-x-y\leqslant m & \twoheadrightarrow\ C(t-1,x,y) \\ & \quad +A(t-1,x,y-1)B(t-1,x-1,y) \end{cases}$$

$$c_{x,y} = \left\{ 0<x\leqslant m \wedge 0<y\leqslant m \wedge t-x-y=m \ \twoheadrightarrow\ C(t,x,y) \right.$$

From the space-time UREs, we can directly extract the following information: the latency, the number of channels required, the number of processors required, the input/output characteristics, the storage requirement, etc. □

The space-time mapping that specifies S. Y. Kung's array happens to map all input data of A and B to the border cells. (C is stationary and, therefore, by definition, at internal cells also. We discuss stationary variables in Chap. 7.) This can be seen clearly from the data dependence graph of the corresponding space-time UREs (Fig. 2–5): the input data of A and B are used for initialisation at the two facets facing us; the points in these two facets are projected to border cells. If some input or output data are not mapped to border cells, border communication can be ensured by an extension of the index space. This is the topic of Sect. 2.5.

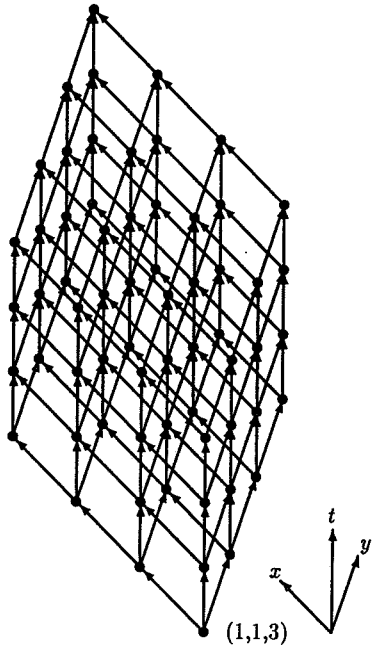


Figure 2–5: The data dependence graph of the space-time UREs that specify S. Y. Kung's systolic array (Fig. 2–3).

2.5 Input and Output Extension

There are space-time mappings that do not project first and last computation points to border cells. The corresponding input and output data are mapped to internal cells. This is costly because the I/O pins of a chip are restricted to the boundary of the chip and, consequently, non-local communication channels may be needed. To impose the restriction of border communication, the index space can be extended in such a way that the resulting first and last computation points are all projected to border cells.

The basic idea is to replicate a data dependence vector in the data dependence graph forward and backward until the images of points so created are at the boundary of the processor space. This extension depends on the projection vector

and works only for moving variables. We give a formal definition of such an extension, based on an informal idea described in [70]. Note that $\text{rays}(\Phi, \pm u)$ denotes the set of all the points in $\mathbf{Z}^n$ whose images under the space-time mapping are in the processor space.

Definition 2.5 The *extended index space* Ψ of the index space Φ with respect to the projection vector u is defined as follows:

$$\Psi_V^{\mathrm{s}} = \begin{cases} \textbf{if } \mathsf{flow}(V)=0 \rightarrow \varnothing \\ [\,]\ \mathsf{flow}(V)\neq 0 \rightarrow \text{rays}(\Phi, \pm u) \cap \text{rays}(\text{fst}(\Phi_V, \vartheta_V) - \vartheta_V, -\vartheta_V) \\ \textbf{fi} \end{cases}$$

$$\Psi_V^{\mathrm{d}} = \begin{cases} \textbf{if } \mathsf{flow}(V)=0 \rightarrow \varnothing \\ [\,]\ \mathsf{flow}(V)\neq 0 \rightarrow \text{rays}(\Phi, \pm u) \cap \text{rays}(\text{lst}(\Phi_V, \vartheta_V) + \vartheta_V, \vartheta_V) \\ \textbf{fi} \end{cases}$$

$$\Psi_V = \Phi_V \cup \Psi_V^{\mathrm{s}} \cup \Psi_V^{\mathrm{d}}$$

$$\Psi^{\mathrm{p}} = (\bigcup\ V : V \in \mathcal{V} : \Psi_V^{\mathrm{s}} \cup \Psi_V^{\mathrm{d}})$$

$$\Psi = \Phi \cup \Psi^{\mathrm{p}}$$

$$\Psi_V^{\perp} = \Psi \backslash \Psi_V$$

The points of Ψ_V^{s} are called the *soaking points* of V; they are generated in the extension of V along direction $-\vartheta_V$. The points of Ψ_V^{d} are called the *draining points* of V; they are generated in the extension of V along direction ϑ_V. The points of $\Psi_V^{\perp}$ are called the *undefined points* of V; they are generated in the extension of the other variables. Ψ_V is the *extended domain of variable* V. Ψ^{p} contains the soaking and draining points of all variables; they are called the *pipelining points.* The data dependence graph resulting from the extension is called the *extended data dependence graph.* □

The following concepts are analogous to those defined for the index space. The set of *new* first computation points of variable V over the extended index space is $\text{fst}(\Psi_V, \vartheta_V)$. The set of *new* last computation points of variable V over the extended index space is $\text{lst}(\Psi_V, \vartheta_V)$.

The two sets $\text{in}(\Psi_V, \vartheta_V)$ and $\text{in}(\Phi_V, \vartheta_V)$ are isomorphic by the bijection

$$\mathcal{I}_V : \text{in}(\Psi_V, \vartheta_V) \rightarrow \text{in}(\Phi_V, \vartheta_V), \quad \mathcal{I}_V(I) = J \quad \text{where } I \xrightarrow{\vartheta_V} J \tag{2.17}$$

($\text{fst}(\Psi_V, \vartheta_V)$ and $\text{fst}(\Phi_V, \vartheta_V)$ are also isomorphic because they are translates of $\text{in}(\Psi_V, \vartheta_V)$ and $\text{in}(\Phi_V, \vartheta_V)$, respectively.) Similarly, the two sets $\text{lst}(\Psi_V, \vartheta_V)$ and

$\mathrm{lst}(\Phi_V, \vartheta_V)$ are isomorphic by the bijection

$$\mathcal{O}_V : \mathrm{lst}(\Psi_V, \vartheta_V) \rightarrow \mathrm{lst}(\Phi_V, \vartheta_V), \quad \mathcal{O}_V(I) = J \quad \text{where } J \xrightarrow{\vartheta_V} I \tag{2.18}$$

The motivation underlying the extension of the index space is to have the input data of V supplied at the points of $\mathrm{in}(\Psi_V, \vartheta_V)$ and the output data of V defined at the points of $\mathrm{lst}(\Psi_V, \vartheta_V)$. The *extended source UREs* defined over the extended index space are a refinedment of the source UREs; they are obtained as follows:

- Replace the input equations as defined in (2.11) by

$$(\forall\, V : V \in \mathcal{V} : I \in \mathrm{in}(\Psi_V, \vartheta_V) \;\twoheadrightarrow\; V(I) = v_{in_V(\mathcal{I}_V(I))}) \tag{2.19}$$

- Replace the output equations as defined in (2.12) by

$$(\forall\, V : V \in \mathcal{V} : I \in \mathrm{lst}(\Psi_V, \vartheta_V) \;\twoheadrightarrow\; v_{out_V(\mathcal{O}_V(I))} = V(I)) \tag{2.20}$$

- Add the following pipelining equations

$$(\forall\, V : V \in \mathcal{V} : I \in \Psi_V \setminus \Phi_V \;\twoheadrightarrow\; V(I) = V(I - \vartheta_V)) \tag{2.21}$$

Let us illustrate how Kung-Leiserson's two-dimensional array for matrix product [41] can be obtained by an extension of the index space.

Example 2.3 $m \times m$ Matrix Product

The following space-time mapping describes Kung-Leiserson's two-dimensional array for matrix product (Fig. 2–6):

$$\Pi = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \qquad \Pi(i,j,k) = (t,x,y)$$

The array has $3m^2 - 3m + 1$ cells and runs in $5m - 4$ steps. Compared with S. Y. Kung's array, $2m - 2$ extra steps are needed for the propagation of input data from border cells and of output data to border cells.

The processor space is obtained by projecting the index space along the projection vector $u = (1,1,1)$. Let us look at the case where the index space is not extended. The set of first (last) computation points for each of the three variables A, B and C are contained in a hyperplane (Fig. 2–2). Since the projection vector is not orthogonal to the normal of any of these hyperplanes, some input and output data need to be handled at internal cells. For example, point $(1,1,1)$ is

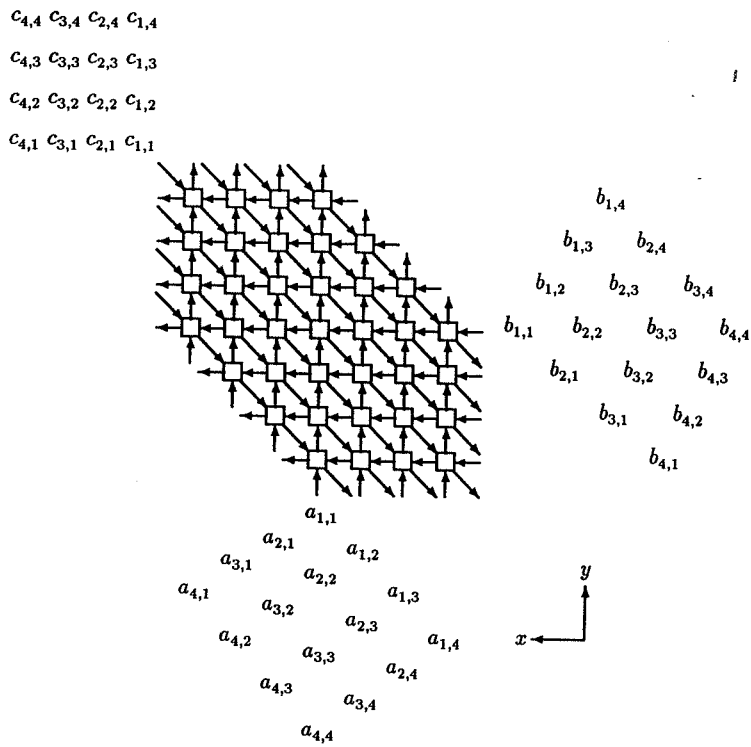


Figure 2–6: Kung-Leiserson's systolic array for the UREs (2.14) of matrix product with data distribution at the first step ($m=4$).

mapped to the cell at the center of the array. As a result, data elements $a_{1,1}$, $b_{1,1}$ and $c_{1,1}$ must be injected at this cell, which by definition is not a border cell. The extension of the index space enforces border communication (Fig. 2–7), because all input and output data are now defined at the boundary of the extended index space, which are projected to the border cells.

The extended source UREs of the source UREs (2.14) for the chosen projection vector $u=(1,1,1)$ follow from (2.19), (2.20) and (2.21):

The Extended Source UREs:

$$A(i,j,k)=\begin{cases} 0<i\leqslant m \wedge ((0<k\leqslant i \wedge i=j+m) \vee (i<k\leqslant m \wedge k=j+m)) \twoheadrightarrow a_{i,k} \\ (0<i<j+m \wedge -m+2\leqslant j\leqslant 0 \wedge 0<k<j+m) \vee \\ (0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m) \twoheadrightarrow A(i,j-1,k) \end{cases} \bullet \blacktriangledown$$

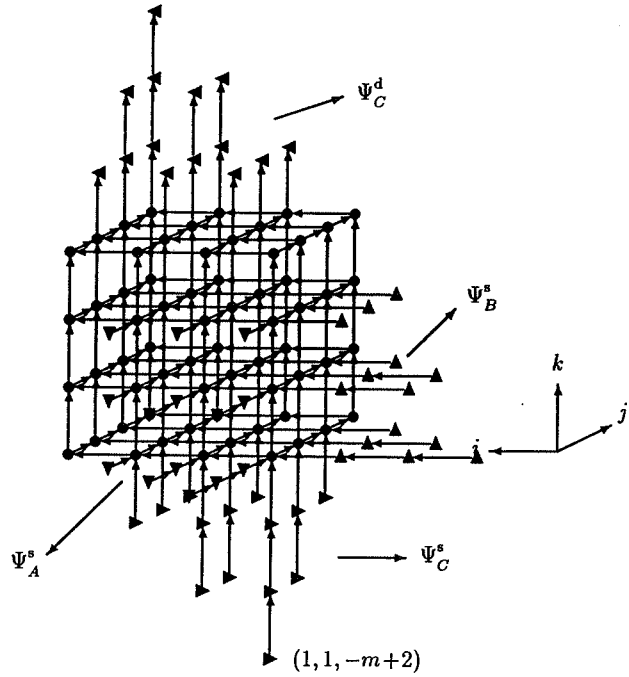


Figure 2–7: The extended data dependence graph of the UREs (2.14) for matrix product ($m = 4$). Extensions along ϑ_A and ϑ_B are not necessary because the output of A and B is of no interest.

$$B(i,j,k) = \begin{cases} 0<k\leqslant m \wedge ((0<j\leqslant k \wedge k=i+m) \vee (k<j\leqslant m \wedge j=i+m)) \twoheadrightarrow b_{k,j} \\ (-m+2\leqslant i\leqslant 0 \wedge 0<j<i+m \wedge 0<k<i+m) \vee \\ (0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m) \twoheadrightarrow B(i-1,j,k) \ \bullet \ \blacktriangle \end{cases}$$

$$C(i,j,k) = \begin{cases} 0<j\leqslant m \wedge ((0<i\leqslant j \wedge j=k+m) \vee (j<i\leqslant m \wedge i=k+m)) \twoheadrightarrow 0 \\ (0<i<k+m \wedge 0<j<k+m \wedge -m+2\leqslant k\leqslant 0) \vee \\ (k-m<i\leqslant m \wedge k-m<j\leqslant m \wedge m<k<2m) \twoheadrightarrow C(i,j,k-1) \ \blacktriangleleft \ \blacktriangleright \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m \twoheadrightarrow C(i,j,k-1) \\ \qquad\qquad +A(i,j-1,k)B(i-1,j,k) \ \bullet \end{cases}$$

$$c_{i,j} = \begin{cases} 0<j\leqslant m \wedge ((0<i\leqslant j \wedge i=k-m+1) \vee (j<i\leqslant m \wedge j=k-m+1)) \\ \qquad\qquad \twoheadrightarrow C(i,j,k) \end{cases} \qquad \square$$

The space-time UREs that specify Kung-Leiserson's array can be obtained straightforwardly from the previous extended source UREs following the procedure described in Sect. 2.4; they are omitted here.

2.6 Uniformisation

The projection vector completely determines the size and shape of the processor space and the channel interconnection pattern between processors. (The number of buffers associated with a channel remains to be determined by the scheduling vector.) Unless uniform, a system of recurrence equations cannot be mapped to a systolic array. For example, no systolic array can be obtained from a projection of the data dependence graph depicted in Fig. 2–1, since channels of unbounded length would be generated.

Uniformisation refers to the process by which AREs are transformed to UREs [63,64,89]. It aims at replacing non-constant data dependence vectors (i.e., broadcast and non-constant pipelining dependences) with constant ones (i.e., constant pipelining dependences). For this reason, uniformisation is also referred to as *data pipelining*. Uniformisation proceeds in two steps. In the first step, one identifies a set of constant data dependence vectors called *pipelining vectors* for the UREs to be constructed. In the second step, one expresses each data dependence vector in the AREs as a linear combination of the pipelining vectors and then replaces the data dependence vector by the pipelining vectors. Uniformisation is a difficult problem because the choice in neither of these two steps is unique.

There are basically three different uniformisation methods. We distinguish them by the way the pipelining vectors are introduced and used. We only describe the idea underlying each method and refer details to the corresponding references.

The method of [63,64] uses the null bases of the linear parts of the index mappings as the pipelining vectors. Consider the AREs (2.13) and the corresponding data dependence graph depicted in Fig. 2–1. $I-\rho_A(I)$ and $I-\rho_B(I)$ are broadcast dependences. Every element of A or B needs to be broadcast to m points. Let us consider variable A first. Element $a_{i,k}$ must be broadcast to the points $(i,1,k)$, $(i,2,k)$, $\cdots$, (i,m,k). We want to find a pipelining vector such

that $a_{i,k}$ can be routed incrementally to the corresponding m points along the pipelining vector. This pipelining vector may be chosen from the null basis of the linear part Δ_A of the index mapping for variable A. Let us choose $(0,1,0)$, since $\Delta_A(0,1,0)=0$. Similarly, we choose $(1,0,0)$ for the pipelining vector for variable B, since $\Delta_B(1,0,0)=0$. Consequently, we have transformed the AREs (2.13) to the UREs (2.14). This method cannot transform non-constant pipelining dependences to constant pipelining dependences. The reason is that the null basis of a non-singular matrix is $\varnothing$, which does not contribute pipelining vectors. The method also breaks down if the point at which a datum is produced has a non-constant distance from the point at which the datum is used for the first time in the pipeline.

The method of [89] uses canonical or non-canonical bases of the vector space $\mathbf{Q}^n$ as pipelining vectors. Let there be a data dependence between two points I and J. Choose n linearly independent integral vectors as canonical or non-canonical bases of the vector space $\mathbf{Q}^n$. We then express the dependence vector $I-J$ as a linear combination of the n pipelining vectors. If the linear parts of the index mappings are either the identity matrix or singular, all dependences of this form can be transformed to constant pipelining dependences. The resulting UREs may not allow the existence of a step function, but it is possible to find multi-step functions; one step function for each variable (Sect. 2.7; for details, see [70]).

The method of [63] uses the extreme rays of the convex cone generated by the data dependence vectors in the AREs as the pipelining vectors. Consider the set of data dependence vectors $\{(i-k,0,1) \mid k<i\leqslant m \wedge k\leqslant j\leqslant m+1 \wedge 0<k\leqslant m\}$ for an ARE specification of Gauss-Jordan elimination for square matrices [63]. There is one extreme ray $(1,0,0)$ and one vertex $(1,0,1)$ in the convex cone generated by these data dependence vectors. We can express the original data dependence vector $(i-k,0,1)$ as a linear combination of the two pipelining vectors $(1,0,0)$ and $(1,0,1)$:

$$(i-k,0,1) \;=\; (i-k-1)(1,0,0)+(1,0,1)$$

Thus, the non-constant dependences are transformed to constant pipelining dependences. Assuming that there exists a step function for the original AREs (i.e., that there exists a mapping of integers to the points in the index space such that the integer mapped to a target variable is always larger than those mapped to its

arguments) and that the original data dependence vectors generate a convex cone, all AREs can be transformed to UREs that permit a step function.

2.7 Conclusion

We have presented the basic techniques for the synthesis of systolic arrays from UREs and for the transformation of AREs to UREs. We conclude this chapter with a brief description of some other related issues in systolic design and comments on some further research topics.

The step function described previously is linear. A more general case is an affine step function: $\mathsf{step}(V, I) = \lambda I + \alpha_V$, where $\alpha_V \in \mathbf{Z}$. There is one step function for each variable $V \in \mathcal{V}$ in the source UREs [70]. The affine step function is more general than the linear step function because it may apply in the presence of zero data dependence vectors. The precedence constraint must be changed accordingly:

$$(\forall\, V, W, I, J : V, W \in \mathcal{V} \wedge I, J \in \Phi : W(I) \dashrightarrow V(J) \implies \mathsf{step}(V, J) > \mathsf{step}(W, I))$$

where $W(I) \dashrightarrow V(J)$ denotes that $W(I)$ is an argument appearing in the defining equation of target variable $V(J)$. For simplicity, we adopt linear rather than affine step functions. The existence of an affine step function in the source UREs guarantees the existence of a linear step function in the UREs resulting from a pre-index translation to the source UREs [70]. A pre-index translation can be obtained from the additive constants in $\{\alpha_V \mid V \in \mathcal{V}\}$. It serves to eliminate zero dependence vectors in the initial UREs. A search for more general forms of **step** and **place** that are capable of describing more general systolic arrays has been one of the research topics in systolic design [62,81].

A systolic array that is described by the space-time mapping as defined here is often referred to as a *parameterised systolic array*, since its size grows with the size of the problem. The *partitioning* of a parameterised systolic array to reduce it to some prescribed shape and size has been an active research topic recently [12,18,59,91]. Partitioning is achieved by first dividing the processor space into a number of blocks such that no two processors in the same block are assigned concurrent computations, and then merging the processors in the same block into

one processor. Partitioning is possible with an extension of the standard space-time mapping [18]. An alternative approach to partitioning takes place at the program level. The idea is to partition the problem into smaller problems that can be mapped directly to individual systolic arrays [33,60].

As a special case, partitioning also solves the problem of designing space-time minimal systolic arrays [10,13,17]. A systolic array is *space-time minimal* when it uses as few processors as any systolic array that has a minimal latency. It is also a parameterised systolic array since we need to retain the latency of the original array. The step and place functions for the partitioned systolic arrays are no longer linear functions (of the indices); they are piecewise linear and generally contain **mod** or **div** operators [25,81].

The space-time mapping technique that has been surveyed in this chapter delivers a systolic array of $n-1$ dimensions. If the systolic array postulates more dimensions than are available, a *projection* of the processor space becomes necessary. Projection can also be considered as a special case of partitioning, but it deserves a solution in its own right. We shall address this topic in Chap. 4, when we deal with the synthesis of data flow for one-dimensional arrays.

There have been some attempts to automate the generation of step and place. Early work on this subject reported in [51] only handles some specific problems. The minimisation of the step function with respect to the computation of one specific variable (rather than all variables) can be formulated as a linear programming problem [76,77]. 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 [87]. A method for minimising the number of processors for a given step function has also been reported [88]. The solution space of projection vectors is bounded; the minimising place function can be obtained by enumeration.

Systolic arrays can also be emulated in software [8,7,23,69,71]. The space-time UREs can be coded in programming languages, which can then be executed in processor networks like Warp [1] and Transputers [29]. Work on the parallelisation of nested loops is relevant here [5,22,44,85]. The formerly distinct areas of the parallelisation of nested loops and the synthesis of systolic arrays from algorithmic descriptions have reached closer proximity [6,24,35,84,90].

Two different methods for the synthesis of systolic arrays deserve particular attention. One is based on graph theory [48]. It transforms a synchronous circuit to a systolic array by redistributing the delays buffers associated with channels using the retiming theorem. The retiming theorem is often transferred to the context of systolic automata [16,34]. The other is based on the theory of functional program transformations [55,56]. It captures the regularity of algorithmic descriptions by the regularity exhibited in function compositions (so-called circuit combinators) rather than the regularity exhibited in data dependence graphs. This method is appealing for its simplicity and elegance in reasoning about various properties of systolic arrays and its facility to support the synthesis of heterogeneous processor arrays.

A lot of issues in systolic design are now well-understood, but much remains to be studied. Each issue mentioned previously needs further study. In addition, we describe several research topics that have not received adequate attention. One topic is the transformation of program specifications of a more general form to UREs. A desirable specification should only allow the existence of data dependences that enforce the correctness of the program, as is possible in GAMMA [3,4] and Unity [14]. But these languages are too general to support an implemented scheme of systolic design. They offer too many implementation choices because they are targeted at a much wider range of architectures than systolic arrays. To support the systematic development of systolic implementations, we need to develop a set of transformation rules that are specifically targeted at systolic arrays.

A second topic is the synthesis of systolic arrays with multi-level pipelining processors [54,80]. A large amount of computations prescribed by systolic algorithms have efficient pipelined implementations [40]. A third topic is the analysis and synthesis of programmable systolic arrays [32,47]. A further topic is the synthesis of control hardware for systolic arrays. This is becoming increasingly important if we are dealing with partitioned, projected or programmable systolic arrays. In subsequent chapters, we shall present a method for the systematic derivation of control signals for parameterised systolic arrays.

3

Control Flow Synthesis for (n–1)-Dimensional Systolic Arrays

3.1 Introductory Remarks

When the notion of a systolic array was introduced a decade ago [38,39], a systolic array was defined to be a network of simple cells, which always perform the same computation. Therefore, no control mechanism was required. Later, the notion of a systolic array was extended to permit a cell to perform different computations at different steps [30]. This made a control mechanism necessary that instructs the cell when to perform what computation. An example in [30] suggested that this control mechanism could be implemented by pipelining control signals, in addition to the data, though.

In Chap. 2, we have reviewed the space-time mapping technique for the synthesis of data flow for $(n-1)$-dimensional arrays. The space-time mapping provides a description of the layout and velocities of data with a guarantee that the right data arrive at the right cells at each step. But it fails to provide the control signals necessary to guarantee that the right computations are executed at each step.

The formal derivation of control flow has not received adequate attention for several reasons. First, there was an initial focus on the synthesis of data flow.

Application problems considered for systolic design were very simple. A manual derivation of control signals was adopted. This is no longer feasible when complex problems are implemented as systolic arrays. Second, a systolic array can be viewed as a system of space-time UREs (Sect. 2.4). Rather than in hardware, it can be realised in software on processor networks like Warp or Transputers [7,23, 71], where the issue of the derivation of control flow becomes irrelevant. Third, the space-time mapping technique does not provide a specification of the control flow. Alternative means must be sought. The necessity and complexity of developing a formal method for the derivation of control flow were not well understood and therefore research on this topic was neglected.

This chapter describes a method for the systematic synthesis of control signals for $(n-1)$-dimensional systolic arrays synthesised from iterative algorithms defined over a convex set of integer coordinates. We first present the method with respect the n-dimensional UREs and then generalise it to algorithms of a more general form in Sect. 3.8.

For the purpose of control flow synthesis, we write all computation equations of a variable V in the following abbreviated form:

$$V(I) = \begin{cases} I \in D_1 & \twoheadrightarrow \quad f_1(W_1(I-\vartheta_1),\ldots) \\ I \in D_2 & \twoheadrightarrow \quad f_2(W_2(I-\vartheta_2),\ldots) \\ & \vdots \\ I \in D_p & \twoheadrightarrow \quad f_p(W_p(I-\vartheta_p),\ldots) \end{cases} \tag{3.1}$$

We assume that $\{D_1, D_2, \cdots, D_p\}$ is a partition of the index space Φ. If not, we can always enforce it by adding the equation $I \in D_\perp \twoheadrightarrow \perp$, where $D_\perp = \Phi \backslash (\bigcup\ i : 0 < i \leqslant p : D_i)$. Therefore, deq$(V)$, which is the set of all domains of computation equations with target variable V (Def. 2.1), is a partition of the index space.

Just like the derivation of the data flow, the derivation of the control flow is a process of program transformation and refinement. In the derivation of the data flow, a problem specification is first transformed to a system of UREs and then the data flow is derived from the UREs by means of a space-time mapping. The derivation of the control flow can proceed along the same lines. The domain predicates in the source UREs constitute the initial specification of the control mechanism for the systolic array. We are motivated to transform these domain predicates to a system of UREs called the *control UREs*. We then replace the

domain predicates by predicates in the variables of the control UREs. We say that the domain predicates are *pipelined*. To derive the control flow, we simply apply the same space-time mapping that was previously applied to the source UREs, now to the control UREs. Of course, the space-time mapping must be valid for both the source UREs and the control UREs.

To make the presentation more precise, we duplicate the terminology of source UREs for control UREs and prefix the words "data" and "control", respectively. That is, we speak of data variables vs. control variables, and so on.

The rest of this chapter is organised as follows. Sect. 3.2 characterises control signals. Sect. 3.3 focuses on the construction of the control UREs and the subsequent replacement of the domain predicates of the source UREs by predicates in control variables. Sect. 3.4 discusses some techniques for optimising the specification of control signals before and after the space-time mapping is chosen. Sect. 3.5 extends the standard space-time mapping technique so that systolic arrays with a description of both data and control flow are directly synthesised from both the source and control UREs. Sect. 3.6 contains a survey of related work. Sect. 3.7 illustrates our method with two examples. Sect. 3.8 contains the conclusion of this chapter. It discusses the generalisation of our method to recurrence equations of a more general form.

3.2 The Requirements on Control Flow

We aim at implementing the control mechanism for a systolic array by pipelining control signals across the array. The control signals received at the input channels of a cell determine the computation to be performed by that cell. The communication of control signals in systolic arrays needs to satisfy certain requirements. Different requirements may lead to different specifications of control signals. We adopt the requirements on the communication of data defined in the systolic array model (Def. 2.2) for the communication of control signals. When viewed as electronic signals, control signals and data need not be distinguished.

The control signals in $(n-1)$-dimensional arrays must satisfy four restrictions:

Rst. 1. Control signals are both input and output at the border cells of the array.

Rst. 2. The number of different control values that travel along a control channel is a constant, i.e., is independent of the size and dimension of the source UREs.

Rst. 3. A control signal moves with a constant velocity.

Rst. 4. Once input, a control signal remains unchanged.

Rst. 1 keeps the number of pins required in the systolic array to a minimum. Rst. 2 ensures that the bit-width of a channel is a constant. Both restrictions cater to the constraints imposed by the current VLSI technology. Rst. 3 stems from the fact that the control signals are specified by UREs and that the space-time mapping is linear. Rst. 4 is not imperative but useful. It happens to be a property of the control UREs constructed for $(n-1)$-dimensional arrays, but will not hold for the control UREs constructed for one-dimensional arrays in Chap. 5.

Rsts. 1 and 3 are imposed on the data in the systolic array model (Def. 2.2). Rst. 2 is guaranteed by the implicit assumption that a data channel has constant bit-width.

Remark. The domain predicates of the source UREs constitute the initial specification of control signals. As far as the synthesis of control signals is concerned, the form of the defining equations of the variables in the source UREs is irrelevant. When we use the source UREs for illustration, it is sufficient to describe them by the sets $\mathrm{deq}(V)$ for all the variables $V \in \mathcal{V}$. We shall not specify these sets in terms of predicates in indices. Rather, we specify them by means of diagrams; one diagram per variable. The diagram associated with variable V depicts the partition of the index space specified by $\mathrm{deq}(V)$. □

3.3 The Synthesis of Control Flow

Our systolic array model requires that the data are exchanged with the external environment only through border cells. The restriction to border communication can be enforced by an extension of the index space with respect to the projection vector. The source UREs defined over the index space Φ can be extended to the extended index space Ψ by adding pipelining equations for all variables at the

newly generated points, i.e., in $\mathsf{C}_\Psi\Phi$. For notational convenience, we write the extension of (3.1) over the extended index space as follows:

$$V(I) = \begin{cases} I \in \Phi \wedge I \in D_1 & \twoheadrightarrow \quad f_1(W_1(I-\vartheta_1),\ldots) \\ I \in \Phi \wedge I \in D_2 & \twoheadrightarrow \quad f_2(W_2(I-\vartheta_2),\ldots) \\ & \vdots \\ I \in \Phi \wedge I \in D_p & \twoheadrightarrow \quad f_p(W_p(I-\vartheta_p),\ldots) \\ I \in \mathsf{C}_\Psi\Phi & \twoheadrightarrow \quad V(I-\vartheta) \end{cases} \tag{3.2}$$

The last defining equation of V corresponds to the pipelining equations in (2.21). In other words, it specifies that the elements of V at the cells of $\mathsf{place}(\mathsf{C}_\Psi\Phi)$ are propagated at the steps $\mathsf{step}(\mathsf{C}_\Psi\Phi)$. This suffices for the purpose of the control flow derivation, because it is the domain $\mathsf{C}_\Psi\Phi$ rather than the dependence vector ϑ that determines the specification of control signals.

We shall construct the control UREs for the extended source UREs. The construction is based on a characterisation of the points in the extended index space. This characterisation aims at partitioning the extended index space into blocks such that all points in one block may be specified by a common set of control signals. For this purpose, we introduce the concept of a type for points. It relies on the following equivalence relation on Φ.

Definition 3.1 $\textcircled{T}$ is the following equivalence relation on Φ:

$$I \textcircled{T} J \iff (\forall\, D : D \in \mathsf{deq}(\Phi) : I \in D \iff J \in D)$$

We say that I and J are of the same *type* if $I \textcircled{T} J$. Points that are not of the same type are said to have different types. □

Recall that the defining equations of the same target variable are distinct (Def. 2.1). Therefore, the defining equations for a variable agree at points of the same type and the defining equations for some variable disagree at points of different types.

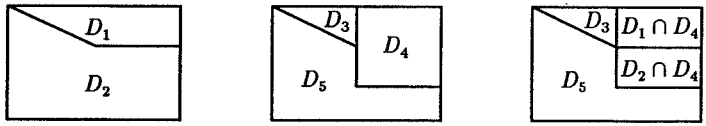
Remark. The definition of $\textcircled{T}$ does not take the input and output equations in the source UREs into account. Remember that $\mathsf{deq}(\Phi)$ does not contain the domains of input and output equations (Sect. 2.2). Input and output equations only serve to declare which elements of variables are input and output data. They do not specify the computations to be performed in the systolic array. □

The quotient set of Φ by $\textcircled{T}$ is given by

$$\Phi/\textcircled{T} \;=\; (\Cap V : V \in \mathcal{V} : \mathrm{deq}(V)) \qquad (3.3)$$

To see why this is so, we note that $\mathrm{deq}(V)$ for a fixed variable V is a partition of the index space. So a subset of Φ is a $\textcircled{T}$-class iff it is the intersection of $|\mathcal{V}|$ blocks, one from every partition in $\{\mathrm{deq}(V) \mid V \in \mathcal{V}\}$.

Example 3.1 Let the source UREs be specified as in Figs. 3–1(a) and (b).



(a) $\mathrm{deq}(U)$ (b) $\mathrm{deq}(W)$ (c) $\Phi/\textcircled{T}$

Figure 3–1: The construction of the quotient set $\Phi/\textcircled{T}$.

The quotient set $\Phi/\textcircled{T}$ depicted in Fig. 3–1(c) follows from (3.3):

$$\begin{aligned}
& \Phi/\textcircled{T} \\
= \;& \{\Phi/\textcircled{T} \text{ of } (3.3)\} \\
& (\Cap V : V \in \mathcal{V} : \mathrm{deq}(V)) \\
= \;& \{\mathcal{V} = \{U, W\}\} \\
& \mathrm{deq}(U) \Cap \mathrm{deq}(W) \\
= \;& \{\text{Definition of } \Cap \text{ in Sect. 1.2}\} \\
& \{D_1 \cap D_3, D_1 \cap D_4, D_1 \cap D_5, D_2 \cap D_3, D_2 \cap D_4, D_2 \cap D_5\} \\
= \;& \{D_1 \cap D_3 = D_3; D_1 \cap D_5 = \varnothing; D_2 \cap D_3 = \varnothing; D_2 \cap D_5 = D_5\} \\
& \{D_3, D_1 \cap D_4, D_2 \cap D_4, D_5\}
\end{aligned}$$

□

Once the control UREs are constructed, we need to replace the domain predicates of the extended source UREs by predicates in control variables. Here, the quotient set $D/\textcircled{T}$ of a domain $D \in \mathrm{deq}(\Phi)$ of equation by $\textcircled{T}$ comes into play.

The quotient sets $D/\textcircled{T}$ for all domains $D \in \mathrm{deq}(\Phi)$ in Ex. 3.1 are:

$$\begin{aligned}
D_1/\textcircled{T} &= \{D_3, D_1 \cap D_4\} \\
D_2/\textcircled{T} &= \{D_2 \cap D_4, D_5\} \\
D_3/\textcircled{T} &= \{D_3\} \\
D_4/\textcircled{T} &= \{D_1 \cap D_4, D_2 \cap D_4\} \\
D_5/\textcircled{T} &= \{D_5\}
\end{aligned}$$

They are subsets of $\Phi/\textcircled{T}$. The following lemma states that this is always so.

Lemma 3.1 $(\forall\, D : D \in \mathrm{deq}(\Phi) : D/\textcircled{T} \subseteq \Phi/\textcircled{T})$.

Proof. The domain D of an equation is a subset of Φ, i.e., $D \subseteq \Phi$. $D/\textcircled{T}$ is the quotient set $\Phi/\textcircled{T}$ restricted to D. That is, $D/\textcircled{T} = \{D\} \Cap \Phi/\textcircled{T}$. By the hypothesis, $D \in \mathrm{deq}(\Phi)$. An application of (3.3) yields $D/\textcircled{T} = \{D\} \Cap \Phi/\textcircled{T} \subseteq \Phi/\textcircled{T}$. □

The set consisting of $\complement_\Psi \Phi$ and all $\textcircled{T}$-classes is a partition of the extended index space Ψ. All points in one block of this partition can be specified by a common set of control signals. The specification of control signals proceeds in two steps (Fig. 1–1):

1. Construct the control UREs from the domain predicates in the extended source UREs. The control UREs are divided into two systems:

   - *Computation control UREs* (CCUREs) distinguish different types of computation points, (i.e., different $\textcircled{T}$-classes). They specify the control signals for the evaluation of the computation points in the index space.
   - *Propagation control UREs* (PCUREs) distinguish pipelining points (of $\complement_\Psi \Phi$) from computation points (of Φ). They specify the control signals for the pipelining of input (output) data that are mapped to the internal cells of the array from (to) the border cells.

   The dichotomy exhibited by the control UREs is natural since $\{\complement_\Psi \Phi, \Phi\}$ is a partition of the extended index space Ψ. The points in $\complement_\Psi \Phi$ are specified by pipelining equations, and the points in Φ are specified by the source UREs.

2. Replace the domain predicates in the extended source UREs by equivalent predicates in the control variables called *guards*: replace those that are in the source UREs by predicates in computation control variables and those that are in the extended source UREs but not in the source UREs, i.e., $I \in \Phi$ and $I \in \complement_\Psi \Phi$, by predicates in propagation control variables. The resulting extended source UREs and the control UREs are called *pipelined UREs*.

If the source UREs are unconditional, i.e., if $I \in \Phi$ is the domain predicate of all computation equations, then $\Phi/\textcircled{T} = \{\Phi\}$. In this case, computation control

is not needed. If all input and output data prescribed by the source UREs are mapped to border cells, propagation control is not needed.

The specification of a control variable V proceeds in three steps:

- Choose a non-zero normalised vector $\vartheta_V \in \mathbf{Z}^n$ as the control dependence vector. (Each control variable is associated with one dependence vector.)
- Define the set of control values, denoted $\mathrm{sig}(V)$, and the input equation that specifies the initialisation of V with control values in $\mathrm{sig}(V)$. The domain of the input equation is $\mathrm{in}(\Psi, \vartheta_V)$.
- Define the computation equation as $I \in \Psi \twoheadrightarrow V(I) = V(I - \vartheta_V)$.

The control signals of control variable V are initialised at $\mathrm{in}(\Psi, \vartheta_V)$ and pipelined along ϑ_V across the extended index space Ψ. We shall address the minimisation of the domains of control variables in Sect. 3.4. The control UREs are unconditional. Otherwise, we would need to add a second level of control UREs that governs the computations prescribed by the initial control UREs, and so on.

Let us discuss how the control UREs specified this way satisfy the four restrictions on control signals described in Sect. 3.2. The sets $\mathrm{fst}(\Psi, \vartheta_V)$ and $\mathrm{lst}(\Psi, \vartheta_V)$ of control variable V are part of the boundary of the extended index space (Def. 2.8). The extension of the index space ensures that all first and last computation points of data variables are mapped to border cells. That is, only points of $(\bigcup V : V \in \mathcal{V} : \mathrm{fst}(\Psi_V, \vartheta_V) \cup \mathrm{lst}(\Psi_V, \vartheta_V))$ are mapped to the border cells. The sets $\mathrm{fst}(\Psi, \vartheta_V)$ and $\mathrm{lst}(\Psi, \vartheta_V)$ may contain points that are mapped to internal cells, since Ψ_V is generally a proper subset of Ψ. To enforce the border communication of control signals, i.e., Rst. 1, we extend the extended index space (with respect to the projection vector) for the control UREs in the same way as we extended the index space for the source UREs in Def. 2.5 (see Figs. 6–1(a) and (c)). Rst. 2 is satisfied if we restrict $|\mathrm{sig}(V)|$ to be a constant, Rst. 3 is satisfied since the space-time mapping is linear, and Rst. 4 is satisfied since the computation equations of control variables are pipelining equations.

The construction of the control UREs relies on the following theorem about the separation of two convex sets in $\mathbf{R}^n$ by hyperplanes [72].

Theorem 3.1 (*Separation Theorem*) *If C_1 and C_2 are non-empty disjoint convex polytopes in $\mathbf{R}^n$, there always exists a hyperplane that strictly separates C_1 and C_2.*

To apply the Separation Theorem, each Ⓣ-class must be a convex polytope in $\mathbf{Z}^n$. If not, it must be partitioned into a union of convex polytopes. We refer to a partition of the index space as a Ⓟ-*partition* if it is finer than the quotient set $\Phi/$Ⓣ and if every block of the partition is a convex polytope. The quotient set $\Phi/$Ⓣ is a Ⓟ-partition if all Ⓣ-classes are convex.

Let us address the problem of obtaining Ⓟ-partitions from the source UREs. We can approach this problem in two different but equivalent ways. In the first approach, we obtain Ⓟ-partitions by partitioning each Ⓣ-class into a union of convex polytopes. We refer to a partition of a Ⓣ-class $C \in \Phi/$Ⓣ as a *convex class partition*, denoted clapar(C), if the blocks of the partition are convex polytopes. This gives rise to a Ⓟ-partition, denoted Ⓟ_{clapar}:

$$\text{Ⓟ}_{\text{clapar}} = (\bigcup C : C \in \Phi/\text{Ⓣ} : \text{clapar}(C)) \tag{3.4}$$

In the second approach, we obtain Ⓟ-partitions by partitioning each domain of equations into a union of convex polytopes. We refer to a partition of a domain $D \in \text{deq}(\Phi)$ as a *convex domain partition*, denoted dompar(D), if the blocks of the partition are convex polytopes. We define

$$\text{condeq}(V) = (\bigcup D : D \in \text{deq}(V) : \text{dompar}(D)) \tag{3.5}$$

$$\text{condeq}(\Phi) = (\bigcup V : V \in \mathcal{V} : \text{condeq}(V)) \tag{3.6}$$

This gives rise to a Ⓟ-partition, denoted Ⓟ_{dompar}:

$$\text{Ⓟ}_{\text{dompar}} = (\Cap V : V \in \mathcal{V} : \text{condeq}(V)) \tag{3.7}$$

Ⓟ_{dompar} is a Ⓟ-partition because condeq(V) is finer than deq(V) and its blocks are convex polytopes.

The previous two ways of obtaining Ⓟ-partitions are equivalent.

Lemma 3.2 *Let Ⓟ_{c} be the set of all Ⓟ-partitions defined by (3.4), and Ⓟ_{d} be the set of all Ⓟ-partitions defined by (3.7). Then, $\text{Ⓟ}_{\text{c}} = \text{Ⓟ}_{\text{d}}$.*

Proof. Every Ⓟ_{clapar} in Ⓟ_{c} is also in Ⓟ_{d} if we choose dompar(D) $= \{D\} \Cap \text{Ⓟ}_{\text{clapar}}$ for every domain of equation $D \in \text{deq}(\Phi)$. Every Ⓟ_{dompar} in Ⓟ_{d} is also in Ⓟ_{c} if we choose clapar(C) $= \{C\} \Cap \text{Ⓟ}_{\text{dompar}}$ for every Ⓣ-class $C \in \Phi/$Ⓣ. □

Remark. To make the Separation Theorem applicable to non-convex Ⓣ-classes, we have suggested to partition them into convex polytopes. In fact, the Separation Theorem is applicable if we decompose (rather than partition) each non-convex Ⓣ-class into a number of not necessarily disjoint convex polytopes whose union is the Ⓣ-class. Similarly, a non-convex domain of an equation can be expressed as a union of not necessarily disjoint convex polytopes. That is, the elements in clapar(C) or dompar(D) may not be disjoint. Consequently, condeq(V), $Ⓟ_{\mathbf{clapar}}$ and $Ⓟ_{\mathbf{dompar}}$ each may not be a partition of the index space (the union of the elements of each set is the index space, though). But, Lemma 3.2 remains valid. (To prove $Ⓟ_c \subseteq Ⓟ_d$, we need to choose dompar(D) $= \{D\} \Cap Ⓟ_{\mathbf{clapar}}$ for every $D \in$ deq(V) for some variable V, and dompar(D) $= \{D\} \Cap \Phi/Ⓣ$ for every $D \in$ deq(Φ)$\setminus$deq(V).)

Besides simplicity of presentation, there is an important reason why we have chosen to partition rather than to decompose non-convex domains of equations or Ⓣ-classes. Let there be a decomposition of a non-convex domain of equation. This decomposition corresponds to the following unique partition of the domain: two elements of the domain are in the same block of the partition iff they are in the same elements of the decomposition. The same is true of a non-convex Ⓣ-class. The partition obtained this way does not introduce any additional conditionals. Proc. 3.1, as will be presented in Sect. 3.3.1.2, delivers the same specification of control signals for both the decomposition and the corresponding partition. □

To find Ⓟ-partitions, we do not need to construct the quotient set $\Phi/Ⓣ$ in order to obtain convex class partitions for non-convex Ⓣ-classes. Lemma 3.2 states that finding different Ⓟ-partitions amounts to finding different convex domain partitions for all domains of the equations in the source UREs. Recall the definition of AREs in Def. 2.1. Each domain of an equation must be expressed as a union of disjoint convex polytopes. This means that the set of these convex polytopes is a convex domain partition. Therefore, the domain predicates in the source UREs uniquely determine a Ⓟ-partition, which is defined by (3.5), (3.6) and (3.7), where convex domain partitions dompar(D) for every $D \in$ deq(Φ) are specified in the source UREs. To distinguish this partition from the others, we denote it by $Ⓟ_{\mathrm{src}}$.

Remark. From now on, when using the source UREs for illustration, we shall describe them by the sets condeq(V) for all variables $V \in \mathcal{V}$ in the source UREs. We continue to use diagrams to depict these sets. In the diagram for a variable V, all

solid lines form the partition deq(V), all dashed lines inside the region that depicts a non-convex polytope $D \in \text{deq}(V)$ form the convex domain partition dompar(D) and, consequently, all solid and dashed lines form the partition condeq(V). □

3.3.1 The Computation Control Flow

The construction of the CCUREs imposes a Ⓟ-partition by means of the computation control variables. To do so, we must find a set of separating hyperplanes that mutually separate all Ⓟ-blocks. Each of these separating hyperplanes divides $\mathbb{Z}^n$ into two halves. We write $\mathfrak{S}$ for the set of half-spaces associated with these hyperplanes, one half-space per hyperplane, and define $\mathfrak{S}^c = \{\complement_{\mathbb{Z}^n} S \mid S \in \mathfrak{S}\}$, i.e., $\mathfrak{S}^c$ contains the complements of the half-spaces of $\mathfrak{S}$.

Definition 3.2 A set S of half-spaces is called a set of *defining half-spaces* of an integral convex polytope C if C is the intersection of these half-spaces, i.e., if $C = (\bigcap H : H \in S : H)$. □

Definition 3.3 A set $\mathfrak{S}$ of half-spaces is called a *separation set* (of a Ⓟ-partition) if $\mathfrak{S} \cup \mathfrak{S}^c$ contains a set of defining half-spaces for every Ⓟ-block. □

Since Ⓟ-blocks are disjoint convex polytopes, the existence of a separation set is guaranteed by the Separation Theorem. In Sect. 3.3.1.1, we present the specification of computation control flow from a separation set for a fixed but arbitrary Ⓟ-partition. In Sect. 3.3.1.2, we describe the construction of a separation set for the Ⓟ-partition Ⓟ_{src}. The way that this separation set is constructed permits the development of a mechanisable procedure for the specification of computation control flow directly from the source UREs.

3.3.1.1 Specifying Computation Control Flow from a Separation Set

The construction of the CCUREs from a separation set $\mathfrak{S}$ is straightforward. We associate a distinct computation control variable, C_S, with every $S = [\pi_S \boxdot \delta_S]$ in $\mathfrak{S}$, where $\pi_S \in \mathbb{Z}^n$ is normalised, $\delta_S \in \mathbb{Z}$, and $\boxdot \in \{<, \leqslant, >, \geqslant\}$. (Conversely, we identify the corresponding half-space in $\mathfrak{S}$ of a computation control variable C by H_C.) The associated control dependence vector ϑ_{C_S} of C_S is a solution of

$\vartheta_{C_S}\pi_S = 0$. C_S takes on two different values: 0 in the half-space S and 1 in the half-space $\complement_{\mathbb{Z}^n}S$. That is, $\mathrm{sig}(C_S) = \{0,1\}$. (This is data pipelining (Sect. 2.6). Instead of performing tests $I \in S$ and $I \in \complement_{\mathbb{Z}^n}S$ for every point in the extended index space, the results of these tests at the corresponding half-space are shared.)

The CCUREs are defined as follows:

$$\left(\forall\, S : S \in \mathfrak{S} \;:\; \left\{ \begin{array}{l} \vartheta_{C_S}\pi_S = 0 \\ C_S(I) \;=\; \left\{ \begin{array}{ll} I \in \mathrm{in}(\Psi, \vartheta_{C_S}) \cap S & \twoheadrightarrow 0 \\ I \in \mathrm{in}(\Psi, \vartheta_{C_S}) \cap \complement_{\mathbb{Z}^n}S & \twoheadrightarrow 1 \\ I \in \Psi & \twoheadrightarrow C_S(I - \vartheta_{C_S}) \end{array} \right. \end{array} \right. \right) \tag{3.8}$$

Remark. We abuse the notion of UREs slightly here. The equation $\vartheta_{C_S}\pi_S = 0$ is not a URE; it specifies that ϑ_{C_S} is any vector that is orthogonal to π_S. □

Once the CCUREs are constructed, we can proceed to replace the domain predicates in the extended source UREs that are also in the source UREs by predicates in the computation control variables.

Definition 3.4 Let $X \subseteq \Psi$. $\mathcal{P}(X)$ denotes the predicate in computation (pipelining) control variables that replaces predicate $I \in X$. X' contains the points of Ψ at which $\mathcal{P}(X)$ holds:

$$X' \;=\; \{I \mid I \in \Psi \wedge \mathcal{P}(X)\}$$ □

The definition of $\mathcal{P}(X)$ aims at the establishment of the equality of X and X'.

Definition 3.5 The computation control flow is *correct* if $(\forall\, D : D \in \mathrm{deq}(\Phi) : D = D')$. □

The following lemma permits a hierarchical definition of predicates in control variables that will replace the domain predicates of the extended source UREs.

Lemma 3.3 *Let $X_1, X_2, \cdots, X_x$ be subsets of a set $X \subseteq \Psi$. Let $\mathcal{P}(X) = (\exists\, i : 0 < i \leqslant x : \mathcal{P}(X_i))$. Then (1) $X' = (\bigcup\, i : 0 < i \leqslant x : X'_i)$. (2) $X = (\bigcup\, i : 0 < i \leqslant x : X'_i) \iff X = X'$.*

Proof. Since (1) implies (2), it suffices to prove (1).

$$\begin{array}{ll} & X' \\ = & \{\text{Def. 3.4}\} \end{array}$$

$$
\begin{aligned}
& \{I \mid I \in \Psi \wedge \mathcal{P}(X)\} \\
= \; & \{\mathcal{P}(X) = (\exists\, i : 0 < i \leqslant x : \mathcal{P}(X_i)) \text{ by hypothesis}\} \\
& \{I \mid I \in \Psi \wedge (\exists\, i : 0 < i \leqslant x : \mathcal{P}(X_i)\} \\
= \; & \{\text{Algebraic manipulation}\} \\
& (\bigcup\, i : 0 < i \leqslant x : \{I \mid I \in \Psi \wedge \mathcal{P}(X_i)\}) \\
= \; & \{\text{Def. 3.4}\} \\
& (\bigcup\, i : 0 < i \leqslant x : X'_i) \qquad \square
\end{aligned}
$$

We need to define predicates $\mathcal{P}(D)$ in order to replace domain predicates $I \in D$ for every $D \in \text{deq}(\Phi)$. By Lemma 3.1, $D/\textcircled{T} \subseteq \Phi/\textcircled{T}$, i.e., all the elements of $D/\textcircled{T}$ are $\textcircled{T}$-classes. $\mathcal{P}(D)$ is defined in terms of predicates $\mathcal{P}(C)$ for every $C \in D/\textcircled{T}$:

$$\mathcal{P}(D) \;=\; (\exists\, C : C \in D/\textcircled{T} : \mathcal{P}(C)) \tag{3.9}$$

Lemma 3.4 $(\forall\, D : D \in \text{deq}(\Phi) : D = D') \iff (\forall\, C : C \in \Phi/\textcircled{T} : C = C')$.

Proof. "$\Longleftarrow$" follows from Lemma 3.3 by using (3.9) and the hypothesis. Let us prove "$\Longrightarrow$". For every $C \in \Phi/\textcircled{T}$, either of the following must be true by the definition of $\textcircled{T}$: (1) $C \in \text{deq}(\Phi)$, and (2) there are at least two elements X and Y in $\text{deq}(\Phi)$ such that $C \subset X$, $C \subset Y$, $C = X \cap Y$, $X \setminus Y \neq \varnothing$ and $Y \setminus X \neq \varnothing$. In (1), $C = C'$ by the hypothesis. In (2), we assume that $C \neq C'$. By (1) of Lemma 3.3 and (3.9), $X' = (\bigcup\, S : S \in X/\textcircled{T} : S')$ and $Y' = (\bigcup\, S : S \in Y/\textcircled{T} : S')$. Since $C \subset X$ and $C \subset Y$, we obtain $C \in X/\textcircled{T}$ and $C \in Y/\textcircled{T}$. We consider two subcases: (a) $C' \setminus C \neq \varnothing$, and (b) $C' \subset C$. In (a), C' cannot be contained in both X and Y, since $X \setminus Y \neq \varnothing$ and $Y \setminus X \neq \varnothing$. Hence, either $X \neq X'$ or $Y \neq Y'$. Let us consider (b). $C' \subset C$ implies $C \setminus C' \neq \varnothing$. In order for $X = X'$ to hold, there must be some $S \neq C$ in $X/\textcircled{T}$ such that S' contains some elements in $C \setminus C'$. This implies that $S' \setminus S \neq \varnothing$. S is a $\textcircled{T}$-class. Regarding S as the previous C, we are back to (a) via (2). $\square$

Next, we define predicates $\mathcal{P}(C)$ for all $\textcircled{T}$-classes C in $\Phi/\textcircled{T}$. C is the union of the convex polytopes in partition $\text{clapar}(C)$. Hence, $\mathcal{P}(C)$ is defined in terms of predicates $\mathcal{P}(P)$ for every $\textcircled{P}$-block $P \in \text{clapar}(C)$:

$$\mathcal{P}(C) \;=\; (\exists\, P : P \in \text{clapar}(C) : \mathcal{P}(P)) \tag{3.10}$$

Lemma 3.5 $(\forall\, C : C \in \Phi/\textcircled{T} : C = C') \iff (\forall\, C : C \in \Phi/\textcircled{T} : C = (\bigcup\, P : P \in \text{clapar}(C) : P'))$.

Proof. Lemma 3.3 and (3.10). □

Recall that clapar(C) is a partition of Ⓣ-class C. The necessary and sufficient condition for $C = C'$, as in Lemma 3.5, does not require that $\{P' \mid P \in \text{clapar}(C)\}$ is a partition of C; the elements in $\{P' \mid P \in \text{clapar}(C)\}$ need not be disjoint and the sets P and P' can differ. However, our specification of predicate $\mathcal{P}(P)$ below enforces the equality of P and P'.

Finally, we define the predicates $\mathcal{P}(P)$ for all the Ⓟ-blocks P in clapar(C). All these blocks are convex polytopes. We define half($\mathfrak{S}, P$) to be any subset of $\mathfrak{S} \cup \mathfrak{S}^c$ that is a set of defining half-spaces of P. If no such set exists, half($\mathfrak{S}, P$) $= \varnothing$. Predicate $\mathcal{P}(P)$ is given by

$$\mathcal{P}(P) \;=\; \left(\forall\, S : S \in \text{half}(\mathfrak{S}, P) : \mathsf{C}_S(I) = \left\{ \begin{array}{ll} \textbf{if } S \in \mathfrak{S} & \rightarrow 0 \\ [\!]\; S \in \mathfrak{S}^c & \rightarrow 1 \\ \textbf{fi} & \end{array} \right. \right) \tag{3.11}$$

The next lemma establishes the containment of a set of defining half-spaces of P in $\mathfrak{S} \cup \mathfrak{S}^c$ as a necessary and sufficient condition for the equality of P and P'.

Lemma 3.6 *Let $P \subseteq \Phi$ be a convex polytope. Then* half($\mathfrak{S}, P$) $\neq \varnothing \iff P = P'$.

Proof.

$$\begin{array}{ll} & P \\ \iff & \{\text{half}(\mathfrak{S}, P) \text{ is a set of defining half-spaces of } P;\ \text{Def. 3.2}\} \\ & (\bigcap\ S : S \in \text{half}(\mathfrak{S}, P) : S) \\ \iff & \{\mathcal{P}(P) \text{ of (3.11); the CCUREs of (3.8)}\} \\ & \{I \mid I \in \Psi \wedge \mathcal{P}(P)\} \\ \iff & \{\text{Def. 3.4}\} \\ & P' \end{array}$$

□

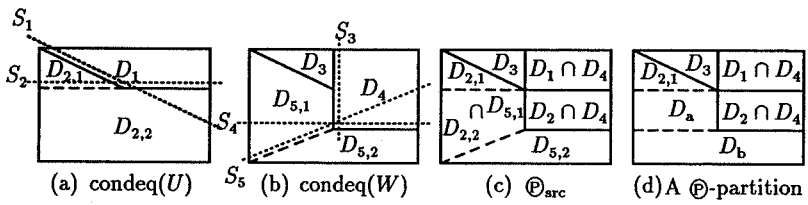
We have completed the definition of predicates $\mathcal{P}(D)$ for every $D \in \text{deq}(\Phi)$. Let us have a look at the situation when these definitions can be simplified. If S in deq(Φ) (Φ/Ⓣ) is a convex polytope, predicate $\mathcal{P}(S)$ defined by (3.9) (by (3.10)) can be defined directly by (3.11).

Lemma 3.7 *Let S in* deq(Φ) *or* Φ/Ⓣ *be a convex polytope. If $\mathfrak{S}$ is a separation set, then* half($\mathfrak{S}, S$) $\neq \varnothing$.

Proof. If $S \in \mathrm{deq}(\Phi)$, then $S = (\bigcup\ X : X \in S/\textcircled{T} : (\bigcup\ Y : Y \in \mathrm{clapar}(X) : Y))$ by Lemma 3.1 and the definition of a Ⓟ-partition. If $\mathfrak{S}$ is a separation set, $\mathrm{half}(\mathfrak{S}, P) \neq \varnothing$ when P is a Ⓟ-block. Thus, $(\bigcup\ X : X \in S/\textcircled{T} : (\bigcup\ Y : Y \in \mathrm{clapar}(X) : \mathrm{half}(\mathfrak{S}, Y)))$ contains a set of defining half-spaces of S. The proof is similar when $S \in \Phi/\textcircled{T}$. □

By definition, the blocks in $\mathrm{dompar}(D)$ for all non-convex domains $D \in \mathrm{deq}(\Phi)$, i.e., in $\mathrm{condeq}(\Phi) \setminus \mathrm{deq}(\Phi)$, are convex polytopes. However, Lemma 3.7 may not hold for every one of them, as the following example demonstrates. This example also provides insight to the formulation of the sufficient condition for Lemma 3.7 to hold for all these blocks.

Example 3.2 Figs. 3–2(a) and (b) depict the source UREs as specified in Ex. 3.1 with $\mathrm{dompar}(D_2) = \{D_{2,1}, D_{2,2}\}$ and $\mathrm{dompar}(D_5) = \{D_{5,1}, D_{5,2}\}$.



(a) $\mathrm{condeq}(U)$ (b) $\mathrm{condeq}(W)$ (c) $\textcircled{P}_{\mathrm{src}}$ (d) A Ⓟ-partition

Figure 3–2: Domain predicate replacement. $\mathrm{deq}(\Phi) = \{D_1, D_2, D_3, D_4, D_5\}$. $\mathrm{condeq}(\Phi) = \{D_1, D_{2,1}, D_{2,2}, D_3, D_4, D_{5,1}, D_{5,2}\}$. S_1, S_2, S_3, S_4 and S_5 are half-spaces whose corresponding hyperplanes are depicted by dotted lines. S_1 and S_2 support D_1, S_3 and S_4 support D_4, and S_5 supports $D_{5,1}$.

Fig. 3–2(c) depicts the Ⓟ-partition $\textcircled{P}_{\mathrm{src}}$. For simplicity, we shall not include the supporting half-spaces of the index space in a separation set. $\mathfrak{S} = \{S_1, S_2, S_3, S_4\}$ is a separation set of the Ⓟ-partition depicted in Fig. 3–2(d), since

$$\begin{aligned}
\mathrm{half}(\mathfrak{S}, D_{2,1}) &= \{\complement_{\mathbb{Z}^n} S_1, S_2\} \\
\mathrm{half}(\mathfrak{S}, D_3) &= \{S_1, \complement_{\mathbb{Z}^n} S_3\} \\
\mathrm{half}(\mathfrak{S}, D_1 \cap D_4) &= \{S_2, S_3\} \\
\mathrm{half}(\mathfrak{S}, D_2 \cap D_4) &= \{\complement_{\mathbb{Z}^n} S_2, S_3, S_4\} \\
\mathrm{half}(\mathfrak{S}, D_a) &= \{\complement_{\mathbb{Z}^n} S_2, \complement_{\mathbb{Z}^n} S_3, S_4\} \\
\mathrm{half}(\mathfrak{S}, D_b S) &= \{\complement_{\mathbb{Z}^n} S_4\}
\end{aligned} \tag{3.12}$$

Lemma 3.7 holds for convex domains D_1, D_3 and D_4 in deq(Φ), since

$$\begin{aligned} \text{half}(\mathfrak{S}, D_1) &= \{S_1, S_2\} \\ \text{half}(\mathfrak{S}, D_3) &= \{S_1, \mathfrak{C}_{\mathbf{Z}^n} S_3\} \\ \text{half}(\mathfrak{S}, D_4) &= \{S_3, S_4\} \end{aligned} \tag{3.13}$$

D_2 and D_5 are non-convex domains in deq(Φ). condeq(Φ)\deq(Φ)$=\{D_{2,1}, D_{2,2}, D_{5,1}, D_{5,2}\}$. Lemma 3.7 holds for $D_{2,1}$ and $D_{2,2}$, since

$$\begin{aligned} \text{half}(\mathfrak{S}, D_{2,1}) &= \{\mathfrak{C}_{\mathbf{Z}^n} S_1, S_2\} \\ \text{half}(\mathfrak{S}, D_{2,2}) &= \{\mathfrak{C}_{\mathbf{Z}^n} S_2\} \end{aligned} \tag{3.14}$$

But it does not hold for $D_{5,1}$ and $D_{5,2}$ since half($\mathfrak{S}, D_{5,1}$)$=\varnothing$ and half($\mathfrak{S}, D_{5,2}$)$=\varnothing$. Thus, we cannot define predicates $\mathcal{P}(D_{5,1})$ and $\mathcal{P}(D_{5,2})$ by (3.11).

The set $\mathfrak{S}$ is a not a separation set of $\mathbb{P}_{\text{src}}$, since half($\mathfrak{S}, D_{2,2} \cap D_{5,1}$) $= \varnothing$ and half($\mathfrak{S}, D_{5,2}$) $= \varnothing$. Adding S_5 to it gives rise to a separation set of $\mathbb{P}_{\text{src}}$: $\mathfrak{S}=\{S_1, S_2, S_3, S_4, S_5\}$, since, in addition to the first four equations of (3.12), we have

$$\begin{aligned} \text{half}(\mathfrak{S}, D_{2,2} \cap D_{5,1}) &= \{\mathfrak{C}_{\mathbf{Z}^n} S_2, \mathfrak{C}_{\mathbf{Z}^n} S_3, S_5\} \\ \text{half}(\mathfrak{S}, D_{5,2}) &= \{\mathfrak{C}_{\mathbf{Z}^n} S_4, \mathfrak{C}_{\mathbf{Z}^n} S_5\} \end{aligned}$$

For the new separation set, Lemma 3.7 is valid for every element in condeq(Φ), since, in addition to (3.13) and (3.14), we have

$$\begin{aligned} \text{half}(\mathfrak{S}, D_{5,1}) &= \{\mathfrak{C}_{\mathbf{Z}^n} S_1, \mathfrak{C}_{\mathbf{Z}^n} S_3, S_5\} \\ \text{half}(\mathfrak{S}, D_{5,2}) &= \{\mathfrak{C}_{\mathbf{Z}^n} S_4, \mathfrak{C}_{\mathbf{Z}^n} S_5\} \end{aligned}$$

Lemma 3.8 *If $\mathfrak{S}$ is a separation set of $\mathbb{P}_{\text{src}}$,* half($\mathfrak{S}, S$) $\neq \varnothing$ *for every* $S \in$ condeq(Φ).

Proof. By (3.7), every $S \in$ condeq(Φ) is a union of $\mathbb{P}_{\text{src}}$-blocks. The rest of proof proceeds exactly as for Lemma 3.7. □

This lemma allows us to define predicates $\mathcal{P}(D)$ for every $D \in$ condeq(Φ) by (3.11). So, predicates $\mathcal{P}(D)$, which replace domain predicates $I \in D$ for every $D \in$ deq(Φ), is given by

$$\mathcal{P}(D) = (\exists\, S : S \in \text{dompar}(D) : \mathcal{P}(S)) \tag{3.15}$$

Lemma 3.8 is the basis for the procedure, presented in Sect. 3.3.1.2, for the synthesis of computation control from the source UREs.

Replacing domain predicates $I \in D_i$ in (3.2) by the new predicates $\mathcal{P}(D_i)$ yields:

$$V(I) \;=\; \{\, I \in \Psi \;\twoheadrightarrow\; \begin{cases} \text{if } I \in \Phi \wedge \mathcal{P}(D_1) & \rightarrow \quad f_1(W_1(I-\vartheta_1),\ldots) \\ [\!]\; I \in \Phi \wedge \mathcal{P}(D_2) & \rightarrow \quad f_2(W_2(I-\vartheta_2),\ldots) \\ & \vdots \\ [\!]\; I \in \Phi \wedge \mathcal{P}(D_p) & \rightarrow \quad f_p(W_p(I-\vartheta_p),\ldots) \\ [\!]\; I \in \complement_{\Psi}\Phi & \rightarrow \quad V(I-\vartheta) \\ \text{fi} \end{cases} \tag{3.16}$$

The new predicates $\mathcal{P}(D_i)$ are part of the guards in this equation. There are as many guards as there are domain predicates in the original equation. Note that the resulting extended source UREs after domain predicate replacement and the CCUREs have the same domain predicate $I \in \Psi$. They are unconditional. Therefore, deriving control signals for the source UREs amounts to transforming the source UREs to a system of unconditional UREs such that the input and output data prescribed by the new system are mapped to border cells.

Theorem 3.2 *If $\mathfrak{S}$ is a separation set of some $\textcircled{P}$-partition, then the computation control flow is correct.*

Proof. We consider only predicates $\mathcal{P}(D)$ for $D \in \mathrm{deq}(\Phi)$ that are defined by (3.9). The proof is similar if they are defined by (3.15).

$\mathfrak{S}$ is a separation set of some $\textcircled{P}$-partition

$\Longleftrightarrow$ {Def. 3.2; Def. 3.3}

$(\forall\, P : P \in \textcircled{P} : (\exists\, S : S \subseteq \mathfrak{S} \cup \mathfrak{S}^c \wedge (S \text{ is a set of defining half-spaces of } P)))$

$\Longleftrightarrow$ {Choose half$(\mathfrak{S}, P)$ as S; P is convex and half$(\mathfrak{S}, P) \neq \varnothing$; Lemma 3.6}

$(\forall\, P : P \in \textcircled{P} : P = P')$

$\Longrightarrow$ {$\Phi/\textcircled{T}$ of (3.3); $\mathcal{P}(C)$ for $C \in \Phi/\textcircled{T}$ by (3.10) ; Def. 3.4; Lemma 3.5}

$(\forall\, C : C \in \Phi/\textcircled{T} : C = C')$

$\Longleftrightarrow$ {$(\forall\, D : D \in \mathrm{deq}(\Phi) : D/\textcircled{T} \subseteq \Phi/\textcircled{T})$; Def. 3.4; Lemma 3.4}

$(\forall\, D : D \in \mathrm{deq}(\Phi) : D = D')$

$\Longleftrightarrow$ {Def. 3.5}

The computation control flow is correct $\square$

The definition of $\mathcal{P}(D_i)$ of (3.9) amounts to redefining the domain D_i of the domain predicate $I \in D_i$ to:

$$(\bigcup X : X \in D_i/\textcircled{T} : (\bigcup Y : Y \in \text{clapar}(X) : (\bigcap H : H \in \text{half}(\mathfrak{S}, Y) : H)))$$

A search for different separation sets is a search for different $\textcircled{P}$-partitions, which amounts to searching for source UREs with different specifications of domain predicates by the virtue of Lemma 3.2. Different separation sets may lead to computation control of varying quality. Take Ex. 3.2 for example. The second separation set is inferior to the first, since it contains one more element (i.e., S_5) and thus results in one more control variable in the CCUREs.

This thesis does not provide systematic solutions to finding the best control UREs. We shall at the end of Sect. 3.3.1.2 describe some heuristics that guide the user in the optimisation of domain predicates. The user is responsible for the choice of domain predicates that yields efficient systolic arrays. Systolic design systems [2,53,61,45,83] generally rely on the user in difficult decision-making and performance optimisation. In the synthesis of data flow, optimisations of **step** and **place** [51,77,87,88] are carried out with respect to the source UREs (rather than the initial problem specification). Also for their choice, the user is responsible.

In Sect. 3.3.1.2, we present a mechanisable procedure for the construction of a separation set of $\textcircled{P}_{\text{src}}$, denoted $\mathfrak{S}_{\text{src}}$, from the source UREs. The construction of $\mathfrak{S}_{\text{src}}$ completely depends on the domain predicates of the source UREs. So does the subsequent domain predicate replacement.

3.3.1.2 Specifying Computation Control Flow from the Source UREs

This section describes a procedure for the synthesis of computation control from the source UREs. We assume that the domain predicates of the source UREs are in normal form (Sect. 2.2), i.e., their constituent conditionals are in one of the following forms:

$$\pi I = \delta,\ \pi I \neq \delta,\ \pi I < \delta,\ \pi I \leqslant \delta,\ \pi I > \delta,\ \pi I \geqslant \delta \tag{3.17}$$

For simplicity and without loss of generality, we further assume that all conditionals are neither equalities nor inequalities. If not, we can always replace every equality $\pi I = \delta$ by $\pi I \leqslant \delta \wedge \pi I \geqslant \delta$, and every inequality $\pi I \neq \delta$ by $\pi I < \delta \vee \pi I > \delta$. App. 9 discusses the normalisation of domain predicates.

Each conditional, $\pi \boxdot \delta$, in one of the rightmost four forms depicted in (3.17) corresponds to half-space $[\pi \boxdot \delta]$. The idea underlying the construction of separation set $\mathfrak{S}_{\text{src}}$ is to just to collect the half-spaces corresponding the conditionals in the domain predicates. If two half-spaces are complements of each other, it is only necessary to put one of them in $\mathfrak{S}_{\text{src}}$ ($\mathfrak{S}^{c}_{\text{src}}$ will contain the other). Once $\mathfrak{S}_{\text{src}}$ is obtained, the CCUREs follow from (3.8) with $\mathfrak{S} = \mathfrak{S}_{\text{src}}$. The replacement of the domain predicates of the source UREs relies on (3.15). That is, we define predicate $\mathcal{P}(D)$ for every $D \in \text{condeq}(\Phi)$ by finding a set of defining half-spaces of D from $\mathfrak{S}_{\text{src}} \cup \mathfrak{S}^{c}_{\text{src}}$. For every conditional $\pi \boxdot \delta$ in the domain predicates, the corresponding half-space $[\pi \boxdot \delta]$ is contained in $\mathfrak{S}_{\text{src}} \cup \mathfrak{S}^{c}_{\text{src}}$. By the specification of the CCUREs, any two complement half-spaces in $\mathfrak{S}_{\text{src}} \cup \mathfrak{S}^{c}_{\text{src}}$ are associated with a unique control variable; the variable takes on the value 0 at the points in the half-space in $\mathfrak{S}_{\text{src}}$ and the value 1 at the points in the half-space in $\mathfrak{S}^{c}_{\text{src}}$. We replace conditional $\pi \boxdot \delta$ by predicate $C_{[\pi \boxdot \delta]}(I) = 0$ if $[\pi \boxdot \delta]$ is in $\mathfrak{S}_{\text{src}}$ and by predicate $C_{[\pi \boxdot \delta]}(I) = 1$ if $[\pi \boxdot \delta]$ is in $\mathfrak{S}^{c}_{\text{src}}$. This amounts to defining $\text{half}(\mathfrak{S}_{\text{src}}, D)$ for $D \in \text{condeq}(\Phi)$ as the set of half-spaces that correspond to the constituent conditionals of $I \in D$. Trivially, $\text{half}(\mathfrak{S}_{\text{src}}, D)$ is a set of half-spaces of C.

Procedure 3.1 (Construction of computation control flow from the source UREs)

INPUT: The source UREs.

OUTPUT: The specification of computation control flow.

1. $\mathfrak{S}_{\text{src}} := \varnothing$.

2. Replace every conditional $\pi I < \delta$, $\pi I \leqslant \delta$, $\pi I > \delta$, or $\pi I \geqslant \delta$ in the domain predicates by $-\pi I > -\delta$, $-\pi I \geqslant -\delta$, $-\pi I < -\delta$, or $-\pi I \leqslant -\delta$ if the first non-zero component of π is not positive. (For fixed π and δ, this prevents simultaneous occurrences of $\pi I \leqslant \delta$ and $-\pi I \geqslant -\delta$ or $\pi I < \delta$ and $-\pi I > -\delta$.)

3. For every conditional $\pi \boxdot \delta$ in the domain predicates:

$$\mathfrak{S}_{\text{src}} := \begin{cases} \textbf{if } \boxdot = < \;\rightarrow\; \mathfrak{S}_{\text{src}} + \{I \mid \pi I < \delta\} \\ [] \;\boxdot = \leqslant \;\rightarrow\; \mathfrak{S}_{\text{src}} + \{I \mid \pi I \leqslant \delta\} \\ [] \;\boxdot = > \;\rightarrow\; \mathfrak{S}_{\text{src}} + \{I \mid \pi I \leqslant \delta\} \\ [] \;\boxdot = \geqslant \;\rightarrow\; \mathfrak{S}_{\text{src}} + \{I \mid \pi I < \delta\} \\ \textbf{fi} \end{cases}$$

4. The CCUREs are given by (3.8) with $\mathfrak{S} = \mathfrak{S}_{\text{src}}$.

5. Replace the domain predicates of the source UREs by predicates in computation control variables. A conditional $\pi \boxdot \delta$ is replaced by $C_{[\pi \boxdot \delta]}(I) = 0$ if $[\pi \boxdot \delta] \in \mathfrak{S}_{\text{src}}$ and by $C_{[\pi \boxdot \delta]}(I) = 1$ otherwise. □

The choice of $\mathfrak{S}_{\text{src}}$ is unique because the first two steps of Proc. 3.1 are deterministic. So is the specification of computation control flow.

Lemma 3.9 *$\mathfrak{S}_{\text{src}}$ is a separation set of $\mathfrak{P}$-partition $\mathfrak{P}_{\text{src}}$.*

Proof. We need to show that $\mathfrak{S}_{\text{src}} \cup \mathfrak{S}^c_{\text{src}}$ contains a set of defining half-spaces for every $S \in \mathfrak{P}_{\text{src}}$. By construction, $\mathfrak{S}_{\text{src}} \cup \mathfrak{S}^c_{\text{src}}$ contains $[\pi \boxdot \delta]$ for every conditional $\pi I \boxdot \delta$ in the domain predicates. Choose half$(\mathfrak{S}_{\text{src}}, S)$ as the half-spaces that are in $\mathfrak{S}_{\text{src}} \cup \mathfrak{S}^c_{\text{src}}$ and that correspond to the constituent conditionals of $I \in S$. □

Theorem 3.3 *The computation control flow specified by Proc. 3.1 is correct.*

Proof. Lemma 3.6, Lemma 3.8 and Thm. 3.2. □

Each conditional of the domain predicates corresponds to a distinct control variable. The more conditionals there are in the domain predicates, the more control variables there are in the CCUREs. We describe some heuristics that guide the user in the minimisation of the number of conditionals when he/she is searching for URE specifications from the initial problem specification.

Procedure 3.2 (Construction of the domain predicates of the source UREs)

INPUT: The set deq(Φ) of the domains of all equations of the source UREs.

OUTPUT: The specification of the domain predicates of the source UREs.

1. $H := \varnothing$.

2. For every convex domain D in deq(Φ), choose a set of defining half-spaces of D from H whenever possible, and add to H the new half-spaces and their complements.

3. For every non-convex domain D in deq(Φ), use the half-spaces in H or their translates to form a convex domain partition of D whenever possible, and add to H the new half-spaces and their complements. □

The motivation for using translates of the half-spaces in convex domain partitions is that we can always combine the control variables whose associated hyperlanes are parallel to one control variable (Sect. 3.4).

Let us use this procedure to obtain convex domain partitions for non-convex domains D_2 and D_5 in Ex. 3.2. Assume that $H = \{S_1, S_2, S_3, S_4\}$ when Step 2 of Proc. 3.2 completes. The choice of dompar$(D_2) = \{D_{2,1}, D_{2,2}\}$ in Ex. 3.2 is permitted by the procedure. It amounts to partitioning D_2 to $\{\Phi \cap \complement_{\mathbb{Z}^n} S_1 \cap S_2, \Phi \cap \complement_{\mathbb{Z}^n} S_2\}$. But, the choice of dompar$(D_5) = \{D_{5,1}, D_{5,2}\}$ is not permitted. This is because we can use the half-spaces in H to obtain a convex domain partition for D_5. One possible solution is dompar$(D_5) = \{D_{2,1} \cup D_a, D_b\} = \{\Phi \cap \complement_{\mathbb{Z}^n} S_1 \cap \complement_{\mathbb{Z}^n} S_3 \cap S_4, \Phi \cap \complement_{\mathbb{Z}^n} S_4\}$. We thus get rid of the conditional corresponding to S_5.

3.3.2 The Propagation Control Flow

The construction of the PCUREs proceeds along similar lines. We redefine the partition $\{\Phi, \complement_\Psi \Phi\}$ of the extended index space Ψ. To do so, we must find a set of defining half-spaces of Φ: $\mathfrak{B} = \{\sup(\Phi, F) \mid F \in \text{facets}(\Phi)\}$. $\mathfrak{B}$ plays the same rôle as $\mathfrak{S}$ previously. We associate a distinct propagation control variable, P_B, with every $B = [\pi_B \boxdot \delta_B]$ in $\mathfrak{B}$, where $\pi_B \in \mathbb{Z}^n$ is normalised and $\delta_B \in \mathbb{Z}$. (As in Sect. 3.3.1, we identify the corresponding half-space in $\mathfrak{B}$ of a propagation control variable P by H_P.) The associated control dependence vector ϑ_{P_B} of B is a solution of $\vartheta_{\mathsf{P}_B} \pi_B = 0$. Each propagation control variable takes on two different values: 0 in the half-space B and 1 in the half-space $\complement_{\mathbb{Z}^n} B$. That is, sig$(\mathsf{P}_B) = \{0, 1\}$.

The PCUREs are defined as follows:

$$\left(\forall\, B : B \in \mathfrak{B} : \left\{ \begin{array}{l} \vartheta_{\mathsf{P}_B} \pi_B = 0 \\ \mathsf{P}_B(I) = \left\{ \begin{array}{ll} I \in \text{in}(\Psi, \vartheta_{\mathsf{P}_B}) \cap B & \twoheadrightarrow 0 \\ I \in \text{in}(\Psi, \vartheta_{\mathsf{P}_B}) \cap \complement_{\mathbb{Z}^n} B & \twoheadrightarrow 1 \\ I \in \Psi & \twoheadrightarrow \mathsf{P}_B(I - \vartheta_{\mathsf{P}_B}) \end{array} \right. \end{array} \right. \right) \tag{3.18}$$

Predicates $\mathcal{P}(\Phi)$ and $\mathcal{P}(\complement_\Psi \Phi)$ replace predicates $I \in \Phi$ and $I \in \complement_\Psi \Phi$ of (3.16), respectively:

$$\begin{aligned} \mathcal{P}(\Phi) &= (\forall\, B : B \in \mathfrak{B} : \mathsf{P}_B(I) = 0) \\ \mathcal{P}(\complement_\Psi \Phi) &= (\exists\, B : B \in \mathfrak{B} : \mathsf{P}_B(I) = 1) \end{aligned} \tag{3.19}$$

$\{\Phi', \mathsf{C}_\Psi\Phi'\}$ is the partition of the extended index space defined by the PCUREs:

$$\begin{aligned} \Phi' &= \{I \mid I \in \Psi \wedge \mathcal{P}(\Phi)\} \\ \mathsf{C}_\Psi\Phi' &= \{I \mid I \in \Psi \wedge \mathcal{P}(\mathsf{C}_\Psi\Phi)\} \end{aligned} \tag{3.20}$$

Replacing the domain predicates $I \in \Phi$ and $I \in \mathsf{C}_\Psi\Phi$ in the extended source UREs by predicates $\mathcal{P}(\Phi)$ and $\mathcal{P}(\mathsf{C}_\Psi\Phi)$ changes (3.16) to:

$$V(I) = \{\ I \in \Psi \ \twoheadrightarrow \begin{cases} \textbf{if } \mathcal{P}(\Phi) \wedge \mathcal{P}(D_1) & \rightarrow \ f_1(W_1(I-\vartheta_1), \ldots) \\ [\!]\ \mathcal{P}(\Phi) \wedge \mathcal{P}(D_2) & \rightarrow \ f_2(W_2(I-\vartheta_2), \ldots) \\ & \vdots \\ [\!]\ \mathcal{P}(\Phi) \wedge \mathcal{P}(D_p) & \rightarrow \ f_p(W_p(I-\vartheta_p), \ldots) \\ [\!]\ \mathcal{P}(\mathsf{C}_\Psi\Phi) & \rightarrow \ V(I-\vartheta) \\ \textbf{fi} \end{cases} \tag{3.21}$$

Definition 3.6 The propagation control flow is *correct* if $\Phi = \Phi'$ and $\mathsf{C}_\Psi\Phi = \mathsf{C}_\Psi\Phi'$.

Theorem 3.4 *The propagation control flow as defined by (3.18), (3.19) and (3.21) is correct iff $\mathfrak{B}$ is a set of defining half-spaces of Φ.*

Proof. Similar to the proof of Thm. 3.2. □

Definition 3.7 The control flow (the specification of the pipelined UREs) is *correct* if the computation control flow and the propagation control flow are correct.

The PCUREs serve to propagate data from the internal cells to (from) the border cells. The half-space $\sup(\Phi, F)$ in $\mathfrak{B}$ is superfluous if all data variables at the pipelining points of $\mathsf{C}_{\mathbf{Z}^n}\sup(\Phi, F)$ are undefined. That is, control signals are unnecessary if no input/output data are pipelined across domain $\Psi \cap \mathsf{C}_{\mathbf{Z}^n}\sup(\Phi, F)$.

The fact that valid space-time mappings are bijections from Ψ to $\overline{\Psi}$ gives rise to the following lemma, which allows us to regard CCUREs also as PCUREs. The image of X' under the space-time mapping is given by

$$\overline{X'} = \{\overline{I} \mid \overline{I} \in \overline{\Psi} \wedge \overline{\mathcal{P}(X)}\}$$

$\overline{\mathcal{P}(X)}$ denotes the image of predicate $\mathcal{P}(X)$; it is the predicate with all the occurrence of index vectors I appearing in $\mathcal{P}(X)$ substituted by $\overline{I}$.

Lemma 3.10 $X = X' \implies \overline{X} = \overline{X'}$.

Proof. The bijectivity of valid space-time mappings from $\mathbf{Q}^n$ to $\mathbf{Q}^n$. □

This lemma allows us to redefine $\mathcal{P}(\Phi)$ and $\mathcal{P}(\mathfrak{C}_\Psi\Phi)$ of (3.19) using computation control variables only.

$$\begin{aligned} \mathcal{P}(\Phi) &= \textsf{true} \\ \mathcal{P}(\mathfrak{C}_\Psi\Phi) &= (\forall\, C : C \in \Phi/\textcircled{T} : \neg\mathcal{P}(C)) \end{aligned} \tag{3.22}$$

Lemma 3.11 *The propagation control flow as defined by (3.18), (3.21) and (3.22) is correct iff* $(\forall\, C : C \in \Phi/\textcircled{T} : \overline{C} = \overline{C'})$.

Proof. To prove necessity, we assume that $\overline{C} \neq \overline{C'}$ for some Ⓣ-class C. By Lemma 3.4, there is some $D \in \mathrm{deq}(\Phi)$ such that $\overline{D} \neq \overline{D'}$. By the facts that $D = D'$ and that a valid space-time mapping is bijective over Φ, we conclude that $\overline{D} \subset \overline{D'}$. So, some pipelining point is treated as a computation point. Let us prove sufficiency. Since $(\forall\, C : C \in \Phi/\textcircled{T} : \overline{C} = \overline{C'})$, we have $(\forall\, D : D \in \mathrm{deq}(\Phi) : \overline{D} = \overline{D'})$ by Lemma 3.4. Thus. $\overline{\mathcal{P}(D)} \Longrightarrow \overline{\mathcal{P}(\Phi)}$ and $\overline{\mathcal{P}(\mathfrak{C}_\Psi\Phi)} \iff (\forall\, C : C \in \Phi/\textcircled{T} : \neg\overline{\mathcal{P}(C)})$. Hence, predicates $\mathcal{P}(\Phi)$ and $\mathcal{P}(\mathfrak{C}_\Psi\Phi)$ of (3.19) can be defined equivalently as in (3.22). The correctness of propagation control follows from Thm. 3.4. □

Theorem 3.5 *The propagation control flow as defined by (3.18), (3.21) and (3.22) is correct.*

Proof. Lemmata 3.10 and 3.11. □

In the synthesis of systolic arrays of reduced dimension, a valid space-time mapping can be non-bijective from Ψ to $\overline{\Psi}$ (Thm. 4.3) – it is bijective from Φ to $\overline{\Phi}$, though. In Chap. 5, we shall see that Lemma 3.10 and, consequently, Thm. 3.10 are not valid for one-dimensional arrays.

If the propagation control flow is specified by (3.18), (3.19) and (3.21), we can disregard the half-spaces of $\mathfrak{S}$ that support the index space. That is, we just need to find a set $\mathfrak{S}$ such that (1) $\mathfrak{S} \cup \mathfrak{S}^c$ contains a set of half-spaces for every Ⓟ-block and (2) the intersection of these half-spaces contains that block only. This is due to the fact that Ψ and $\mathfrak{C}_\Psi\Phi$ can be identified by propagation control.

3.4 The Optimisation of Control Flow

This section describes three optimisation techniques for the purpose of eliminating redundant control hardware in the array. The first two are independent of the space-time mapping, while the third is dependent on the projection vector.

The first optimisation is to merge control variables in order to reduce the number of control signals. Assume that $\mathfrak{S} \cup \mathfrak{B}$ contains p half-spaces $S_1, S_2, \cdots, S_p$ whose corresponding hyperplanes are parallel. Instead of introducing p control variables, one per hyperplane, we introduce only one control variable, C, for the p hyperplanes. The control dependence vector ϑ_C is perpendicular to the normal of these hyperplanes. The p hyperplanes partition $\text{in}(\Psi, \vartheta_C)$ into $p+1$ subsets; each is initialised with a distinct control value, which is pipelined along the pipelining vector ϑ_C across the extended index space Ψ. Let $\text{sig}(C) = \{c_1, c_2, \cdots, c_{p+1}\}$ and $S_1 \subset S_2 \subset \cdots \subset S_p$. The specification of C is as follows:

$$C(I) = \begin{cases} I \in \text{in}(\Psi, \vartheta_C) \cap S_1 & \twoheadrightarrow \quad c_1 \\ I \in \text{in}(\Psi, \vartheta_C) \cap \complement_{\mathbb{Z}^n} S_1 \cap S_2 & \twoheadrightarrow \quad c_2 \\ & \vdots \\ I \in \text{in}(\Psi, \vartheta_C) \cap \complement_{\mathbb{Z}^n} S_p & \twoheadrightarrow \quad c_{p+1} \\ I \in \Psi & \twoheadrightarrow \quad C(I - \vartheta_C) \end{cases} \tag{3.23}$$

This optimisation reduces the number of control signals from $2p$ for the p variables to $p+1$ for the variable that replaces them.

The second optimisation is to minimise the domain of each control variable. The values of a control variable C at some points may not appear in the guards of the pipelined UREs. Thus, C need not be defined at these points. Let D_C be the set of all points I such that $C(I)$ appears in the guards of the pipelined UREs. D_C is the union of all domains of equations $D_1, D_2, \cdots, D_d$ such that every predicate $\mathcal{P}(D_i)$ refers to $C(I)$ (see (3.9)). Thus, we can substitute $\text{rays}(\Phi, \pm u) \cap \text{rays}(D_C, \pm\vartheta_C)$ for Ψ in the defining equation of C. The domain of C is given by

$$\Psi_C = \text{rays}(\Phi, \pm u) \cap \text{rays}(D_C, \pm\vartheta_C) \tag{3.24}$$

The CCUREs serve to distinguish different types of computation points. If C is a computation control variable, the $D_1, D_2, \cdots, D_d$ are domains of computation

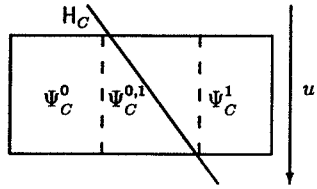


Figure 3–3: The optimisation of pipelined UREs ($n = 2$). The solid rectangle depicts the extended index space. $\mathsf{H}_C \in \mathfrak{S}$ is a half-space that supports Ψ_0.

equations. Hence, $\text{rays}(D_C, \pm\vartheta_C) \subseteq \text{rays}(\Phi, \pm\vartheta_C)$. The PCUREs serve to distinguish pipelining points from computation points. If C is a propagation control variable, the $D_1, D_2, \cdots, D_d$ become $\complement_\Psi \Phi$ and Φ. By (3.19), predicate $\mathcal{P}(\Phi)$ refers to $C(I)$ for the points I of the index space Φ and predicate $\mathcal{P}(\complement_\Psi \Phi)$ refers to $C(I)$ for the points I of $\complement_\Psi \Phi$ that are contained in the half-space $\complement_{\mathbb{Z}^n} \mathsf{H}_C$. Hence, $\text{rays}(D_C, \pm\vartheta_C) = \text{rays}(\Psi \cap \complement_{\mathbb{Z}^n} \mathsf{H}_C, \pm\vartheta_C) \cup \text{rays}(\Phi, \pm\vartheta_C)$.

Remark. To apply Proc. 3.1, we must replace $\pi I = \delta$ by $\pi I \leqslant \delta \wedge \pi I \geqslant \delta$. This implies that the corresponding control variable must be defined at the two half-spaces. If the control variable is only referenced at the points of the hyperplane $[\pi : \delta]$, the domain of the control variable can be reduced to the intersection of the hyperlane and the index space. □

The third optimisation can be applied after the projection vector u has been chosen (Fig. 3–3). It removes conditionals in the space-time UREs that become redundant due to the chosen projection vector. For control variable C, we can partition the extended index space into three blocks:

$$\begin{aligned} \Psi_C^0 &= \{I \mid \text{ray}(I, \pm u) \subseteq \mathsf{H}_C \wedge \text{ray}(I, \pm u) \not\subseteq \complement_{\mathbb{Z}^n} \mathsf{H}_C\} \\ \Psi_C^1 &= \{I \mid \text{ray}(I, \pm u) \not\subseteq \mathsf{H}_C \wedge \text{ray}(I, \pm u) \subseteq \complement_{\mathbb{Z}^n} \mathsf{H}_C\} \\ \Psi_C^{0,1} &= \Psi \backslash (\Psi_C^0 \cup \Psi_C^1) \end{aligned} \tag{3.25}$$

Imagine that we draw lines parallel to the projection vector u in the extended index space. The three sets defined in (3.25) are the union of lines that only contain points in H_C, the union of lines that only contain points in $\complement_{\mathbb{Z}^n} \mathsf{H}_C$ and the union of the remaining lines. Thus, we can substitute **true** for predicate $C(I) = 0$ at Ψ_C^0 and for predicate $C(I) = 1$ at Ψ_C^1, but we must retain predicates $C(I) = 0$ and $C(I) = 1$ at $\Psi_C^{0,1}$.

If after this optimisation some guard of an equation with domain D is true, the processors at place(D) can be pre-designed to apply that equation. An extreme case occurs when the projection vector is orthogonal to the normal of the separating hyperplane associated with control variable C. In this case, $\Psi_C^{0,1} = \varnothing$. Then control variable C is said to be *redundant*. This is because $\Psi \cap \mathsf{H}_C$ and $\Psi \cap \mathsf{C}_{\mathbb{Z}^n} \mathsf{H}_C$ are projected to two disjoint sets of processors. They are already separated spatially and do not have to be separated temporally.

Theorem 3.6 *If all computation (propagation) control variables are redundant under the space-time mapping, then the CCUREs (PCUREs) are unnecessary.*

Proof. If all computation control variables are redundant, all Ⓣ-classes are projected to disjoint sets of processors. The CCUREs are unnecessary because a cell can be pre-designed to compute the computation points mapped to it. The PCUREs are unnecessary, if all propagation control variables are redundant. Then there are no pipelining points, i.e., $\Psi = \Phi$. □

A redundant control variable becomes stationary if the projection vector is chosen as its control dependence vector. Conversely, a stationary control variable is redundant, since the projection vector that is parallel to its control dependence vector is orthogonal to the normal of the separating hyperplane associated with the control variable. Unfortunately, the advantage of making control variables stationary does not carry over to data variables. Complications arise when some data variables become stationary. A local processor memory is required to store the stationary data allocated to the processor. The control UREs do not provide a specification for handling stationary variables. In Chap. 8, we shall present a systematic method for handling stationary variables. The basic idea is to rewrite the source UREs to encompass the specification for handling stationary variables.

3.5 The Extension of the Space-Time Mapping

The mapping conditions for the space-time mapping stated in Thm. 2.1 must be augmented such that the specification of the control UREs is taken into account. We write C for the matrix whose column vectors are the control dependence vectors. (Each control variable is associated with one control dependence vector.)

Theorem 3.7 *A space-time mapping is valid if*

- *The scheduling vector λ satisfies:*
  - $(\forall\, \vartheta : \vartheta \in \mathcal{D} : \lambda\vartheta > 0)$ (3.26)
  - $(\forall\, \vartheta : \vartheta \in \mathcal{C} : \lambda\vartheta > 0)$ (3.27)
- *The projection vector u satisfies $\lambda u \neq 0$.*

Proof. The pipelined UREs are UREs. Apply Thm. 2.1. □

Let C^* be the set of scheduling vectors satisfying (3.26), and C^{**} the set of scheduling vectors satisfying (3.26) and (3.27). Both C^* and C^{**} are convex cones. First, we show that, if there is a valid scheduling vector for the source UREs, there must be a valid scheduling vector for the pipelined UREs.

Theorem 3.8 $C^* \neq \varnothing \implies C^{**} \neq \varnothing$.

Proof. For every control dependence vector $\vartheta \in \mathcal{C}$, it makes no difference if we replace it by $-\vartheta$. This is because the control signals of the corresponding control variable can be pipelined along either of the two directions (see (3.8) and (3.18)). Let Θ^* be the convex cone generated by the data dependence vectors in $\mathcal{D}$. To prove that C^{**} is not empty, it suffices to prove that we can obtain a convex cone, Θ^{**} which contains Θ^* and either of the two vectors ϑ and $-\vartheta$ for every $\vartheta \in \mathcal{C}$. This is indeed the case, since either $\Theta^* \cup \{\vartheta\}$ or $\Theta^* \cup \{-\vartheta\}$ must be a convex cone. □

Thm. 3.8 indicates that all domain predicates can be pipelined. The proof of the theorem has the following implication. If we choose all control dependence vectors from Θ^*, then a space-time mapping that is valid for the source UREs must also be valid for the pipelined UREs. The advantage of doing so is that the latency of the systolic array can be retained when the PCUREs are not needed.

Next, we show that the control signals of a control variable need to be broadcast iff the scheduling vector is orthogonal to the normal of the separating hyperplane associated with the control variable. This assertion assumes that the control dependence vectors in $\mathcal{C}$ are not fixed but are variable vectors that satisfy the conditions in (3.8) and (3.18).

Theorem 3.9 *Let $S=[\pi_S \boxdot \delta_S]$ be a half-space in $\mathfrak{S} \cup \mathfrak{B}$. The control signals of control variable C_S need to be broadcast iff λ and π_S are linearly dependent (i.e., parallel).*

Proof. By (3.8) and (3.18), the control dependence vector ϑ_{C_S} of C_S is a solution of $\vartheta_{\mathsf{C}_S}\pi_S=0$. The control signals of C_S need to be broadcast iff $(\forall\ \vartheta : \vartheta\pi_S=0 : \lambda\vartheta=0)$. The rest of the proof follows from the fact that λ and π_S are linearly dependent iff $(\forall\ \vartheta : \vartheta\pi_S=0 : \lambda\vartheta=0)$. □

Thm. 3.9 indicates that, if we want to pipeline the control signals of all control variables, we must choose the scheduling vector such that no separating hyperplane is parallel to the temporal hyperplanes. Thus, in the formulation of the domain predicates, we should try to avoid conditionals that exclude desirable scheduling vectors from C^*. This theorem may also be used to selectively broadcast the control signals of certain control variables in order to optimise some aspects of the array, e.g., its latency or number of delay buffers.

3.6 Related Work

Chen [15] proposed a method that replaces a time-dependent conditional of a space-time equation by a control signal pipelined from the boundary of the array. We explain the basic idea by considering the pipelining of a generic time-dependent conditional $bt \boxdot (\sum i\ : 0<i<n : a_i p_i)$, where t represents time and $p_1, p_2, \cdots, p_{n-1}$ represent processor coordinates. If the conditional evaluates to true at processor $(x_1, \cdots, x_j, \cdots, x_{n-1})$ at step t_0, it also evaluates to true at processors $(x_1, \cdots, x_j + mb, \cdots, x_{n-1})$ at steps $t_0 + ma_j$ $(m \in \mathbf{Z})$. Instead of performing the test $bt \boxdot (\sum i\ : 0<i<n : a_i p_i)$ at each of these processors, the results of these tests can be shared. This is achieved by injecting a one-bit control signal at the appropriate border cell and pipelining it along direction $(0, \cdots, 0, b/a_j, 0, \cdots, 0)$ across the array at a velocity of b/a_j, i.e., passing b processors per a_j clock ticks.

This method does not apply for time-dependent conditionals of the form $bt \boxdot a$; they do not consist of processor coordinates. In this case, the corresponding control signals must be broadcast, because the scheduling vector is orthogonal to the normals of the hyperplanes associated with these conditionals (Thm. 3.9). But,

Chen's method provides no means for preventing these conditionals with the choice of the space-time mapping. Our method does away with these conditionals by the introduction of control dependence vectors. Chen's method cannot enforce design constraints on the control signals since there is no notion of control dependence vector. In the previous example conditional, $(\forall\ i : 0 < i < n : b/a_i > 1)$ will result in non-neighbouring channels. The question whether we can get away with only neighbouring channels cannot be answered. Chen's method also does not address control for the propagation of input/output data that are mapped to internal cells of the array. Our method specifies this control with the PCUREs. Finally, Chen's method does not generalise to systolic arrays of reduced dimension because it relies on space-time mappings for $(n-1)$-dimensional arrays.

The method reported in [78] is similar to Chen's method. The method reported in [64] only deals with conditionals that are equalities.

3.7 Examples

This section illustrates the synthesis of control signals with two examples: dynamic programming and LU-decomposition. The construction of the PCUREs is similar to that of the CCUREs. We shall therefore restrict ourselves mainly to the latter. For the former, we only present a set $\mathfrak{B}$ of defining half-spaces of the index space. So, the presentation of the extended source UREs is unnecessary. Proc. 3.1 is completely mechanical. We go through its steps to highlight the rôles that the various concepts and theorems introduced so far play in the construction of the control UREs. For each example, we present the Ⓣ-classes and the separation set $\mathfrak{S}_{\text{src}}$ (with manual optimisations) returned by Proc. 3.1. We then describe the pipelined UREs (without the propagation UREs and the PCUREs). We shall omit the half-spaces in a separation set which are the supporting half-spaces to the index space. We shall continue to depict every point of the index space by a graphical symbol (e.g., ■, ▲, etc). When we label a computation equation of the source UREs by a number of graphical symbols, we mean that the equation is only defined for the points depicted by those symbols (Sect. 2.2). When we write Φ_s for a graphical symbol s, we mean the set of all the points of the index space depicted by that symbol.

We use the first example to show how control complexity should be taken into account in the appraisal of the systolic array, and the second example to show how the CCUREs or PCUREs can be avoided by an appropriate choice of space-time mapping. These two examples will also be used for illustration in Chap. 5.

3.7.1 Dynamic Programming

We apply dynamic programming to solve the optimal string parenthesisation problem [30]. Let a string of items be indexed by integers 1 through m from left to right. A parenthesisation of a string of m items has $m-1$ pairs; each parenthesis pair encloses two elements each of which is either an item or another parenthesis pair. The optimal cost, denoted $c_{i,j}$, of parenthesising the substring consisting items i through $j-1$ is defined recursively as follows [15,27,64]:

Specification: $c_{i,j} = (\min\ k : i<k<j : c_{i,k}+c_{k,j}) + w_{i,j}.$

where $w_{i,j}$ is the additional cost for the outermost parenthesis pair. $c_{1,m+1}$ is the optimal cost for parenthesising a string of m items.

3.7.1.1. The Source UREs

This specification can be transformed into a system of UREs using the methods described in [63,64,89]. The following UREs are adapted from [64].

UREs:

$$A(i,j,k) = \begin{cases} j-i=2k \twoheadrightarrow D(i,j,k) & \blacktriangleleft\ \blacklozenge \\ j-i\neq 2k \twoheadrightarrow A(i,j-1,k) & \blacktriangleright\ \bullet\ \blacksquare\ \bigstar \end{cases}$$

$$B(i,j,k) = \begin{cases} k=1 \twoheadrightarrow C(i+1,j,k) & \blacktriangleleft\ \blacktriangleright\ \blacksquare \\ k\neq 1 \twoheadrightarrow B(i+1,j,k-1) & \blacklozenge\ \bullet\ \bigstar \end{cases}$$

$$D(i,j,k) = \begin{cases} k=1 \twoheadrightarrow C(i,j-1,k) & \blacktriangleleft\ \blacktriangleright\ \blacksquare \\ k\neq 1 \twoheadrightarrow D(i,j-1,k-1) & \blacklozenge\ \bullet\ \bigstar \end{cases}$$

$$E(i,j,k) = \begin{cases} j-i=2k \twoheadrightarrow B(i,j,k) & \blacktriangleleft \blacklozenge \\ j-i\neq 2k \twoheadrightarrow E(i+1,j,k) & \blacktriangleright \bullet \blacksquare \bigstar \end{cases}$$

$$c_{1,m+1} = C(1,m+1,1)$$

$$C(i,j,k) = \begin{cases} k=1 \wedge j-i=2k-1 \twoheadrightarrow w_{i,j} & \blacksquare \\ k=1 \wedge j-i\geqslant 2k \twoheadrightarrow \min \begin{pmatrix} A(i,j,k)+B(i,j,k) \\ C(i,j,k+1) \\ D(i,j,k)+E(i,j,k) \end{pmatrix} + w_{i,j} & \blacktriangleleft \blacktriangleright \\ k\neq 1 \wedge j-i\geqslant 2k \twoheadrightarrow \min \begin{pmatrix} A(i,j,k)+B(i,j,k) \\ C(i,j,k+1) \\ D(i,j,k)+E(i,j,k) \end{pmatrix} & \blacklozenge \bullet \\ k\neq 1 \wedge j-i<2k \wedge j-i\geqslant 2k-2 \twoheadrightarrow \infty & \bigstar \end{cases}$$

Index Space: $\Phi=\{(i,j,k) \mid 0<i<j \leqslant m+1 \wedge 0<k\leqslant (j-i)/2+1\}$

Data Dependence Matrix: $\mathcal{D} = [\vartheta_A\ \vartheta_B\ \vartheta_C\ \vartheta_D\ \vartheta_E] = \begin{bmatrix} 0 & -1 & 0 & 0 & -1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 1 & 0 \end{bmatrix}$

Data Dependence Graph ($m=7$)*:*

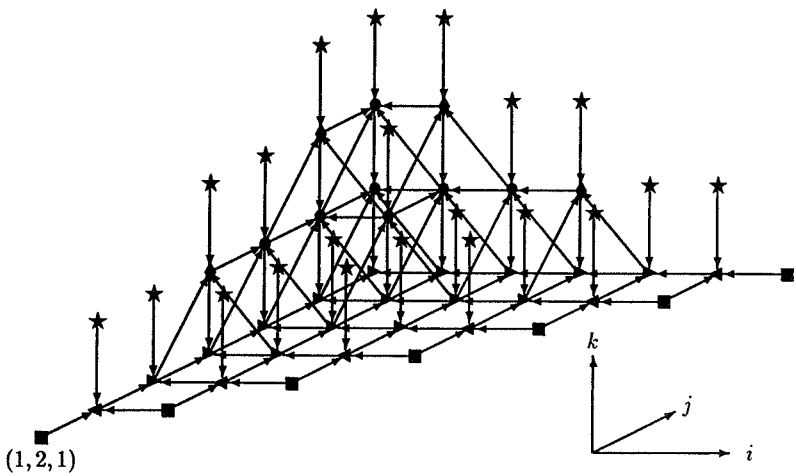


Figure 3–4: The data dependence graph for dynamic programming ($m=7$).

3.7.1.2. The Computation Control Flow

$\Phi/\textcircled{T}=\{\Phi_{\blacksquare},\Phi_{\blacktriangleleft},\Phi_{\blacktriangleright},\Phi_{\bigstar},\Phi_{\blacklozenge},\Phi_{\bullet}\}$; the $\textcircled{T}$-classes are:

$$\begin{aligned}
\Phi_{\blacksquare} &= \Phi\cap\{(i,j,k)\mid j-i=k\wedge k=1\}\\
\Phi_{\blacktriangleleft} &= \Phi\cap\{(i,j,k)\mid j-i=2k\wedge k=1\}\\
\Phi_{\blacktriangleright} &= \Phi\cap\{(i,j,k)\mid j-i>2k\wedge k=1\}\\
\Phi_{\bigstar} &= \Phi\cap\{(i,j,k)\mid j-i<2k\wedge j-i\geqslant 2k-2\}\\
\Phi_{\blacklozenge} &= \Phi\cap\{(i,j,k)\mid j-i=2k\wedge k\neq 1\}\\
\Phi_{\bullet} &= \Phi\cap\{(i,j,k)\mid j-i>2k\wedge k\neq 1\}
\end{aligned}$$

All six $\textcircled{T}$-classes are convex polytopes. Thus, $\Phi/\textcircled{T}$ is a $\textcircled{P}$-partition.

The set $\mathfrak{S}_{\text{src}}=\{S_1,S_2,S_3\}$ given by

$$\begin{aligned}
S_1 &= \{(i,j,k)\mid k\leqslant 1\}\\
S_2 &= \{(i,j,k)\mid -i+j-2k\leqslant -1\}\\
S_3 &= \{(i,j,k)\mid -i+j-2k\leqslant 0\}
\end{aligned}$$

is a separation set of $\Phi/\textcircled{T}$, since

$$\begin{aligned}
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\blacksquare}) &= \{S_1,S_2\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\blacktriangleleft}) &= \{S_1,\complement_{\mathbb{Z}^n}S_2,S_3\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\blacktriangleright}) &= \{S_1,\complement_{\mathbb{Z}^n}S_3\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\bigstar}) &= \{\complement_{\mathbb{Z}^n}S_1,S_2\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\blacklozenge}) &= \{\complement_{\mathbb{Z}^n}S_1,\complement_{\mathbb{Z}^n}S_2,S_3\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\bullet}) &= \{\complement_{\mathbb{Z}^n}S_1,\complement_{\mathbb{Z}^n}S_3\}
\end{aligned}$$

The hyperplanes associated with S_2 and S_3 are parallel. The two control variables associated with them can be merged (Sect. 3.4). Thus, the CCUREs must specify two computation control variables: P and Q. P is associated with S_1. $\text{sig}(P)=\{p_1,p_2\}$. Q is associated with S_2 and S_3. $\text{sig}(Q)=\{q_1,q_2,q_3\}$. An application of (3.8) and (3.23) yields the following CCUREs:

$$\begin{aligned}
(0,0,1)\vartheta_P &= 0\\
P(I) &= \begin{cases} I\in\text{in}(\Psi,\vartheta_P)\cap S_1 & \twoheadrightarrow\ p_1\\ I\in\text{in}(\Psi,\vartheta_P)\cap\complement_{\mathbb{Z}^n}S_1 & \twoheadrightarrow\ p_2\\ I\in\Psi & \twoheadrightarrow\ P(I-\vartheta_P)\end{cases}\\
(-1,1,-2)\vartheta_Q &= 0
\end{aligned}$$

$$Q(I) \;=\; \begin{cases} I \in \mathrm{in}(\Psi, \vartheta_Q) \cap S_2 & \twoheadrightarrow \; q_1 \\ I \in \mathrm{in}(\Psi, \vartheta_Q) \cap \complement_{\mathbb{Z}^n} S_2 \cap S_3 & \twoheadrightarrow \; q_2 \\ I \in \mathrm{in}(\Psi, \vartheta_Q) \cap \complement_{\mathbb{Z}^n} S_3 & \twoheadrightarrow \; q_3 \\ I \in \Psi & \twoheadrightarrow \; Q(I - \vartheta_Q) \end{cases}$$

The previous six $\textcircled{T}$-classes are redefined in the computation control variables by (3.10) and (3.11):

$$\begin{aligned}
\Phi'_{\blacksquare} &= \{I \mid I \in \Psi \wedge \mathcal{P}(\Phi_{\blacksquare})\} &&= \{I \mid I \in \Psi \wedge P(I) = p_1 \wedge Q(I) = q_1\} \\
\Phi'_{\blacktriangleleft} &= \{I \mid I \in \Psi \wedge \mathcal{P}(\Phi_{\blacktriangleleft})\} &&= \{I \mid I \in \Psi \wedge P(I) = p_1 \wedge Q(I) = q_2\} \\
\Phi'_{\blacktriangleright} &= \{I \mid I \in \Psi \wedge \mathcal{P}(\Phi_{\blacktriangleright})\} &&= \{I \mid I \in \Psi \wedge P(I) = p_1 \wedge Q(I) = q_3\} \\
\Phi'_{\bigstar} &= \{I \mid I \in \Psi \wedge \mathcal{P}(\Phi_{\bigstar})\} &&= \{I \mid I \in \Psi \wedge P(I) = p_2 \wedge Q(I) = q_1\} \\
\Phi'_{\blacklozenge} &= \{I \mid I \in \Psi \wedge \mathcal{P}(\Phi_{\blacklozenge})\} &&= \{I \mid I \in \Psi \wedge P(I) = p_2 \wedge Q(I) = q_2\} \\
\Phi'_{\bullet} &= \{I \mid I \in \Psi \wedge \mathcal{P}(\Phi_{\bullet})\} &&= \{I \mid I \in \Psi \wedge P(I) = p_2 \wedge Q(I) = q_3\}
\end{aligned}$$

By Lemma 3.1, $D/\textcircled{T} \subseteq \Phi/\textcircled{T}$ for every $D \in \mathrm{deq}(\Phi)$. $D/\textcircled{T}$ for the domain D of an equation in the source UREs in Sect. 3.7.1.1 contains the $\textcircled{T}$-classes whose subscripts are those labeling the equation. The calculation of predicate $\mathcal{P}(D)$ follows from (3.9). Replacing the domain predicates in the source UREs by predicates in the computation control variables as in (3.16) with optimisations yields ("$I \in \Psi \;\twoheadrightarrow$" that appears in each equation of (3.21) is omitted):

$$A(i,j,k) \;=\; \begin{cases} \textbf{if } Q(i,j,k) = q_2 \;\rightarrow\; D(i,j,k) & \blacktriangleleft \blacklozenge \\ [\!]\; Q(i,j,k) \neq q_2 \;\rightarrow\; A(i,j-1,k) & \blacktriangleright \bullet \blacksquare \bigstar \\ \textbf{fi} \end{cases}$$

$$B(i,j,k) \;=\; \begin{cases} \textbf{if } P(i,j,k) = p_1 \;\rightarrow\; C(i+1,j,k) & \blacktriangleleft \blacktriangleright \blacksquare \\ [\!]\; P(i,j,k) = p_2 \;\rightarrow\; B(i+1,j,k-1) & \blacklozenge \bullet \bigstar \\ \textbf{fi} \end{cases}$$

$$D(i,j,k) \;=\; \begin{cases} \textbf{if } P(i,j,k) = p_1 \;\rightarrow\; C(i,j-1,k) & \blacktriangleleft \blacktriangleright \blacksquare \\ [\!]\; P(i,j,k) = p_2 \;\rightarrow\; D(i,j-1,k-1) & \blacklozenge \bullet \bigstar \\ \textbf{fi} \end{cases}$$

$$E(i,j,k) \;=\; \begin{cases} \textbf{if } Q(i,j,k) = q_2 \;\rightarrow\; B(i,j,k) & \blacktriangleleft \blacklozenge \\ [\!]\; Q(i,j,k) \neq q_2 \;\rightarrow\; E(i+1,j,k) & \blacktriangleright \bullet \blacksquare \bigstar \\ \textbf{fi} \end{cases}$$

$$
\begin{aligned}
c_{1,m+1} &= C(1,m+1,1) \\
C(i,j,k) &= \\
&\begin{cases}
\textbf{if } P(i,j,k)=p_1 \wedge Q(i,j,k)=q_1 \;\rightarrow\; w_{i,j} \;\blacksquare \\
[\!]\; P(i,j,k)=p_1 \wedge Q(i,j,k)\neq q_1 \;\rightarrow\; \min \begin{pmatrix} A(i,j,k)+B(i,j,k) \\ C(i,j,k+1) \\ D(i,j,k)+E(i,j,k) \end{pmatrix} + w_{i,j} \;\blacktriangleleft\; \blacktriangleright \\
[\!]\; P(i,j,k)=p_2 \wedge Q(i,j,k)\neq q_1 \;\rightarrow\; \min \begin{pmatrix} A(i,j,k)+B(i,j,k) \\ C(i,j,k+1) \\ D(i,j,k)+E(i,j,k) \end{pmatrix} \;\blacklozenge\; \bullet \\
[\!]\; P(i,j,k)=p_2 \wedge Q(i,j,k)=q_1 \;\rightarrow\; \infty \;\bigstar \\
\textbf{fi}
\end{cases}
\end{aligned}
$$

3.7.1.3. The Propagation Control Flow

There are four supporting half-spaces to the index space:

$$
\begin{aligned}
B_1 &= \{(i,j,k) \mid k \geqslant 1\} \\
B_2 &= \{(i,j,k) \mid i \geqslant 1\} \\
B_3 &= \{(i,j,k) \mid j \leqslant m+1\} \\
B_4 &= \{(i,j,k) \mid -i+j-2k \geqslant -2\}
\end{aligned}
$$

No data variables are defined at the points of $\Psi \cap \complement_{\mathbb{Z}^n} B_2$ and $\Psi \cap \complement_{\mathbb{Z}^n} B_3$. Hence $\mathfrak{B} = \{B_1, B_4\}$.

3.7.1.4. The Space-Time Mapping

The following space-time matrices describe the systolic arrays depicted in Fig. 3–5; Π_s ($s \in \{\mathrm{a,b,c,d}\}$) describes the array shown in Fig. 3–5(s):

$$
\Pi_{\mathrm{a}} = \begin{bmatrix} -2 & 2 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad \Pi_{\mathrm{b}} = \begin{bmatrix} -2 & 2 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}
$$

$$
\Pi_{\mathrm{c}} = \begin{bmatrix} -2 & 2 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \Pi_{\mathrm{d}} = \begin{bmatrix} -2 & 2 & -1 \\ 0 & -1 & 1 \\ 1 & 0 & 1 \end{bmatrix}
$$

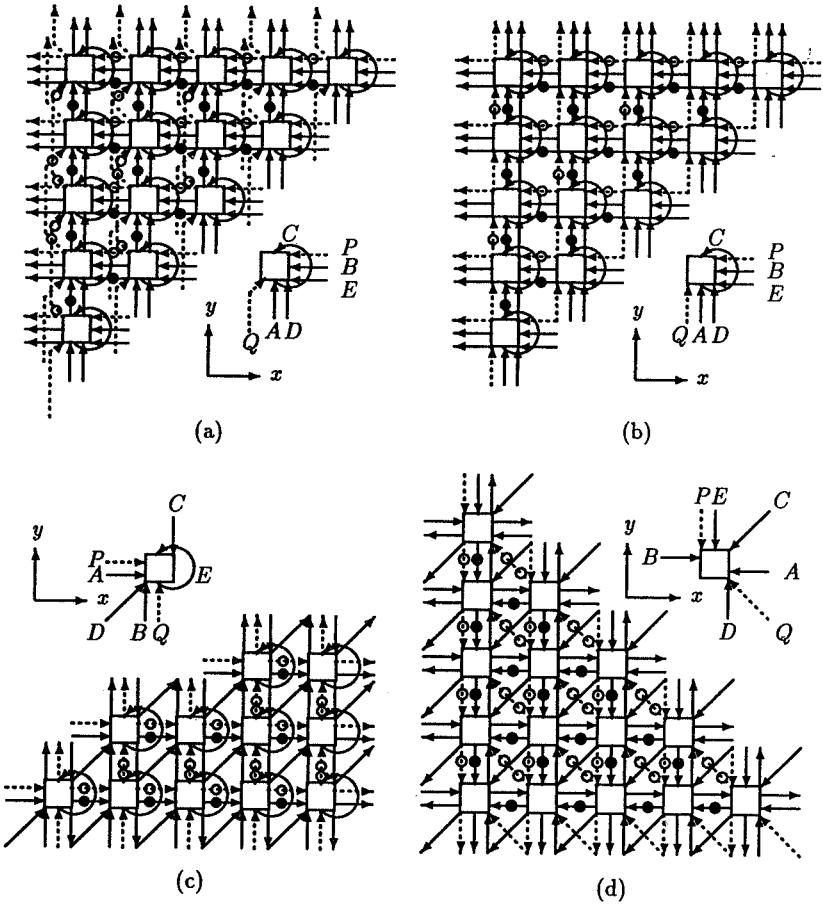


Figure 3–5: Four systolic arrays for dynamic programming ($m = 5$). Propagation control is not shown. (a) Guibas-Kung-Thompson's array [30] is described by $\lambda = (-2, 2, -1)$ and $u = (0, 0, 1)$ with $\vartheta_P = (-1, 0, 0)$ and $\vartheta_Q = (0, 2, 1)$. (b) Chen's array [15] is a modification of Guibas-Kung-Thompson's array (see text). (c) Gachet-Joinnault-Quinton's array [27] is described by $\lambda = (-2, 2, -1)$ and $u = (1, 0, 0)$ with $\vartheta_P = (0, 1, 0)$ and $\vartheta_Q = (-2, 0, 1)$. (d) A new systolic array that we propose is described by $\lambda = (-2, 2, -1)$ and $u = (-1, 1, 1)$ with $\vartheta_P = (-1, 0, 0)$ and $\vartheta_Q = (0, 2, 1)$.

The four arrays each run in $2m-1$ steps. Array (c) has $\lfloor m/2 \rfloor(\lfloor m/2 \rfloor+1)+m$ cells if m is odd and $m^2/4+m-1$ cells if m is even. The other three arrays each have $m(m+1)/2$ cells.

The space-time mapping of array (a) is chosen with respect to the source UREs only; it restricts the choice of the control dependence vectors. The best choice for control variable Q results in non-neighbouring channels, each connecting every other processors along y axis. A control signal of Q passes two processors in three time steps.

Array (b) can be regarded as a modification of array (a) as follows. First, eliminate all channels that connect processors (x, y), where $x+y$ is even. Second, remove all buffers from the remaining channels, make the remaining channels connect neighbours and then place one buffer on each output channel of the processors (x, y), where $x+y$ is even. This adaptation is a significant improvement, but it is hard to generalise. Note that the control flow of Q is not constant. Optimisations of this kind are problem-specific.

The motivations for array (c) is to reduce the number of processors. Again, the space-time mapping is chosen with respect to the source UREs only. The control channels of control variables P and Q are neighbouring channels.

Array (d) is synthesised from the pipelined UREs. We have two reasons for proposing it. First, we want to enforce neighbouring communications for both data and control variables. Second, we do not allow the existence of stationary variables. The concept of pipelined UREs enables all these design constraints to be taken into account straightforwardly. We derived this array by hand, but it could be derived using the methods in [26,58,92]. These methods allow the synthesis of systolic arrays with prescribed channel interconnections.

Let us review the control complexity of the four arrays. The control complexity of array (b) is similar to that of array (a). In arrays (a) and (d), different weights $w_{i,j}$ reside in different cells. Accessing these weights is simple. In array (c), weights $w_{i,j}$ with fixed j reside in the same cell. Additional access control is needed to choose the correct i. Array (a) needs to recover C, which is stationary. The stationary E in array (c) does not cause any problem because E needs not be recovered.

3.7.2 LU-Decomposition

LU-decomposition is the unique decomposition of a non-singular $m \times m$ matrix C into a lower-triangular matrix A and an upper-triangular matrix B such that $AB = C$. The elements of the upper triangle of A and the elements of the lower triangle of B (excluding the diagonal) are 0; the diagonal elements of A are 1.

Specification: $(\forall\, i,j : 0<i\leqslant m \wedge 0\leqslant j\leqslant m : (\sum\, k : 0<k\leqslant m : a_{i,k}b_{k,j} = c_{i,j}))$

3.7.2.1. The Source UREs

UREs:

$$A(i,j,k) = \begin{cases} k<i\leqslant m \wedge k=j \wedge 0<k\leqslant m & \twoheadrightarrow C(i,j,k-1)B(i,j,k) \quad \blacktriangle \\ k<i\leqslant m \wedge k<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow A(i,j-1,k) \quad \bullet \end{cases}$$

$$B(i,j,k) = \begin{cases} k=i \wedge k<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow C(i,j,k-1) \quad \blacktriangledown \\ k=i \wedge k=j \wedge 0<k\leqslant m & \twoheadrightarrow C(i,j,k-1)^{-1} \quad \blacksquare \\ k<i\leqslant m \wedge k\leqslant j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow B(i-1,j,k) \quad \bullet\ \blacktriangle \end{cases}$$

$$C(i,j,k) = \begin{cases} 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0=k & \twoheadrightarrow c_{i,j} \\ k\leqslant i\leqslant m \wedge k\leqslant j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow C(i,j,k-1)-A(i,j-1,k) \\ & \quad B(i-1,j,k) \quad \blacksquare\ \bullet\ \blacktriangle\ \blacktriangledown \end{cases}$$

$$a_{i,k} = \left\{ k<i\leqslant m \wedge m=j \wedge 0<k\leqslant m \twoheadrightarrow A(i,j,k) \right.$$

$$b_{k,j} = \left\{ m=i \wedge k\leqslant j\leqslant m \wedge 0<k\leqslant m \twoheadrightarrow B(i,j,k) \right.$$

Remark. Element $b_{i,i}$ for every i in the source UREs is the inverse of its corresponding element in the previous specification. □

Index Space: $\Phi = \{(i,j,k) \mid k\leqslant i\leqslant m \wedge k\leqslant j\leqslant m \wedge 0<k\leqslant m\}$

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

Data Dependence Graph ($m=4$)*:*

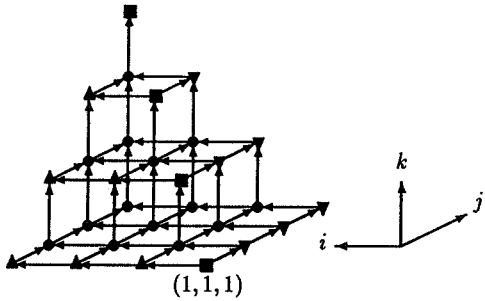


Figure 3–6: The data dependence graph for LU-decomposition ($m=4$). The dotted line will be referred to in Chap. 5.

3.7.2.2. The Computation Control Flow

$\Phi/\textcircled{T}=\{\Phi_{\blacksquare},\Phi_{\blacktriangle},\Phi_{\blacktriangledown},\Phi_{\bullet}\}$, the $\textcircled{T}$-classes are:

$$\begin{aligned}
\Phi_{\blacksquare} &= \{(i,j,k) \mid k=i \wedge k=j \wedge 0<k\leqslant m\}\\
\Phi_{\blacktriangle} &= \{(i,j,k) \mid k<i\leqslant m \wedge k=j \wedge 0<k\leqslant m\}\\
\Phi_{\blacktriangledown} &= \{(i,j,k) \mid k=i \wedge k<j\leqslant m \wedge 0<k\leqslant m\}\\
\Phi_{\bullet} &= \{(i,j,k) \mid k<i\leqslant m \wedge k<j\leqslant m \wedge 0<k\leqslant m\}
\end{aligned}$$

All four $\textcircled{T}$-classes are convex polytopes. Thus, $\Phi/\textcircled{T}$ is a $\textcircled{P}$-partition.

The set $\mathfrak{S}_{\text{src}}=\{S_1,S_2\}$ given by

$$\begin{aligned}
S_1 &= \{(i,j,k) \mid i-k\leqslant 0\}\\
S_2 &= \{(i,j,k) \mid j-k\leqslant 0\}
\end{aligned}$$

is a separation set, since

$$\begin{aligned}
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\blacksquare}) &= \{S_1,S_2\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\blacktriangle}) &= \{\complement_{\mathbb{Z}^n}S_1,S_2\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\blacktriangledown}) &= \{S_1,\complement_{\mathbb{Z}^n}S_2\}\\
\text{half}(\mathfrak{S}_{\text{src}},\Phi_{\bullet}) &= \{\complement_{\mathbb{Z}^n}S_1,\complement_{\mathbb{Z}^n}S_2\}
\end{aligned}$$

The CCUREs specify two computation control variables: P and Q. P is associated with S_1. $\mathrm{sig}(P)=\{p_1,p_2\}$. Q is associated with S_2. $\mathrm{sig}(Q)=\{q_1,q_2\}$. An application of (3.8) yields the following CCUREs:

$$
\begin{aligned}
(1,0,-1)\vartheta_P &= 0 \\
P(I) &= \begin{cases} I\in\mathrm{in}(\Psi,\vartheta_P)\cap S_1 & \twoheadrightarrow\ p_1 \\ I\in\mathrm{in}(\Psi,\vartheta_P)\cap \complement_{\mathbb{Z}^n} S_1 & \twoheadrightarrow\ p_2 \\ I\in\Psi & \twoheadrightarrow\ P(I-\vartheta_P) \end{cases} \\
(0,1,-1)\vartheta_Q &= 0 \\
Q(I) &= \begin{cases} I\in\mathrm{in}(\Psi,\vartheta_Q)\cap S_2 & \twoheadrightarrow\ q_1 \\ I\in\mathrm{in}(\Psi,\vartheta_Q)\cap \complement_{\mathbb{Z}^n} S_2 & \twoheadrightarrow\ q_2 \\ I\in\Psi & \twoheadrightarrow\ Q(I-\vartheta_Q) \end{cases}
\end{aligned}
$$

The previous four Ⓣ-classes are redefined in the computation control variables by (3.10) and (3.11):

$$
\begin{aligned}
\Phi'_{\blacksquare} &= \{I\in\Psi\wedge\mathcal{P}(\Phi_{\blacksquare})\} &&= \{I \mid I\in\Psi\wedge P(I)=p_1\wedge Q(I)=q_1\} \\
\Phi'_{\blacktriangle} &= \{I\in\Psi\wedge\mathcal{P}(\Phi_{\blacktriangle})\} &&= \{I \mid I\in\Psi\wedge P(I)=p_2\wedge Q(I)=q_1\} \\
\Phi'_{\blacktriangledown} &= \{I\in\Psi\wedge\mathcal{P}(\Phi_{\blacktriangledown})\} &&= \{I \mid I\in\Psi\wedge P(I)=p_1\wedge Q(I)=q_2\} \\
\Phi'_{\bullet} &= \{I\in\Psi\wedge\mathcal{P}(\Phi_{\bullet})\} &&= \{I \mid I\in\Psi\wedge P(I)=p_2\wedge Q(I)=q_2\}
\end{aligned}
$$

The calculation of predicates $\mathcal{P}(D)$ for every $D\in\mathrm{deq}(\Phi)$ proceeds exactly as in Sect. 3.7.1.2. Replacing the domain predicates of the source UREs by predicates in the computation control variables as in (3.16) yields (input and output equations are omitted):

$$
A(i,j,k) = \begin{cases} \textbf{if } P(i,j,k)=p_2\wedge Q(i,j,k)=q_1 \rightarrow C(i,j,k-1)B(i,j,k) & \blacktriangle \\ [\!]\ P(i,j,k)=p_2\wedge Q(i,j,k)=q_2 \rightarrow A(i,j-1,k) & \bullet \\ \textbf{fi} \end{cases}
$$

$$
B(i,j,k) = \begin{cases} \textbf{if } P(i,j,k)=p_1\wedge Q(i,j,k)=q_2 \rightarrow C(i,j,k-1) & \blacktriangledown \\ [\!]\ P(i,j,k)=p_1\wedge Q(i,j,k)=q_1 \rightarrow C(i,j,k-1)^{-1} & \blacksquare \\ [\!]\ P(i,j,k)=p_2 \rightarrow B(i-1,j,k) & \bullet\ \blacktriangle \\ \textbf{fi} \end{cases}
$$

$$
C(i,j,k) = C(i,j,k-1)-A(i,j-1,k)B(i-1,j,k) \quad \blacksquare\ \bullet\ \blacktriangle\ \blacktriangledown
$$

3.7.2.3. The Propagation Control Flow

There are five supporting half-spaces to the index space:

$$\begin{aligned} B_1 &= \{(i,j,k) \mid i \leqslant m\} \\ B_2 &= \{(i,j,k) \mid j \leqslant m\} \\ B_3 &= \{(i,j,k) \mid k \geqslant 1\} \\ B_4 &= \{(i,j,k) \mid i-k \geqslant 0\} \\ B_5 &= \{(i,j,k) \mid j-k \geqslant 0\} \end{aligned}$$

The input of A and B is undefined and the output of C is not of interest. That is, all data variables are undefined at the points in $\Psi \cap \mathbb{C}_{\mathbb{Z}^n} B_4$ and $\Psi \cap \mathbb{C}_{\mathbb{Z}^n} B_5$. Hence, $\mathfrak{B} = \{B_1, B_2, B_3\}$.

3.7.2.4. The Space-Time Mapping

Choose the following space-time mapping (Fig. 3–7(a)):

$$\Pi_0 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The projection vector is $u = (0,0,1)$. By Thm. 3.6, the propagation control variables P_1 and P_2 are redundant. P_3 serves to propagate the input data of variable C to the array; it is unnecessary since C is stationary variable, i.e., $\Psi \cap \mathbb{C}_{\mathbb{Z}^n} B_3 = \varnothing$. Let us choose the data dependence vectors as control dependence vectors: $\vartheta_P = (0,1,0)$ and $\vartheta_Q = (1,0,0)$.

Choose the following space-time mapping (Fig. 3–7(b)):

$$\Pi_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

The projection vector is $u = (1,1,1)$. By Thm. 3.6, all computation control variables are redundant. Three propagation control variables, P_1, P_2 and P_3 (associated with B_1, B_2 and B_3, respectively), are necessary for propagating the input data of C to the array and the output data of A and B from the array. For convenience, we set $\mathrm{sig}(P_i) = \{p_i, \overline{p}_i\}$, where $p_i = 0$ and $\overline{p}_i = 1$. Following the previous line of reasoning, we set $\vartheta_{P_1} = \vartheta_{P_3} = (0,1,0)$ and $\vartheta_{P_2} = (1,0,0)$.

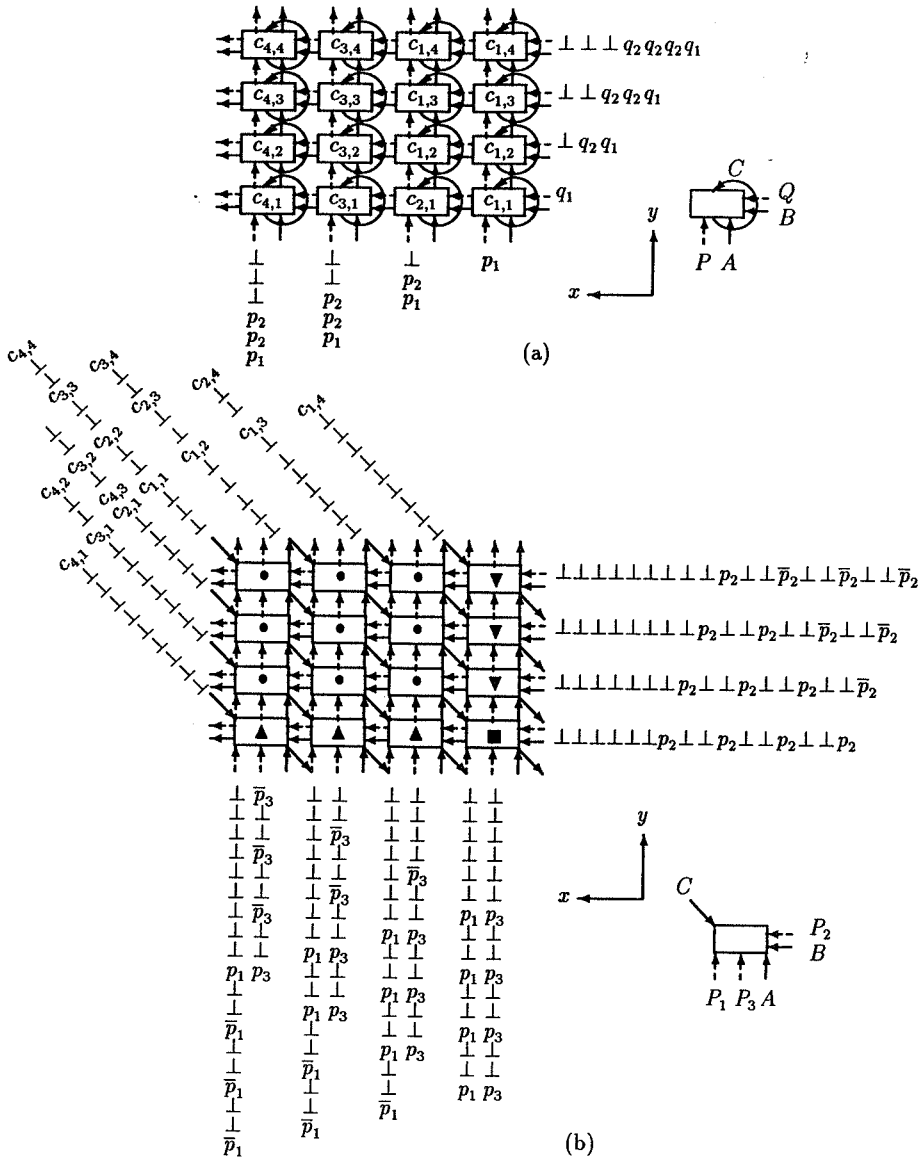


Figure 3–7: Two systolic arrays for LU-decomposition with data and control distribution at the first step ($m=4$). (a) The space-time mapping is Π_0. (b) The space-time mapping is Π_1. The highlighting of a cell by a symbol indicates that only computation points depicted by that symbol are mapped to the cell.

3.8 Conclusion

We have presented a method that enables a specification of control signals for systolic arrays in terms of control UREs. We have given necessary and sufficient conditions for the correctness of the pipelined UREs. These conditions allow the user to rewrite the domain predicates in order to obtain a better control flow. We have also shown how the pipelined UREs can be optimised to eliminate redundant control hardware in the systolic array.

Our method has the following advantages. First, it is based on linear algebra and the theory of polyhedra. Specifying control signals amounts to choosing separating hyperplanes. This enables the user of interactive systolic design systems to exercise his geometrical intuition to find a good control flow. Once the control flow is specified in the form of UREs, its optimisation with respect to some prescribed design criteria can be formulated as a linear and integer programming or combinatorial optimisation problem. Second, the notion of separating hyperplanes and control dependence vectors is explicit in the specification of control signals. Therefore, we can choose space-time mappings following prescribed design criteria with respect to not only the data but also the control flow. For example, we can enforce the pipelining of control signals by simply choosing the scheduling vector such that no separating hyperplane is parallel to the temporal hyperplanes (Thm.3.7). We can also enforce neighbouring communication for both the data and control flow, if possible, because both the data and control dependence vectors are explicit in the pipelined UREs [92]. Third, the correctness of the pipelined UREs is independent of the space-time mapping. The pipelined UREs constitute the specification of control signals for not just one but a class of systolic arrays that can be obtained by the choice of different space-time mappings. The pipelined UREs can be viewed as a refinement of the source UREs with the domain predicates substituted by predicates in variables of control UREs. Therefore, the pipelined UREs can be manipulated like the source UREs by available synthesis methods [28,45]. For example, they can be easily mapped to fixed-size arrays [12,18]. Of course, such a partitioning calls for extra control signals due to the change in the specification of the pipelined UREs. These extra control signals can be specified to preserve the behaviour of the pipelined UREs.

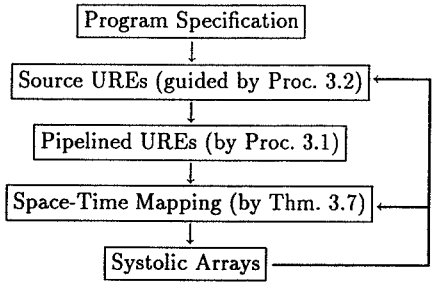


Figure 3–8: An iterative design process for the systhesis of systolic arrays.

The synthesis of systolic arrays from a program specification is an iterative process (Fig. 3–8). If the systolic array does not satisfy the prescribed criteria, we need to search for either a different space-time mapping or a different specification of the source UREs.

Our method for the construction of control UREs applies for any recurrence equations that are defined over a domain in $\mathbf{Z}^n$. This class of recurrence equations is an extension of AREs (Def 2.1); each equation is of the following form:

$$I \in D \;\rightarrow\; V(I) = f(W(\rho_W(I)), \cdots), \quad \text{where } \rho_W(I) : \mathbf{Z}^n \rightarrow \mathbf{Z}^\ell \; (0 \leqslant \ell \leqslant n) \quad (3.28)$$

ρ_W is any function from $\mathbf{Z}^n$ to $\mathbf{Z}^\ell$. The domain predicate $I \in D$ is defined as in Tab. 9–1, if we apply Proc. 3.1 to derive control signals. Otherwise it can be any predicate in the index vector $I \in \mathbf{Z}^n$, provided that one can construct a separation set. Recall that only UREs map directly to systolic arrays. If we are concerned with the design of systolic arrays, we must first transform the specifying recurrence equations to UREs and then add control UREs (say by applying Proc. 3.1) in order to obtain pipelined UREs.

Equivalence relation $\textcircled{T}$ plays an important rôle in the construction of the control UREs. This relation can be replaced by the equivalence relation $\textcircled{T}_\perp$:

$$I \textcircled{T}_\perp J \iff (\forall\, V : V \in \mathcal{V} : (\forall\, D : D \in \mathrm{deq}(V) : I \in D \Longrightarrow J \in D \vee J \in D_V^\perp))$$

where $D_V^\perp = \Phi \backslash \Phi_V$ is the set of the points I of the index space at which $V(I) = \perp$. That is, two points are of the same type if, for every data variable of the source

UREs, either the defining equations at both points agree or the defining equation at one of the two points is undefined. $\textcircled{T}$ is finer than $\textcircled{T}_\perp$. The use of $\textcircled{T}_\perp$ is feasible because the defining equation of a variable at a computation point, if undefined, is of free interpretation. The cell that evaluates the computation point can be instructed to perform any computation for the variable. Thus, this relaxation increases the space of separation sets at no extra cost.

If the index space itself is the only $\textcircled{T}$-class ($\Phi/\textcircled{T} = \{\Phi\}$), the CCUREs are unnecessary. In this case, the PCUREs may be eliminated by exploiting some algebraic properties of operators in the source UREs. This is the topic of Chap. 6.

The notion of an extended index space serves us well in the construction of the control UREs by (3.8) and (3.18). But our method does not depend on the space-time mapping. This follows from the fact that the construction of $\mathfrak{S}$ in Sect. 3.3.1 and $\mathfrak{B}$ in Sect. 3.3.2 are independent of the space-time mapping. The knowledge of the projection vector does allow us to remove redundant elements in $\mathfrak{S}$ and $\mathfrak{B}$ (Thm. 3.6), though. Both the CCUREs and the PCUREs are defined conceptually for the extended index space. An explicit extension of the index space is unnecessary. To see which of two control values of a control variable, C, needs to be input at border cell $p = (p_1, p_2, \cdots, p_{n-1})$ at step t, we only need to perform the test $(t, p_1, p_2, \cdots, p_{n-1}) \in \overline{\mathsf{H}_C}$ ($\overline{\mathsf{H}_C}$ denotes the image of H_C under the space-time mapping). We input control value **0** if the test succeeds and **1** if it fails.

There have been a number of publications on the optimisation of the latency and processor count of a systolic array [76,87,88]. In these papers, latency is defined with respect to data only. An accurate definition should also include control signals. The concept of pipelined UREs enables us to synthesise systolic arrays that satisfy prescribed design criteria with respect to both data and control signals.

4

Data Flow Synthesis for One-Dimensional Systolic Arrays

4.1 Introductory Remarks

The space-time mapping technique described in Chap. 2 is for the synthesis of time-minimal $(n-1)$-dimensional arrays from n-dimensional UREs. The $(n-1)$-dimensional processor space is obtained by a projection of the data dependence graph along the projection vector. If the resulting systolic array requires more dimensions than are available, further projections of the processor space become necessary. A sequence of projections is called a *multi-projection* [42,70]. The dimensions that are eliminated by the multiprojection are traded to time.

In practice, one-dimensional arrays have the following advantages [40]: 100% utilisation of non-faulty cells on a wafer, a constant I/O bandwidth that can be achieved by restricting external communication to the two boundary cells, and a clock rate that is independent of the size of the array. This chapter is concerned with the synthesis of data flow for one-dimensional arrays. There are two aspects to this study. The first one is an extension and improvement of previous results, and the second is a characterisation of some properties of one-dimensional arrays that will be used in the synthesis of control flow for one-dimensional arrays presented in Chap. 5.

The rest of this chapter is organised as follows. Sect. 4.2 defines the two most frequently used one-dimensional systolic array models: one allows non-neighbouring channel connections, the other does not. Sect. 4.3 formulates conditions for the validity of a space-time mapping in both models. Sect. 4.4 discusses the extension of the index space for the purpose of enforcing border and neighbouring communication. Sect. 4.5 characterises the computations for a fixed variable in one-dimensional arrays. Sects. 4.4 and 4.5 also describes several different formulations of the communication constraint. This leads to the realisation that the non-injectivity of the space-time mapping plays a crucial rôle in the analysis and synthesis of one-dimensional arrays. Sect. 4.6 describes a procedure for the synthesis of data flow for both models. Sect. 4.7 reviews related work. Sect. 4.8 contains the conclusion of the chapter.

4.2 One-Dimensional Systolic Array Models

Recall that a space-time mapping that describes an r-dimensional systolic array consists of two components: step and place.

- $\mathsf{step} : \mathbf{Z}^n \to \mathbf{Z}$, $\mathsf{step}(I) = \lambda I$, $\lambda \in \mathbf{Z}^n$. step specifies the temporal distribution. $\mathcal{S} = \{t \mid t_{\mathrm{fst}} \leqslant t \leqslant t_{\mathrm{lst}}\}$ is the scheduling space.
- $\mathsf{place} : \mathbf{Z}^n \to \mathbf{Z}^r$, $\mathsf{place}(I) = \sigma I$, $\sigma \in \mathbf{Z}^{r \times n}$. place specifies the spatial distribution. $\mathcal{P} = \{\sigma I \mid I \in \Phi\}$ is the processor space.

This chapter considers only space-time mappings that describe one-dimensional arrays ($r = 1$). In this case, σ is a vector, the *allocation vector*. It determines the coordinate of the leftmost cell $p_{\min}$ and the rightmost cell $p_{\max}$ of the array:

$$\begin{aligned} p_{\min} &= (\min\ I : I \in \Phi : \sigma I) \\ p_{\max} &= (\max\ I : I \in \Phi : \sigma I) \end{aligned}$$

By convention, variable V moves to the right if $\mathsf{flow}(V) > 0$, to the left if $\mathsf{flow}(V) < 0$.

To distinguish them from space-time mappings Π that describe $(n-1)$-dimensional arrays, we denote the space-time mappings for one-dimensional arrays by π:

$$\pi = \begin{bmatrix} \lambda \\ \sigma \end{bmatrix} = \begin{bmatrix} \lambda_1 & \lambda_2 & \dots & \lambda_n \\ \sigma_1 & \sigma_2 & \dots & \sigma_n \end{bmatrix}$$

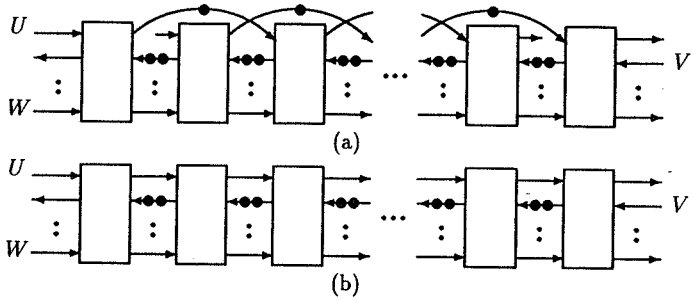


Figure 4–1: Two one-dimensional systolic array models. $\lambda\vartheta_U = 2$, $\sigma\vartheta_U = 2$, $\lambda\vartheta_V = 3$, $\sigma\vartheta_V = -1$, $\lambda\vartheta_W = 1$ and $\sigma\vartheta_W = 1$. (a) The π-model. (b) The τ-model.

The space-time mapping can be interpreted geometrically. The interpretation of the scheduling vector λ is the same as for $(n-1)$-dimensional arrays. That is, the points that are scheduled concurrently belong to a hyperplane whose normal is λ. The processor space is obtained by projecting the index space along the $(n-1)$-dimensional space that is orthogonal to the allocation vector σ. The points that are mapped to the same cell belong to a hyperplane whose normal is σ.

We study space-time mappings with respect to two one-dimensional array models. They differ in channel topology. One model is called the π-model; in it both the direction and length of a channel depend on the space-time mapping π. This model is an instance of the r-dimensional systolic array model of Def. 2.2 when $r=1$. The other model is called the τ-model; it allows only neighbouring communication. In the τ-model, only the direction of a channel depends on the space-time mapping; its length is constant.

Definition 4.1 (π-model) A one-dimensional systolic array consists of a finite sequence of cells with the following properties (Fig. 4–1(a)):

Prop. 1 (Synchrony of Computation) The array is driven by a global clock that ticks in unit time. Each cell is active at every clock cycle.

Prop. 2 (Uniqueness of Channel Connections) The image $\overline{\vartheta}_V$ of data dependence vector ϑ_V is given by

$$\overline{\vartheta}_V \;=\; \textbf{if } \mathsf{flow}(V)=0 \;\rightarrow\; \begin{bmatrix} 1 \\ 0 \end{bmatrix} \;[]\; \mathsf{flow}(V)\neq 0 \;\rightarrow\; \begin{bmatrix} \lambda\vartheta_V \\ \sigma\vartheta_V \end{bmatrix} \textbf{ fi}$$

For variable V, every cell p has an unique input (output) channel, which connects to cell $p-\sigma\vartheta_V$ ($p+\sigma\vartheta_V$). The number of buffers associated with the channel is the first component of $\overline{\vartheta}_V$ decremented by 1. If $p-\sigma\vartheta_V \notin \mathcal{P}$, cell p is an *input cell* of V. If $p+\sigma\overline{\vartheta}_V \notin \mathcal{P}$, cell p is an *output cell* of V. Both input and output cells of V are called *border cells* of V; they are connected to the external environment. The cells that are not border cells of V are called *internal cells* of V.

Prop. 3 (Linearity of Velocity) A variable moves with a constant velocity. □

Prop. 1 is the standard assumption that the evaluation of a point takes unit time. Prop. 2 ensures that a cell receives elements of a fixed variable from a fixed channel. Depending on the length of the channel, a border cell may be a processor inside the array – a few processors away from the real border. (In Fig. 4–1(a), the two cells on the left (right) are the input (output) cells of U. All four cells are the border cells of U.)

The length of the channels for a variable V is $|\sigma\vartheta_V|$. Therefore, there are $|\sigma\vartheta_V|$ input (output) cells for variable V. We can view a one-dimensional array as a graph whose nodes represent the cells of the array and whose directed arcs represent the channels of the array. A *link* of a variable is a directed path whose arcs are associated with the variable and whose first and last arcs connect border cells of the variable. Intuitively, a link represents the sequence of channels that an input value traverses across the array. Clearly, there are $|\sigma\vartheta_V|$ links, one per input (output) cell of V.

The definition of $\overline{\vartheta}_V$ for stationary variable V implies that each element of V travels along a loop channel without delay buffers. This definition conforms to the space-time diagram that will be introduced in Sect. 4.5. (In the systolic array model of Def. 2.2, the definition of $\overline{\vartheta}_V$ does not distinguish stationary from moving variables. There, each element of stationary variable V can be viewed as travelling along a loop channel associated with $\lambda\vartheta_V-1$ delay buffers.) Prop. 3 is guaranteed by the linearity of space-time mappings.

Next, we determine where and when an input (output) value is injected (ejected). Recall that pattern specifies the distribution of the input data in the processor

space at the first step. For one-dimensional arrays, it is more convenient to reason about the temporal and spatial distribution of input (output) data separately.

Function pi specifies the coordinates of the input cells:

$$\mathsf{pi} : \mathcal{V} \to \Phi_{\mathsf{fst}} \to \mathbf{Z},\ \Phi_{\mathsf{fst}} = (\bigcup V : V \in \mathcal{V} : \mathsf{fst}(\Phi_V, \vartheta_V))$$

$$\mathsf{pi}(V(I)) = \begin{cases} \textbf{if } \mathsf{flow}(V) > 0 & \to \quad p_{\mathsf{min}} + (\mathsf{place}(I) - p_{\mathsf{min}}) \bmod |\overline{\vartheta}_V.2| \\ [\!]\ \mathsf{flow}(V) < 0 & \to \quad p_{\mathsf{max}} + (\mathsf{place}(I) - p_{\mathsf{max}}) \bmod |\overline{\vartheta}_V.2| \\ \textbf{fi} \end{cases} \tag{4.1}$$

where the notation $x.i$ stands for the i-th component of an n-vector x. We write $\textcircled{1}_V$ for the equivalence relation on $\mathsf{fst}(\Phi_V, \vartheta_V)$ such that two points I and J are in the same $\textcircled{1}_V$-class iff $V(I)$ and $V(J)$ are injected at the same input cell, i.e.,

$$(\forall\, I, J : I, J \in \mathsf{fst}(\Phi_V, \vartheta_V) : I \textcircled{1}_V J \iff \mathsf{pi}(V(I)) = \mathsf{pi}(V(J)))$$

There are $|\sigma\vartheta_V|$ $\textcircled{1}_V$-classes, as many as there are links for variable V.

Function po specifies the coordinates of the output cells:

$$\mathsf{po} : \mathcal{V} \to \Phi_{\mathsf{lst}} \to \mathbf{Z},\ \Phi_{\mathsf{lst}} = (\bigcup V : V \in \mathcal{V} : \mathsf{lst}(\Phi_V \vartheta_V))$$

$$\mathsf{po}(V(I)) = \begin{cases} \textbf{if } \mathsf{flow}(V) > 0 & \to \quad p_{\mathsf{max}} + (\mathsf{place}(I) - p_{\mathsf{max}}) \bmod |\overline{\vartheta}_V.2| \\ [\!]\ \mathsf{flow}(V) < 0 & \to \quad p_{\mathsf{min}} + (\mathsf{place}(I) - p_{\mathsf{min}}) \bmod |\overline{\vartheta}_V.2| \\ \textbf{fi} \end{cases} \tag{4.2}$$

We write $\textcircled{0}_V$ for the equivalence relation on $\mathsf{lst}(\Phi_V, \vartheta_V)$ given by

$$(\forall\, I, J : I, J \in \mathsf{lst}(\Phi_V, \vartheta_V) : I \textcircled{0}_V J \iff \mathsf{po}(V(I)) = \mathsf{po}(V(J)))$$

There are $|\sigma\vartheta_V|$ $\textcircled{0}_V$-classes, as many as there are links for variable V.

Having defined functions that determine the cells that perform input (output), we next define functions that determine the steps at which input (output) is performed.

Function input specifies the steps at which input data are injected into the array:

$$\begin{aligned} &\mathsf{input} : \mathcal{V} \to \Phi_{\mathsf{fst}} \to \mathbf{Z},\ \Phi_{\mathsf{fst}} = (\bigcup V : V \in \mathcal{V} : \mathsf{fst}(\Phi_V, \vartheta_V)) \\ &\mathsf{input}(V(I)) = \mathsf{step}(I) - (\mathsf{place}(I) - \mathsf{pi}(V(I)) / \mathsf{flow}(V) \end{aligned} \tag{4.3}$$

Function output specifies the steps at which output data are ejected from the array:

$$\begin{aligned} &\mathsf{output} : \mathcal{V} \to \Phi_{\mathsf{lst}} \to \mathbf{Z},\ \Phi_{\mathsf{lst}} = (\bigcup V : V \in \mathcal{V} : \mathsf{lst}(\Phi_V, \vartheta_V)) \\ &\mathsf{output}(V(I)) = \mathsf{step}(I) - (\mathsf{place}(I) - \mathsf{po}(V(I)) / \mathsf{flow}(V) \end{aligned} \tag{4.4}$$

The first step t_{fst} at which an input value is injected and the last step t_{lst} at which an output value is ejected are given by

$$\begin{aligned} t_{\text{fst}} &= (\min\ V, I : V \in \mathcal{V} \wedge I \in \Phi_{\text{fst}} : \mathsf{input}(V(I))) \\ t_{\text{lst}} &= (\max\ V, I : V \in \mathcal{V} \wedge I \in \Phi_{\text{lst}} : \mathsf{output}(V(I))) \end{aligned}$$

Definition 4.2 (τ-model) A one-dimensional systolic array consists of a finite sequence of cells with the following properties (Fig. 4–1(b)):

Prop. 1 (Synchrony of Computation) The array is driven by a global clock that ticks in unit time. Each cell is active at every clock cycle.

Prop. 2 (Uniqueness of Channel Connections) The image $\overline{\vartheta}_V$ of data dependence vector ϑ_V is given by

$$\overline{\vartheta}_V = \text{if } \mathsf{flow}(V)=0 \rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix} [\!]\ \mathsf{flow}(V)\neq 0 \rightarrow \begin{bmatrix} \lambda\vartheta_V/|\sigma\vartheta_V| \\ \mathsf{sign}(\mathsf{flow}(V)) \end{bmatrix} \text{ fi}$$

For variable V, there is a unique channel between every two neighbouring cells. The number of buffers associated with the channel is the first component of $\overline{\vartheta}_V$ decremented by 1. $p_{\min}$ is the *input* (*output*) *cell* of V and $p_{\max}$ is the *output* (*input*) *cell* of V if $\mathsf{flow}(V)>0$ ($\mathsf{flow}(V)<0$). Both input and output cells of V are called *border cells* of V; they are connected to the external environment. The cells that are not border cells of V are called *internal cells* of V.

Prop. 3 (Linearity of Velocity) A variable moves with a constant velocity. □

Prop. 2 requires that, for variable V travelling along non-neighbouring channels in the π-model, neighbouring communication must be enforced in the τ-model. This is achieved by breaking a non-neighbouring channel into $|\sigma\vartheta_V|$ neighbouring channels, making the new channels connect $|\sigma\vartheta_V|-1$ intermediate cells, and evenly distributing the $\lambda\vartheta_V-1$ buffers associated with the original channel over the new channels (see variable U in Fig. 4–1). Since each intermediate cell acts as a delay buffer, the number of buffers associated with a new channel is $(\lambda\vartheta_V-1-(|\sigma\vartheta_V|-1))/|\sigma\vartheta_V|$, which simplifies to $\lambda\vartheta_V/|\sigma\vartheta_V|-1$. The definition of $\overline{\vartheta}_V$ for stationary variable V is consistent with the previous interpretation for moving variables. Each element of V travels along a loop channel without delay buffers.

Remark. The definitions of pi, po, input, output, $\textcircled{I}_V$ and $\textcircled{O}_V$ for the π-model carry over to the τ-model. In the τ-model, $|\overline{\vartheta}_V.2| = 1$. The right operand of each addition in (4.1) and (4.2) reduces to zero. Thus, functions pi and po are as desired. Since the τ-model enforces neighbouring communication, $\mathrm{fst}(\Phi_V, \vartheta_V)/\textcircled{I}_V = \{\mathrm{fst}(\Phi_V, \vartheta_V)\}$ and $\mathrm{lst}(\Phi_V, \vartheta_V)/\textcircled{O}_V = \{\mathrm{lst}(\Phi_V, \vartheta_V)\}$.

The functions pi po, input and output are undefined for stationary variables. Whenever we apply these functions, it is understood that the variable they are applied to is moving. For example, when we write $(\forall\ V : V \in \mathcal{V} : \sigma\vartheta_V | \lambda\vartheta_V)$, we mean $(\forall\ V : V \in \mathcal{V} \wedge \mathsf{flow}(V) \neq 0 : \sigma\vartheta_V | \lambda\vartheta_V)$. □

4.3 The Mapping Conditions

We cannot ensure the computation and communication rules of Def. 2.4 by simply choosing non-singular space-time matrices, because these matrices are of size $2 \times n$ and are not square when $n > 2$. The previous mapping conditions for $(n-1)$-dimensional arrays must be modified and extended.

Theorem 4.1 *A space-time mapping π is valid for the source UREs in the π-model iff*

- $(\forall\ V : V \in \mathcal{V} : \lambda\vartheta_V \geqslant 1)$ *(Precedence Constraint)*
- $\pi : \Phi \rightarrowtail \mathbf{Z}^2$ *(Computation Constraint)*
- $(\forall\ V : V \in \mathcal{V} : (\forall\ S : S \in \mathrm{fst}(\Phi_V, \vartheta_V)/\textcircled{I}_V : \mathsf{input} : S \rightarrowtail \mathbf{Z}))$ *(Communication Constraint)*

Proof. The precedence, computation and communication constraint are equivalent to the respective rules of Def. 2.4. The delay rule is enforced by Prop. 2 of the π-model. □

Theorem 4.2 *A space-time mapping π is valid for the source UREs in the τ-model iff*

- $(\forall\ V : V \in \mathcal{V} : \lambda\vartheta_V \geqslant 1)$ *(Precedence Constraint)*

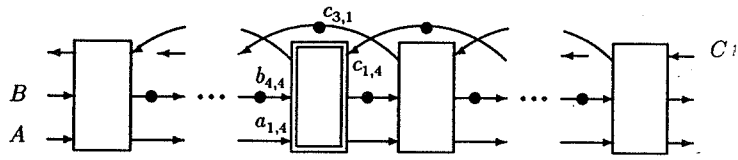


Figure 4–2: A systolic array in which the neighbouring communication cannot be enforced. The left dangling arrow depicts the output channel of the first output cell of C. The right dangling arrow depicts the input channel of the second input cell of C. The cell highlighted by a doubly-bounded box is computing point $(1,4,4)$; it is performing the last accumulation of element $c_{1,4}$. The delay buffer above the cell is propagating element $c_{3,1}$, which is the value represented by $C(3,1,3)$. If the two links associated with C were merged, the cell would have to both accumulate $C_{1,4}$ and propagate $c_{3,1}$ at the same step.

- $\pi : \Phi \rightarrowtail \mathbb{Z}^2$ *(Computation Constraint)*
- $(\forall\, V : V \in \mathcal{V} : \sigma\vartheta_V | \lambda\vartheta_V)$ *(Delay Constraint)*
- $(\forall\, V : V \in \mathcal{V} : (\forall\, S : S \in \mathrm{fst}(\Phi_V, \vartheta_V)/\textcircled{1}_V : \mathsf{input} : S \rightarrowtail \mathbb{Z}))$ *(Communication Constraint)*

Proof. By Thm. 4.1, we only need to consider the delay constraint. By Prop. 2 of the τ-model, the delay constraint is equivalent to the delay rule of Def. 2.4. □

The communication constraint in Thm. 4.2 can be written equivalently as $(\forall\, V : V \in \mathcal{V} : \mathsf{input} : \mathrm{fst}(\Phi_V, \vartheta_V) \rightarrowtail \mathbb{Z})$, since $\mathrm{fst}(\Phi_V, \vartheta_V)/\textcircled{1}_V = \{\mathrm{fst}(\Phi_V, \vartheta_V)\}$. The fact that the communication constraint can be phrased equivalently in both models allows us to talk about it without an explicit reference to either model.

It is easy to see that the set of mapping conditions is stronger for the τ-model than for the π-model. Put another way, space-time mappings that are valid in the τ-model are also valid in the π-model, but the converse is not true. Let us use matrix product as example to illustrate this fact and the essential rôle that the communication constraint plays in ensuring the validity of the space-time mapping.

Example 4.1 4×4 Matrix Product.

Choose the following space-time mapping:

$$\pi = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 1 & -2 \end{bmatrix}$$

The precedence, delay and computation constraint are satisfied. The communication constraint is satisfied for variables A and B. Let us consider variable C with $\lambda\vartheta_C = 2$ and $\sigma\vartheta_C = -2$. In the τ-model, only one link per variable is allowed. The communication constraint is violated since $\mathsf{input}(C(1,4,0)) = \mathsf{input}(C(3,1,0)) = 5$ and $\mathsf{input}(C(2,4,0)) = \mathsf{input}(C(4,1,0)) = 8$ (Fig. 4–2). In the π-model, there are two links associated with variable C; each link consist of channels that connect every other cells. The communication constraint is satisfied because $\mathsf{pi}(C(1,4,0)) = \mathsf{pi}(C(4,1,0)) = p_{\max} - 1$ and $\mathsf{pi}(C(3,1,0)) = \mathsf{pi}(C(2,4,0)) = p_{\max}$. That is, elements $C(1,4,0)$ and $C(3,1,0)$ are injected at the same step but at different input cells. They travel along different links. The same is true of elements $C(2,4,0)$ and $C(4,1,0)$.

Choose the following space-time mapping:

$$\pi = \begin{bmatrix} 6 & 2 & 2 \\ 1 & -2 & 1 \end{bmatrix}$$

The precedence, delay and computation constraint are satisfied. The communication constraint is satisfied for variables A and B. Let us consider variable C with $\lambda\vartheta_C = 2$ and $\sigma\vartheta_C = 1$. There is only one link consisting of neighbouring channels associated with variable C in both models. Since $\mathsf{input}(C(1,3,0)) = \mathsf{input}(C(4,1,0)) = 34$ and $\mathsf{input}(C(1,4,0)) = \mathsf{input}(C(4,2,0)) = 40$, the communication constraint for variable C is violated in both models. □

The next lemma provides conditions under which the communication constraint implies the computation constraint. (The converse is generally not true as has been demonstrated by Ex. 4.1.) Intuitively, if two distinct points are computed simultaneously at the same cell, the two elements of a fixed variable indexed by the two points must travel along a common link and, consequently, must be input simultaneously at the corresponding input cell of the link.

Lemma 4.1 *Let* $\Phi = \Phi_V$ *and* $\lambda\vartheta_V \neq 0$.

$$(\forall\, S : S \in \mathsf{fst}(\Phi_V, \vartheta_V)/\textcircled{1}_V : \mathsf{input} : S \rightarrowtail \mathbb{Z}) \implies \pi : \Phi \rightarrowtail \mathbb{Z}^2 \qquad (4.5)$$

Proof. Assume $\pi I = \pi J$ for two distinct points $I \neq J \wedge I, J \in \Phi$.

$$
\begin{aligned}
& \mathsf{true} \\
\Longleftrightarrow\ & \{\text{Assumption}\} \\
& \pi I = \pi J \wedge I \neq J \\
\Longrightarrow\ & \{\text{Definition of } \mathsf{input}\} \\
& \mathsf{input}(V(I)) = \mathsf{input}(V(J)) \wedge I \neq J \\
\Longleftrightarrow\ & \{\lambda \vartheta_V \neq 0;\ \lambda I = \lambda J\} \\
& \mathsf{input}(V(I)) = \mathsf{input}(V(J)) \wedge I \overset{\vartheta_V}{\not\rightarrow} J \\
\Longleftrightarrow\ & \{\text{Choose } K, L \in \mathrm{fst}(\Phi_V, \vartheta_V) \text{ such that } K \overset{\vartheta_V}{\rightarrow} I \wedge L \overset{\vartheta_V}{\rightarrow} J\} \\
& \mathsf{input}(V(K)) = \mathsf{input}(V(L)) \wedge K \overset{\vartheta_V}{\not\rightarrow} L \\
\Longleftrightarrow\ & \{\text{Definition of } \textcircled{\mathsf{I}}_V\} \\
& (\exists\, S : S \in \mathrm{fst}(\Phi_V, \vartheta_V) / \textcircled{\mathsf{I}}_V : K, L \in S) \\
\Longleftrightarrow\ & \{\text{Antecedent of (4.5)}\} \\
& \mathsf{false}
\end{aligned}
$$

□

4.4 The Extension of the Index Space

Sect. 2.5 discussed the extension of the index space for the purpose of enforcing border communication for $(n-1)$-dimensional arrays. Here, the extension of the index space plays an additional rôle: it enforces neighbouring communication in the τ-model.

Remark. The π-model permits non-neighbouring communication. The extension of the index space that aims at the enforcement of neighbouring communication in the τ-model does not apply for the π-model. It would be confusing to switch constantly between the two models in the presentation. We avoid this confusion by giving the definition of a concept for both array models even if it is only meaningful for one of them: the concepts of δ_V in (4.6) and Γ_V in (4.10), that are intended for the τ-model, are also defined for the π-model. All our results apply to both array models unless specified otherwise. □

The extension of the index space proceeds in two steps:

1. The index space is extended such that the input and output data are mapped to border cells. This follows from Def. 2.5.

2. The data dependence vectors are redefined such that the new data dependence vectors are mapped to neighbouring communication channels in the τ-model. The new data dependence vector δ_V of ϑ_V is given by

$$\begin{aligned} \delta_V &= \textbf{if } \mathsf{flow}(V)=0 \rightarrow \vartheta_V/\lambda\vartheta_V \ [] \ \mathsf{flow}(V)\neq 0 \rightarrow \vartheta_V \ \textbf{fi} && (\pi\text{-model}) \\ \delta_V &= \textbf{if } \mathsf{flow}(V)=0 \rightarrow \vartheta_V/\lambda\vartheta_V \ [] \ \mathsf{flow}(V)\neq 0 \rightarrow \vartheta_V/|\sigma\vartheta_V| \ \textbf{fi} && (\tau\text{-model}) \end{aligned} \tag{4.6}$$

Each data dependence arc ϑ_V, connecting points I and J in the data dependence graph, that was created in the first step is sliced into as many consecutive arcs of equal length as specified by the corresponding denominator of ϑ_V, denoted D, in (4.6). This creates $D-1$ points on the original arc. Consider one such point K; at step $\mathsf{step}(K)$, cell $\mathsf{place}(K)$ serves as a delay buffer to propagate an element of V from the cell at location $\mathsf{place}(I)$ to the cell at location $\mathsf{place}(J)$.

Remark. $\overline{\vartheta}_V$ as defined in two one-dimensional array models is equal to the image of δ_V under the space-time mapping as defined in (2.16). Hence, the two definitions are consistent. □

To formalise the extension of the index space, we need the following notation. Let $L \subset \mathbf{Q}^{m\times n}$ be a linear transformation from $\mathbf{Q}^n$ to $\mathbf{Q}^m$ and $S \subset \mathbf{Q}^n$. planes(S, L) denotes the set of all points in $\mathbf{Q}^n$ that shares the same image as S under L:

$$\text{planes}(S, L) \ = \ \{I \mid I \in \mathbf{Q}^n \wedge LI = LJ \wedge J \in S\} \tag{4.7}$$

If $L_1, L_2, \cdots, L_m$ are the row vectors of L, then planes$(S, L) = (\cap\ i : 0 < i \leqslant m :$ planes$(S, L_i))$, where planes(S, L_i) is the union of all parallel hyperplanes whose normal is L_i and that intersect S.

The allocation matrix σ has size $r \times n$. planes(Φ, σ) denotes the inverse image of the processor space in $\mathbf{Q}^n$. There are two special cases. First, if σ is an $(n-1)\times n$ allocation matrix, it describes an $(n-1)$-dimensional array; then planes$(\Phi, \sigma) =$ rays$(\Phi, \pm u)$. Second, if σ an allocation vector, it describes an one-dimensional array; then planes(Φ, σ) plays the rôle in one-dimensional arrays that rays$(\Phi, \pm u)$ plays in $(n-1)$-dimensional arrays.

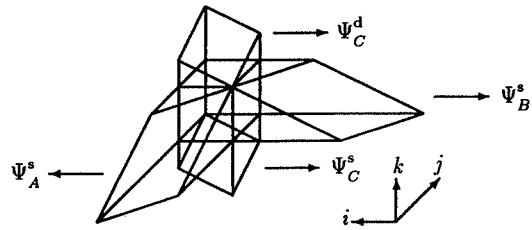


Figure 4–3: The extended index space with respect to the allocation vector of Fig. 4–4 for matrix product ($m=4$). The cube depicts the index space. Extensions along ϑ_A and ϑ_B are not necessary because the output of A and B is of no interest.

Definition 4.3 The *extended index space* Ψ of the index space Φ is defined as follows (Fig. 4–3):

$$\Psi^{s}_{V} = \begin{cases} \textbf{if } \mathsf{flow}(V)=0 & \rightarrow \quad \varnothing \\ [] \ \mathsf{flow}(V)\neq 0 & \rightarrow \quad \mathrm{planes}(\Phi,\sigma) \cap \mathrm{rays}(\mathrm{fst}(\Phi_V,\vartheta_V)-\delta_V,-\delta_V) \\ \textbf{fi} \end{cases}$$

$$\Psi^{d}_{V} = \begin{cases} \textbf{if } \mathsf{flow}(V)=0 & \rightarrow \quad \varnothing \\ [] \ \mathsf{flow}(V)\neq 0 & \rightarrow \quad \mathrm{planes}(\Phi,\sigma) \cap \mathrm{rays}(\mathrm{lst}(\Phi_V,\vartheta_V)+\delta_V,\delta_V) \\ \textbf{fi} \end{cases}$$

$$\begin{aligned}
\Psi^{r}_{V} &= (\mathrm{rays}(\Phi_V,-\delta_V) \cap \mathrm{rays}(\Phi_V,\delta_V)) \setminus \Phi_V \\
\Psi_V &= \Phi_V \cup \Psi^{s}_{V} \cup \Psi^{r}_{V} \cup \Psi^{d}_{V} \\
\Psi^{p} &= (\forall\, V : V \in \mathcal{V} : \Psi^{s}_{V} \cup \Psi^{r}_{V} \cup \Psi^{d}_{V}) \\
\Psi &= \Phi \cup \Psi^{p} \\
\Psi^{\perp}_{V} &= \Psi \setminus \Psi_V
\end{aligned}$$

The portion attributed to variable V is Ψ_V (compare Def. 2.5). □

We have given a formal definition of the extended index space for $(n-1)$-dimensional arrays in Def. 2.5. Every set defined in Def. 4.3, except Ψ^{r}_{V}, has a parallel in Def. 2.5. The interpretation of the points of the two corresponding sets is identical. The points of Ψ^{r}_{V} are called *relaying points of* V. In the presence of relaying points, the extension of the index space depends on the scheduling vector.

Let V be a moving variable. In the π-model, $\Psi_V^r = \varnothing$. In the τ-model, $\Psi_V^r = \varnothing$ only if $|\sigma\vartheta_V| = 1$, i.e., only if V is mapped directly to neighbouring channels. The relaying points serve to employ immediate cells as delay buffers for relaying the data of V between non-neighbouring cells. The relaying points may be rational (Fig. 4–1). So $\Psi \subset \mathbf{Z}^n$ may not hold.

The extended source UREs over the extended index space are as defined in Sect. 2.5 except that all data dependence vectors ϑ_V are replaced by δ_V.

The following concepts are analogous to those defined for the index space. The set of *new* first computation points of variable V over the extended index space is $\mathrm{fst}(\Psi_V, \delta_V)$. The set of *new* last computation points of variable V over the extended index space is $\mathrm{lst}(\Psi_V, \delta_V)$.

Remark. The concept of $\mathrm{fst}(\Psi_V, \delta_V)$ ($\mathrm{lst}(\Psi_V, \delta_V)$) for one-dimensional arrays is parallel to the concept of $\mathrm{fst}(\Psi_V, \vartheta_V)$ ($\mathrm{lst}(\Psi_V, \vartheta_V)$) for $(n-1)$-dimensional systolic arrays. For compatibility and ease of reference, we shall write $\mathrm{fst}(\Psi_V, \vartheta_V)$ for $\mathrm{fst}(\Psi_V, \delta_V)$ and $\mathrm{lst}(\Psi_V, \vartheta_V)$ for $\mathrm{lst}(\Psi_V, \delta_V)$. □

The points of $\mathrm{fst}(\Psi_V, \vartheta_V)$ are mapped to input cells. We write $\textcircled{I}'_V$ for the equivalence relation on $\mathrm{fst}(\Psi_V, \vartheta_V)$ given by

$$(\forall\, I, J : I, J \in \mathrm{fst}(\Psi_V, \vartheta_V) : I \textcircled{I}'_V J \iff \mathsf{pi}(V(I)) = \mathsf{pi}(V(J))) \tag{4.8}$$

The points in a $\textcircled{I}'_V$-class are mapped to the same input cell. Thus, the communication constraint ensures the injectivity of step for each $\textcircled{I}'_V$-class. This formulation of the communication constraint is useful in the validity proof of the space-time mappings generated by the procedure described in Sect. 4.6.

Lemma 4.2 *The communication constraint is satisfied iff*

$$(\forall\, V : V \in \mathcal{V} : (\forall\, S : S \in \mathrm{fst}(\Psi_V, \vartheta_V)/\textcircled{I}'_V : \mathsf{step} : S \mapsto \mathbf{Z}))$$

Proof.

$$\begin{aligned}
& (\forall\, V : V \in \mathcal{V} : (\forall\, S : S \in \mathrm{fst}(\Psi_V, \vartheta_V)/\textcircled{I}'_V : \mathsf{step} : S \mapsto \mathbf{Z})) \\
\iff\ & \{(\forall\, I, J : I, J \in S : \mathsf{place}(I) = \mathsf{place}(J))\} \\
& (\forall\, V : V \in \mathcal{V} : (\forall\, S : S \in \mathrm{fst}(\Psi_V, \vartheta_V)/\textcircled{I}'_V : \mathsf{input} : S \mapsto \mathbf{Z})) \\
\iff\ & \{\text{Define } \rho_V : \mathrm{fst}(\Phi_V, \vartheta_V) \to \mathrm{fst}(\Psi_V, \vartheta_V),\ \rho_V(I) = J, \text{ where } J \xrightarrow{\vartheta_V} I. \\
& \ \text{Then } \rho_V \text{ is a bijection and } \mathsf{input}(V(I)) = \mathsf{input}(V(J))\} \\
& (\forall\, V : V \in \mathcal{V} : (\forall\, S : S \in \mathrm{fst}(\Phi_V, \vartheta_V)/\textcircled{I}_V : \mathsf{input} : S \mapsto \mathbf{Z}))
\end{aligned}$$

□

The next lemma rephrases the communication constraint by viewing the space-time mapping as a mapping from Ψ_V to Υ.

Lemma 4.3 *The communication constraint is satisfied iff*

$$(\forall\, V : V \in \mathcal{V} : \pi : \Psi_V \;\rightarrowtail\; \mathbf{Z}^2)$$

Proof. Similar to the proof of Lemma 4.1. (The idea is to show that, if two distinct points of Ψ_V share the same image, then the two elements of V indexed by the two points must travel along a common link and, consequently, must be input simultaneously at the corresponding input cell of the link.) □

This lemma provides the basis for the characterisation of computations associated with a variable in one-dimensional arrays (Lemma 4.4).

4.5 The Space-Time Diagram

It is convenient to represent a one-dimensional array as a *space-time diagram*, which is obtained by viewing the points of $\overline{\Psi}$ as forming a two-dimensional lattice, where the horizontal axis represents time and the vertical axis space. The points in the space-time diagram are called *space-time points*. The set Υ of space-time points is given by

$$\Upsilon \;=\; \{(t,p) \mid t_{\mathrm{fst}} \leqslant t \leqslant t_{\mathrm{lst}} \wedge p_{\min} \leqslant p \leqslant p_{\max} \wedge t \in \mathbf{Z} \wedge p \in \mathbf{Z}\}$$

where t represents time and p represents space. $\Upsilon \backslash \overline{\Psi}$ contains points whose computations are undefined. These points represent the situation when the corresponding cells are idle at the corresponding steps. They are also called *pipelining points*, for they can be interpreted as propagating the undefined value $\perp$.

The classification of the points in the extended domain of a fixed variable, as in Def. 4.3 (and Def. 2.5), also applies to the space-time diagram. We shall carry over type designations from the points in the extended domain of a variable to their images under the space-time mapping. For example, the images of soaking points of a variable are also called soaking points of that variable, and so on. This is justified by the injectivity of valid space-time mappings over the extended domain of a variable (Lemma 4.3). Therefore, for any variable V, the set of its soaking

points $\overline{\Psi}^{s}_{V}$, the set of its relaying points $\overline{\Psi}^{r}_{V}$, the set of its draining points $\overline{\Psi}^{d}_{V}$ and the set of computation points $\overline{\Phi}_V$ form a partition of $\overline{\Psi}_V$. We refer to the points of $\complement_{\Upsilon}\overline{\Psi}_V$ as the *undefined points of* V.

Lemma 4.4 $\{\overline{\Psi}^{s}_{V}, \overline{\Psi}^{r}_{V}, \overline{\Psi}^{d}_{V}, \overline{\Phi}_V, \complement_{\Upsilon}\overline{\Psi}_V\}$ *is a partition of* Υ.

Proof. By Def. 4.3, $\{\Psi^{s}_{V}, \Psi^{r}_{V}, \Psi^{d}_{V}, \Phi_V\}$ is a partition of Ψ_V. Since valid space-time mappings are injective over Ψ_V, $\{\overline{\Psi}^{s}_{V}, \overline{\Psi}^{r}_{V}, \overline{\Psi}^{d}_{V}, \overline{\Phi}_V\}$ is a partition of $\overline{\Psi}_V$. The rest of the proof follows from the fact that $\complement_{\Upsilon}\overline{\Psi}_V$ and $\overline{\Psi}_V$ are complements in Υ. □

We introduce the concept of a path. Let $S \subset \mathbf{Q}^n$ and $\vartheta \in \mathbf{Q}^n$. We refer to the sequence of all points in S such that the difference between any point in the sequence and its immediate predecessor (if any) is ϑ as a *ϑ-path* or *path of S*. $p(S, \vartheta, I)$ denotes the ϑ-path of S that passes through point I. For convenience, we shall identify ϑ-path $p(S, \vartheta, I)$ by its constitute elements:

$$p(S, \vartheta, I) \;=\; S \cap \mathrm{ray}(I, \pm\vartheta)$$

We write $\mathrm{paths}(S, \vartheta)$ for the set of all ϑ-paths associated with a set S:

$$\mathrm{paths}(S, \vartheta) \;=\; \{p(S, \vartheta, I) \mid I \in S\}$$

So $p(S, \vartheta, I) = p(S, \vartheta, J)$ iff I and J are in the same ϑ-path. When we speak of a ϑ_V-path for variable V, we mean a ϑ_V-path of Φ_V. The source (target) of a ϑ_V is a first (last) computation point of V. Operationally, a ϑ_V-path contains the points at which a given element of variable V is successively computed. When we speak of a $\overline{\vartheta}$-path (with an overbar), we mean a path of Υ.

From the space-time diagram for a one-dimensional array (Fig. 4–4), we can directly extract the following information: the latency, the number of cells required, the velocities of variables, and input and output characteristics. To find out how the input and output data are handled in the systolic array, we note that the elements of variable V must be injected at the first points of $\overline{\vartheta}_V$-paths and ejected at the last points of $\overline{\vartheta}_V$-paths. Some $\overline{\vartheta}_V$-paths of V may not start at input cells in the following two cases:

- V is a moving. The input data for these paths must be $\perp$; they can be disregarded; i.e., they need not be supplied from the external environment.

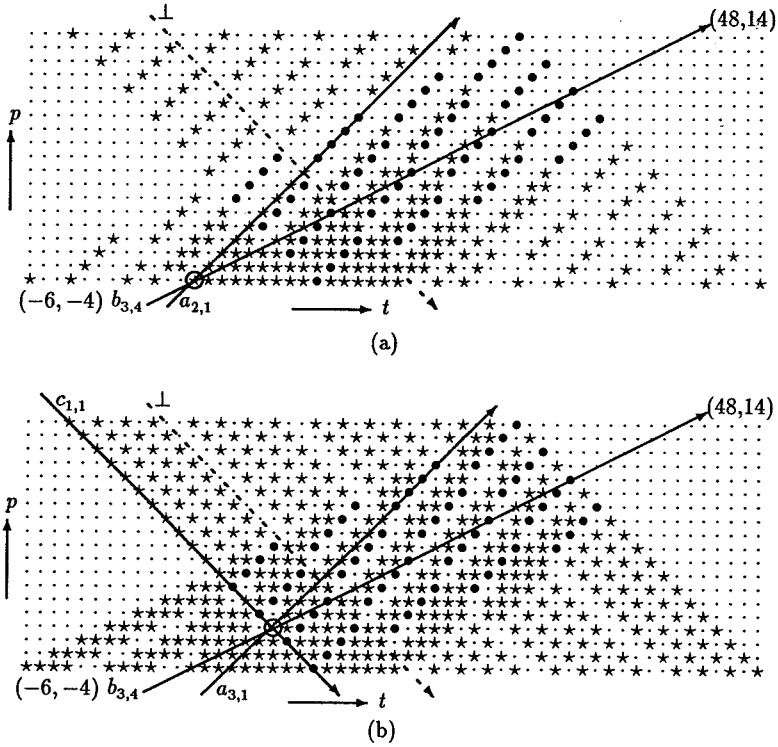


Figure 4–4: The space-time diagrams with respect to (a) the π-model and (b) the τ-model that are described by the following space-time mapping [46] ($m=4$):

$$\pi_{\text{even}} = \begin{bmatrix} 2m-2 & 1 & m/2 \\ m-1 & 1 & -m/2 \end{bmatrix} \qquad \pi_{\text{odd}} = \begin{bmatrix} 2m & 1 & (m+1)/2 \\ m & 1 & -(m+1)/2 \end{bmatrix}$$

π_{even} applies for even m and π_{odd} for odd m. The point at the bottom-left corner is $(t_{\text{fst}}, p_{\text{min}}) = (-6, -4)$, that at the top-right corner is $(t_{\text{lst}}, p_{\text{max}}) = (48, 14)$. The fat dots represent computation points. The regular dots and stars represent pipelining points; the stars depict the pipelining points at which at least one element is propagated but exclude those that only propagate data elements of A and B which are no longer used, since the output of A and B is of no interest. The pipelining points highlighted by circles are referred to in this section and in Sect. 4.4. Arrows depict related paths. Their tails are labelled with the corresponding input data.

- V is a stationary. The input data for these paths must be loaded before the computation starts, i.e., before step t_{fst}.

The two sets $\text{fst}(\Phi_V, \vartheta_V)$ and $\text{paths}(\Phi_V, \vartheta_V)$ are isomorphic by the bijection:

$$\eta_V : \text{fst}(\Phi_V, \vartheta_V) \rightarrow \text{paths}(\Phi_V, \vartheta_V), \quad \eta_V(I) = p(\Phi_V, \vartheta_V, I) \tag{4.9}$$

The communication constraint states that at most one element of a variable can be injected at any input cell at a given step. Expressed in terms of the mapping from $\text{paths}(\Phi_V, \vartheta_V)$ to $\text{paths}(\Upsilon, \overline{\vartheta}_V)$, this means that different ϑ_V-paths must have different images, i.e., different $\overline{\vartheta}_V$-paths in the space-time diagram. This is not true if V is stationary, because more than one input value, i.e., more than one ϑ_V-path may be projected to the same cell.

Lemma 4.5 *For a moving variable V, let $\mu_V : \text{paths}(\Phi_V, \vartheta_V) \rightarrow \text{paths}(\Upsilon, \overline{\vartheta}_V)$, $\mu_V(p(\Phi_V, \vartheta_V, I)) = p(\Upsilon, \overline{\vartheta}_V, \overline{I})$. The communication constraint is satisfied iff*

$$(\forall\, V : V \in \mathcal{V} : \mu_V : \text{paths}(\Phi_V, \vartheta_V) \rightarrowtail \text{paths}(\Upsilon, \overline{\vartheta}_V))$$

Proof.

$$\begin{aligned}
& (\forall\, V : V \in \mathcal{V} : \mu_V : \text{paths}(\Phi_V, \vartheta_V) \rightarrowtail \text{paths}(\Upsilon, \overline{\vartheta}_V)) \\
\Longleftrightarrow\ & \{\eta_V \text{ of } (4.9)\} \\
& \mu_V \circ \eta_V : \text{fst}(\Phi_V, \vartheta_V) \rightarrowtail \text{paths}(\Upsilon, \overline{\vartheta}_V) \\
\Longleftrightarrow\ & \{\text{Definition of } \overline{\vartheta}_V\text{-path}\} \\
& (\forall\, V : V \in \mathcal{V} : (\forall\, S : S \in \text{fst}(\Phi_V, \vartheta_V) / \textcircled{I}_V : \textsf{input} : S \rightarrowtail \mathbf{Z})) \qquad \square
\end{aligned}$$

Next, we characterise the distribution of pipelining points in the space-time diagram. There are two types of $\overline{\vartheta}_V$-paths (Fig. 4–4):

- Paths that contain computation points. Each such path can be viewed as consisting of the following three consecutive subpaths:
  - The first subpath consists of soaking points of V.
  - The second subpath consists of relaying points of V and computation points. There are $\Gamma_V - 1$ relaying points between every two neighbouring computation points:

$$\begin{aligned}
\Gamma_V &= \textbf{if } \textsf{flow}(V) = 0 \rightarrow \lambda\vartheta_V \ [\!]\ \textsf{flow}(V) \neq 0 \rightarrow 1 \textbf{ fi} && (\pi\text{-model}) \\
\Gamma_V &= \textbf{if } \textsf{flow}(V) = 0 \rightarrow \lambda\vartheta_V \ [\!]\ \textsf{flow}(V) \neq 0 \rightarrow |\sigma\vartheta_V| \textbf{ fi} && (\tau\text{-model})
\end{aligned} \tag{4.10}$$

(E.g., see the ϑ_B-path depicted by the solid arrow in Fig. 4–4(b); in this case, $\Gamma_B = 3$.)

- The third subpath consists of draining points of V.

- Paths that contain undefined points of V only. (E.g., see the ϑ_C-paths depicted by the dashed arrows in Figs. 4–4(a) and (b).)

Remark. If V is stationary variable, the definitions of $\overline{\vartheta}_V$ in both models are consistent with the definitions of Γ_V. In this case, each element of V is accessed every Γ_V steps. For any $\overline{\vartheta}_V$-path in the space-time diagram, there are $\Gamma_V - 1$ relaying points between every two neighbouring computation points. □

Lemma 4.3 indicates that the communication constraint is to ensure that the space-time mapping is injective over Ψ_V. However, valid space-time mappings from Ψ to Υ are generally not injective; one example is the mapping displayed in Fig. 4–4. Take the pipelining point $\overline{I} = (12, -1)$ highlighted in Fig. 4–4(b) as an example. Elements $a_{3,1}$, $b_{3,4}$, and $c_{1,1}$ are propagated at this point, since the paths $p(\Upsilon, \overline{\vartheta}_A, \overline{I})$, $p(\Upsilon, \overline{\vartheta}_B, \overline{I})$ and $p(\Upsilon, \overline{\vartheta}_C, \overline{I})$ intersect there. In the extended source UREs, $A(3, -8, 1) = a_{3,1}$, $B(1/3, 4, 3) = b_{3,4}$ and $C(1, 1, 5/2) = c_{1,1}$, where $\pi(3, -8, 1) = \pi(1/3, 4, 3) = \pi(1, 1, 5/2) = \overline{I}$. Point $\overline{I}$ has three inverse images in the extended index space. In general, a space-time point may have as many inverse images (in the extended index space) as there are data variables – one inverse image per variable. We also note that the inverse images of the three paths, i.e., $p(\Psi_A, \delta_A, (3, -8, 1))$, $p(\Psi_B, \delta_B, (1/3, 4, 3))$, and $p(\Psi_C, \delta_C, (1, 1, 5/2))$ do not intersect. A similar analysis applies for the pipelining point $\overline{I} = (6, -4)$ highlighted in Fig. 4–4(a). Elements $a_{2,1}$ and $b_{3,4}$ are propagated at this point.

Theorem 4.3 *A valid space-time mapping may not be injective from Ψ to Υ.*

Proof. The space-time mapping in Fig. 4–4 is not injective from Ψ to Υ. □

This theorem is not valid for $(n-1)$-dimensional arrays because a valid space-time mapping, in this case, is a bijection from $\mathbf{Q}^n$ to $\mathbf{Q}^n$.

From now on, when we speak of the non-injectivity of space-time mappings for one-dimensional arrays without an explicit reference to the domains of the space-time mappings, we mean the non-injectivity of the space-time mappings over the extended index space Ψ.

In general, the non-injectivity of the space-time mapping increases the importance of control signals in one-dimensional arrays. Take the array shown in Fig. 4–4(b). For example, cell $p=-1$ must be instructed at step $t=12$ to propagate rather than accumulate element $c_{1,1}$. However, the non-injectivity of the space-time mapping is a virtue rather than an evil. It permits the parallelism in the propagation of data between cells to be fully exploited. If (t,p) is a pipelining point, cell p at step t is allowed to propagate up to $|\mathcal{V}|$ elements; one element for every variable in $\mathcal{V}$. If the injectivity of the space-time mapping from Ψ to Υ is enforced, a cell can propagate at most one element at a given step. This will certainly increase the latency of the array. On the other hand, propagation control signals may be eliminated for a special class of systolic algorithms; this is the topic of Chap. 6.

4.6 A Synthesis Procedure

Based on the results developed in the previous sections, we now present a synthesis procedure for the generation of valid space-time mappings for both models. The procedure first constructs $n\times n$ matrices and then converts them to $2\times n$ matrices, which are valid space-time matrices.

First, we construct a space-time matrix, denoted Π, that is an $n\times n$ integer matrix, whose first $n-1$ rows, denoted by Λ, form a *scheduling matrix* and whose last row is the allocation vector.

$$\Pi = \begin{bmatrix} \Lambda \\ \sigma \end{bmatrix} = \begin{bmatrix} \Lambda_1 \\ \Lambda_2 \\ \cdots \\ \Lambda_{n-1} \\ \sigma \end{bmatrix} = \begin{bmatrix} \Lambda_{1,1} & \Lambda_{1,2} & \cdots & \Lambda_{1,n} \\ \Lambda_{2,1} & \Lambda_{2,2} & \cdots & \Lambda_{2,n} \\ & & \cdots & \\ \Lambda_{n-1,1} & \Lambda_{n-1,2} & \cdots & \Lambda_{n-1,n} \\ \sigma_1 & \sigma_2 & \cdots & \sigma_n \end{bmatrix}$$

$$\overline{I} = \Pi I = (t_1, t_2, \cdots, t_{n-1}, p)$$

Point I is computed at cell p at multi-dimensional time step $(t_1, t_2, \cdots, t_{n-1})$. To realise the systolic array in hardware, we need an $(n-1)$-dimensional clock, whose most (least) significant hand represents t_1 (t_{n-1}). The scales for the $n-1$

hands of the clock are represented by a vector $(s_1, s_2, \cdots, s_{n-1})$. One unit of the i-th hand is equal to s_{i+1} units of the $(i+1)$-st hand. s_i is the difference between the maximum and minimum of the i-th time components of all space-time points. We shall later provide a procedure for deriving these scales.

Theorem 4.4 *Let the validity of space-time mappings be defined in the absence of the delay rule of Def. 2.4. A space-time mapping Π is valid for the source UREs if*

- $(\forall\ V : V \in \mathcal{V} : \Lambda\vartheta_V > 0)$ *(Precedence Constraint)*
- $\text{rank}(\Pi) = n$ *(Computation and Communication Constraint)*

Proof. The proof of the precedence constraint is similar to that of Thms. 4.1 and 4.2, except we need to consider $(n-1)$-dimensional time. When Π is non-singular, it is a bijection from $\mathbf{Q}^n$ to $\mathbf{Q}^n$. The satisfaction of the computation and communication constraint is obvious. □

This theorem is due to Moldovan [57]. The mapping conditions depend on the data dependence vectors but not on the index space. The search for a space-time mapping Π that satisfies these conditions can be formulated as an integer programming problem [70]. The delay rule of Def. 2.4 is not part of mapping conditions of Thm. 4.4. It can be trivially satisfied by an appropriate scaling of the scheduling vector that is converted from a scheduling matrix.

Next, we convert the $(n-1) \times n$ scheduling matrix to a scheduling vector by a procedure that depends on the index space or, more precisely, on the extended index space. This is accomplished by a calculation of scales $s_1, s_2, \cdots, s_{n-1}$, from which we obtain a vector $(g_1, g_2, \cdots, g_{n-1})$ such that one unit of the i-th component is equal to g_i units of the $(n-1)$-st component. This vector is calculated by setting $g_{n-1} = 1$ and $g_i = g_{i+1} s_{i+1}$. By setting g_{n-1} to 1, we assume that $(0, 0, \cdots, 0, 1)$ is the smallest unit of the $(n-1)$-dimensional clock. (Scale s_1 does not contribute to the calculation of the vector $(g_1, g_2, \cdots, g_{n-1})$.) Then $(n-1)$-dimensional time $(t_1, t_2, \cdots, t_{n-1})$ is converted to scalar time $t = (\sum\ i : 0 < i < n : t_i g_i)$.

Procedure 4.1 (Conversion of $(n-1)$-dimensional to one-dimensional time)

INPUT: The extended index space Ψ and a scheduling matrix Λ.

OUTPUT: A scheduling vector λ.

1. $\left(\forall\, i : 0<i<n : \Delta_i = \begin{cases} \textbf{if } (\forall\, I,J : I,J \in \Psi : \Lambda_i I = \Lambda_i J) \rightarrow 1 \\ [] \textbf{ else} \rightarrow (\min\, I,J : I,J \in \Psi \wedge \Lambda_i I \neq \Lambda_i J : |\Lambda_i(I-J)|) \\ \textbf{fi} \end{cases}\right)$

2. $(\forall\, i : 1<i<n : s_i = (\max\ I,J : I,J \in \Psi : |\Lambda_i(I-J)| + \Delta_i))$

3. $g_{n-1} = 1/\Delta_{n-1},\ (\forall\, i : 0<i<n-1 :\ g_i = g_{i+1}s_{i+1}/\Delta_i)$

4. $(\forall\, k : 0<k \leqslant n :\ \lambda_k = (\sum\ i : 0<i<n : g_i \Lambda_{i,k}))$ □

Let us explain the rôle that quantities $\Delta_1, \Delta_2, \cdots, \Delta_{n-1}$ play in Proc. 4.1. Δ_i represents the smallest difference among the i-th time components of all space-time points. If all the i-th time components are integers, then $\Delta_i \in \mathbf{Z}^+$. Δ_i can be disregarded (i.e., treated as 1). But, retaining Δ_i minimises the components $g_1, g_2, \cdots, g_i$ and thus the latency of the systolic array. If the i-th time components of some space-time points are not integers (because some points of Ψ for the τ-model are not integers), Δ_i may not be integral. Δ_i is essential in ensuring that different multi-dimensional time steps are converted to different scalar time steps.

Lemma 4.6 *If* Π *satisfies Thm. 4.4, the transformation of* Λ *to* λ *by Proc. 4.1 preserves the following properties:*

1. $(\forall\, I,J : I,J \in \Psi : \Lambda I = \Lambda J \implies \lambda I = \lambda J)$ *(Injectivity)*

2. $(\forall\, I,J : I,J \in \Psi : \Lambda I < \Lambda J \implies \lambda I < \lambda J)$ *(Precedence)*

3. $(\forall\, V : V \in \mathcal{V} : (\forall\, I,J : I,J \in \Psi : \Lambda I = \Lambda J + \Lambda \vartheta_V \implies \lambda I = \lambda J + \lambda \vartheta_V))$ *(Linearity)*

Proof. Letting $G = (g_1, g_2, \cdots, g_{n-1})$ gives rise to $\lambda = G\Lambda$ (Step 4 of Proc. 4.1). Stats. 1 and 3 of the theorem are true because λ is a linear combination of the row vectors of Λ. To prove Stat. 2 of the theorem, we assume that $\Lambda I < \Lambda J$ for two arbitrary points I and J in Ψ.

$$
\begin{aligned}
& \lambda I - \lambda J \\
= & \ \{\lambda = G\Lambda \text{ (Step 4 of Proc. 4.1)}\} \\
& G\Lambda I - G\Lambda J
\end{aligned}
$$

$= \{\text{Algebraic manipulation}\}$

$(\sum\ i : 0<i<n : g_i\Lambda_i(I-J))$

$= \{\text{Assume } \Lambda_k(I-J)<0 \text{ for } 0<k<n \text{ and } (\forall\ i : 0<i<k : \Lambda_i(I-J)=0)\}$

$g_k\Lambda_k(I-J)+(\sum\ i : k<i<n : g_i\Lambda_i(I-J))$

$\leqslant \{\text{By the definition of } \Delta_k \text{ in Step 1 of Proc. 4.1, } \Lambda_k(I-J)\leqslant -\Delta_k\}$

$-g_k\Delta_k+(\sum\ i : k<i<n : g_i\Lambda_i(I-J))$

$\leqslant \{\text{By the definition of } s_i \text{ in Step 2 of Proc. 4.1, } \Lambda_i(I-J)\leqslant (s_i-\Delta_i)\}$

$-g_k\Delta_k+(\sum\ i : k<i<n : g_i(s_i-\Delta_i))$

$= \{\text{By Step 3 of Proc. 4.1, } g_i\Delta_i=g_{i+1}s_{i+1}\text{; algebraic manipulation}\}$

$-g_{n-1}\Delta_{n-1}$

$= \{\text{By Step 3 of Proc. 4.1, } g_{n-1}=1/\Delta_{n-1}\}$

-1

$< \{-1<0\}$

0

□

Procedure 4.2 (Construction of the space-time mappings for the source UREs with respect to either the π-model or the τ-model)

INPUT: The source UREs.

OUTPUT: Valid space-time mappings π.

1. Find space-time mappings Π that satisfy Thm. 4.4.
2. Obtain the extended index spaces Ψ by Def. 4.3 (with respect to either the π-model or the τ-model).
3. Transform Λ to λ by Proc. 4.1.
4. Scale λ to satisfy the delay constraint for the τ-model. □

Theorem 4.5 *The space-time mapping π returned by Proc. 4.1 is valid.*

Proof. If a space-time mapping Π satisfies Thm. 4.4, π satisfies the precedence and computation constraint by Lemma 4.6. Scaling λ ensures that π satisfies the delay constraint. We remain to prove that π satisfies the communication constraint.

Π is non-singular

$\Longrightarrow$ {Non-singularity}

$\Pi : \mathbf{Q}^n \rightarrowtail \mathbf{Q}^n$

$\Longrightarrow$ {Restriction of Π to a subset of $\mathbf{Q}^n$}

$\Pi : \mathrm{fst}(\Psi_V, \vartheta_V) \rightarrowtail \mathbf{Q}^n$

$\Longrightarrow$ {$\textcircled{1}'_V$ of (4.8)}

$(\forall\ S : S \in \mathrm{fst}(\Phi_V, \vartheta_V) / \textcircled{1}'_V : \Pi : S \rightarrowtail \mathbf{Q}^n)$

$\Longrightarrow$ {Lemma 4.6}

$(\forall\ S : S \in \mathrm{fst}(\Phi_V, \vartheta_V) / \textcircled{1}'_V : \pi : S \rightarrowtail \mathbf{Q}^2)\}$

$\Longleftrightarrow$ $\{(\forall\ I, J : I, J \in S : \mathsf{place}(I) = \mathsf{place}(J))\}$

$(\forall\ S : S \in \mathrm{fst}(\Phi_V, \vartheta_V) / \textcircled{1}'_V : \mathsf{step} : S \rightarrowtail \mathbf{Q}))\}$

$\Longrightarrow$ {Appropriate scaling of λ; Lemma 4.2}

The communication constraint is satisfied □

The space-time mappings generated by Proc. 4.2 may be inefficient because Proc. 4.1 preserves the bijectivity of the space-time mapping Π over the extended index space (Stats. 1 and 2 of Lemma 4.6). This follows from two observations. First, the parallel injection and ejection of the elements of different variables at the same cell is excluded by Proc. 4.1. But the communication constraint only requires the injectivity of **step** over every $\textcircled{1}'_V$-class. Second, Thm. 4.5 still holds if we allow two points scheduled at different multi-dimensional time steps and at different cells to be converted to the same scalar time step, provided that the precedence rule is not violated.

4.7 Related Work

Depending on how the scheduling vector is derived, we distinguish two methods for the synthesis of data flow for one-dimensional arrays: the one-dimensional method searches the scheduling vector directly, and the multi-dimensional method does so via an $(n-1)$-dimensional scheduling matrix. In the one-dimensional method, one applies a $2 \times n$ space-time matrix π to the source UREs. In the transformed UREs, one index represents time and the other space. In the multi-dimensional method,

one applies an $n \times n$ space-time matrix Π to the source UREs. In the transformed UREs, $n-1$ indices represent time and the remaining index represents space. Then, the $(n-1)$-dimensional scheduling matrix is converted to a one-dimensional scheduling vector.

Ramakrishnan, Fussell and Silberschatz [67] were the first to introduce the τ-model. Their synthesis of the data flow amounts to embedding two- or three-dimensional data dependence graphs in a space-time diagram. The correctness of an embedding is ensured by certain embedding rules, which are equivalent in nature to the mapping conditions specified in Thm. 4.1. However, this graph embedding approach does not lead to a systematic embeddings of any UREs. This method has been recently improved by the one-dimensional method proposed by Lee and Kedem [46]. They adopt the τ-model and formulate the previous embedding rules in terms of relations on the data dependence vectors (Thm. 4.2). This data dependence approach is superior to the previous graph embedding approach in that it applies for any n-dimensional UREs and that the search for valid space-time mappings can be formulated as linear and integer programming. However, Lee and Kedem did not give an intuitive explanation for the necessity of the mapping condition that is referred to here as the communication constraint. This might be the reason why they failed to recognise that, in general, the communication constraint implies the computation constraint (Lemma 4.1). (The way that the communication constraint is formulated here clearly indicates its irrelevance for $(n-1)$-dimensional arrays and its necessity for systolic arrays of reduced dimension.)

Rao [70] presents a multi-dimensional method without providing a model; his concept of extended index space conforms to the π-model but not the τ-model. Kumar and Tsai [37] present a method that converts two-dimensional arrays to one-dimensional arrays. They adopt the π-model, although this is not explicitly mentioned. Wong and Delosme [86] present a procedure for converting $(n-2)$-dimensional to one-dimensional time in the context of the synthesis of two-dimensional arrays. The two-dimensional arrays they consider conform to the systolic array model of Def. 2.2. The index space rather than the extended index space is used in the conversion. Hence, input and output data may have to be handled at internal cells.

4.8 Conclusion

We have formally defined the two most frequently used one-dimensional array models: the π-model allows non-neighbouring connections and the τ-model does not. We have extended previously known mapping conditions to both models. This allows synthesis methods formulated for one model to be applied to the other model. For example, Lee and Kedem's one-dimensional method [46] can be extended to the π-model. The only requirement is that we must choose the allocation vector before the scheduling vector, because the definition of $\textcircled{I}_V$ (or $\textcircled{I}'_V$) in the communication constraint depends on the allocation vector.

The synthesis procedure (Proc. 4.2) is built on previously known results, but it applies for both models. Valid space-time mappings can be obtained for either of the two models, depending on how the extended index space is defined.

We have represented the communication constraint in a variety of forms: each equivalently enforces the restriction that a channel cannot transfer more than one datum per step but conveys the properties of the space-time mapping from a distinct perspective. For example, Lemma 4.5 justifies the geometrical approach of [11] in which one-dimensional arrays are constructed by a manual embedding of ϑ_V-paths in the space-time diagram.

We have investigated properties characteristic of one-dimensional arrays. First, we have characterised the regular distribution of pipelining points over the space-time diagram. This will be the basis of the construction of propagation control signals for one-dimensional arrays in Chap. 5. Second, we have shown how the non-injectivity of a space-time mapping can be understood by an extension of the index space. The non-injectivity of the space-time mapping makes the implementation of the control UREs inefficient. This is our reason to look for a different but more efficient specification of propagation control signals for one-dimensional arrays in Chap. 5.

The concept of the extended index space can be used in the optimisation of a schedule. The points that are mapped to t_{fst} and t_{lst} must be extreme points in $(\bigcup\ V : V \in \mathcal{V} : \text{fst}(\Psi_V, \vartheta_V) \cup \text{lst}(\Psi_V, \vartheta_V))$. Since a valid space-time mapping must satisfy the precedence constraint, some extreme points can be excluded as

candidates. Assume that F (L) is the set of extreme points that may be scheduled at the first (last) step t_{fst} (t_{lst}). A space-time mapping that minimises the latency must minimise the objective function:

$$(\max\ I : I \in L : \lambda I) - (\min\ I : I \in F : \lambda I)$$

The computation constraint aims at preventing intra-cell parallelism. This is enforced by requiring too much: no two points that are scheduled simultaneously can be mapped to the same cell. The necessary and sufficient condition should state that no concurrent computations *for a fixed variable* can be mapped to the same cell. (So, two points can share the same image iff every variable is undefined at at least one of the two points). This is guaranteed by the communication constraint. Therefore, the computation constraint can be neglected.

The communication constraint defined in this chapter is overconstrained. Two ϑ_V-paths may be mapped to the same $\overline{\vartheta}_V$-path if, for example, variable V is undefined at all the points in one of the two paths. In general, two ϑ_V-paths can share the same image iff, for every pair of points (one point per path) that share the same image, variable V is undefined at at least one of the two points, i.e., iff

$$(\forall\ x, y : x, y \in \text{paths}(\Psi_V, \vartheta_V) :$$
$$\overline{x} = \overline{y} \implies (\forall\ I, J : I \in x \wedge J \in y : \overline{I} = \overline{J} \implies V(\overline{I}) = \bot \vee V(\overline{J}) = \bot))$$

Unfortunately, the verification of this formula depends on both the space-time mapping and the index space. In addition, more complex control is required.

Finally, let us consider the impact of the one-dimensional method and the multi-dimensional method on the latency of the array. With the one-dimensional method, one searches for the scheduling vector directly. Hence, the set of points that are scheduled concurrently can be $(n-1)$-dimensional. With the multi-dimensional method, one first searches for a scheduling matrix and then converts it to a scheduling vector. Because the scheduling matrix is of rank $n-1$, the set of points that are scheduled concurrently can only be one-dimensional. This reduced amount of parallelism is retained when the scheduling matrix is converted to a vector using Proc. 4.1 (Stats. 1 and 2 of Lemma 4.6). Therefore, it is useful to find a new conversion procedure that increases the reduced parallelism specified by the scheduling matrix.

5

Control Flow Synthesis for One-Dimensional Systolic Arrays

5.1 Introductory Remarks

In Chap. 3, we have provided a constructive method for the synthesis of control signals for $(n-1)$-dimensional arrays from n-dimensional UREs. The basic idea is to transform the source UREs to pipelined UREs. Then, finding $(n-1)$-dimensional arrays is just finding space-time mappings that are valid with respect to the pipelined UREs and the $(n-1)$-dimensional array model.

In Chap. 4, we have defined two one-dimensional array models and stated mapping conditions for the synthesis of data flow from n-dimensional UREs for these two models. The pipelined UREs constructed in Chap. 3 are a special form of UREs. Just like for the synthesis of $(n-1)$-dimensional arrays, finding one-dimensional arrays is just finding valid space-time mappings with respect to the pipelined UREs and one-dimensional array models.

This chapter is concerned with the construction of control UREs for one-dimensional arrays. To illustrate why the specification of the control UREs constructed previously may result in inefficient one-dimensional arrays, we state in a theorem the mapping conditions for the validity of space-time mappings with

respect to the pipelined UREs and the two one-dimensional array models; this theorem is a direct application of Thms. 4.1 and 4.2 to the pipelined UREs. From now on, we write $\mathcal{V}_c$ ($\mathcal{V}_p$) for the set of computation (pipelining) control variables, and $\mathcal{V}_d$ for the set of data variables. $\mathcal{V}$ is the union of $\mathcal{V}_c$, $\mathcal{V}_p$ and $\mathcal{V}_d$.

Theorem 5.1 *A space-time mapping is valid for the pipelined UREs (in either the π-model or the τ-model) iff*

- $(\forall\, V : V \in \mathcal{V} : \lambda\vartheta_V > 0)$ *(Precedence Constraint)*
- $\pi : \Phi \rightarrowtail \mathbf{Z}^2$ *(Computation Constraint)*
- $(\forall\, V : V \in \mathcal{V} : \ \sigma\vartheta_V \mid \lambda\vartheta_V)$ *(Delay Constraint for the χ-Model)*
- $(\forall\, V : V \in \mathcal{V}_d : (\forall\, S : S \in \mathrm{fst}(\Phi_V, \vartheta_V)/\mathbb{I}_V : \mathsf{input} : S \rightarrowtail \mathbf{Z}))$ (5.1)

  $(\forall\, V : V \in \mathcal{V}_c : (\forall\, S : S \in \mathrm{fst}(\Psi_V, \vartheta_V)/\mathbb{I}'_V : \mathsf{input} : S \rightarrowtail \mathbf{Z}))$ (5.2)

  $(\forall\, V : V \in \mathcal{V}_p : (\forall\, S : S \in \mathrm{fst}(\Psi_V, \vartheta_V)/\mathbb{I}'_V : \mathsf{input} : S \rightarrowtail \mathbf{Z}))$ (5.3)

  (Communication Constraint)

Proof. Thms. 4.1 and 4.2. □

Domain Ψ_V of a control variable V in (5.2) and (5.3) is defined in (3.24):

$$\Psi_V \;=\; \mathrm{planes}(\Phi, \sigma) \cap \mathrm{rays}(D_V, \pm\vartheta_V) \tag{5.4}$$

D_V is the set of points I such that $V(I)$ is referenced in the guards of the pipelined UREs. It has the following relationship with the index space Φ and the extended index space Ψ (Sect. 3.4):

$$(\forall\, V : V \in \mathcal{V}_c : \mathrm{rays}(D_V, \pm\vartheta_V) \subseteq \mathrm{rays}(\Phi, \pm\vartheta_V)) \tag{5.5}$$

$$(\forall\, V : V \in \mathcal{V}_p : \mathrm{rays}(D_V, \pm\vartheta_V) = \mathrm{rays}(\Psi \cap \mathbb{C}_{\mathbf{Z}^n} \mathsf{H}_V, \pm\vartheta_V) \cup \mathrm{rays}(\Phi, \pm\vartheta_V)) \tag{5.6}$$

The communication constraint for a variable ensures that all input data of the variable are input at distinct time steps. In general, the higher the number of input data of a variable, the higher the latency of the systolic array. This makes it desirable to keep the number of inputs of a control variable as small as possible.

The CCUREs serve to distinguish different types of computation points. They need only be defined for the index space. $\mathrm{fst}(\Psi_V, \vartheta_V)$ of a computation control variable V contains no more elements than $\mathrm{fst}(\Phi, \vartheta_V)$. The latency of the systolic

array may be retained when it is possible to choose data dependence vectors as the control dependence vectors for computation control variables (Sect. 3.5). The PCUREs serve to distinguish pipelining points from computation points. They must be defined for the extended index space. However, $\mathrm{fst}(\Psi_V, \vartheta_V)$ of a propagation control variable V contains more elements than $\mathrm{fst}(\Phi, \vartheta_V)$, leading to a higher latency of the systolic array. (This is not true for $(n-1)$-dimensional arrays because the corresponding mapping conditions depend only on data and control dependence vectors but not on the (extended) index space.) We avoid the higher latency by providing a different construction of the PCUREs. The domain of a propagation control variable V in the new PCUREs is defined for the extended index space Ψ. But, the construction of the new PCUREs will allow us to replace Ψ_V in the communication constraint (5.3) by a subset of Ψ called the *communication constraint domain* of V and denoted Ω_V:

$$\Omega_V = \mathrm{planes}(\Phi, \sigma) \cap \mathrm{rays}(D_V, \pm\vartheta_V) \tag{5.7}$$

where D_V satisfies (5.5), eliminating the need for (5.6). Before doing so, we illustrate with LU-decomposition why the behaviour of the original PCUREs is not preserved if we directly substitute Ω_V for Ψ_V of the communication constraint (5.3). This also indicates that the PCUREs play a more important rôle for one-dimensional arrays than for $(n-1)$-dimensional arrays.

Example 5.1 $m \times m$ LU-Decomposition

Recall the pipelined UREs constructed in Sect. 3.7.2. The CCUREs consist of two computation control variables P and Q. The PCUREs were not presented. They follow from the set $\mathfrak{B} = \{B_1, B_2, B_3\}$ given in Sect. 3.7.1.3 and (3.18). By our convention, we write P_{B_1}, P_{B_2} and P_{B_3} for the three corresponding propagation control variables. By (3.19), we obtain

$$\begin{aligned} \mathcal{P}(\Phi) &= (\forall\, i : 0 < i \leqslant 3 : \mathsf{P}_{B_i}(I) = 0) \\ \mathcal{P}(\complement_\Psi \Phi) &= (\exists\, i : 0 < i \leqslant 3 : \mathsf{P}_{B_i}(I) = 1) \end{aligned} \tag{5.8}$$

$\mathcal{P}(\Phi)$ holds at point I iff $I \in \Phi$. (That is, $\mathcal{P}(\complement_\Psi \Phi)$ holds at point I iff $I \in \complement_\Psi \Phi$.) For the purpose of illustration, we choose $\vartheta_P = \vartheta_{\mathsf{P}_{B_1}} = \vartheta_{\mathsf{P}_{B_3}} = (0,1,0)$ and $\vartheta_Q = \vartheta_{\mathsf{P}_{B_2}} = (1,0,0)$. The following space-time mapping, which describes Ramakrishnan and Varman's one-dimensional array for matrix product, is valid by Thm. 5.1 if we

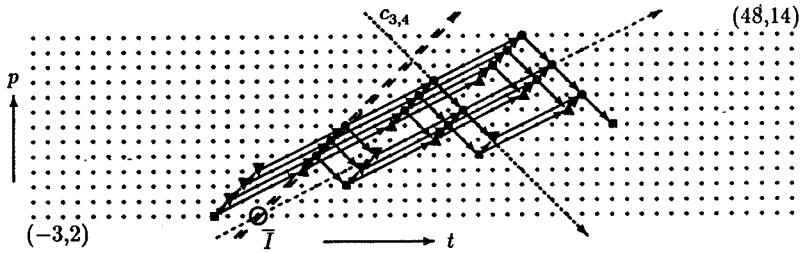


Figure 5–1: The space-time diagram for LU-decomposition for both one-dimensional array models ($m = 4$). The two (identical) long-dashed paths depict $p(\Upsilon, \overline{\vartheta}_{\mathsf{P}_{B_1}}, \overline{I})$ and $p(\Upsilon, \overline{\vartheta}_{\mathsf{P}_{B_3}}, \overline{I})$; they are equal and should be viewed as overlapping each other. The short-dashed path depicts $p(\Upsilon, \overline{\vartheta}_{\mathsf{P}_{B_2}}, \overline{I})$. The dotted path will be referred to in Sect. 5.3.

replace Ψ_V of (5.3) by Ω_V of (5.7) (Fig. 5–1):

$$\pi_{\text{even}} = \begin{bmatrix} 2m-2 & 1 & m/2 \\ m-1 & 1 & -m/2 \end{bmatrix} \quad \pi_{\text{odd}} = \begin{bmatrix} 2m & 1 & (m+1)/2 \\ m & 1 & -(m+1)/2 \end{bmatrix} \tag{5.9}$$

In the resulting systolic array, control variables P, P_{B_1} and P_{B_3} each travel at the same velocity as data variable A, and control variables Q and P_{B_2} each travel at the same velocity as data variable B.

Two observations can be made on this example. First, the behaviour of the PCUREs is not preserved by the chosen space-time mapping. Consider the three dashed paths pointing towards the northeast shown in Fig. 5–1. If we input control signal 0 at each of these three paths, the pipelining point highlighted by a circle will be interpreted as a computation point, since $\overline{\mathcal{P}(\Phi)}$ holds at this point. If we input control signal 1 at one of these three paths, then the computation points in the path will be interpreted as pipelining points, since $\overline{\mathcal{P}(\mathfrak{C}_\Psi \Phi)}$ holds at all points of the path. The problem is that two different $\vartheta_{\mathsf{P}_{B_i}}$-paths for a fixed i are mapped to the same $\overline{\vartheta}_{\mathsf{P}_{B_i}}$-path – a condition that is not permitted by the premises of Thm. 4.5.

Second, Lemma. 3.10 does not apply due to the non-injectivity of the space-time mapping. For example, $\overline{\mathcal{P}(\Phi_\bullet)}$ holds not only at the computation points highlighted by fat dots $\bullet$, as intended, but also at the pipelining point highlighted

by a circle in Fig. 5–1. This implies that $\overline{\Phi_\bullet} \neq \overline{\Phi'_\bullet}$, albeit $\Phi_\bullet = \Phi'_\bullet$. Thus, predicate $\mathcal{P}(\Phi)$ in the pipelined equation of (3.21) is essential and cannot be disregarded as in the case of $(n-1)$-dimensional arrays. So Thm. 3.3.2 is not valid. This indicates that propagation control is more important for one-dimensional arrays than for $(n-1)$-dimensional arrays. □

In this chapter, we focus on the construction of PCUREs for one-dimensional arrays. To do so, we need to lift one of the four restrictions imposed on the specification of control signals in Sect. 3.2, namely, Rst. 4. That is to say, we shall allow the control value of some propagation control variable to change during its propagation throughout the array.

Definition 5.1 A control variable is called an *evolution control variable* if some value of the variable changes before it is ejected from the array.

The rest of this chapter is organised as follows. Sect. 5.2 describes some notation and the basic idea underlying the construction of the PCUREs. Sect. 5.3 presents the construction of the PCUREs for three-dimensional UREs. Sect. 5.4 generalises this construction to n-dimensional UREs. (We do not consider two-dimensional UREs here; they have already been covered in Chap. 3.) Sect. 5.5 comments on some related work. Sect. 5.6 illustrates our method with two examples. Sect. 5.7 contains the conclusion of the chapter.

5.2 The Synthesis of Propagation Control Flow

To avoid unduly complex notation, we present the specification of propagation control signals in the form of programs (rather than in the form of UREs). The program for the host, called the *host program*, corresponds to the input equations and injects propagation control signals to the array. The program for the cells of the array, called the *cell program*, corresponds to the computation equations and specifies the control signals of the output channels of a cell based on the control signals of the input channels of the cell. The cell program is identical for all cells and is executed by every cell at every time step.

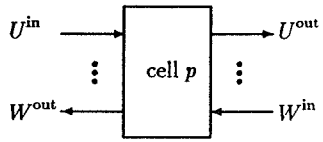


Figure 5–2: The notation for specifying the cell program.

How can UREs be expressed as programs? Since all cells are given the same cell program, we can abstract from the index vectors of propagation control variables. This results in one program that will be executed by every cell of the array.

Let us consider the situation where cell p is computing point $\overline{I}=(t,p)$ at step t (Fig. 5–2). The cell receives its input signal $V(\overline{I}-\overline{\vartheta}_V)$ of propagation control variable V at step $t-1$ and sends its output signal $V(\overline{I})$ at step t. We abstract from the index vectors of V by writing V^{in} for $V(\overline{I}-\overline{\vartheta}_V)$ and V^{out} for $V(\overline{I})$. That is, V^{in} (V^{out}) denotes the input (output) channel for V at this cell.

The domain of each propagation control variable V is the extended index space: $\Psi_V=\Psi$. So the domain of V in the index space is just Φ: $\Phi_V=\Phi$.

There are two types of propagation control variables V:

- *Pipelining Control Variables.* Every cell directly sends the control signal received at the input channel of V to its corresponding output channel. That is, the computation equation is given by

$$V^{out} = V^{in} \tag{5.10}$$

  If the cell is an input cell, V^{in} represents one of the values in sig(V) (which is the set of control values of V in Sect. 3.3) to be injected by the host program.

- *Evolution Control Variables.* Every cell may change the control signal received at the input channel of V before sending the (changed) control signal to the corresponding output channel. The computation equation is given by

$$V^{out} = \begin{cases} \text{if } B_1(W^{in},\cdots) & - \quad f_1(V^{in}) \\ [\!] \; B_2(W^{in},\cdots) & - \quad f_2(V^{in}) \\ & \cdots \\ [\!] \; B_b(W^{in},\cdots) & \rightarrow \quad f_b(V^{in}) \\ \text{fi} \end{cases} \tag{5.11}$$

The guard $B_i(W^{\text{in}}, \cdots)$ can always be written in disjunctive normal form, where each disjunct consists of a conjunction of tests of an argument (i.e., a propagation control variable) for a control signal. $f_i(V^{\text{in}})$ is a function that recursively defines the control value of V^{out} in terms of argument V^{in}.

Similarly, if the cell is an input cell, V^{in} represents one of the values in $\text{sig}(V)$ to be injected by the host program.

The propagation control flow is *correct* if there exists a construction of a characteristic function $\chi_{\overline{\Phi}}$ in the propagation control variables of $\overline{\Phi}$ in Υ:

$$\chi_{\overline{\Phi}} : \Upsilon \rightarrow \{\chi_c, \chi_p\}, \quad \chi_{\overline{\Phi}}(\overline{I}) = \textbf{if } \overline{I} \in \overline{\Phi} \rightarrow \chi_c \ [] \ \textbf{else} \rightarrow \chi_p \ \textbf{fi} \tag{5.12}$$

That is, $\overline{I}$ is a computation point if $\chi_{\overline{\Phi}}(\overline{I}) = \chi_c$ and a pipelining point otherwise. This amounts to defining predicates $\mathcal{P}(\Phi)$ and $\mathcal{P}(\complement_{\Psi}\Phi)$ of (3.21) as follows:

$$\begin{aligned} \overline{\mathcal{P}(\Phi)} &= \chi_{\overline{\Phi}}(\overline{I}) = \chi_c \\ \overline{\mathcal{P}(\complement_{\Psi}\Phi)} &= \chi_{\overline{\Phi}}(\overline{I}) = \chi_p \end{aligned}$$

The notation $\varphi(\overline{I})$ with $\overline{I} = (t, p)$ stands for the tuple of propagation control signals associated with the input channels of cell p at step $t-1$, one component per input channel.

$$\varphi(\overline{I}) \ = \ (V^{\text{in}}, \cdots) \ = \ (V(\overline{I} - \overline{\vartheta}_V), \cdots) \tag{5.13}$$

The basic idea underlying the construction of the PCUREs is to exploit the regular distribution of pipelining points in the space-time diagram. Recall that the space-time points of Υ are divided into five categories with respect to a fixed variable V (Lemma 4.4): the set of soaking points $\overline{\Psi}^{\text{s}}_V$, the set of relaying points $\overline{\Phi}^{\text{r}}_V$, the set of draining points $\overline{\Psi}^{\text{d}}_V$, the set of computation points $\overline{\Phi}_V$, and the set of undefined points $\complement_{\Upsilon}\overline{\Psi}_V$. By choosing one propagation control variable, say, V as a reference, we shall construct the PCUREs in such a way that the five sets

$$\{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Psi}^{\text{s}}_V\}, \{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Psi}^{\text{r}}_V\}, \{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Psi}^{\text{d}}_V\}, \{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Phi}_V\}, \{\varphi(\overline{I}) \mid \overline{I} \in \complement_{\Upsilon}\overline{\Psi}_V\}$$

are disjoint. Thus, the five types of points in the space-time diagram with respect to V are set apart. So are computation and pipelining points, since $\overline{\Phi}_V = \overline{\Phi}$.

The PCUREs constructed this way apply for both one-dimensional array models. We shall conduct our presentation with respect to the τ-model only. This is

justified since a space-time mapping that is valid with respect to the τ-model is also be valid with respect the π-model.

The construction of the PCUREs depends on the dimension of the index space. To provide a better understanding, we shall first present the PCUREs for three-dimensional UREs and then describe the generalisation to n-dimensional UREs.

We say a space-time mapping is *valid for a variable* if the computation constraint is satisfied and the precedence, delay and communication constraint for that variable are satisfied. Our construction of the PCUREs requires that all propagation control variables are moving. In Sect. 5.3.4, we shall state this constraint explicitly as part of the mapping conditions for the validity of space-time mappings. In the sections before, we shall assume this constraint implicitly.

5.3 The PCUREs for Three-Dimensional UREs

5.3.1 The Evolution Control Flow

We need to use only one evolution control variable; we name it E. Its specification exploits the regular distribution of pipelining points over the space-time diagram. In this section and subsequent sections, when we refer to soaking points, draining points, relaying points, undefined points, first computation points and last computation points with no explicit reference to a variable, we mean the points associated with the evolution control variable.

The PCUREs are supposed to ensure that the five sets

$$\{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Psi}^{\mathrm{s}}_E\}, \{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Psi}^{\mathrm{r}}_E\}, \{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Psi}^{\mathrm{d}}_E\}, \{\varphi(\overline{I}) \mid \overline{I} \in \overline{\Phi}_E\}, \{\varphi(\overline{I}) \mid \overline{I} \in \complement_{\Upsilon} \overline{\Psi}_E\}$$

are disjoint. Since $\Phi_E = \Phi$, we abbreviate $\mathrm{fst}(\Phi_E, \vartheta_E)$ to $\mathrm{fst}(\Phi, \vartheta_E)$ and $\mathrm{lst}(\Phi_E, \vartheta_E)$ to $\mathrm{lst}(\Phi, \vartheta_E)$. We classify the remaining propagation control variables, which are pipelining control variables, into two categories:

- *Initialisation Control Variables* $F_{1,1}, F_{1,2}, F_{2,1}, F_{2,2}, \cdots, F_{f,1}, F_{f,2}$. Let $\mathcal{B}(F)$ be a predicate in the initialisation control variables. We shall construct

initialisation control variables such that $\mathcal{B}(F)$ is a characteristic function of the first computation points of E:

$$\mathcal{B}(F) \;=\; \textbf{if } \overline{I} \in \overline{\mathrm{fst}(\Phi, \vartheta_E)} \rightarrow \textbf{true } [] \textbf{ else} \rightarrow \textbf{false fi} \qquad (5.14)$$

The idea behind the construction of the initialisation control variables is to partition $\mathrm{fst}(\Phi, \vartheta_E)$ into a number of subsets, $\mathrm{fst}(\Phi, \vartheta_E)_1$, $\mathrm{fst}(\Phi, \vartheta_E)_2$, $\cdots$, $\mathrm{fst}(\Phi, \vartheta_E)_f$, such that $\dim(\mathrm{fst}(\Phi, \vartheta_E)_i) \leqslant n-1$. We write

$$\textcircled{F} \;=\; \{\mathrm{fst}(\Phi, \vartheta_E)_i \mid 0 < i \leqslant f\}$$

One possible solution is to first decompose $\mathrm{fst}(\Phi, \vartheta_E)$ into the union of facets of the index space Φ and then build a partition from this decomposition. (These facets intersect at their boundaries.) In our construction, we associate two initialisation control variables, $F_{i,1}$ and $F_{i,2}$, with $\mathrm{fst}(\Phi, \vartheta_E)_i$. $\mathrm{sig}(F_{i,1}) = \{f_{i,1}, \overline{f}_{i,1}\}$. $\mathrm{sig}(F_{i,2}) = \{f_{i,2}, \overline{f}_{i,2}\}$. The reason for employing exactly two initialisation control variables will become clear later on in this section (Thm. 5.3).

- *Termination Control Variables* $L_1, L_2, \cdots, L_\ell$. Let $\mathcal{B}(L)$ be a predicate in the termination control variables. We shall construct termination control variables such that $\mathcal{B}(L)$ is a characteristic function of the last computation points of E:

$$\mathcal{B}(L) \;=\; \textbf{if } \overline{I} \in \overline{\mathrm{lst}(\Phi, \vartheta_E)} \rightarrow \textbf{true } [] \textbf{ else} \rightarrow \textbf{false fi} \qquad (5.15)$$

For reasons of symmetry, the construction of the termination control variables proceeds in a similar way as that of the initialisation control variables. We decompose $\mathrm{lst}(\Phi, \vartheta_E)$ into a number of unnecessarily disjoint subsets, $\mathrm{lst}(\Phi, \vartheta_E)_1, \mathrm{lst}(\Phi, \vartheta_E)_2, \cdots, \mathrm{lst}(\Phi, \vartheta_E)_\ell$, such that $\dim(\mathrm{lst}(\Phi, \vartheta_E)_i) \leqslant n-1$ and the union of these subsets is $\mathrm{lst}(\Phi, \vartheta_E)$. We write

$$\textcircled{L} \;=\; \{\mathrm{lst}(\Phi, \vartheta_E)_i \mid 0 < i \leqslant \ell\}$$

One possible solution is to decompose $\mathrm{lst}(\Phi, \vartheta_E)$ into the union of facets of the index space Φ. In our construction, we associate one termination control variable, L_i, with $\mathrm{lst}(\Phi, \vartheta_E)_i$. $\mathrm{sig}(L_i) = \{\ell_i, \overline{\ell}_i\}$. The reason for employing one termination control variable will be explained in Sect. 5.3.3.

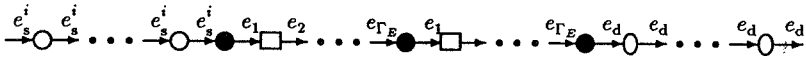


Figure 5–3: The evolution of the evolution control variable. Arrows depict $\overline{\vartheta}_E$. Fat dots are computation points, circles are soaking points, boxes are relaying points and ovals are draining points.

Let us describe the specification of evolution control variable E. The communication constraint domain of E is given by

$$\Omega_E \;=\; \text{planes}(\Phi, \sigma) \cap \text{rays}(\Phi, \pm\vartheta_E) \tag{5.16}$$

This is to ensure that all ϑ_E-paths that contain computation points are mapped to different $\overline{\vartheta}_E$-paths in the space-time diagram. Thus, each $\overline{\vartheta}_E$-path that contains computation points contains a unique first (last) computation point of E (Lemma 4.5).

We consider two types of $\overline{\vartheta}_E$-paths in the space-time diagram:

- Path $p(\Upsilon, \overline{\vartheta}_E, \overline{I})$ contains computation points (Fig. 5–3). It must contain one first computation point of $\overline{\text{fst}(\Phi, \vartheta_E)}_i$ for some i. We input e_s^i at the first point of the path and adopt this value for all soaking points of E. (The superscript i in e_s^i identifies the soaking points in the $\overline{\vartheta}_E$-paths that contain first computation points of $\overline{\text{fst}(\Phi, \vartheta_E)}_i$.) $\mathcal{B}(F)$ holds at the first computation point of the path. There, e_s^i is converted to e_1. Then, if a point receives element e_k of E, it sends $e_{(k \bmod \Gamma_E)+1}$; k is the distance of a relaying point from its preceding computation point. Thus, the considered element of E periodically adopts the values $e_1, e_2, \cdots, e_{\Gamma_E}$. $\mathcal{B}(L)$ holds at the last computation point of the path. There, e_{Γ_E} is converted to e_d, the value for all draining points of E. If path $p(\Upsilon, \overline{\vartheta}_E, \overline{I})$ contains only one computation point, which is, therefore, both a first and last computation point, e_s^i changes to e_d at that point.

- Path $p(\Upsilon, \overline{\vartheta}_E, \overline{I})$ contains undefined points of E only. We input e_u at the first point of the path and adopt this value for the remaining points.

By the construction of E, we inject e_s^i at steps $\text{input}(E(I))$ for $I \in \text{fst}(\Phi, \vartheta_E)_i$ for every i and e_u at the remaining steps. $\text{sig}(E) = \{e_u, e_s^1, e_s^2, \cdots, e_s^f, e_1, e_2, \cdots, e_{\Gamma_E}, e_d\}$.

```
PROGRAM: HostProg
(∀ t,i : t_fst ≤ t ≤ t_lst ∧ 0 < i ≤ f : f_{i,1}(t) ∈ sig(F_{i,1}))
(∀ t,i : t_fst ≤ t ≤ t_lst ∧ 0 < i ≤ f : f_{i,2}(t) ∈ sig(F_{i,2}))
(∀ t,i : t_fst ≤ t ≤ t_lst ∧ 0 < i ≤ ℓ : ℓ_i(t) ∈ sig(L_i))
for t from t_fst to t_lst do
    for i from 1 to f do
        inject(f_{i,1}(t), pi(F^in_{i,1}))
        inject(f_{i,2}(t), pi(F^in_{i,2}))
    for i from 1 to ℓ do
        inject(ℓ_i(t), pi(L^in_i))
    if t ∉ {input(E(I)) | I ∈ fst(Φ, ϑ_E)} → inject(e_u, E^in)
    [] else → for i from 1 to f do
                  if t ∈ {input(E(I)) | I ∈ fst(Φ, ϑ_E)_i} → inject(e^i_s, pi(E^in))
                  [] else → skip
                  fi
    fi
```

Table 5–1: The host program for three-dimensional UREs. The specification of the initialisation and termination control variables are refined in Sects. 5.3.2 and 5.3.3.

For the host and cell program for three-dimensional UREs, see Tabs. 5–1 and 5–2. The notation $\mathsf{inject}(s, \mathsf{pi}(V^{\mathrm{in}}))$ stands for the injection of control value $s \in \mathrm{sig}(V)$ to input cell $\mathsf{pi}(V^{\mathrm{in}})$.

In Tab. 5–2, the first guard of E^{out} selects first computation points of E but excludes points that are both a first and last computation point of E. The second guard of E^{out} establishes whether a point is a last computation point of E. $\mathcal{B}(F)$ holds at first computation points of E and $E^{\mathrm{in}} = e_{\Gamma_E}$ holds at computation points that are not first computation points of E. Thus, if $\mathrm{fst}(\Phi, \vartheta_E)$ and $\mathrm{lst}(\Phi, \vartheta_E)$ are disjoint, the first guard of E can be simplified to $\mathcal{B}(F)$. The third guard handles soaking points, draining points and undefined points of E. The **else** guard of E^{out} captures points that are relaying or computation points but neither first nor last computation points of E.

PROGRAM: *CellProg*

$$
\begin{array}{rcl}
(\forall\, i : 0 < i \leqslant \Gamma_E : e_i & = & i) \\
(\forall\, i : 0 < i \leqslant f : F^{\text{out}}_{i,1} & = & F^{\text{in}}_{i,1}) \\
(\forall\, i : 0 < i \leqslant f : F^{\text{out}}_{i,2} & = & F^{\text{in}}_{i,2}) \\
(\forall\, i : 0 < i \leqslant \ell : L^{\text{out}}_{i} & = & L^{\text{in}}_{i}) \\
E^{\text{out}} & = & \textbf{if } \mathcal{B}(F) \wedge \neg\mathcal{B}(L) \rightarrow e_{\text{l}} \\
 & & [\!]\ \mathcal{B}(L) \rightarrow e_{\text{d}} \\
 & & [\!]\ ((\exists\, i : 0 < i \leqslant f : E^{\text{in}} = e^{i}_{s}) \wedge \neg\mathcal{B}(F)) \vee \\
 & & \quad E^{\text{in}} = e_{\text{d}} \vee E^{\text{in}} = e_{\text{u}} \rightarrow E^{\text{in}} \\
 & & [\!]\ \textbf{else} \rightarrow (E^{\text{in}} \textbf{ mod } \Gamma_E) + 1 \\
 & & \textbf{fi}
\end{array}
$$

$$
\chi^{\text{in}}_{\Phi} \;=\; \textbf{if } \mathcal{B}(F) \vee E^{\text{in}} = e_{\Gamma_E} \;\rightarrow\; \chi_{\text{c}}\ [\!]\ \textbf{else} \;\rightarrow\; \chi_{\text{p}}\ \textbf{fi}
$$

Table 5–2: The cell program for three-dimensional UREs.

The following lemma states certain properties of the cell and host programs. It will play a technical rôle later on.

Lemma 5.1 *Let the space-time mapping be valid for the evolution control variable.*

1. $E^{\text{in}} \neq e^{i}_{s}$ *for every* i *holds at the points of* $\mathfrak{C}_{\text{T}}\overline{\Psi}_E$.

2. *Let* $\overline{I}$ *be a point in a* $\overline{\vartheta}_E$*-path that contains a first computation point of* $\overline{\text{fst}(\Phi, \vartheta_E)}_i$. *If* $\overline{I}$ *is the first point in the* $\overline{\vartheta}_E$*-path at which* $\mathcal{B}(F)$ *or* $\mathcal{B}(L)$ *holds, then* $E^{\text{in}} = e^{i}_{s}$ *holds at* $\overline{I}$ *and all points that precede* $\overline{I}$ *but not at any point that succeeds* $\overline{I}$ *in the* $\overline{\vartheta}_E$*-path.*

3. *If* $\overline{I}_1, \overline{I}_2, \cdots, \overline{I}_p$ *is a subpath of a* $\overline{\vartheta}_E$*-path such that (a)* $\mathcal{B}(F)$ *holds at* $\overline{I}_1$ *but not at the remaining points of the* $\overline{\vartheta}_E$*-path and (b)* $\overline{I}_p$ *is the first point of the* $\overline{\vartheta}_E$*-path at which* $\mathcal{B}(L)$ *holds, then* $E^{\text{in}} = e_{\Gamma_E}$ *holds at all points* $\overline{I}_i$ *of the subpath satisfying* $i \neq 1 \wedge (i-1) \textbf{ mod } \Gamma_E = 0$ *but not at the remaining points of the* $\overline{\vartheta}_E$*-path.*

4. *If* $\overline{I}$ *is the first point in a* $\overline{\vartheta}_E$*-path at which* $\mathcal{B}(L)$ *holds and* $\mathcal{B}(F)$ *does not hold at any point that succeeds* $\overline{I}$ *of the* $\overline{\vartheta}_E$*-path, then* $E^{\text{in}} = e_{\text{d}}$ *does not hold at* $\overline{I}$ *nor at any point that precedes* $\overline{I}$ *but holds at all points that succeed* $\overline{I}$ *in the* $\overline{\vartheta}_E$*-path.*

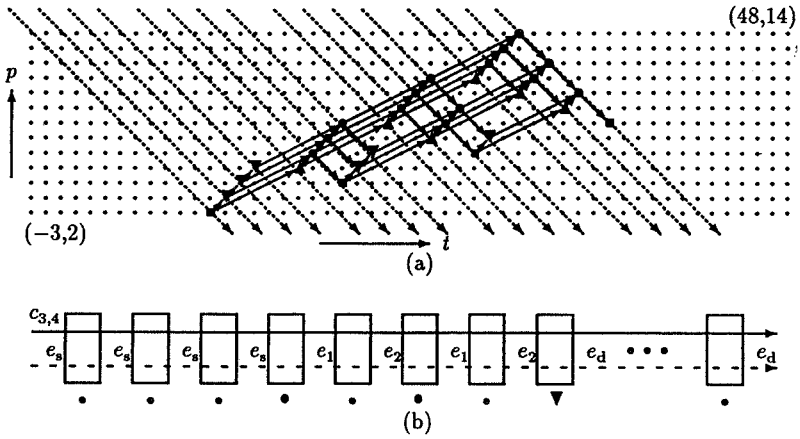


Figure 5–4: The evolution control variable for LU-decomposition ($m = 4$). (a) e_s is input at the $\overline{\vartheta}_E$-paths depicted by the dotted lines and e_u at the remaining $\overline{\vartheta}_E$-paths. (b) The evolution of the evolution control variable along the $\overline{\vartheta}_E$-path depicted by the dotted line in Fig. 5–1.

Proof. Follows from the host and cell programs as in Tabs. 5–1 and 5–2. □

Theorem 5.2 *Let $\mathcal{B}(F)$ and $\mathcal{B}(L)$ be characteristic functions of first and last computation points, respectively. If the space-time mapping is valid for the evolution control variable, the PCUREs are correct.*

Proof. Proving the correctness of the PCUREs means proving that $\chi_{\overline{\Phi}}$, as defined in the cell program, is a characteristic function of $\overline{\Phi}$. By the hypothesis, $\mathcal{B}(F)$ is a characteristic function of $\overline{\mathrm{fst}(\Phi, \vartheta_E)}$. It suffices to prove that $E^{\mathrm{in}} = e_{\Gamma_E}$ is a characteristic function of the set of the computation points that are not first computation points of E. This follows from (3) of Lemma 5.1, since $\mathcal{B}(F)$ ($\mathcal{B}(L)$) is a characteristic function of $\overline{\mathrm{fst}(\Phi, \vartheta_E)}$, ($\overline{\mathrm{lst}(\Phi, \vartheta_E)}$). (The validity of the space-time mapping for E ensures that Lemma 5.1 applies.) □

Remark. If $|Ⓕ| = 1$, there are two initialisation control variables $F_{1,1}$ and $F_{1,2}$ and one control value e_s^1 for the soaking points of E. The index $i = 1$ will be omitted. Similarly, we write L instead of L_1 for the termination control variable when $|Ⓛ| = 1$. □

Example 5.2 $m \times m$ LU-Decomposition

Without loss of generality, we choose $\vartheta_E = \vartheta_C$. $\mathrm{fst}(\Phi, \vartheta_E)$ by itself is a facet. We choose $\textcircled{F} = \{\mathrm{fst}(\Phi, \vartheta_E)\}$. Because of the choice of ϑ_E, the space-time mapping (5.9) is valid for evolution control variable E (Fig. 5–4(a)). The ϑ_C-path depicted by the dashed line in the data dependence graph in Fig. 3–6 is mapped to the $\overline{\vartheta}_C$-path depicted by the dotted line in Fig. 5–1. This path is also depicted at the bottom of the array shown in Fig. 5–4(b). Let us consider the evaluation of the points in this path. Element $c_{3,4}$ is input at this path. By the specification of the evolution control variable and the fact that $\vartheta_E = \vartheta_C$, we input e_s at the time step at which $c_{3,4}$ is input. e_s and $c_{3,4}$ travel with the same velocity. e_s changes to e_1 at the first computation point. Then the control value changes alternatively from e_1 to e_2 ($\mathrm{sig}(E) = \{e_u, e_s, e_1, e_2, e_d\}$ since $\Gamma_E = 2$). e_2 changes to e_d at the last computation point; it is propagated to the output cell. The same reasoning applies for the other $\overline{\vartheta}_E$-paths. The correctness of the PCUREs is straightforward and is independent of the choice of ϑ_E. □

Once evolution control variable E is constructed, or more precisely, once control dependence vector ϑ_E is chosen, the specification of initialisation and termination control variables is completely determined. The following two sections present their respective constructions.

5.3.2 The Initialisation Control Flow

Initialisation control variables are pipelining control variables. Their specification must enable us to define $\mathcal{B}(F)$ as a characteristic function of $\overline{\mathrm{fst}(\Phi, \vartheta_E)}$. We shall construct $F_{i,1}$ and $F_{i,2}$ such that $\mathcal{B}(F_i)$, defined below, is a characteristic function of $\overline{\mathrm{fst}(\Phi, \vartheta_E)}_i$:

$$\mathcal{B}(F_i) \;=\; E^{\mathrm{in}} = e_s^i \wedge F_{i,1}^{\mathrm{in}} = f_{i,1} \wedge F_{i,2}^{\mathrm{in}} = f_{i,2} \tag{5.17}$$

Note that evolution control variable E also acts as an initialisation control variable. This is because a space-time point is a first computation point only if $(\exists\, i : 0 < i \leqslant f : E^{\mathrm{in}} = e_s^i)$ ((1) and (2) of Lemma 5.1). Clearly, if $\mathcal{B}(F_i)$ is a characteristic function of $\overline{\mathrm{fst}(\Phi, \vartheta_E)}_i$, for every i, $\mathcal{B}(F)$ defined below is a characteristic function of $\overline{\mathrm{fst}(\Phi, \vartheta_E)}$:

$$\mathcal{B}(F) \;=\; (\exists\, i : 0 < i \leqslant f : \mathcal{B}(F_i)) \tag{5.18}$$

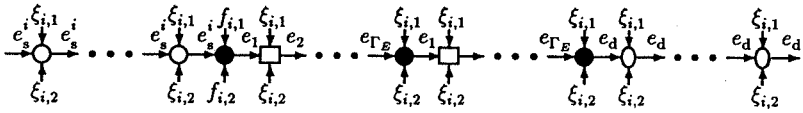


Figure 5–5: The interplay between the evolution control variable and the initialisation control variables. The arrows pointing down depict $\vartheta_{F_{i,1}}$. The arrows pointing up depict $\vartheta_{F_{i,2}}$. $\xi_{i,1}$ and $\xi_{i,2}$ satisfy $\neg(\xi_{i,1}=f_{i,1} \wedge \xi_{i,2}=f_{i,2})$.

Let $\mathrm{soak}(E,i)$ be the set of soaking points in the $\overline{\vartheta}_E$-paths that contain first computation points of $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$. $\{\mathrm{soak}(E,i) \mid 0<i\leqslant f\}$ is a partition of $\overline{\Psi}^{\mathrm{s}}_E$. The next lemma is the basis of the construction of the initialisation control variables.

Lemma 5.2 *Let the space-time mapping be valid for the evolution control variable. $\mathcal{B}(F_i)$ is a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$ iff $F^{\mathrm{in}}_{i,1}=f_{i,1} \wedge F^{\mathrm{in}}_{i,2}=f_{i,2}$ holds at $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$ but not at the soaking points of* $\mathrm{soak}(E,i)$.

Proof. The validity of the space-time mapping for evolution control variable E ensures that Lemma 5.1 applies.

"$\Longrightarrow$": If $F^{\mathrm{in}}_{i,1}=f_{i,1} \wedge F^{\mathrm{in}}_{i,2}=f_{i,2}$ does not hold at $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$, then $\mathcal{B}(F_i)$ is not a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$. If it holds at some soaking points of $\mathrm{soak}(E,i)$, then $\mathcal{B}(F_i)$ must hold at some soaking points of $\mathrm{soak}(E,i)$, by (2) of Lemma 5.1. Thus, $\mathcal{B}(F_i)$ is not a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$.

"$\Longleftarrow$": (1) of Lemma 5.1 asserts that $\mathcal{B}(F_i)$ does not hold at the undefined points of E. A further application of (2) of Lemma 5.1 completes the proof. □

Let us describe the specification of initialisation control variables $F_{i,1}$ and $F_{i,2}$. The communication constraint domains of $F_{i,1}$ and $F_{i,2}$ are given by

$$\begin{aligned}(\forall\, i : 0<i\leqslant f : \Omega_{F_{i,1}} &= \mathrm{planes}(\Phi,\sigma) \cap \mathrm{rays}(\mathrm{fst}(\Phi,\vartheta_E)_i, \pm\vartheta_{F_{i,1}})) \\ (\forall\, i : 0<i\leqslant f : \Omega_{F_{i,2}} &= \mathrm{planes}(\Phi,\sigma) \cap \mathrm{rays}(\mathrm{fst}(\Phi,\vartheta_E)_i, \pm\vartheta_{F_{i,2}}))\end{aligned} \tag{5.19}$$

This is to ensure that all $\vartheta_{F_{i,1}}$-paths ($\vartheta_{F_{i,2}}$-paths) that contain first computation points of $\mathrm{fst}(\Phi,\vartheta_E)_i$ are mapped to different $\overline{\vartheta}_{F_{i,1}}$-paths ($\overline{\vartheta}_{F_{i,2}}$-paths) in the space-time diagram.

To make $\mathcal{B}(F_i)$ a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$, we must establish $\mathcal{B}(F_i)$ at $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$. To establish $\mathcal{B}(F_i)$ at $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$, we must ensure that $F^{\mathrm{in}}_{i,1}=f_{i,1} \wedge$

```
for t from t_fst to t_lst do
    for i from 1 to f do
        if t ∈ {input(F_{i,1}(I)) | I ∈ fst(Φ, ϑ_E)_i} → inject(f_{i,1}, pi(F^in_{i,1}))
        [] else → inject(f̄_{i,1}, pi(F^in_{i,1}))
        fi
        if t ∈ {input(F_{i,2}(I)) | I ∈ fst(Φ, ϑ_E)^in_i} → inject(f_{i,2}, pi(F^in_{i,2}))
        [] else → inject(f̄_{i,2}, pi(F^in_{i,2}))
        fi
```

Table 5–3: The specification of the initialisation control variables.

$F^{\text{in}}_{i,2}=f_{i,2}$ holds at $\overline{\text{fst}(\Phi,\vartheta_E)}_i$ (Lemma 5.2). Then, we must inject $f_{i,k}$ $(k=0,1)$ at steps $\mathsf{input}(F_{i,k}(I))$ for $I\in\text{fst}(\Phi,\vartheta_E)_i$ and $\overline{f}_{i,k}$ at the remaining steps (Fig. 5–5). For the refined specification of the initialisation control variables in the host program, see Tab. 5–3. However, this specification is insufficient to ensure that $\mathcal{B}(F_i)$ is a characteristic function of $\overline{\text{fst}(\Phi,\vartheta_E)}_i$ by Lemma 5.2, since $F^{\text{in}}_{i,1}=f_{i,1}\wedge F^{\text{in}}_{i,2}=f_{i,2}$ and, consequently, $\mathcal{B}(F_i)$ may also hold at some soaking points of $\text{soak}(E,i)$ due to the non-injectivity of the space-time mapping. This happens, for example, if $f_{i,1}$ which is injected at step $\mathsf{input}(F_{i,1}(I))$ meets $f_{i,2}$ which is injected at step $\mathsf{input}(F_{i,2}(J))$ for different $I,J\in\text{fst}(\Phi,\vartheta_E)_i$. We avoid this situation by means of an appropriate choice of control dependence vectors for initialisation control variables.

We choose two linearly independent control dependence vectors $\vartheta_{F_{i,1}}$ and $\vartheta_{F_{i,2}}$ from $\text{lin}(\text{fst}(\Phi,\vartheta_E)_i)$. We write $\mathfrak{F}_i$ for the intersection of $\text{rays}(\text{fst}(\Phi,\vartheta_E)_i,\vartheta_{F_{i,1}})$ and $\text{rays}(\text{fst}(\Phi,\vartheta_E)_i,\vartheta_{F_{i,2}})$, and $\mathfrak{F}$ for the union of $\mathfrak{F}_1,\mathfrak{F}_2,\cdots,\mathfrak{F}_f$:

$$\begin{aligned}\mathfrak{F}_i &= \text{rays}(\text{fst}(\Phi,\vartheta_E)_i,\vartheta_{F_{i,1}})\cap\text{rays}(\text{fst}(\Phi,\vartheta_E)_i,\vartheta_{F_{i,2}})\\ \mathfrak{F} &= (\bigcup\ i:0<i\leqslant f:\mathfrak{F}_i)\end{aligned}\tag{5.20}$$

It is easy to see that $\mathfrak{F}_i\supseteq\text{fst}(\Phi,\vartheta_E)_i$ and thus $\mathfrak{F}\supseteq\text{fst}(\Phi,\vartheta_E)$. This choice of control dependence vectors ensures that $\mathcal{B}(F_i)$ only holds at $\overline{\mathfrak{F}}_i$ (by Lemma 5.3, which will be presented shortly). Still, $\mathcal{B}(F_i)$ is not a characteristic function of $\overline{\text{fst}(\Phi,\vartheta_E)}_i$ if $\overline{\mathfrak{F}}_i\backslash\overline{\text{fst}(\Phi,\vartheta_E)}_i$ contains soaking points of $\text{soak}(E,i)$, as the following example shows.

Example 5.3 Consider source UREs with the data dependence graph depicted in Fig. 5–6(a).

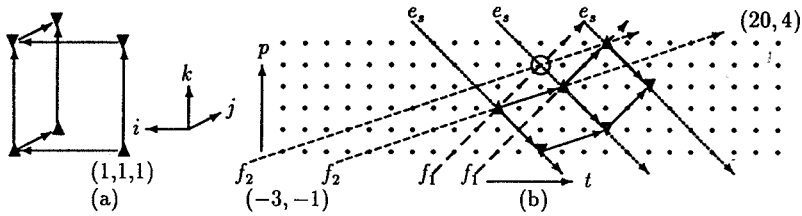


Figure 5–6: The construction of the initialisation control variables.

Let us choose $\vartheta_E = (0,0,1)$; then, $\mathrm{fst}(\Phi, \vartheta_E)$ is a facet. We choose $\textcircled{F} = \{\mathrm{fst}(\Phi, \vartheta_E)\}$. The three first (last) computation points of E are depicted by ▲ (▼). We need two initialisation control variables, F_1 and F_2. We choose $\vartheta_{F_1} = (0,1,0)$ and $\vartheta_{F_2} = (1,0,0)$. This gives rise to $\mathfrak{F} = \{(1,1,1),(1,2,1),(2,1,1),(2,2,1)\}$. $\mathfrak{F} \backslash \mathrm{fst}(\Phi, \vartheta_E) = \{(2,2,1)\}$. The following space-time mapping is valid for the source UREs and the three control variables:

$$\pi = \begin{bmatrix} 3 & 2 & 2 \\ 1 & 2 & -2 \end{bmatrix}$$

The corresponding space-time diagram is depicted in Fig. 5–6(b). Note that $\mathcal{B}(F)$ holds at the soaking point of E highlighted by a circle. This means that $(\overline{\mathfrak{F}} \backslash \overline{\mathrm{fst}(\Phi, \vartheta_E)}) \cap \mathrm{soak}(E,1) \neq \varnothing$ ($|\textcircled{F}| = 1$), because the two ϑ_E-paths (of $\mathrm{paths}(\mathfrak{F}, \vartheta_E)$) that pass through points $(1,2,1)$ and $(2,1,1)$ are mapped to the same $\overline{\vartheta}_E$-path. □

In order for $\overline{\mathfrak{F}}_i \backslash \overline{\mathrm{fst}(\Phi, \vartheta_E)}_i$ and $\mathrm{soak}(E,i)$ to be disjoint, we simply redefine (enlarge) the communication constraint domain of evolution control variable E to:

$$\Omega_E = \mathrm{planes}(\Phi, \sigma) \cap \mathrm{rays}(\mathfrak{F}, \pm\vartheta_E) \tag{5.21}$$

(Compare (5.16).) Under a valid space-time mapping for evolution control variable E, all ϑ_E-paths of $\mathrm{paths}(\mathfrak{F}_i, \vartheta_E)$ are mapped to different $\overline{\vartheta}_E$-paths. Thus, no two points of $\mathfrak{F}_i$ can have images in the same $\overline{\vartheta}_E$-path. That is, no point of $\mathfrak{F}_i \backslash \mathrm{fst}(\Phi, \vartheta_E)_i$ is mapped to a $\overline{\vartheta}_E$-path that contains a first computation point of $\overline{\mathrm{fst}(\Phi, \vartheta_E)}_i$, i.e., $(\overline{\mathfrak{F}}_i \backslash \overline{\mathrm{fst}(\Phi, \vartheta_E)}_i) \cap \mathrm{soak}(E,i) = \varnothing$.

The advantage of choosing control dependence vectors this way is that we know precisely that $\mathcal{B}(F_i)$ can hold only at the image of $\mathfrak{F}_i$. $\mathfrak{F}_i$ depends on the choice of $\vartheta_{F_{i,1}}$ and $\vartheta_{F_{i,2}}$ but is independent of the space-time mapping. This enables us to enforce the disjointness of $\overline{\mathfrak{F}}_i \backslash \overline{\mathrm{fst}(\Phi, \vartheta_E)}_i$ and $\mathrm{soak}(E,i)$ by an appropriate choice of the communication constraint domain for the evolution control variable.

Remark. We have chosen two different control values. e_s^i and e_s^j, of E for $\overline{\vartheta}_E$-paths that pass through $\mathrm{fst}(\Phi,\vartheta_E)_i$ and $\mathrm{fst}(\Phi,\vartheta_E)_j$ for different i and j. This is to take into account the fact that $\mathfrak{F}$ and Ψ_E^s are not disjoint. In this case, $\mathcal{B}(F)$ would hold at the soaking points of $\overline{\mathfrak{F}\cap\Psi_E^s}$ if we adopted the same control value. Whenever $\mathfrak{F}$ and Ψ_E^s are disjoint for the source UREs under consideration, we can replace control values $e_s^1, e_s^2, \cdots, e_s^f$ by one value. In addition, the elements of Ⓕ need not be disjoint.

By Lemma 5.2, $\xi_{i,1}$ and $\xi_{i,2}$, as shown in Fig. 5–5 at all points succeeding the first computation point can take any value. However, this does not seem to be practically useful, because the verification of a space-time mapping that satisfies this constraint depends on the space-time mapping. Our construction of the initialisation control variables ensures the satisfaction of $\neg(\xi_{i,1}=f_{i,1}\wedge\xi_{i,2}=f_{i,2})$ at the points that are not first computation points of $\overline{\mathrm{fst}(\Phi,\vartheta_E)_i}$. □

Theorem 5.3 *If the space-time mapping is valid for the evolution and initialisation control variable, $\mathcal{B}(F)$ is a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)}$.*

The proof of this theorem requires the following lemma, which states that two paths of variables moving with different velocities only share an image at the intersection point. This lemma is the reason why we use two initialisation control variables for every $\mathrm{fst}(\Phi,\vartheta_E)_i$ and why the communication constraint for the initialisation control variables can be disregarded.

Lemma 5.3 *Assume* $0\neq\mathsf{flow}(V)\neq\mathsf{flow}(W)\neq 0$.

$$(\forall\, I,J,K : I,J,K\in\mathbf{Z}^n\wedge I\in\mathrm{ray}(K,\pm\vartheta_V)\wedge J\in\mathrm{ray}(K,\pm\vartheta_W) : \overline{I}=\overline{J} \Longrightarrow I=J=K)$$

Proof. $\overline{I}=\overline{J} \iff \mathsf{step}(I)=\mathsf{step}(J)\wedge\mathsf{place}(I)=\mathsf{place}(J) \iff \lambda I=\lambda J\wedge\sigma I=\sigma J$. $I\in\mathrm{ray}(K,\pm\vartheta_V) \iff (\exists\, m : m\in\mathbf{Z} : K=m\vartheta_V+I)$ and $J\in\mathrm{ray}(K,\pm\vartheta_W) \iff (\exists\, n : n\in\mathbf{Z} : K=n\vartheta_W+J)$. A simple algebraic calculation establishes

$$m\lambda\vartheta_V = n\lambda\vartheta_W \tag{5.22}$$

$$m\sigma\vartheta_V = n\sigma\vartheta_W \tag{5.23}$$

We consider all four possible geometric relationships between I and J relative to K (the proof proceeds to show that only $I=J=K$ can hold).

- $I=K \wedge J \neq K$. This implies $m=0$ and consequently $\lambda\vartheta_W=0$, contradicting the hypothesis $\mathsf{flow}(W)\neq 0$.
- $I\neq K \wedge J=K$. This implies $n=0$ and consequently $\lambda\vartheta_V=0$, contradicting the hypothesis $\mathsf{flow}(V)\neq 0$.
- $I\neq K \wedge J\neq K$. By hypothesis we infer $m\lambda\vartheta_V \neq 0 \neq n\lambda\vartheta_W$. Dividing (5.23) by (5.22) yields $\sigma\vartheta_V/\lambda\vartheta_V = \sigma\vartheta_W/\lambda\vartheta_W$, i.e., $\mathsf{flow}(V)=\mathsf{flow}(W)$, contradicting the hypothesis $\mathsf{flow}(V)\neq\mathsf{flow}(W)$.
- $I=K \wedge J=K$. Trivially true. □

This proof does not require the containment of I, J and K in the index space; it only relies on the geometric relationship of the three points. It is easy to see that, if ϑ_V and ϑ_W are not co-linear for two moving variables V and W, then they must travel with distinct velocities, i.e., $\mathsf{flow}(V)\neq\mathsf{flow}(W)$. This is why we require $\vartheta_{F_{i,1}}$ and $\vartheta_{F_{i,2}}$ to be linearly independent.

Proof of Thm. 5.3. By the construction of the initialisation control variables, $\vartheta_{F_{i,1}}$ and $\vartheta_{F_{i,2}}$ are linearly independent vectors from the linear space $\mathrm{lin}(\mathrm{fst}(\Phi,\vartheta_E)_i)$. Thus, for every $\vartheta_{F_{i,1}}$-path that passes through some point of $\mathrm{fst}(\Phi,\vartheta_E)_i$, there must exist a $\vartheta_{F_{i,2}}$-path that passes through some point of $\mathrm{fst}(\Phi,\vartheta_E)_i$ such that the two paths intersect, and vice versa. By definition, $\mathfrak{F}_i$ denotes the set of these intersection points. By Lemma 5.3, the images of all these paths intersect only at the points of $\overline{\mathfrak{F}}_i$ since the space-time mapping is valid for the initialisation control variables and by our assumption that all propagation control variables are moving. If the space-time mapping is valid for evolution control variable E, no point contained in $\overline{\mathfrak{F}}_i\backslash\overline{\mathrm{fst}(\Phi,\vartheta_E)}_i$ is in the same $\overline{\vartheta}_E$-path as a point contained in $\overline{\mathrm{fst}(\Phi,\vartheta_E)}_i$. This means that $(\overline{\mathfrak{F}}_i\backslash\overline{\mathrm{fst}(\Phi,\vartheta_E)}_i)\cap\mathrm{soak}(E,i)=\varnothing$. That is, $F^{\mathrm{in}}_{i,1}=f_{i,1}\wedge F^{\mathrm{in}}_{i,2}=f_{i,2}$ does not hold at $\mathrm{soak}(E,i)$. But it does hold at $\overline{\mathrm{fst}(\Phi,\vartheta_E)}_i$ by the construction of the initialisation control variables. An application of Lemma 5.2 establishes that $\mathcal{B}(F_i)$ is a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)}_i$. By the definition of $\mathcal{B}(F)$, $\mathcal{B}(F)$ is a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)}$. □

Example 5.4 $m\times m$ LU-Decomposition

Let us continue to choose $\vartheta_E=\vartheta_C$ and $\textcircled{F}=\{\mathrm{fst}(\Phi,\vartheta_E)\}$. We need two initialisation control variables F_1 and F_2. We choose $\vartheta_{F_1}=\vartheta_A$ and $\vartheta_{F_2}=\vartheta_B$. This gives rise to $\mathfrak{F}=\mathrm{fst}(\Phi,\vartheta_E)$ (Fig. 5–7). □

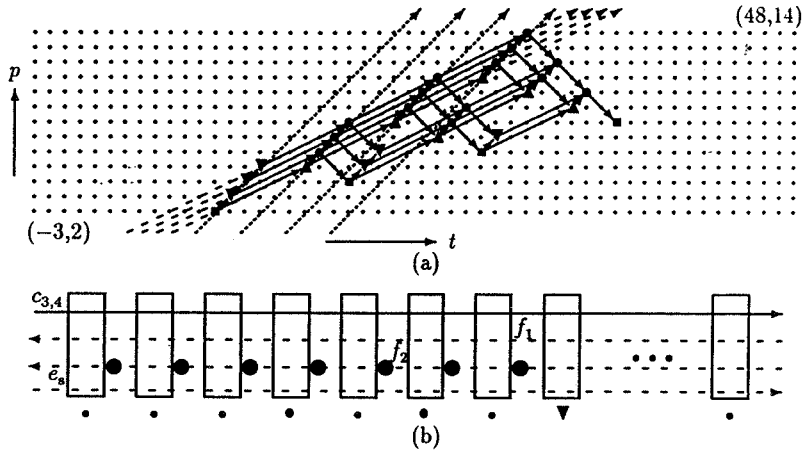


Figure 5–7: The initialisation control variables for LU-decomposition ($m = 4$). (a) f_1 (f_2) is input at the $\overline{\vartheta}_{F_1}$-paths ($\overline{\vartheta}_{F_2}$-paths) depicted by the dotted (dashed) lines and $\overline{f}_1$ ($\overline{f}_2$) at the remaining $\overline{\vartheta}_{F_1}$-paths ($\overline{\vartheta}_{F_2}$-paths). (b) The interplay between evolution control variable E and initialisation control variables F_1 and F_2 in the ϑ_E-path depicted in Fig. 5–1. Control signals e_s, f_1 and f_2 will meet at the fourth cell from the left three time steps after $c_{3,4}$ is input. At this step, $\mathcal{B}(F)$ holds at that cell, indicating that the point to be computed by the cell is a first computation point.

5.3.3 The Termination Control Flow

Termination control variables are pipelining control variables. Their specification must enable us to define $\mathcal{B}(L)$ as a characteristic function of $\overline{\mathrm{lst}(\Phi, \vartheta_E)}$. However, the specification of the termination control variables can be greatly simplified for two reasons. First, we can construct them incrementally based on the specification of evolution and initialisation control variables. A space-time point is a last computation point only if it is a computation point, i.e., only when $\mathcal{B}(F) \vee E^{\mathrm{out}} = e_{\Gamma_E}$ holds at the point. Second, we have dealt with the non-injectivity of the space-time mapping during the construction of initialisation control variables (Thm. 5.3). This enables us to associate one rather than two control variables with every $\mathrm{lst}(\Phi, \vartheta_E)_i$. Besides, $\mathbb{L}$ needs not be a partition of $\mathrm{lst}(\Phi, \vartheta_E)$.

We shall construct L_i such that $\mathcal{B}(L_i)$, defined below, is a characteristic func-

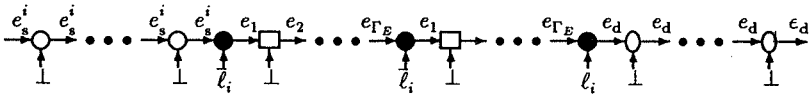


Figure 5–8: The interplay between the evolution control variable and the termination control variables. The arrows pointing up depict ϑ_{L_i}. $\perp$ denotes any value in $\mathrm{sig}(L_i)$.

tion of $\overline{\mathrm{lst}(\Phi,\vartheta_E)_i}$:

$$\mathcal{B}(L_i) \;=\; (\mathcal{B}(F) \vee E^{\mathrm{in}} = e_{\Gamma_E}) \wedge L_i^{\mathrm{in}} = \ell_i$$

Note that the evolution and initialisation control variables also act as termination control variables. If $\mathrm{lst}(\Phi,\vartheta_E)_i$ does not contain first computation points of E, i.e., if $\mathrm{fst}(\Phi,\vartheta_E)$ and $\mathrm{lst}(\Phi,\vartheta_E)_i$ are disjoint, $\mathcal{B}(F) \vee E^{\mathrm{in}} = e_{\Gamma_E}$ can be simplified to $E^{\mathrm{in}} = e_{\Gamma_E}$. Clearly, if $\mathcal{B}(L_i)$ is a characteristic function of $\overline{\mathrm{lst}(\Phi,\vartheta_E)_i}$ for every i, $\mathcal{B}(L)$, defined below, is a characteristic function of $\overline{\mathrm{lst}(\Phi,\vartheta_E)}$:

$$\mathcal{B}(L) \;=\; (\exists\, i : 0 < i \leqslant \ell : \mathcal{B}(L_i))$$

The next lemma is the basis of the construction of the termination control variables.

Lemma 5.4 *Let the space-time mapping be valid for the evolution and initialisation control variables. $\mathcal{B}(L_i)$ is a characteristic function of $\overline{\mathrm{lst}(\Phi,\vartheta_E)_i}$ iff $L_i^{\mathrm{in}} = \ell_i$ holds at $\overline{\mathrm{lst}(\Phi,\vartheta_E)_i}$ and $L_i^{\mathrm{in}} = \overline{\ell}_i$ holds at $\overline{\Phi} \backslash \overline{\mathrm{lst}(\Phi,\vartheta_E)_i}$.*

Proof. Sufficiency follows from the definition of a characteristic function (Sect. 5.2). Let us prove necessity. Since the space-time mapping is valid for the evolution and initialisation control variables, $\mathcal{B}(F)$ is a characteristic function of $\overline{\mathrm{fst}(\Phi,\vartheta_E)}$ by Thm. 5.3. An application of (3) and (4) of Lemma 5.1 completes the proof. □

Let us describe the construction of the termination control variables. The communication constraint domain of L_i is given by:

$$(\forall\, i : 0 < i \leqslant \ell : \Omega_{L_i} \;=\; \mathrm{planes}(\Phi,\sigma) \cap \mathrm{rays}(\Phi, \pm\vartheta_{L_i})) \qquad (5.24)$$

This is to ensure that all ϑ_{L_i}-paths that contain computation points are mapped to different $\overline{\vartheta}_{L_i}$-paths. Following the same reasoning as in the specification of initialisation control variables, we inject ℓ_i at time steps $\mathsf{input}(L_i(I))$ for $I \in \mathrm{lst}(\Phi,\vartheta_E)_i$.

```
PROGRAM: HostProg
for t from t_fst to t_lst do
    for i from 1 to f do
        if t ∈ {input(F_{i,1}(I)) | I ∈ fst(Φ, ϑ_E)_i} → inject(f_{i,1}, pi(F^in_{i,1}))
        [] else → inject(f̄_{i,1}, pi(F^in_{i,1}))
        fi
        if t ∈ {input(F_{i,2}(I)) | I ∈ fst(Φ, ϑ_E)_i} → inject(f_{i,2}, pi(F^in_{i,2}))
        [] else → inject(f̄_{i,2}, pi(F^in_{i,2}))
        fi
    for i from 1 to ℓ do
        if t ∈ {input(L_i(I)) | I ∈ lst(Φ, ϑ_E)_i} → inject(ℓ_i, pi(L^in_i))
        [] else → inject(ℓ̄_i, pi(L^in_i))
        fi
    if t ∉ {input(E(I)) | I ∈ fst(Φ, ϑ_E)} → inject(e_u, E^in)
    [] else → for i from 1 to f do
                  if t ∈ {input(E(I)) | I ∈ fst(Φ, ϑ_E)_i} → inject(e^i_s, pi(E^in))
                  [] else → skip
                  fi
    fi
```

Table 5–4: The host program for three-dimensional UREs (the final version).

and $\overline{\ell}_i$ at the remaining steps (Fig. 5–8). For the final version of the host program for three-dimensional UREs, see Tab. 5–4.

Similarly to the choice of control dependence vectors for initialisation control variables, we choose ϑ_{L_i} from the linear space $\mathrm{lin}(\mathrm{lst}(\Phi, \vartheta_E)_i$. We introduce domain $\mathfrak{L}_i$ analogously to $\mathfrak{F}_i$ of (5.20). Since each $\mathrm{lst}(\Phi, \vartheta_E)_i$ is associated with one termination control variable, $\mathfrak{L}_i$ is defined as the intersection of the index space and the ϑ_{L_i}-paths that pass through some point of $\mathrm{lst}(\Phi, \vartheta_E)_i$, and $\mathfrak{L}$ for the union of $\mathfrak{L}_1, \mathfrak{L}_2, \cdots, \mathfrak{L}_\ell$:

$$\begin{aligned} \mathfrak{L}_i &= \Phi \cap \mathrm{rays}(\mathrm{lst}(\Phi, \vartheta_E)_i, \pm\vartheta_{L_i}) \\ \mathfrak{L} &= (\bigcup\ i : 0 < i \leqslant \ell : \mathfrak{L}_i) \end{aligned} \tag{5.25}$$

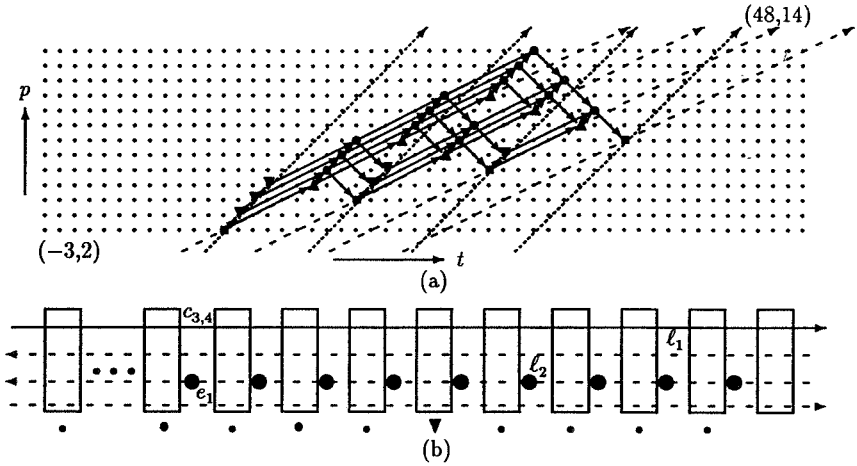


Figure 5–9: The termination control variables for LU-decomposition ($m=4$). (a) ℓ_1 (ℓ_2) is input at the $\overline{\vartheta}_{L_1}$-paths ($\overline{\vartheta}_{L_2}$-paths) depicted by the dotted (dashed) lines and $\overline{\ell}_1$ ($\overline{\ell}_2$) at the remaining $\overline{\vartheta}_{L_1}$-paths ($\overline{\vartheta}_{L_2}$-paths). (b) The interplay between evolution control variable E and termination control variables L_1 and L_2 in the ϑ_E-path depicted in Fig. 5–1. ℓ_1 and ℓ_2 and the control value of E will meet at the sixth cell from the right in three time steps. Then, $\mathcal{B}(L)$ evaluates to true, indicating that the point to be computed by the cell is a last computation point.

Theorem 5.4 *If the space-time mapping is valid for all propagation control variables, then $\mathcal{B}(L)$ is a characteristic function of $\overline{\mathrm{lst}(\Phi,\vartheta_E)}$.*

Proof. Under a valid space-time mapping for the termination control variables, $L_i^{\mathrm{in}}=\ell_i$ holds at $\overline{\mathrm{lst}(\Phi,\vartheta_E)}_i$ and $L_i^{\mathrm{in}}=\overline{\ell}_i$ holds at $\overline{\Phi}\setminus\overline{\mathrm{lst}(\Phi,\vartheta_E)}_i$. By Lemma 5.4, $\mathcal{B}(L_i)$ is a characteristic function of $\overline{\mathrm{lst}(\Phi,\vartheta_E)}_i$. By the definition of $\mathcal{B}(L)$, $\mathcal{B}(L)$ is a characteristic function of $\overline{\mathrm{lst}(\Phi,\vartheta_E)}$. □

Remark. It is not difficult to see that $\mathfrak{L}=\mathrm{lst}(\Phi,\vartheta_E)$ is necessary and sufficient for $\mathcal{B}(L)$ to be characteristic function of $\overline{\mathrm{lst}(\Phi,\vartheta_E)}$. Decomposing $\mathrm{lst}(\Phi,\vartheta_E)$, as described here, and choosing ϑ_{L_i} from the linear space $\mathrm{lin}(\mathrm{lst}(\Phi,\vartheta_E)_i)$ is unnecessary, but there is an advantage to doing so: $\mathfrak{L}=\mathrm{lst}(\Phi,\vartheta_E)$ is automatically satisfied. Otherwise, we may have to verify the equality of $\mathfrak{L}$ and $\mathrm{lst}(\Phi,\vartheta_E)$ explicitly. □

Example 5.5 $m \times m$ LU-Decomposition

We continue to choose $\vartheta_E = \vartheta_C$, $\vartheta_{F_1} = \vartheta_A$ and $\vartheta_{F_2} = \vartheta_B$. We choose $\textcircled{L} = \{\mathrm{lst}(\Phi, \vartheta_E)_1, \mathrm{lst}(\Phi, \vartheta_E)_2\}$, where $\mathrm{lst}(\Phi, \vartheta_E)_1$ and $\mathrm{lst}(\Phi, \vartheta_E)_2$ are two facets of the index space:

$$\begin{aligned} \mathrm{lst}(\Phi, \vartheta_E)_1 &= \Phi \cap \{(i,j,k) \mid i-k=0\} \\ \mathrm{lst}(\Phi, \vartheta_E)_2 &= \Phi \cap \{(i,j,k) \mid j-k=0\} \end{aligned}$$

We choose $\vartheta_{L_1} = \vartheta_A \in \mathrm{lin}(\mathrm{fst}(\Phi, \vartheta_E)_1)$ and $\vartheta_{L_2} = \vartheta_B \in \mathrm{lin}(\mathrm{fst}(\Phi, \vartheta_E)_2)$. The space-time mapping (5.9) is valid for the termination control variables (Fig. 5–9). □

5.3.4 The Space-Time Mapping

Let us summarise the previous construction of the PCUREs in a procedure.

Procedure 5.1 (Construction of the PCUREs for one-dimensional arrays from three-dimensional source UREs)

1. Choose control dependence vector $\vartheta_E \in \mathbf{Z}^n$ for evolution control variable E.

2. Find a partition $\textcircled{F} = \{\mathrm{fst}(\Phi, \vartheta_E)_i \mid 0 < i \leqslant f\}$ of $\mathrm{fst}(\Phi, \vartheta_E)$ such that $\dim(\mathrm{fst}(\Phi, \vartheta_E)_i) \leqslant n-1$. Choose two linearly independent vectors $\vartheta_{F_{i,1}}$ and $\vartheta_{F_{i,2}}$ for initialisation control variables $F_{i,1}$ and $F_{i,2}$ from the linear space $\mathrm{lin}(\mathrm{fst}(\Phi, \vartheta_E)_i)$ for every $\mathrm{fst}(\Phi, \vartheta_E)_i$.

3. Find a set $\textcircled{L} = \{\mathrm{lst}(\Phi, \vartheta_E)_i \mid 0 < i \leqslant \ell\}$ such that $\dim(\mathrm{lst}(\Phi, \vartheta_E)_i) \leqslant n-1$ and the union of its elements is $\mathrm{lst}(\Phi, \vartheta_E)$. Choose control dependence vector ϑ_{L_i} for termination control variable L_i from the linear space $\mathrm{lin}(\mathrm{lst}(\Phi, \vartheta_E)_i)$ for every $\mathrm{lst}(\Phi, \vartheta_E)_i$.

4. The cell program and host program are given in Tabs. 5–2 and 5–4. □

The communication constraint domains of the propagation control variables are given in (5.19), (5.21) and (5.24). Our objective for a different specification of PCUREs stated in Sect. 5.1 was to replace Ψ_V in the formulation of the communication constraint (5.3) by Ω_V, which satisfies (5.7). This objective has been achieved.

Lemma 5.5 *If* $0 \neq \mathsf{flow}(F_{i,1}) \neq \mathsf{flow}(F_{i,2}) \neq 0$, *then the communication constraint for the initialisation control variables* $F_{i,1}$ *and* $F_{i,2}$ *is satisfied.*

Proof. $0 \neq \mathsf{flow}(F_{i,1}) \neq \mathsf{flow}(F_{i,2}) \neq 0$ implies that $\overline{\vartheta}_{F_{i,1}}$ and $\overline{\vartheta}_{F_{i,2}}$ are linearly independent. This implies that all $\vartheta_{F_{i,1}}$-paths ($\vartheta_{F_{i,2}}$-paths) containing first computation points of $\mathrm{fst}(\Phi, \vartheta_E)_i$ are mapped to different $\overline{\vartheta}_{F_{i,1}}$-paths ($\overline{\vartheta}_{F_{i,2}}$-paths) in the space-time diagram. A further application of Lemma 4.5 completes the proof. □

This lemma allows us to disregard the communication constraint for the initialisation control variables. Equivalently, it amounts to defining the communication constraint domains of the initialisation control variables as follows:

$$(\forall\, i : 0 < i \leqslant f : \Omega_{F_{i,1}} \;=\; \Omega_{F_{i,2}} \;=\; \varnothing) \tag{5.26}$$

The mapping conditions for the validity of space-time mappings are stated in Thm. 5.1. Remember that we require all propagation control variables to be moving (Sect. 5.1). We call this constraint the *velocity constraint.* For completeness, we state in a theorem the mapping conditions, including the velocity constraint, for the validity of space-time mappings. We rewrite the communication constraint for propagation control variables to dispense with the concept of the extended index space. We write $\mathcal{V}_\ell$ for the set of termination control variables.

Theorem 5.5 *A space-time mapping is valid for the pipelined UREs (in either the* π*-model or the* τ*-model) iff*

- $(\forall\, V : V \in \mathcal{V} : \lambda\vartheta_V > 0)$ *(Precedence Constraint)*
- $\pi : \Phi \;\rightarrowtail\; \mathbf{Z}^2$ *(Computation Constraint)*
- $(\forall\, V : V \in \mathcal{V} :\; \sigma\vartheta_V \mid \lambda\vartheta_V)$ *(Delay Constraint for the* χ*-Model)*
- $(\forall\, V : V \in \mathcal{V}_\mathsf{d} : (\forall\, S : S \in \mathrm{fst}(\Phi_V, \vartheta_V)/\textcircled{I}_V : \mathsf{input} : S \rightarrowtail \mathbf{Z}))$
  $(\forall\, V : V \in \mathcal{V}_\mathsf{c} : (\forall\, S : S \in \mathrm{fst}(\Psi_V, \vartheta_V)/\textcircled{I}'_V : \mathsf{input} : S \rightarrowtail \mathbf{Z}))$
  $(\forall\, S : S \in \mathrm{fst}(\mathfrak{F}, \vartheta_E)/\textcircled{I}_E : \mathsf{input} : S \rightarrowtail \mathbf{Z})$
  $(\forall\, V : V \in \mathcal{V}_\ell : (\forall\, S : S \in \mathrm{fst}(\Phi, \vartheta_V)/\textcircled{I}_V : \mathsf{input} : S \rightarrowtail \mathbf{Z}))$
  (Communication Constraint)
- $(\forall\, V : V \in \mathcal{V}_\mathsf{p} : \mathsf{flow}(V) \neq 0)$ *(Velocity Constraint)*

Proof. Thms. 5.1, 5.2, 5.3 and 5.4. □

Evolution control variable E may be stationary. Then, additional control is needed to direct its loading, access and recovery, just like for stationary data variables. In general, we should choose ϑ_E and the space-time mapping such that E is moving. Initialisation and termination control variables must be moving for the following reasons. Their (control) values serve to change the control values of E; they must be supplied from the outside environment such that they can arrive at the right cells at the right time. If they are stationary, we have to pre-load their control signals – but we cannot, because the presence of these control signals ahead of time will validate some guards of E^{out} in the cell program at the wrong time (unless we add a second level of control variables, this one governing the access of stationary control variables, and so on).

Let us have a look at the requirements that the PCUREs impose on hardware. Recall our discussions about the injection of input data and the ejection of output data in the systolic array in Sect. 4.5. Some $\overline{\vartheta}_V$-paths may start at internal cells (see Figs. 5–4, 5–7 and 5–9). If the corresponding input values at these paths are required to be input, the input can be implemented by a system reset or by pipelining before the first step t_{fst}. In the construction of the PCUREs, the specification of the evolution control variable requires that all corresponding input channels of E be reset to e_{u}. Similarly, the specification of initialisation control variables requires that all corresponding input channels of $F_{i,1}$ be reset to $\overline{f}_{i,1}$ and of $F_{i,2}$ to $\overline{f}_{i,2}$, and the specification of termination control variables requires that all corresponding input channels be reset to $\overline{\ell}_i$.

The bit width required to communicate values of V is $\lceil \log_2 |\text{sig}(V)| \rceil$. For example, evolution control variable E needs $\lceil \log_2(\Gamma_E + |Ⓕ| + 2) \rceil$ bits – $\Gamma_E - 1$ for relaying points, one for computation points, $|Ⓕ|$ for soaking points, one for draining points and one for undefined points. We should choose ϑ_E and the space-time mapping such that Γ_E becomes a constant in order to satisfy Rst. 2 of Sect. 3.2. The modulo operation in the cell program can be implemented by a circular shift (or rotate) operation if Γ_E is a power of 2. It is superfluous if $\Gamma_E = 1$.

5.4 The PCUREs for n-Dimensional UREs

Both Thm. 5.2 and Thm. 5.4 work for n-dimensional UREs. But, unfortunately, Thm. 5.3 cannot be generalised to n-dimensional UREs. Recall that, in the specification of the initialisation control variables for three-dimensional UREs, $\mathrm{fst}(\Phi, \vartheta_E)$ is partitioned into a union of subsets each of which is contained in a hyperplane. We associate two initialisation control variables with each subset. The proof of Thm. 5.3 relies on Lemma 5.3, which assumes that each subset has no more than two dimensions. This is because the two linearly independent vectors, which are associated with the two initialisation control variables of a subset forms a basis of two-dimensional space. We failed to find an analogue of Lemma 5.3 that applies for more than two control variables.

There is a brute-force application of Thm. 5.3 to n-dimensional UREs. All we need to do is to partition $\mathrm{fst}(\Phi, \vartheta_E)$ into a union of subsets each of which has no more than two dimensions. The number of initialisation control variables needed will be twice the number of these subsets. Clearly, this number will depend on the size of the problem. In the extreme, each subset contains only one first computation point. Such a simple-minded scheme is rather impractical.

We choose a construction scheme for initialisation control variables for n-dimensional UREs that is based on the observation that our previous construction of the PCUREs supports a hierarchical specification of control signals. To identify the first computation points in $\mathrm{fst}(\Phi, \vartheta_E)$, we treat $\mathrm{fst}(\Phi, \vartheta_E)$ as a new index space and subject it to the previous construction of the PCUREs. We repeat this procedure until we obtain an index space of three dimensions. We then apply Thm. 5.3 to the three-dimensional index space. This decomposition produces a binary tree structure. The left (right) node at a level denotes the set of first (last) computation points of the set denoted by the father node. Since the initial index space is bounded and the new index space resulting from each decomposition always contains less points than the preceding one, this decomposition must eventually lead to a three-dimensional domain.

To speed up the convergence and thus reduce the number of control variables, we make the new index space at one level one dimension less than that at the level

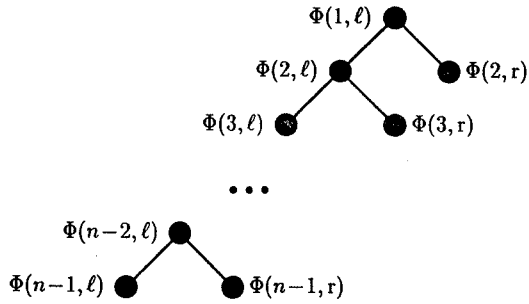


Figure 5–10: A hierarchical construction of the PCUREs for n-dimensional UREs.

above, by an appropriate choice of control dependence vector for the evolution control variable at every level in the decomposition tree. This choice is always possible if we extend the initial index space appropriately prior to the decomposition, as will be discussed in Sect. 5.7.

Let us assume that the new index space at one level is one dimension less than the index space at the level above. Let us denote the evolution control variable associated with the resulting index space at level i by $E(i)$. The decomposition tree has $n-2$ levels; the first level is labelled 1. The following recursive equation assigns a unique identifier to each node in the decomposition tree (Fig. 5–10):

$$\begin{aligned} \Phi(1,\ell) &= \Phi \\ (\forall\, i : 1<i<n : \quad \Phi(i,\ell) &= \mathrm{fst}(\Phi(i-1,\ell), \vartheta_{E(i-1)}) \\ (\forall\, i : 1<i<n : \quad \Phi(i,\mathrm{r}) &= \mathrm{lst}(\Phi(i-1,\ell), \vartheta_{E(i-1)}) \end{aligned} \tag{5.27}$$

$\Phi(i,\ell)$ is called a *left* node and $\Phi(i,\mathrm{r})$ a *right* node. Note that the range of i is extended by 1 so that the specification of the PCUREs for three-dimensional UREs, as presented previously, becomes a special case.

Next, we present the specification of the PCUREs as constructed previously, for an index space that is an n-parallelepiped. Let $\vartheta_{E(1)}, \vartheta_{E(2)}, \cdots, \vartheta_{E(n)}$ be the n linearly independent vectors that generate the parallelepiped. $\Phi(i+1,\ell)$ and $\Phi(i+1,\mathrm{r})$ are the sets of first and last computation points of $\Phi(i,\ell)$. We choose $\textcircled{F}_i = \{\Phi(i+1,\ell)\}$ and $\textcircled{L}_i = \{\Phi(i+1,\mathrm{r})\}$ for $\Phi(i,\ell)$; $\textcircled{F}_i$ plays the rôle of $\textcircled{F}$ and $\textcircled{L}_i$ the rôle of $\textcircled{L}$ in the case of a three-dimensional index space.

- *Evolution Control Variables.* Every left non-leaf node $\Phi(i,\ell)$ is a facet of its father node $\Phi(i-1,\ell)$. It is associated with one evolution control variable $E(i)$. $\mathrm{sig}(E(i)) = \{e(i)_{\mathrm{u}}, e(i)_{\mathrm{s}}, e(i)_1, e(i)_2, \cdots, e(i)_{\Gamma_{E(i)}}, e(i)_{\mathrm{d}}\}$. The communication constraint domains of the evolution control variables are given by

$$(\forall\, i : 0 < i \leqslant n-2 : \Omega_{E(i)} \;=\; \mathrm{planes}(\Phi,\sigma) \cap \mathrm{rays}(\Phi(i,\ell), \pm\vartheta_{E(i)}))$$

- *Initialisation Control Variables.* The left leaf node $\Phi(n-1,\ell)$ is a (two-dimensional) facet of $\Phi(n-2,\ell)$. It is associated with two initialisation control variables, $F(n-1)_1$ and $F(n-1)_2$. $\mathrm{sig}(F(n-1)_1) = \{f(n-1)_1, \overline{f}(n-1)_1\}$. $\mathrm{sig}(F(n-1)_2) = \{f(n-1)_2, \overline{f}(n-1)_2\}$. We choose $\vartheta_{F(n-1)_1}$ and $\vartheta_{F(n-1)_2}$ such that they are parallel to two adjacent edges of $\Phi(n-1,\ell)$, which are parallel to $\vartheta_{E(n-1)}$ and $\vartheta_{E(n)}$, respectively. Clearly, this ensures that $\vartheta_{F(n-1)_1}, \vartheta_{F(n-1)_2} \in \mathrm{lin}(\Phi(n-1,\ell))$. The communication constraint domains of the initialisation control variables are given by

$$\Omega_{F(n-1)_1} \;=\; \Omega_{F(n-1)_2} \;=\; \varnothing$$

- *Termination Control Variables.* Every right node $\Phi(i,\mathrm{r})$ is a facet of its father node $\Phi(i-1,\ell)$. It is associated with one termination control variable, $L(i)$. $\mathrm{sig}(L(i)) = \{\ell(i), \overline{\ell}(i)\}$. We choose $\vartheta_{L(i)}$ from the linear space $\mathrm{span}(\vartheta_{E(i+1)}, \vartheta_{E(i+2)}, \cdots, \vartheta_{E(n)})$. Clearly, $\vartheta_{L(i)} \in \mathrm{lin}(\Phi(i,\mathrm{r}))$. The communication constraint domains of the termination control variables are given by

$$(\forall\, i : 1 < i < n : \Omega_{L(i)} \;=\; \mathrm{planes}(\Phi,\sigma) \cap \mathrm{rays}(\Phi(i-1,\ell), \pm\vartheta_{L(i)}))$$

Because of the recursive nature of the specification of the PCUREs, the control variables associated with all the nodes in the subtree rooted at node $\Phi(i,\ell)$ act as initialisation control variables for the evolution control variable associated with the father node $\Phi(i-1,\ell)$.

Next, we define the characteristic functions of all nodes in the decomposition tree. Let $\mathcal{B}(F(i))$ be the characteristic function of every left node $\Phi(i,\ell)$, except the root $\Phi(1,\ell)$, and $\mathcal{B}(L(i))$ the characteristic function of every right node $\Phi(i,\mathrm{r})$ in $\Phi(i-1,\ell)$. These characteristic functions are recursively defined as follows:

$$\mathcal{B}(F(n-1)) = E(n-2)^{\mathrm{in}} = e(n-2)_{\mathrm{s}} \wedge F(n-1)_1^{\mathrm{in}} = f(n-1)_1 \wedge F(n-1)_2^{\mathrm{in}} = f(n-1)_2$$

$$(\forall\, i : 1 < i < n-1 : \mathcal{B}(F(i)) = E(i-1)^{\mathrm{in}} = e(i-1)_{\mathrm{s}} \wedge \mathcal{B}(F(i+1)) \wedge E(i)^{\mathrm{in}} = e(i)_{\Gamma_{E(i)}})$$

$$(\forall\, i : 1 < i < n : \mathcal{B}(L(i)) = (\mathcal{B}(F(i)) \vee E(i-1)^{\mathrm{in}} = e(i-1)_{\Gamma_{E(i-1)}}) \wedge L(i)^{\mathrm{in}} = \ell(i))$$

PROGRAM: *HostProg*

for t **from** t_{fst} **to** t_{lst} **do**

  if $t \in \{\text{input}(F(n-1)_1(I)) \mid I \in \Phi(n-1,\ell)\} \rightarrow \text{inject}(f(n-1)_1, \text{pi}(F(n-1)_1^{\text{in}}))$

  [] **else** $\rightarrow$ $\text{inject}(\overline{f}(n-1)_1, \text{pi}(F(n-1)_1^{\text{in}}))$

  fi

  if $t \in \{\text{input}(F(n-1)_2(I)) \mid I \in \Phi(n-1,\ell)\} \rightarrow \text{inject}(f(n-1)_2, \text{pi}(F(n-1)_2^{\text{in}}))$

  [] **else** $\rightarrow$ $\text{inject}(\overline{f}(n-1)_2, \text{pi}(F(n-1)_2^{\text{in}}))$

  fi

  for i **from** 2 **to** $n-1$ **do**

    if $t \in \{\text{input}(L(i)(I)) \mid I \in \Phi(i,\text{r})\} \rightarrow \text{inject}(\ell(i), \text{pi}(L(i)^{\text{in}}))$

    [] **else** $\rightarrow$ $\text{inject}(\overline{\ell}(i), \text{pi}(L(i)^{\text{in}}))$

    fi

  for i **from** 1 **to** $n-2$ **do**

    if $t \in \{\text{input}(E(i)(I)) \mid I \in \Phi(i,\ell)\} \rightarrow \text{inject}(e(i)_{\text{s}}, \text{pi}(E(i)^{\text{in}}))$

    [] **else** $\rightarrow$ $\text{inject}(e(i)_{\text{u}}, \text{pi}(E(i)^{\text{in}}))$

    fi

Table 5–5: The host program for n-dimensional UREs ($n>2$).

The characteristic function $\chi_{\overline{\Phi}}$ of $\overline{\Phi}$ in Υ is given by

$$\chi_{\overline{\Phi}}^{\text{in}} \;=\; \textbf{if } \mathcal{B}(F(2)) \vee E(1)^{\text{in}} = e(1)_{\Gamma_{E(1)}} \;\rightarrow\; \chi_{\text{c}} \;[]\; \textbf{else} \;\rightarrow\; \chi_{\text{p}} \;\textbf{fi} \qquad (5.28)$$

$\mathcal{B}(F(n-1))$ is the previous $\mathcal{B}(F)$ and $\mathcal{B}(L(n-1))$ is the previous $\mathcal{B}(L)$. These characteristic functions can be simplified by removing redundant predicates.

For the host and cell program for an n-dimensional parallelepiped index space, see Tabs. 5–5 and 5–6. Thm. 5.1 provides the mapping conditions for the validity of space-time mappings for the pipelined UREs whose index space is an n-dimensional parallelepiped index space. For completeness, we state in a theorem the mapping conditions, including the velocity constraint, for the validity of space-time mappings. We rephrase the communication constraint for propagation control variables to dispense with the concept of the extended index space.

Theorem 5.6 *A space-time mapping is valid for the pipelined UREs (in either the π-model or the τ-model) iff*

PROGRAM: *CellProg*

$(\forall\, i,k : 0<i\leqslant n-2 \wedge 0<k\leqslant \Gamma_{E(i)} : e(i)_k = k)$

$F(n-1)_1^{\text{out}} = F(n-1)_1^{\text{in}}$

$F(n-1)_2^{\text{out}} = F(n-1)_2^{\text{in}}$

$(\forall\, i : 1<i<n : L(i)^{\text{out}} = L(i)^{\text{in}})$

$E(n-2)^{\text{out}} = \textbf{if } \mathcal{B}(F(n-1)) \wedge \neg\mathcal{B}(L(n-1)) \rightarrow e(n-2)_{\text{l}}$

$[]\ \mathcal{B}(L(n-1)) \rightarrow e(n-2)_{\text{d}}$

$[]\ (E(n-2)^{\text{in}} = e(n-2)_{\text{s}} \wedge \neg\mathcal{B}(F(n-1))) \vee$

$E(n-2)^{\text{in}} = e(n-2)_{\text{d}} \vee E(n-2)^{\text{in}} = e(n-2)_{\text{u}} \rightarrow E(n-2)^{\text{in}}$

$[]\ \textbf{else} \rightarrow (E(n-2)^{\text{in}} \bmod \Gamma_{E(n-2)})+1$

$\textbf{fi}$

$(\forall\, i : 0<i<n-2 :$

$E(i)^{\text{out}} = \textbf{if } \mathcal{B}(F(i-1)) \wedge \neg\mathcal{B}(L(i-1)) \rightarrow e(i)_{\text{l}}$

$[]\ \mathcal{B}(L(i-1)) \rightarrow e(i)_{\text{d}}$

$[]\ (E(i)^{\text{in}} = e(i)_{\text{s}} \wedge \neg\mathcal{B}(F(i-1)))$

$\vee E(i)^{\text{in}} = e(i)_{\text{d}} \vee E(i)^{\text{in}} = e(i)_{\text{u}} \rightarrow E(i)^{\text{in}}$

$[]\ \textbf{else} \rightarrow (E(i)^{\text{in}} \bmod \Gamma_{E(i)})+1$

$\textbf{fi} \qquad)$

$\chi_{\Phi}^{\text{in}} = \textbf{if } \mathcal{B}(F(2)) \vee E(1)^{\text{in}} = e(1)_{\Gamma_{E(1)}} \rightarrow \chi_{\text{c}}\ []\ \textbf{else} \rightarrow \chi_{\text{p}}\ \textbf{fi}$

Table 5–6: The cell program for n-dimensional UREs ($n>2$).

- $(\forall\, V : V \in \mathcal{V} : \lambda\vartheta_V > 0)$ *(Precedence Constraint)*
- $\pi : \Phi \rightarrowtail \mathbf{Z}^2$ *(Computation Constraint)*
- $(\forall\, V : V \in \mathcal{V} :\ \sigma\vartheta_V \mid \lambda\vartheta_V)$ *(Delay Constraint for the χ-Model)*
- $(\forall\, V : V \in \mathcal{V}_{\text{d}} : (\forall\, S : S \in \text{fst}(\Phi_V, \vartheta_V)/\textcircled{I}_V : \text{input} : S \rightarrowtail \mathbf{Z}))$
  $(\forall\, V : V \in \mathcal{V}_{\text{c}} : (\forall\, S : S \in \text{fst}(\Psi_V, \vartheta_V)/\textcircled{I}'_V : \text{input} : S \rightarrowtail \mathbf{Z}))$
  $(\forall\, i : 0<i\leqslant n-2 : (\forall\, S : S \in \text{fst}(\Phi(i,\ell), \vartheta_{E(i)}) : \text{input} : S \rightarrowtail \mathbf{Z}))$
  $(\forall\, i : 1<i<n : (\forall\, S : S \in \text{fst}(\Phi(i-1,\ell), \vartheta_{L(i)}) : \text{input} : S \rightarrowtail \mathbf{Z}))$
  (Communication Constraint)
- $(\forall\, V : V \in \mathcal{V}_{\text{p}} : \text{flow}(V) \neq 0)$ *(Velocity Constraint)*

Proof. The proof is an induction on the reverse order of the decomposition of the index space (Fig. 5–10). To prove that $\mathcal{B}(F(i))$ is a characteristic function of $\Phi(i,\ell)$ in $\Phi(i-1,\ell)$, it suffices to show that $\mathcal{B}(F(i+1))$ is a characteristic function of $\Phi(i+1,\ell)$ and $\mathcal{B}(L(i+1))$ is a characteristic function of $\Phi(i+1,\mathrm{r})$ in $\Phi(i,\ell)$ by Thm. 5.2. To prove that $\mathcal{B}(L(i+1))$ is a characteristic function of $\Phi(i+1,\mathrm{r})$ in $\Phi(i,\ell)$, it suffices to show that $\mathcal{B}(F(i+1))$ is a characteristic function of $\Phi(i+1,\ell)$ in $\Phi(i,\ell)$ by Thm. 5.4. Thus, to prove the correctness of the PCUREs, it suffices to show that $\mathcal{B}(F(n-1))$ is a characteristic function of $\Phi(n-1,\ell)$ in $\Phi(n-2,\ell)$. This follows from Thm. 5.3. □

5.5 Related Work

Ramakrishnan and Varman [9] present a one-dimensional array for matrix product. They supply the flow of data and control and use linear algebra to prove the correctness of the array. Kumar and Tsai [10] propose a method that transforms two-dimensional to one-dimensional arrays. Their method requires an explicit choice of the communication topology and the sequencing of input data. The flow of data and control is then derived by solving a set of constraint equations on timing. Lang [11] shows how control signals can be pipelined to solve problems such as matrix product and merging two matrices. This method only applies for iterative algorithms defined over three-dimensional index spaces. All these methods do not include procedures that allow the systematic derivation of control signals. Rather, the synthesis of control signals is ad hoc and problem-specific.

5.6 Examples

5.6.1 Matrix Product

We recommend to choose a data dependence vector as the control dependence vector ϑ_E. For reasons of symmetry, it makes no difference which data dependence vector we choose: ϑ_A, ϑ_B or ϑ_C. Let us choose $\vartheta_E = \vartheta_C$. Both $\mathrm{fst}(\Phi,\vartheta_E)$ and $\mathrm{lst}(\Phi,\vartheta_E)$ are facets of the index space Φ. We choose $\textcircled{F} = \{\mathrm{fst}(\Phi,\vartheta_E)\}$

and $\textcircled{L} = \{\mathrm{lst}(\Phi, \vartheta_E)\}$. We need two initialisation control variables, F_1 and F_2, and one termination control variable, L. Hence, $\vartheta_{F_1}, \vartheta_{F_2} \in \mathrm{lin}(\mathrm{fst}(\Phi, \vartheta_E))$ and $\vartheta_L \in \mathrm{lin}(\mathrm{lst}(\Phi, \vartheta_E))$. We should choose two data dependence vectors ϑ_A and ϑ_B as ϑ_{F_1} and ϑ_{F_2}. This makes $\mathfrak{F} = \mathrm{fst}(\Phi, \vartheta_E)$. Without loss of generality, we choose $\vartheta_{F_1} = \vartheta_A$ and $\vartheta_{F_2} = \vartheta_B$. We should choose either ϑ_A or ϑ_B as ϑ_L. Let us take $\vartheta_L = \vartheta_A$. Because $\Phi/\textcircled{T} = \{\Phi\}$, the CCUREs are unnecessary. The PCUREs specify the control signals for all one-dimensional systolic arrays that realise matrix product whose three data variables A, B and C are moving.

Choose the space-time mapping that describes Ramakrishnan and Varman's array (Fig. 4–4). Our array runs in $(9m^2-9m+2)/2$ steps. The array of Ramakrishnan and Varman runs in $(12m^2+7m+2)/2$ steps. Both arrays require the same hardware for data flow. But our array is superior in terms of control requirement: it has a total of bit-width of four bits for control signals. Ramakrishnan and Varman's array requires five bits and an encoder and decoder for every cell.

As discussed in Sect. 5.3.4, the evolution control variable can be stationary. If all ϑ_E-paths of $\mathrm{paths}(\Phi, \vartheta_E)$ are mapped to distinct cells, we only need one buffer in each cell to store the corresponding element of E mapped to the cell. Here is a space-time mapping that has this property:

$$\pi = \begin{bmatrix} m+1 & m & 1 \\ 1 & m & 0 \end{bmatrix} \tag{5.29}$$

Both data variable C and control variable E are stationary. Each cell needs a buffer for accumulating one element of C. Similarly, each cell needs a buffer to store one element of E. The buffers for C must be initialised to zero and the buffers for E to e_s before the first step.

The systolic array consists of m^2 cells and runs in m^3+m^2-1 steps. There is an array proposed by Kumar and Tsai [37] that has the same number of cells but runs in $3m^2-m+3$ steps. The lower latency is obtained at the expense of one additional link for data variable B. Elements of B must be frequently routed from one link to the other, complicating the control circuitry.

To compare the hardware required for the control flow: both arrays use a two-bit buffer for E in each cell. In Kumar and Tsai's array, there are three one-bit moving control variables; two control variables need two buffers per channel;

one needs one buffer. In our array, there are also three one-bit moving control variables: F_1, F_2, and L; F_1 and F_2 do not need buffers; L needs m buffers per channel.

5.6.2 Dynamic Programming

Recall that the CCUREs consist of two computation control variables P and Q (Sect. 3.7.1.2). We choose $\vartheta_P=(-1,0,0)$ and $\vartheta_Q=(0,2,1)$.

Since data variable E has the same variable name as the evolution control variable, we rename it ot E' to avoid confusion. It is desirable to choose one of the three data dependence vectors ϑ_A, ϑ_C and $\vartheta_{E'}$ as the evolution control dependence vector ϑ_E. We choose $\vartheta_E=\vartheta_{E'}$.

Let us first consider the specification of the initialisation control variables. We choose $\textcircled{F}=\{\mathrm{fst}(\Phi,\vartheta_E)_1,\mathrm{fst}(\Phi,\vartheta_E)_2\}$:

$$\begin{aligned}\mathrm{fst}(\Phi,\vartheta_E)_1 &= \Phi\cap\{(i,j,k)\mid j-i=2k-1\}\\ \mathrm{fst}(\Phi,\vartheta_E)_2 &= \Phi\cap\{(i,j,k)\mid j-i=2k-2\}\end{aligned}$$

We need to choose $\vartheta_{F_{1,1}},\vartheta_{F_{1,2}}\in\mathrm{lin}(\mathrm{fst}(\Phi,\vartheta_E)_1)$ and $\vartheta_{F_{2,1}},\vartheta_{F_{2,2}}\in\mathrm{lin}(\mathrm{fst}(\Phi,\vartheta_E)_2)$. Let us choose $\vartheta_{F_{1,1}}=\vartheta_{F_{2,1}}=(0,2,1)$ and $\vartheta_{F_{1,2}}=\vartheta_{F_{2,2}}=(-2,0,1)$.

The specification of termination control is simple. $\mathrm{lst}(\Phi,\vartheta_E)$ is a facet of the index space. We choose $\textcircled{L}=\{\mathrm{lst}(\Phi,\vartheta_E)\}$. We name the associated termination control variable L and choose $\vartheta_L=(0,1,0)\in\mathrm{lin}(\mathrm{lst}(\Phi,\vartheta_E))$.

The following space-time mapping describes the one-dimensional array in [68]:

$$\pi = \begin{bmatrix} -4 & 2(m+2) & -2 \\ -1 & 1 & 0 \end{bmatrix}$$

Data variable C is stationary. Elements of C must be loaded before the computation starts and recovered after the computation completes. Elements of A, B, D, E', E $F_{1,1}$, $F_{1,2}$, $F_{2,1}$, $F_{2,2}$ and L encounter a delay of $2n+3$, 1, $2n+1$, 3, 3, $2n+2$, $2n+2$, 2, 2 and $2n+3$ at the respective channel (remember that our analysis is conducted with respect to the τ-model).

5.7 Conclusion

We have provided a construction of propagation control flow for one-dimensional arrays from n-dimensional UREs. For notational convenience, we have presented the PCUREs in terms of two programs: a host program that is executed by the host and that is responsible for the injection of the propagation control signals, and a cell program that is executed by all cells of the array and that specifies the output control signals of a cell based on the input control signals received by that cell. Similarly to the control UREs constructed in Chap. 3, the control dependence vectors of the propagation control variables in the PCUREs remain to be determined. We have summarised in a procedure how to choose these dependence vectors. We have stated the mapping conditions for the pipelined UREs constructed for one-dimensional arrays. Once the control dependence vectors have been chosen, as demonstrated by two examples in Sect. 5.6, finding one-dimensional arrays is just finding valid space-time mappings.

The idea underlying the construction of the PCUREs may also shed light on solving other issues in systolic design. As has been proved in [70], a systolic array is a system of space-time UREs. Once we have formulated a systolic array model and the corresponding mapping conditions for the validity of space-time mappings, we can conduct our analysis at the source level, i.e., independently of the space-time mapping. This has the benefit that we do not have to be concerned with temporal and spatial issues of the systolic array. Rather, the synthesis of control signals can be considered as a process of program construction and transformation.

Let us now have a look at how an adaptation of the index space can simplify the specification of the initialisation and termination control variables. We describe the basic idea with respect to an artificial example shown in Fig. 5–11. $\mathrm{fst}(\Phi, \vartheta_E)$ and $\mathrm{lst}(\Phi, \vartheta_E)$ each contain two facets of the index space. Therefore, $|Ⓕ| \geqslant 2$ and $|Ⓛ| \geqslant 2$. Assume that $|Ⓕ| = |Ⓛ| = 2$. We then need four initialisation control variables and two termination control variables. A simple adaptation of the index space can reduce the number of each type of control variable by half. Let us denote the two initialisation control variables by F_1 and F_2 and the termination control variable by L. Let their control dependence vectors be as shown in Fig. 5–11(a).

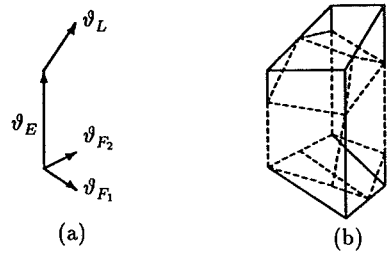


Figure 5–11: Adaptation of the index space for the construction of the PCUREs (a) The control dependence vectors ϑ_E, ϑ_{F_1}, ϑ_{F_2}, ϑ_L are shown. (b) The dashed polytope depicts the index space Φ. Both $\mathrm{fst}(\Phi, \vartheta_E)$ and $\mathrm{lst}(\Phi, \vartheta_E)$ are not facets of Φ. The polytope that encloses Φ depicts the adapted index space Φ_A.

Let $H(F_1, F_2)$ be the unique hyperplane that is spanned by ϑ_{F_1} and ϑ_{F_2} and that bounds the index space from below:

$$H(F_1, F_2) \;=\; \{I \mid (\vartheta_{F_1} \times \vartheta_{F_2})I = \alpha\}$$

where "$\times$" denotes vector product. We extend the index space along $-\vartheta_E$ until the points intersect $H(F_1, F_2)$. Similarly, let $H(L)$ be a hyperplane whose normal is orthogonal to ϑ_L and that bounds the index space from above (the choice of this hyperplane is not unique):

$$H(L) \;=\; \{I \mid \pi_L I = \beta \wedge \pi_L \vartheta_L = 0\}$$

We then extend the index space along ϑ_E until the extension intersects $H(L)$. The *adapted* index space, denoted by Φ_A, is given by

$$\begin{aligned}\Phi_A \;=\; & (\mathrm{planes}(\mathrm{fst}(\Phi, \vartheta_E), \vartheta_{F_1} \times \vartheta_{F_2}) \cap \mathrm{rays}(\mathrm{fst}(\Phi, \vartheta_E), -\vartheta_E)) \;\dot\cup\; \Phi \;\cup \\ & (\mathrm{planes}(\mathrm{lst}(\Phi, \vartheta_E), \pi_L) \cap \mathrm{rays}(\mathrm{lst}(\Phi, \vartheta_E), \vartheta_E))\end{aligned}$$

Both $\mathrm{fst}(\Phi_A, \vartheta_E)$ and $\mathrm{lst}(\Phi_A, \vartheta_E)$ are facets of the adapted index space Φ_A.

Φ_A is a superset of Φ. A fixed space-time mapping may yield the same latency and processor space in both cases. This happens when the extension does not change the longest dependence path and the size of the projection space. If such an adaptation of the index space degrades the performance of the systolic array too much, it had better be avoided.

Finally, the PCUREs presented in this chapter work for systolic arrays of r-dimensions $(1 < r < n)$ that conform to the systolic array model defined by Def. 2.2

and that are described by the space-time mappings defined in Def. 2.4, whose validity is given by: Def. 2.4. The reason is that Lemma 5.3, on which the construction of the PCUREs (or more precisely, the construction of the initialisation control variables) depends, is independent of the dimensionality of the systolic array. Lemma 5.3 is still valid if σ is an $r \times n$ allocation matrix rather than an allocation vector. The presentation of the PCUREs is based on the space-time diagram. The space-time diagram for an r-dimensional array is just the combination of the r-dimensional processor space $\mathcal{P}$ and the one-dimensional scheduling space $\mathcal{S}$; it is of dimension $r+1$. The host programs in Tabs. 5–1, 5–3, 5–4 and 5–5 rely on functions **input** and **inject**, which depend on function **pi**. To make these host programs applicable to r-dimensional arrays, **pi** can be generalised to:

$$\begin{aligned} &\mathsf{pi} : \mathcal{V} \to \Phi_{\mathsf{fst}} \to \mathbf{Z},\ \Phi_{\mathsf{fst}} = (\bigcup V : V \in \mathcal{V} : \mathsf{fst}(\Phi_V, \vartheta_V)) \\ &\mathsf{pi}(V(I)) \ = \ P, \ \text{ where } \ P \in \mathsf{fst}(\mathcal{P}, \beta_V) \wedge P \overset{\beta_V}{\to} \sigma I \end{aligned} \tag{5.30}$$

where $\beta_V = (\overline{\vartheta}_V.2, \overline{\vartheta}_V.3, \cdots, \overline{\vartheta}_V.r)$ represents the r-vector formed by the last r processor coordinates of $\overline{\vartheta}_V$. Similarly, function **po** can be generalised to:

$$\begin{aligned} &\mathsf{po} : \mathcal{V} \to \Phi_{\mathsf{lst}} \to \mathbf{Z},\ \Phi_{\mathsf{lst}} = (\bigcup V : V \in \mathcal{V} : \mathsf{lst}(\Phi_V, \vartheta_V)) \\ &\mathsf{po}(V(I)) \ = \ P, \ \text{ where } \ P \in \mathsf{lst}(\mathcal{P}, \beta_V) \wedge \sigma I \overset{\beta_V}{\to} P \end{aligned} \tag{5.31}$$

However, the PCUREs constructed here can be greatly simplified in the case of $(n-1)$-dimensional systolic arrays, since valid space-time mappings are bijections from $\mathbf{Q}^n$ to $\mathbf{Q}^n$. This has the following two consequences. First, the hierarchical decomposition of the index space, as in Sect. 5.4, in order to construct the PCUREs for n-dimensional UREs is not necessary. The PCUREs that are presented with respect to three-dimensional UREs apply for n-dimensional UREs. Second, the construction of the initialisation control variables can be simplified. We only need to associate one initialisation control variable with every block in Ⓟ. The construction of the initialisation control variables can be conducted in exactly the same way as the termination control variables. This halves the number of initialisation control variables required.

Now, we have two different constructions of the PCUREs for $(n-1)$-dimensional arrays. In general, the PCUREs that were constructed in Chap. 3 are superior to the PCUREs constructed here, partly because they are simpler and partly because they do not degrade the latency of the systolic array as they do in the case of one-dimensional arrays (Sect. 5.1).

6

The Elimination of Propagation Control Flow

6.1 Introductory Remarks

Why would we want to eliminate propagation control signals for systolic arrays? Consider source UREs that do not require any CCUREs, i.e, that satisfy $\Phi/\textcircled{T} = \{\Phi\}$. That is, all cells in the systolic array are identical. At any given step, a cell is in one of two possible states: the *computation state* when it is performing a common computation as specified by the source UREs and the *propagation state* when it is propagating input/output data. The PCUREs specify when a cell is in which of these two states.

Recall that pattern specifies the distribution of the input data of a variable for r-dimensional arrays. (pattern has also been defined equivalently by input of (4.3) and pi of (5.30), which specify the temporal and spatial distribution of the input data, respectively.) If a variable is not given a value at a given step, the undefined value $\perp$ is injected. If we substitute any value for $\perp$, we still obtain the output data specified by the source UREs from the systolic array. We use this fact to eliminate the propagation control signals in the systolic array. Rather than replacing $\perp$ arbitrarily, we choose the substitute value carefully. We then let all cells of the array adopt the computation state at every step. If we still obtain the

output data specified by the source UREs, we can do without propagation control signals.

Let us examine this scheme formally. Because of our assumption that $\Phi/\textcircled{T} = \{\Phi\}$, i.e., that $(\forall\ V : V \in \mathcal{V} : \Phi_V = \Phi)$. a variable V in the extended source UREs is specified by the two equations:

$$I \in \Phi \ \twoheadrightarrow\ V(I) \ = \ f(W(I - \vartheta_W), \cdots) \tag{6.1}$$

$$I \in \Psi_V \setminus \Phi_V \ \twoheadrightarrow\ V(I) \ = \ V(I - \vartheta_V) \tag{6.2}$$

The first equation is from the source UREs. The second equation results from the extension of the index space (Defs. 2.5 and 4.3). Propagation control is required iff the two equations for some data variable differ.

Since the space-time mappings for one-dimensional arrays are not injective, we discuss the elimination of propagation control with respect to space-time UREs (Sect. 2.4) rather than the source UREs. Assume that $W(\overline{I} - \overline{\vartheta}_W) = \perp$ at cell $\mathsf{place}(I)$ at time step $\mathsf{step}(I)$. Instead of letting the cell evaluate equation $V(\overline{I}) = V(\overline{I} - \overline{\vartheta}_V)$ at that time step, we let it evaluate the equation $V(\overline{I}) = f(W(\overline{I} - \overline{\vartheta}_W), \cdots)$ by substituting $W(\overline{I} - \overline{\vartheta}_W) = w_{\overline{I}}$ for $W(\overline{I} - \overline{\vartheta}_W) = \perp$. The choice of value $w_{\overline{I}}$ will be explained later. We still obtain the output data specified by the source UREs from the systolic array iff the following two conditions are satisfied:

- Cond. 1. W is undefined at all points of the $\overline{\vartheta}_W$-path that passes through the point $\overline{I}$, i.e., $(\forall\ \overline{J} : \overline{J} \in p(\overline{\Psi}, \overline{\vartheta}_W, \overline{I}) : W(\overline{J}) = \perp)$. (This ensures that value $w_{\overline{I}}$ can be injected from input cells, as is required in systolic array models.)
- Cond. 2. $V(\overline{I} - \overline{\vartheta}_V) = f(W(\overline{I} - \overline{\vartheta}_W), \cdots)$. (This preserves the behaviour of the (extended) source UREs.)

To satisfy Cond. 1, we choose the space-time mapping carefully. To satisfy Cond. 2, we exploit the algebraic properties of the operators in the equation (6.1).

There cannot be a general method for all systolic algorithms, since the algebraic properties of operators are problem-specific. We shall present our method with respect to a restricted class of UREs called *inner-product UREs*. The inner-product UREs are a subclass of UREs as defined in [62]. The basic idea can be generalised to other systolic algorithms.

Inner-Product UREs:

$$A(I) = \begin{cases} I \in \mathrm{in}(\Phi_A, \vartheta_A) & \twoheadrightarrow \quad A(I) = a_{in_A(I)} \\ I \in \Phi_A & \twoheadrightarrow \quad A(I) = A(I - \vartheta_A) \end{cases}$$

$$B(I) = \begin{cases} I \in \mathrm{in}(\Phi_B, \vartheta_B) & \twoheadrightarrow \quad B(I) = b_{in_B(I)} \\ I \in \Phi_B & \twoheadrightarrow \quad B(I) = B(I - \vartheta_B) \end{cases}$$

$$C(I) = \begin{cases} I \in \mathrm{in}(\Phi_C, \vartheta_C) & \twoheadrightarrow \quad C(I) = c_{in_C(I)} \\ I \in \Phi_C & \twoheadrightarrow \quad C(I) = C(I - \vartheta_C) \oplus (A(I - \vartheta_A) \otimes B(I - \vartheta_B)) \end{cases}$$

where $\Phi_A = \Phi_B = \Phi_C = \Phi$, and the functions in_A, in_B and in_C are defined in Sect. 2.2. The output equations are irrelevant for our discussion and are thus omitted. We assume that the values represented by variables A, B and C belong to a set M. $\otimes$ and $\oplus$ are binary operations on the set M. To satisfy Cond. 2, we postulate the existence of elements u, z_1, z_2 (which may be equal) in M satisfying:

$$\begin{aligned} &(\forall\, s : s \in M : s \oplus u = s) \\ &(\forall\, s : s \in M : z_1 \otimes s = u) \\ &(\forall\, s : s \in M : s \otimes z_2 = u) \end{aligned} \tag{6.3}$$

The element u is a right unit of $\oplus$. The idea is to replace $A(\overline{I}) = \perp$ ($B(\overline{I}) = \perp$) by $A(\overline{I}) = z_1$ ($B(\overline{I}) = z_2$). Examples that are instances of inner-product UREs are 1-D convolution (Sect. 6.2) and matrix product. In this case, $\oplus$ becomes addition, $\otimes$ becomes multiplication, M becomes $\mathbf{R}$, and $z_1 = z_2 = u = 0$. 0 is both the unit element for addition and the zero element for multiplication.

The rest of this chapter is organised as follows. Sect. 6.2 motivates the elimination of propagation control signals with the example of 1-D convolution. Sect. 6.3 discusses the elimination of propagation control signals for $(n-1)$-dimensional systolic arrays. Sect. 6.4 continues our presentation for one-dimensional arrays.

6.2 Example: 1-D Convolution

Given two input sequences $x_1, x_2 \cdots, x_m$ and $w_1, w_2 \cdots, w_m$, 1-D convolution computes an output sequence $y_1, y_2 \cdots, y_{2m-1}$ where y_i is given by

Specification: $(\forall\, i : 0 < i < 2m : \; y_i = (\sum\, k : 0 < k \leqslant m \wedge 0 \leqslant i - k < m : x_{i-k+1} w_k))$

Source UREs:

$$X(i,k) = \begin{cases} 0 \leqslant i < m \wedge k = 0 & \twoheadrightarrow x_{i+1} \\ m \leqslant i \leqslant 2m-2 \wedge k = 0 & \twoheadrightarrow 0 \\ 0 = i \wedge 0 < k < m & \twoheadrightarrow 0 \\ 0 < i < 2m \wedge 0 < k \leqslant m & \twoheadrightarrow X(i-1,k-1) \quad \bullet \end{cases}$$

$$W(i,k) = \begin{cases} 0 = i \wedge 0 < k \leqslant m & \twoheadrightarrow w_k \\ 0 < i < 2m \wedge 0 < k \leqslant m & \twoheadrightarrow W(i-1,k) \quad \bullet \end{cases}$$

$$Y(i,k) = \begin{cases} 0 < i < 2m \wedge k = 0 & \twoheadrightarrow 0 \\ 0 < i < 2m \wedge 0 < k \leqslant m & \twoheadrightarrow Y(i,k-1) + X(i-1,k-1)W(i-1,k) \quad \bullet \end{cases}$$

$$y_i = \begin{cases} 0 < i < 2m \wedge k = m & \twoheadrightarrow Y(i,k) \end{cases}$$

Index Space: $\Phi = \{(i,k) \mid 0 < i < 2m \wedge 0 < k \leqslant m\}$

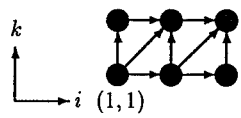
Data Dependence Matrix: $\mathcal{D} = [\vartheta_X \ \vartheta_Y \ \vartheta_W] = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$

First Computation Points:

$$\begin{aligned} \mathrm{fst}(\Phi, \vartheta_X) &= \{(i,k) \mid (0 < i < 2m \wedge 1 = k) \vee \\ &\qquad (1 = i \wedge 1 < k \leqslant m)\} \\ \mathrm{fst}(\Phi, \vartheta_Y) &= \{(i,k) \mid 0 < i < 2m \wedge 1 = k\} \\ \mathrm{fst}(\Phi, \vartheta_W) &= \{(i,k) \mid 1 = i \wedge 0 < k \leqslant m\} \end{aligned}$$

Last Computation Points: $\mathrm{lst}(\Phi, \vartheta_Y) = \{(i,k) \mid 0 < i < 2m \wedge k = m\}$

Data Dependence Graph $(m=2)$:



The UREs for 1-D convolution satisfy $\Phi/\textcircled{T} = \{\Phi\}$. Thus, CCUREs are not required. Only PCUREs are necessary to tell whether a cell is in a state of computation or propagation. Let us consider the specification of the PCUREs. $\mathfrak{B}$ contains the following supporting half-spaces to the index space:

$$\begin{aligned} B_1 &= \{(i,j,k) \mid i \geqslant 1\} \\ B_2 &= \{(i,j,k) \mid i \leqslant 2m-1\} \\ B_3 &= \{(i,j,k) \mid k \geqslant 1\} \\ B_4 &= \{(i,j,k) \mid k \leqslant m\} \end{aligned}$$

The PCUREs follow from (3.18). If we observe that the two hyperplanes corresponding to B_1 and B_2 are parallel, we can combine the two associated control variables into one, denoted by P_1. This reduces the number of control values from four to three. Variable P_1 defines three control values at the domains $\Psi \cap \complement_{\mathbb{Z}^n} B_1$, $\Psi \cap B_1 \cap B_2$ and $\Psi \cap \complement_{\mathbb{Z}^n} B_2$. It suffices to use the same control value for the two domains $\Psi \cap \complement_{\mathbb{Z}^n} B_1$ and $\Psi \cap \complement_{\mathbb{Z}^n} B_2$, because both contain pipelining points. We denote the control value defined at $\Psi \cap B_1 \cap B_2$ by p_1 and the control value at $\Psi \cap \complement_{\mathbb{Z}^n} B_1$ and $\Psi \cap \complement_{\mathbb{Z}^n} B_2$ by $\overline{p}_1$. For reasons of symmetry, we use one control variable, P_2, for B_3 and B_4. We denote the control value defined at $\Psi \cap B_3 \cap B_4$ by p_2 and the control value defined at $\Psi \cap \complement_{\mathbb{Z}^n} B_3$ and $\Psi \cap \complement_{\mathbb{Z}^n} B_4$ by $\overline{p}_2$. By (3.19), a cell is in the computation state at a time step if $P_1(\overline{I}) = p_1 \wedge P_2(\overline{I}) = p_2$ holds at the cell at the step and in the propagation state otherwise.

We choose the space-time mapping (Figs. 6–1(a), (c) and (d)).

$$\pi = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The projection vector is $u = (1, 1)$.

Both X and W are specified by pipelining equations over the extended index space. The only reason why we need propagation control is that Y is defined by the equation $Y(I) = Y(I - \vartheta_Y)$ at the pipelining points and by the equation $Y(I) = Y(I - \vartheta_Y) + X(I - \vartheta_X) W(I - \vartheta_W)$ at the computation points. We can eliminate propagation control signals in the systolic array if we do the following:

- Inject 0 instead of $\perp$ for variable W at steps $t_{\text{fst}}, t_{\text{fst}}+2, t_{\text{fst}}+4, \cdots$ excluding those at which $w_1, w_2, \cdots, w_m$ are injected.

- Evaluate Y at the pipelining points $\overline{I}$ using equation $Y(\overline{I}) = Y(\overline{I} - \overline{\vartheta}_Y) + X(\overline{I} - \overline{\vartheta}_X) W(\overline{I} - \overline{\vartheta}_W)$ rather than $Y(\overline{I}) = Y(\overline{I} - \overline{\vartheta}_Y)$.

To see why Conds. 1 and 2 are satisfied, we classify the pipelining points of Ψ^{P} (Def. 2.5) into two categories:

- $\Psi_Y^{\perp}$ contains the points at which Y is undefined. At these points I, we can replace the equation $Y(I) = Y(I - \vartheta_Y)$ by any equation.

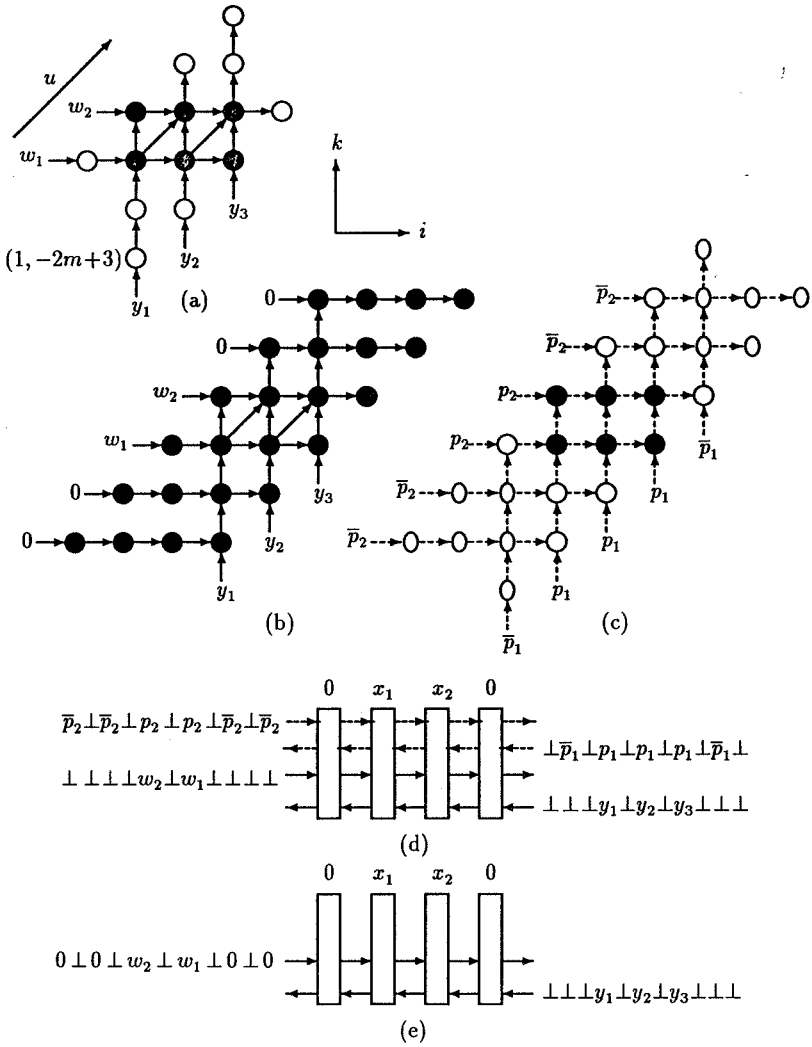


Figure 6–1: (a) The extended dependence graph with respect to projection vector $u=(1,1)$ $(m=2)$. (b) The extension of the extended data dependence graph of (a) for the purpose of substituting $W(\overline{I}-\overline{\vartheta}_W)=0$ for $W(\overline{I}-\overline{\vartheta}_W)=\perp$. (c) The extension of the extended data dependence graph of (a) by Def. 2.5 for propagation control variables P_1 and P_2. The ovals depict the new points generated in the extension. (d) The systolic array with the distribution of both data and control signals at the first step. (e) The systolic array without the propagation control signals.

- $\Psi_Y^s \cup \Psi_Y^d$ contains the points at which input/output data of Y are propagated. Because ϑ_Y and ϑ_W are orthogonal, W is undefined at these points I:

$$(\forall\, I : I \in \Psi_Y^s \cup \Psi_Y^d : (\forall\, J : J \in p(\Psi, \vartheta_W, I) : W(J) = \perp))$$

The space-time mapping (6.4) is a bijection from $\mathbb{Q}^n$ to $\mathbb{Q}^n$ $(n=2)$. So

$$(\forall\, \overline{I} : \overline{I} \in \overline{\Psi}_Y^s \cup \overline{\Psi}_Y^d : (\forall\, \overline{J} : \overline{J} \in p(\overline{\Psi}, \overline{\vartheta}_W, \overline{I}) : W(\overline{J}) = \perp))$$

Thus, Cond. 1 is satisfied. 0 is both the unit element for addition and the zero element for multiplication. Substituting $W(\overline{I}-\overline{\vartheta}_W)=0$ for $W(\overline{I}-\overline{\vartheta}_W)=\perp$ at the points of $\overline{\Psi}_Y^s \cup \overline{\Psi}_Y^d$ gives rise to:

$$Y(\overline{I}-\overline{\vartheta}_Y) \;=\; Y(\overline{I}-\overline{\vartheta}_Y)+X(\overline{I}-\overline{\vartheta}_X)W(\overline{I}-\overline{\vartheta}_W)$$

Thus, Cond. 2 is satisfied.

The previous illustration can be understood by an extension of the extended index space (Fig. 6–1(b)); the extension is carried out by first replicating ϑ_W throughout the extended index space and then extending ϑ_W and $-\vartheta_W$ until the points so created are at the boundary of the processor space. Fig. 6–1(e) displays the systolic array without propagation control signals.

6.3 Systolic Arrays of $n-1$ Dimensions

We present necessary and sufficient conditions for the elimination of propagation control signals for $(n-1)$-dimensional arrays synthesised from inner-product UREs. These conditions depend on the data dependence structure of the source UREs and on the projection vector.

Theorem 6.1 *The PCUREs are unnecessary for inner-product UREs iff*

$$\begin{aligned}&(\forall\, I : I \in \Psi_C^s \cup \Psi_C^d : C(I) \neq \perp \implies \\ &(\forall\, J : J \in p(\Psi, \vartheta_A, I) : A(J) = \perp) \vee (\forall\, J : J \in p(\Psi, \vartheta_B, I) : B(J) = \perp))\end{aligned} \tag{6.4}$$

Proof. We show that the inner-product UREs can be refined to a system of unconditional UREs defined over the extended index space and, consequently, propagation control is unnecessary, iff hypothesis (6.4) holds.

The extended source UREs of the inner-product UREs follow from Def. 2.5:

$$
\begin{aligned}
A(I) &= \begin{cases} I \in \mathrm{in}(\Psi_A, \vartheta_A) & \twoheadrightarrow \quad A(I) = a_{\mathrm{in}_A(\mathcal{I}(I))} \\ I \in \Psi_A & \twoheadrightarrow \quad A(I) = A(I - \vartheta_A) \end{cases} \\
B(I) &= \begin{cases} I \in \mathrm{in}(\Psi_B, \vartheta_B) & \twoheadrightarrow \quad B(I) = b_{\mathrm{in}_B(\mathcal{I}(I))} \\ I \in \Psi_B & \twoheadrightarrow \quad B(I) = B(I - \vartheta_B) \end{cases} \\
C(I) &= \begin{cases} I \in \mathrm{in}(\Psi_C, \vartheta_C) & \twoheadrightarrow \quad C(I) = c_{\mathrm{in}_C(\mathcal{I}(I))} \\ I \in \Phi_C & \twoheadrightarrow \quad C(I) = C(I - \vartheta_C) \oplus (B(I - \vartheta_A) \otimes B(I - \vartheta_B)) \\ I \in \Psi_C^{\perp} \cup \Psi_C^{s} \cup \Psi_C^{d} & \twoheadrightarrow \quad C(I) = C(I - \vartheta_C) \end{cases}
\end{aligned}
\tag{6.5}
$$

The computation equations of both A and B over the extended index space Ψ are pipelining equations. The only reason why we need propagation control signals is that C is defined by different equations at pipelining and computation points. To eliminate propagation control signals, as discussed informally in Sect. 6.2, we must replace the defining equation at the pipelining points by the defining equation at the computation points. To preserve the behaviour of the initial source UREs, we introduce appropriately some new input equations for A or B or both.

Let $A_{\perp}$ ($B_{\perp}$) be the union of the ϑ_A-paths (ϑ_B-paths) that intersect $\Psi_C^{s} \cup \Psi_C^{d}$ and whose points are undefined points of A (B). The extension of the index space ensures

$$
\begin{aligned}
\mathrm{in}(A_{\perp}, \vartheta_A) \cap \mathrm{in}(\Psi_A, \vartheta_A) &= \varnothing \\
\mathrm{in}(B_{\perp}, \vartheta_B) \cap \mathrm{in}(\Psi_B, \vartheta_B) &= \varnothing \\
I \in \mathrm{in}(A_{\perp}, \vartheta_A) \twoheadrightarrow A(I) &= \perp \\
I \in \mathrm{in}(B_{\perp}, \vartheta_B) \twoheadrightarrow B(I) &= \perp
\end{aligned}
$$

Hence, the following defining equations of A and B are a refinement of their counterparts of (6.5):

$$
\begin{aligned}
A(I) &= \begin{cases} I \in \mathrm{in}(A_{\perp}, \vartheta_A) & \twoheadrightarrow \quad A(I) = z_1 \\ I \in \mathrm{in}(\Psi_A, \vartheta_A) & \twoheadrightarrow \quad A(I) = a_{\mathrm{in}_A(\mathcal{I}(I))} \\ I \in \Psi_A & \twoheadrightarrow \quad A(I) = A(I - \vartheta_A) \end{cases} \\
B(I) &= \begin{cases} I \in \mathrm{in}(B_{\perp}, \vartheta_B) & \twoheadrightarrow \quad B(I) = z_2 \\ I \in \mathrm{in}(\Psi_B, \vartheta_B) & \twoheadrightarrow \quad B(I) = b_{\mathrm{in}_B(\mathcal{I}(I))} \\ I \in \Psi_B & \twoheadrightarrow \quad B(I) = B(I - \vartheta_B) \end{cases}
\end{aligned}
\tag{6.6}
$$

Propagation control is unnecessary iff the following defining equations of C are a refinement of their counterparts of (6.5):

$$C(I) = \begin{cases} I \in \mathrm{in}(\Psi_C, \vartheta_C) & \twoheadrightarrow \quad C(I) = c_{in_C(\mathcal{I}(I))} \\ I \in \Psi_C & \twoheadrightarrow \quad C(I) = C(I-\vartheta_C) \oplus (A(I-\vartheta_A) \otimes B(I-\vartheta_B)) \end{cases} \tag{6.7}$$

Variable C is undefined for the domain $\Psi_C^{\perp}$. For the points of $\Psi_C^{\perp}$ and the points I of $\Psi_C^{s} \cup \Psi_C^{d}$ at which $C(I)=\perp$, the defining equation of C is free of interpretation.

Finally, we consider the points $I \in \Psi_C^{s} \cup \Psi_C^{d}$ at which $C(I) \neq \perp$:

"$\Longleftarrow$" Hypothesis (6.4) states

$$(\forall\, J : J \in p(\Psi, \vartheta_A, I) : A(J)=\perp) \text{ or}$$
$$(\forall\, J : J \in p(\Psi, \vartheta_B, I) : B(J)=\perp)$$

So, the introduction of the two new equations in (6.6) gives rises to:

$$(\forall\, J : J \in p(\Psi, \vartheta_A, I) : A(J)=z_1) \text{ or}$$
$$(\forall\, J : J \in p(\Psi, \vartheta_B, I) : B(J)=z_2)$$

In either case, assumption (6.3) ensures the equality of $C(I-\vartheta_C)$ and $C(I-\vartheta_C) \oplus (A(I-\vartheta_A) \otimes B(I-\vartheta_B))$. (The satisfaction of Cond. 1 is enforced by hypothesis (6.4) and the bijectivity of space-time mappings. The satisfaction of Cond. 2 is enforced by assumption (6.3).)

"$\Longrightarrow$" Assume that hypothesis (6.4) does not hold. Then there must be a point I in $\Psi_C^{s} \cup \Psi_C^{d}$ such that

- $C(I) \neq \perp$,
- there must be a point J in $p(\Psi, \vartheta_A, I)$ such that $A(J) \neq \perp$, and
- there must be a point K in $p(\Psi, \vartheta_B, I)$ such that $B(K) \neq \perp$.

This means that $p(\Psi, \vartheta_A, I) \not\subseteq A_{\perp}$ and $p(\Psi, \vartheta_B, I) \not\subseteq B_{\perp}$. Thus, Cond. 1 is not satisfied. □

The PCUREs are not needed for a special instance of inner-product UREs, independently of the projection vector, namely, when

$$\mathrm{rays}(\Phi_A, \vartheta_A) \cap \mathrm{rays}(\Phi_C, \vartheta_C) \;=\; \mathrm{rays}(\Phi_B, \vartheta_B) \cap \mathrm{rays}(\Phi_C, \vartheta_C) \;=\; \Phi$$

In this case, hypothesis (6.4) is always satisfied, independently of the projection vector. Examples include matrix product and 1-D convolution.

6.4 Systolic Arrays of One Dimension

Thm. 6.1 is not valid for one-dimensional arrays due to the non-injectivity of the space-time mapping. A counterexample is Ramakrishnan and Varman's one-dimensional array for matrix product (Fig. 4-4). Hypothesis (6.4) is only necessary but no longer sufficient for the elimination of propagation control in one-dimensional arrays. A sufficient condition must depend on the scheduling vector.

Theorem 6.2 *The PCUREs are unnecessary for inner-product UREs iff*

$$\begin{aligned}&(\forall\, \overline{I} : \overline{I} \in \overline{\Psi}^{\mathrm{s}}_C \cup \overline{\Psi}^{\mathrm{r}}_C \cup \overline{\Psi}^{\mathrm{d}}_C : C(\overline{I}) \neq \perp \implies \\ &(\forall\, \overline{J} : \overline{J} \in p(\Upsilon, \overline{\vartheta}_A, \overline{I}) : A(\overline{J}) = \perp) \vee (\forall\, \overline{J} : \overline{J} \in p(\Upsilon, \overline{\vartheta}_B, \overline{I}) : B(\overline{J}) = \perp))\end{aligned} \tag{6.8}$$

Proof. Similar to that of Thm. 6.1, except that the proof is conducted with respect to the space-time diagram Υ instead of the extended index space Ψ (In one-dimensional arrays, the pipelining points at which C may be defined also contain the relaying points of $\overline{\Psi}^{\mathrm{r}}_C$.) □

Take the UREs of (2.14) for matrix product as an example. The following valid space-time mapping satisfies hypothesis (6.8):

$$\pi = \begin{bmatrix} 6m-1 & 1 & 1 \\ 1 & 1 & -1 \end{bmatrix}$$

The elimination of propagation control may increase the latency of the array in order to satisfy the hypothesis (6.8). In the presence of propagation control signals, this hypothesis need not be satisfied. One valid improvement of the previous space-time mapping that violates the hypothesis is:

$$\pi = \begin{bmatrix} 2m-2 & 1 & 1 \\ 1 & 1 & -1 \end{bmatrix}$$

The latency of the former space-time mapping is $18m^2-18m+1$ steps, the latency of the latter is only $6m^2-9m+4$. According to the PCUREs for matrix product in Sect. 5.6.1, the latter array needs four one-bit links for propagation control signals.

6.5 Conclusion

We have made an attempt of a formal treatment of the elimination of propagation control for systolic arrays that do not require computation control. We first described informally two prerequisites, Conds. 1 and 2, for the elimination of the propagation control. We then presented necessary and sufficient conditions for the elimination of propagation control with respect to a restricted class of UREs: inner-product UREs. We distinguish two cases: the synthesis of $(n-1)$-dimensional arrays and of one-dimensional arrays. We have not given a general method for n-dimensional UREs. But the underlying idea is general enough to be applied to other systolic algorithms.

We have stated before that the synthesis of systolic arrays from programs is a process of program transformation. Most transformations that are reported in the literature are syntactic adaptations of the source program; they amount to a rearrangement of the data dependences in the source program. Transformations such as uniformisation and space-time mapping are typical syntactic adaptations. The transformations that we have applied here to inner-product UREs for the purpose of eliminating propagation control are semantic adaptations; they take semantic aspects of the source programs into account. The relevance of semantic adaptations to systolic design cannot be overemphasised. By realising that some variables always denote the same value, wrap-around channels can be eliminated from systolic arrays for transitive closure [43]. The latency of two-dimensional arrays for band matrix product can be reduced by counting the accumulating index (k in Ex. 2.1) down rather than up [31]. We succeeded in eliminating diagonal channels in systolic arrays for pyramidal algorithms by means of removing redundant computations in the source program [50]. A more formal treatment of the exploitation of commutativity and associativity of operators in systolic design is reported in [65]. We expect to see more work along this line.

7

The Loading, Recovery and Access of Stationary Data

7.1 Introductory Remarks

The control UREs provide a complete specification of control signals for systolic arrays if all the data variables are moving. In the presense of stationary data variables, a local processor memory is required. The space-time mapping does not provide any help in handling stationary data. Thus, a scheme for loading, recovering and accessing stationary data and the according control signals required remain to be developed. In practice, it is sometimes advantageous to selectively make certain data variables stationary in order to satisfy certain performance requirements, e.g., to reduce the latency and/or the size of the systolic array. Thus, the systematic derivation of a scheme for handling stationary data is of practical importance.

We describe a systematic method for the loading, recovery and access of stationary data in systolic arrays; the method exhibits a continuation of the idea underlying the construction of the control UREs, especially the idea underlying the construction of the PCUREs for one-dimensional arrays. The method is applied after the place function has been chosen, because the place function determines

which data variables are stationary. The basic idea is to add new variables, which constitute a specification for the loading, recovery and access of stationary data, to the source UREs. This specification leaves open the choice of dependence vectors of the new variables and the step function. Therefore, the search of schemes for the loading, recovery and access of stationary data reduces to a search of dependence vectors for the new variables and of step functions.

The new variables added to the source UREs are pipelining variables, each is associated with one dependence vector. They are divided into three classes:

- *Loading (data) variables* are responsible for the loading of the elements of stationary data variables of the source UREs.
- *Recovery (data) variables* are responsible for the recovery of the elements of stationary data variables of the source UREs.
- *Address (control) variables* provide the addresses for the access of the elements of stationary data variables of the source UREs.

We describe the construction of loading, recovery and address variables for r-dimensional arrays $(0<r<n)$ described by n-dimensional UREs and the space-time mappings of Def. 2.3. The construction relies on the notion of planes(Φ, σ), i.e., the inverse image of the r-dimensional processor space in $\mathbf{Q}^n$. Once a specification for the loading, recovery and access of stationary data variables is added to the source UREs, the synthesis of control signals for the new source UREs follows directly from the methods described in Chaps. 3 and 5. We shall use matrix product and dynamic programming for illustration.

The rest of this chapter is organised as follows. Sect. 7.2 describes the construction of loading and recovery variables. Sect. 7.3 describes the construction of address variables. Sect. 7.4 contains the conclusion of this chapter.

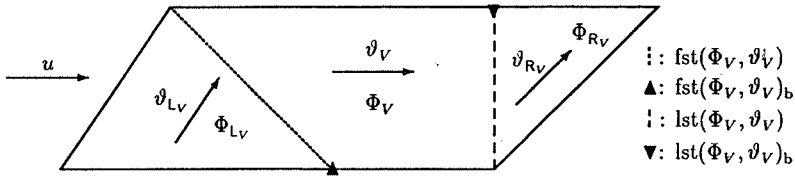


Figure 7–1: The construction of loading and recovery variables ($n=2$).

7.2 The Loading and Recovery of Stationary Data

We associate two variables with every stationary data variable in the source UREs: loading variable L_V for loading the elements of V and recovery variable R_V for recovering the elements of V. We must choose dependence vectors ϑ_{L_V} and ϑ_{R_V} such that L_V and R_V are moving.

Recall that the input data of V are initialised at the points of $\mathrm{in}(\Phi_V, \vartheta_V)$ and used for the first time at the points of $\mathrm{fst}(\Phi_V, \vartheta_V)$, and the output data of V are available at the points of $\mathrm{lst}(\Phi_V, \vartheta_V)$. The construction of L_V and R_V is symmetrical. Let us consider the loading of the input data first (Fig. 7–1). The domain Φ_{L_V} of loading variable L_V contains the points that propagate the input data of V during the loading phase:

$$\Phi_{\mathsf{L}_V} \;=\; (\mathrm{planes}(\Phi, \sigma) \cap \mathrm{rays}(\mathrm{fst}(\Phi_V, \vartheta_V) - \vartheta_{\mathsf{L}_V})) \setminus \mathrm{fst}(\Phi_V, \vartheta_V)$$

Since one ϑ_{L_V}-path serves to load one element of stationary variable V, we must choose dependence vector ϑ_{L_V} such that each ϑ_{L_V}-path contains only one point of $\mathrm{fst}(\Phi_V, \vartheta_V)$.

To determine the initialisation of variable L_V with the input data of V, we partition $\mathrm{fst}(\Phi_V, \vartheta_V)$ into two subsets: the set $\mathrm{fst}(\Phi_V, \vartheta_V)_b$ contains the points that are mapped to border cells and the set $\mathrm{fst}(\Phi_V, \vartheta_V) \setminus \mathrm{fst}(\Phi_V, \vartheta_V)_b$ contains the points that are mapped to internal cells, where

$$\mathrm{fst}(\Phi_V, \vartheta_V)_b \;=\; \{I \mid I \in \mathrm{fst}(\Phi_V, \vartheta_V) \wedge I - \vartheta_{\mathsf{L}_V} \notin \mathrm{planes}(\Phi, \sigma)\}$$

The loading of the input data $V(I-\vartheta_V)$ for $I \in \mathrm{fst}(\Phi_V, \vartheta_V)_\mathrm{b}$ is unnecessary; $\mathsf{L}_V(I-\vartheta_{\mathsf{L}_V})$ is initialised with the input value that $V(I-\vartheta_V)$ is initialised with. The initialisation of L_V with the remaining input data is defined as follows. The two sets $\mathrm{in}(\Phi_{\mathsf{L}_V}, \vartheta_{\mathsf{L}_V})$ and $\mathrm{fst}(\Phi_V, \vartheta_V) \backslash \mathrm{fst}(\Phi_V, \vartheta_V)_\mathrm{b}$ are isomorphic by the bijection

$$\ell_V : \mathrm{in}(\Phi_{\mathsf{L}_V}, \vartheta_{\mathsf{L}_V}) \;\rightarrow\; \mathrm{fst}(\Phi_V, \vartheta_V) \backslash \mathrm{fst}(\Phi_V, \vartheta_V)_\mathrm{b}, \quad \ell_V(I) = J, \quad \text{where } I \overset{\vartheta_{\mathsf{L}_V}}{\rightarrow} J$$

$\mathsf{L}_V(I)$ is initialised with the input value that $V(J-\vartheta_V)$ is initialised with. It is then pipelined along ϑ_{L_V} to the corresponding point of $\mathrm{fst}(\Phi_{\mathsf{L}_V}, \vartheta_{\mathsf{L}_V})$.

Finally, we initialise variable V with its input data loaded via variable L_V by substituting $\mathsf{L}_V(I-\vartheta_{\mathsf{L}_V})$ for all arguments $V(I-\vartheta_V)$, for every $I \in \mathrm{fst}(\Phi_V, \vartheta_V)$.

Next, we consider the recovery of the output data of V (Fig. 7-1). The domain Φ_{R_V} of recovery variable R_V contains the points that propagate the output data of V during the recovery phase:

$$\Phi_{\mathsf{R}_V} \;=\; (\mathrm{planes}(\Phi, \sigma) \cap \mathrm{rays}(\mathrm{lst}(\Phi_V, \vartheta_V), \vartheta_{\mathsf{R}_V})) \setminus \mathrm{lst}(\Phi_V, \vartheta_V)$$

Since one ϑ_{R_V}-path serves to recover one element of stationary variable V, we must choose dependence vector ϑ_{R_V} such that each ϑ_{R_V}-path contains only one point of $\mathrm{lst}(\Phi_V, \vartheta_V)$.

The recovery of the output data of V is symmetric to the loading of the input data of V. We partition $\mathrm{lst}(\Phi_V, \vartheta_V)$ into two subsets: the set $\mathrm{lst}(\Phi_V, \vartheta_V)_\mathrm{b}$ contains the points that are mapped to border cells and the set $\mathrm{lst}(\Phi_V, \vartheta_V) \backslash \mathrm{lst}(\Phi_V, \vartheta_V)_\mathrm{b}$ contains the points that are mapped to internal cells, where

$$\mathrm{lst}(\Phi_V, \vartheta_V)_\mathrm{b} \;=\; \{I \mid I \in \mathrm{lst}(\Phi_V, \vartheta_V) \wedge I + \vartheta_{\mathsf{R}_V} \notin \mathrm{planes}(\Phi, \sigma)\}$$

The recovery of the output data $V(I)$ for $I \in \mathrm{lst}(\Phi_V, \vartheta_V)_\mathrm{b}$ is unnecessary. The recovery of the output data $V(I)$ for $I \in \mathrm{lst}(\Phi_V, \vartheta_V) \setminus \mathrm{lst}(\Phi_V, \vartheta_V)_\mathrm{b}$ is defined as follows. The two sets $\mathrm{lst}(\Phi_{\mathsf{R}_V}, \vartheta_{\mathsf{R}_V})$ and $\mathrm{lst}(\Phi_V, \vartheta_V) \setminus \mathrm{lst}(\Phi_V, \vartheta_V)_\mathrm{b}$ are isomorphic by the bijection

$$r_V : \mathrm{lst}(\Phi_{\mathsf{R}_V}, \vartheta_{\mathsf{R}_V}) \;\rightarrow\; \mathrm{lst}(\Phi_V, \vartheta_V) \setminus \mathrm{lst}(\Phi_V, \vartheta_V)_\mathrm{b}, \quad r_V(I) = J, \quad \text{where } J \overset{\vartheta_{\mathsf{R}_V}}{\rightarrow} I$$

To recover stationary variable V, we must initialise R_V with the output data of V. We substitute $\mathsf{R}_V(I)$ for all target variables $V(I)$, for every $I \in \mathrm{lst}(\Phi_V, \vartheta_V)$. $\mathsf{R}_V(I)$ is then pipelined along ϑ_{R_V} to the corresponding point of $\mathrm{lst}(\Phi_{\mathsf{R}_V}, \vartheta_{\mathsf{R}_V})$.

The specification of stationary variable V must be updated to reflect the change caused by the introduction of loading variable L_V and recovery variable R_V in the source UREs. This update and the specification of L_V and R_V are summarised in a procedure.

Procedure 7.1 (Construction of loading variable L_V and recovery variable R_V for stationary variable V of the source UREs given a place function)

- Delete the input and output equations of variable V.
- The specification of loading variable L_V is as follows:
  - Substitute $\mathsf{L}_V(I-\vartheta_{\mathsf{L}_V})$ for all arguments $V(I-\vartheta_V)$ of the source UREs, where $I \in \mathrm{fst}(\Phi_V, \vartheta_V)$.
  - The UREs for loading variable L_V are:

$$\begin{aligned} I \in \mathrm{fst}(\Phi_V, \vartheta_V)_{\mathsf{b}} &\twoheadrightarrow \mathsf{L}_V(I-\vartheta_{\mathsf{L}_V}) = v_{in_V(I-\vartheta_V)} \\ I \in \mathrm{in}(\Phi_{\mathsf{L}_V}, \vartheta_{\mathsf{L}_V}) &\twoheadrightarrow \mathsf{L}_V(I) = v_{in_V(\ell_V(I)-\vartheta_V)} \\ I \in \Phi_{\mathsf{L}_V} &\twoheadrightarrow \mathsf{L}_V(I) = \mathsf{L}_V(I-\vartheta_{\mathsf{L}_V}) \end{aligned}$$

- The specification of recovery variable R_V is as follows:
  - Substitute $\mathsf{R}_V(I)$ for all target variables $V(I)$ of the source UREs, where $I \in \mathrm{lst}(\Phi_V, \vartheta_V)$.
  - The UREs for recovery variable R_V are:

$$\begin{aligned} I \in \Phi_{\mathsf{R}_V} &\twoheadrightarrow \mathsf{R}_V(I) = \mathsf{R}_V(I-\vartheta_{\mathsf{R}_V}) \\ I \in \mathrm{lst}(\Phi_V, \vartheta_V)_{\mathsf{b}} &\twoheadrightarrow v_{out_V(I)} = \mathsf{R}_V(I) \\ I \in \mathrm{lst}(\Phi_{\mathsf{R}_V}, \vartheta_{\mathsf{R}_V}) &\twoheadrightarrow v_{out_V(r_V(I))} = \mathsf{R}_V(I) \end{aligned}$$

□

Let us examine the impact of the choice of dependence vectors of loading and recovery variables on the quality of the systolic array. If the domain of a stationary variable V, and the domains of its loading and recovery variables are mutually disjoint (Fig. 7–1), then each cell is either loading, recovering or computing one element of V at each step. (If the domains of loading and recovery variables are disjoint, the two variables can be merged.) Of course, the loading, recovery and computation of V can still be performed concurrently among different cells of the

Scheme	ϑ_{L_C}	ϑ_{R_C}	Can L_C and R_C be Merged?	λ	Latency
1	$(1,0,1)$	$(1,0,1)$	yes	$(1,1,1)$	$5m-2$
2	$(2,0,-1)$	$(2,0,-1)$	yes	$(1,1,1)$	$3m-2$
3	$(1,0,-1)$	$(1,0,-1)$	no	$(2,1,1)$	$4m-3$

Table 7–1: Several schemes for the loading and recovery of stationary variable C of the UREs (2.14) for matrix product for the projection vector $u=(0,0,1)$.

array. To minimise the latency (i.e., the total number of execution steps), it is preferable to choose dependence vectors for loading and recovery such that the loading and recovery domains are contained in the index space. However, the choice of these dependence vectors may have to cater for other factors beside the minimisation of latency, e.g., the satisfaction of the requirements on the length and direction of the channels for loading and recovery variables.

To illustrate the specification of the loading and recovery of stationary data, we consider two-dimensional array implementations of the UREs (2.14) for matrix product. We choose the projection vector $u=(0,0,1)$. Variables A and B are moving and C is stationary.

Tab. 7–1 lists several schemes for the loading and recovery of stationary variable C. In Scheme 1, Φ_{L_C}, Φ_{R_C} and Φ are disjoint. The channels associated with L_C and R_C connect neighbouring cells; each has one delay buffer. In Scheme 2, $\Phi_{\mathsf{L}_C} \subset \Phi$, $\Phi_{\mathsf{R}_C} \subset \Phi$, and $\Phi_{\mathsf{L}_C} \cap \Phi_{\mathsf{R}_C} = \varnothing$. The channels associated with L_C and R_C connect non-neighbouring cells and have no delay buffers; each connects every other cells along x axis. In Scheme 3, $\Phi_{\mathsf{L}_C} \subset \Phi$, $\Phi_{\mathsf{R}_C} \subset \Phi$, and $\Phi_{\mathsf{L}_C} \cap \Phi_{\mathsf{R}_C} \neq \varnothing$. The channels associated with L_C and R_C connect neighbouring cells and have no delay buffers.

Let us present the UREs for Scheme 1 of Tab. 7–1. The corresponding two-dimensional array without the specification of the loading and recovery of C is depicted in Fig. 2–3. Its space-time mapping is:

$$\Pi = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{7.1}$$

Let us write P_C for the variable resulting from the merge of L_C and R_C. An application of Proc. 7.1 rewrites the UREs (2.14) to the following set of equations:

UREs with the specification of the loading and recovery of C:

$$
\begin{aligned}
A(i,j,k) &= \begin{cases} 0<i\leqslant m \wedge 0=j \wedge 0<k\leqslant m & \twoheadrightarrow a_{i,k} \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow A(i,j-1,k) \quad \blacktriangle \blacktriangledown \bullet \end{cases} \\
B(i,j,k) &= \begin{cases} 0=i \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow b_{k,j} \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow B(i-1,j,k) \quad \blacktriangle \blacktriangledown \bullet \end{cases} \\
\mathsf{P}_C(i,j,k) &= \begin{cases} i=0 \wedge 0<j\leqslant m \wedge -m<k\leqslant 0 & \twoheadrightarrow 0 \\ (0<i \wedge 0<j\leqslant m \wedge i-m<k\leqslant 0) \vee & \\ (i\leqslant m \wedge 0<j\leqslant m \wedge m<k<i+m) & \twoheadrightarrow \mathsf{P}_C(i-1,j,k-1) \ \blacksquare \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge k=m & \twoheadrightarrow C(i,j,k-1) \\ & \quad +A(i,j-1,k)B(i-1,j,k) \ \blacktriangledown \end{cases} \qquad (7.2) \\
C(i,j,k) &= \begin{cases} 0<i\leqslant m \wedge 0<j\leqslant m \wedge k=1 & \twoheadrightarrow \mathsf{P}_C(i-1,j,k-1) \\ & \quad +A(i,j-1,k)B(i-1,j,k) \ \blacktriangle \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 1<k<m & \twoheadrightarrow C(i,j,k-1) \\ & \quad +A(i,j-1,k)B(i-1,j,k) \ \bullet \end{cases} \\
c_{2m-k,j} &= \left\{ i=m \wedge 0<j\leqslant m \wedge m\leqslant k<2m \twoheadrightarrow \mathsf{P}_C(i,j,k) \right.
\end{aligned}
$$

Fig. 7–2 depicts the corresponding data dependence graph. The cube depicts the original index space. The input elements of C are now initialised at the right facet translated by $-\vartheta_{\mathsf{P}_C}$ of the lower triangular block and propagated towards the bottom facet of the cube. This implements the loading of C. Similarly, the output elements of C that are available at the top facet of the cube are now propagated towards the left facet of the top triangular block. This implements the recovery of C.

Once the loading and recovering scheme is devised, the derivation of the corresponding control signals is a matter of construction of the control UREs. There are four Ⓣ-classes for the UREs (7.2):

$$\Phi/\text{Ⓣ} \quad = \quad \{\Phi_{\blacksquare}, \Phi_{\bullet}, \Phi_{\blacktriangle}, \Phi_{\blacktriangledown}\}$$

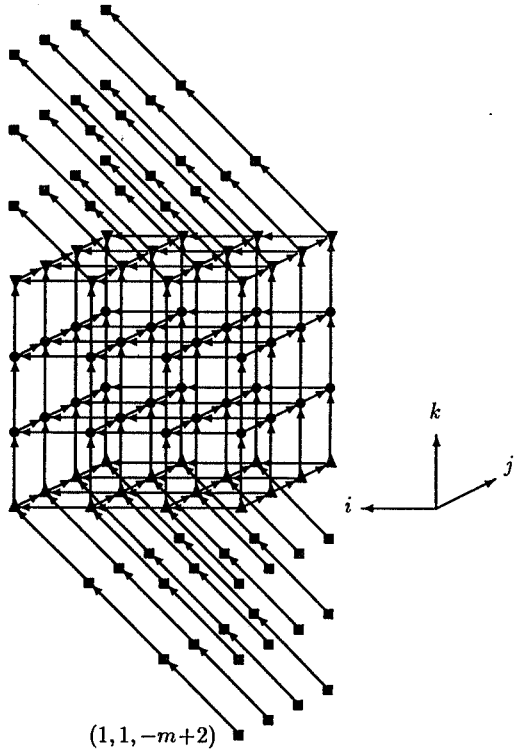


Figure 7–2: The data dependence graph of the UREs (7.2) for matrix product ($m=4$). The projection vector $u=(0,0,1)$.

The PCUREs are unnecessary because of the particular choice of the projection vector: $\Phi = \Psi$. We construct the CCUREs by identifying a separation set $\mathfrak{S} = \{S_1, S_2, S_3, S_4\}$:

$$\begin{aligned} S_1 &= \{(i,j,k) \mid k \leqslant 0\} \\ S_2 &= \{(i,j,k) \mid k \leqslant 1\} \\ S_3 &= \{(i,j,k) \mid k \leqslant m-1\} \\ S_4 &= \{(i,j,k) \mid k \leqslant m\} \end{aligned}$$

Since the corresponding hyperplanes of these three half-spaces are parallel, we

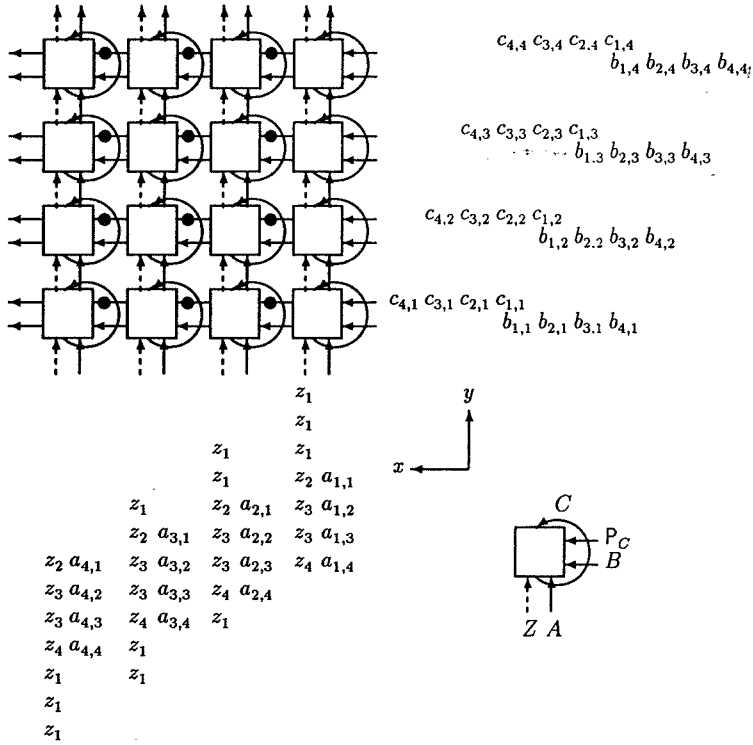


Figure 7–3: A two-dimensional array for matrix product that handles the loading and recovery of stationary variable C ($m=4$). The distribution of the data and control signals is the snapshot at the first step. Compare Fig. 2–3.

can combine the three control variables associated with these half-spaces into one, namely Z. $\mathrm{sig}(Z)=\{z_1, z_2, z_3, z_4\}$. The CCUREs follow from (3.8) and (3.23):

$$
\begin{aligned}
(0,0,1)\vartheta_Z &= 0 \\
Z(I) &= \begin{cases}
I \in \mathrm{in}(\Psi, \vartheta_Z)(\cap S_1 \cup \complement_{\mathbb{Z}^n} S_4) & \twoheadrightarrow \quad z_1 \\
I \in \mathrm{in}(\Psi, \vartheta_Z) \cap \complement_{\mathbb{Z}^n} S_1 \cap S_2 & \twoheadrightarrow \quad z_2 \\
I \in \mathrm{in}(\Psi, \vartheta_Z) \cap \complement_{\mathbb{Z}^n} S_2 \cap S_3 & \twoheadrightarrow \quad z_3 \\
I \in \mathrm{in}(\Psi, \vartheta_Z) \cap \complement_{\mathbb{Z}^n} S_3 \cap S_4 & \twoheadrightarrow \quad z_4 \\
I \in \Psi & \twoheadrightarrow \quad Z(I-\vartheta_Z)
\end{cases}
\end{aligned}
$$

The replacement of the domain predicates by predicates in control variable Z is straightforward and omitted. Let us choose $\vartheta_Z = (0,1,0)$. The previous space-time mapping (7.1) describes the systolic array shown in Fig. 7–3. Following the notation of Chap. 5, we write V^{in} and V^{out} for the input and output channel of a cell. The following program is executed by every cell of the array at every step:

$$
\begin{array}{lll}
 & & Z^{\text{out}} = Z^{\text{in}} \\
\textbf{if } Z^{\text{in}} = z_1 & \rightarrow & \mathsf{P}_C^{\text{out}} = \mathsf{P}_C^{\text{in}} \\
[\,]\ Z^{\text{in}} = z_2 & \rightarrow & A^{\text{out}} = A^{\text{in}},\ B^{\text{out}} = B^{\text{in}},\ C^{\text{out}} = \mathsf{P}_C^{\text{in}} + A^{\text{in}} B^{\text{in}} \\
[\,]\ Z^{\text{in}} = z_3 & \rightarrow & A^{\text{out}} = A^{\text{in}},\ B^{\text{out}} = B^{\text{in}},\ C^{\text{out}} = C^{\text{in}} + A^{\text{in}} B^{\text{in}} \\
[\,]\ Z^{\text{in}} = z_4 & \rightarrow & A^{\text{out}} = A^{\text{in}},\ B^{\text{out}} = B^{\text{in}},\ \mathsf{P}_C^{\text{out}} = C^{\text{in}} + A^{\text{in}} B^{\text{in}} \\
\textbf{fi} & &
\end{array}
$$

Sometimes, it is advantageous to reuse the communication channels for moving variables to load and recover the elements of stationary variables. Let there be s stationary variables. In addition to the source UREs (Proc. 7.1), there will be s systems of UREs for loading variables and s systems of UREs for recovery variables. Instead of choosing one step function for these $2s+1$ systems of UREs, we choose one step function for each system. We denote the scheduling vector for the source UREs by λ and the scheduling vector for the UREs of loading or recovery variable P by λ_{P}. To reuse the channels for a moving variable M, we must choose λ, λ_{P} and ϑ_{P} such that

- $\lambda\vartheta_M = \lambda_{\mathsf{P}}\vartheta_{\mathsf{P}}$ (this ensures that the channels associated with P and M have the same number of delay buffers).
- $\sigma\vartheta_M = \sigma\vartheta_{\mathsf{P}}$ (this ensures that the channels associated with P and M have the same length and direction), and

Finally, the $2s+1$ systems of UREs must be scheduled globally such that the dependence relations among them are preserved and the loading and/or recovery of different stationary variables using the channels for the same moving variable are performed in sequence. For example, if the P is a loading variable, the computation of the source UREs cannot start before the loading phase corresponding P

completes. Similarly, if the P is a recovery variable, the recovery phase corresponding P cannot start before the computation of the source UREs completes. These multiple systems of UREs are called *multi-step algorithms* in [20], where a method is provided to map them to systolic arrays by finding a space-time mapping for each individual system and a space-time mapping for the whole.

Let us use the systolic array depicted in Fig. 2–3 for illustration. We can reuse the channels for moving variable B to load and recover the elements of stationary variable C if we choose $\lambda = (1,1,1)$, $\lambda_{\mathsf{L}_C} = \lambda_{\mathsf{R}_C} = (0,0,1)$, and $\vartheta_{\mathsf{L}_C} = \vartheta_{\mathsf{R}_C} = (1,0,1)$. (A simple calculation verifies that $\lambda\vartheta_B = \lambda_{\mathsf{L}_C}\vartheta_{\mathsf{L}_C} = \lambda_{\mathsf{R}_C}\vartheta_{\mathsf{R}_C} = 1$ and $\sigma\vartheta_B = \sigma\vartheta_{\mathsf{L}_C} = \sigma\vartheta_{\mathsf{R}_C} = 1$.) We must choose the global scheduling such that the loading, computation and recovery of stationary variable C are performed in sequence. The latency of the array is the same as for Scheme 1 of Tab. 7–1: the loading and recovery of C each take m steps and the computation of C takes $3m-2$ steps. For this specific example, the method on reusing channels for moving variables for the loading and recovery of stationary data is appealing.

7.3 The Access of Stationary Data

Address variables provide the addresses for accessing the stationary elements stored in the local memory of a cell. They can be dispensed with in $(n-1)$-dimensional (parameterised) arrays, because at most one element per variable can be mapped to a cell. The size of the local processor memory associated with a stationary variable is at most 1. The access of the corresponding stationary element – if any – can be predesigned in hardware. However, address variables are essential for systolic arrays of reduced dimension, because more than one element per variable can be mapped to the local memory of a cell. We need to supply the right addresses at the right cells at the right steps so that the right stationary elements can be accessed.

We associate $n-r-1$ address variables, $\mathsf{A}_V^1, \cdots, \mathsf{A}_V^{n-r-1}$, with every stationary variable V. Element $V(I)$ is assigned with location $(\mathsf{A}_V^1(I), \cdots, \mathsf{A}_V^{n-r-1}(I))$ of the memory at cell $\mathsf{place}(I)$. Most systolic algorithms in the application world are

specified by three-dimensional UREs. Their one-dimensional array implementations require one address variable per stationary variable.

Let $I, I+\vartheta_V, \cdots, I+p\vartheta_V$ be a ϑ_V-path. If V is moving, $V(I)$ is computed at cell $\mathsf{place}(I)$ at time $\mathsf{step}(I)$, and then pipelined along channels with the length and direction $\sigma\vartheta_V$ across the cells $\mathsf{place}(I+\vartheta_V), \cdots, \mathsf{place}(I+p\vartheta_V)$ at a velocity of $\mathsf{flow}(V)$, and the final output value $V(I+p\vartheta_V)$ is available at cell $\mathsf{place}(I+p\vartheta_V)$ at time $\mathsf{step}(I+p\vartheta_V)$. If V is stationary, all points of the path are mapped to cell $\mathsf{place}(I)$. Since the values of V at all but the last point of the path are intermediate data, it suffices to allocate one memory location for all the points of the path. The final output value $V(I+p\vartheta_V)$ will be available at that memory location.

We shall construct address variables $\mathsf{A}_V^1, \cdots, \mathsf{A}_V^{n-r-1}$ such that all the points in one ϑ_V-path are given the same memory location and different stationary elements that are mapped to the same cell are given different memory locations. The construction of address variable A_V^i is as follows. We choose an n-vector $\pi_{\mathsf{A}_V^i}$ and, thus, obtain a sequence of parallel hyperplanes $\mathrm{addr}_{\ell_{\min}^{V^i}}$, $\mathrm{addr}_{\ell_{\min}^{V^i}+1}$, $\cdots$, $\mathrm{addr}_{\ell_{\max}^{V^i}}$ ($\ell_{\min}^{V^i} \leqslant \ell_{\max}^{V^i}$) whose normals are $\pi_{\mathsf{A}_V^i}$ and which intersect the domain of stationary variable V. We choose the dependence vector $\vartheta_{\mathsf{A}_V^i}$ of address variable A_V^i such that $\vartheta_{\mathsf{A}_V^i}\pi_{\mathsf{A}_V^i} = 0$ and construct A_V^i such that $\mathsf{A}_V^i(I) = \ell$ ($\ell_{\min}^{V^i} \leqslant \ell \leqslant \ell_{\max}^{V^i}$) for all points I in hyperplane addr_ℓ:

$$\mathsf{A}_V^i(I) = \begin{cases} I \in \mathrm{in}(\Phi_V, \vartheta_{\mathsf{A}_V^i}) \cap \mathrm{addr}_{\ell_{\min}^{V^i}} & \twoheadrightarrow \ \ell_{\min}^{V^i} \\ I \in \mathrm{in}(\Phi_V, \vartheta_{\mathsf{A}_V^i}) \cap \mathrm{addr}_{\ell_{\min}^{V^i}+1} & \twoheadrightarrow \ \ell_{\min}^{V^i}+1 \\ & \vdots \\ I \in \mathrm{in}(\Phi_V, \vartheta_{\mathsf{A}_V^i}) \cap \mathrm{addr}_{\ell_{\max}^{V^i}} & \twoheadrightarrow \ \ell_{\max}^{V^i} \\ I \in \Phi_V & \twoheadrightarrow \ \mathsf{A}_V^i(I-\vartheta_{\mathsf{A}_V^i}) \end{cases} \tag{7.3}$$

This implies that the domain of address variable A_V^i is the same as that of stationary variable V, i.e., $\Phi_{A_V^i} = \Phi_V$.

Finally, we must choose $\pi_{\mathsf{A}_V^i}$ and $\vartheta_{\mathsf{A}_V^i}$ such that

- $\sigma\vartheta_{\mathsf{A}_V^i} \neq 0$ (this ensures that A_V^i is a moving variable).
- $\pi_{\mathsf{A}_V^i}\vartheta_V = 0$ (this ensures that $\mathsf{A}_V^i(I)$ is the same address for all points of a ϑ_V-path).

- $\pi_{\mathsf{A}_V^1}, \cdots, \pi_{\mathsf{A}_V^{n-r-1}}, \sigma_1, \cdots, \sigma_r$ are linearly independent (this ensures that different elements of stationary variable V that are mapped to the same cell are given different memory locations, because the intersection of $n-1$ mutually non-parallel hyperplanes is one-dimensional).

The specification of stationary variable V must be updated to reflect the change caused by the introduction of address variables $\mathsf{A}_V^1, \cdots, \mathsf{A}_V^{n-r-1}$ in the source UREs. This update and the specification of $\mathsf{A}_V^1, \cdots, \mathsf{A}_V^{n-r-1}$ are summarised in a procedure.

Procedure 7.2 (Construction of address variables $\mathsf{A}_V^1, \cdots, \mathsf{A}_V^{n-r-1}$ for stationary variable V of the source UREs given a place function)

- Replace all occurrences of $V(I)$ in the source UREs by $V(\mathsf{A}_V^1(I), \cdots, \mathsf{A}_V^{n-r-1}(I))$.
- The specification of address variables $\mathsf{A}_V^1, \cdots, \mathsf{A}_V^{n-r-1}$ is given by (7.3). □

To retain the latency, it is always best to choose the dependence vector of a moving variable as the dependence vector of an address variable.

To illustrate the specification of the access of stationary data, we consider one-dimensional array implementations of the UREs (2.14) for matrix product. We have presented a one-dimensional array in Sect. 5.6.1. The corresponding space-time mapping (5.29) makes variable C stationary, but only one element of C is mapped to each cell. Hence, addressing control is not needed.

Let us choose allocation vector $\sigma = (1, -1, 0)$. Variables A and B are moving and C is stationary. There are m elements of C mapped to cell 0, and $m-k$ elements of C mapped to cell $-k$ or k $(0 < k < m)$. Only one address variable is needed (since $r = 1$); it is named A_C. We arbitrarily choose $\vartheta_{\mathsf{A}_C} = \vartheta_A = (0, 1, 0)$ and $\pi_{\mathsf{A}_C} = (1, 0, 0)$. A simple calculations shows that $\ell^C_{\min} = 1$ and $\ell^C_{\max} = m$. For the loading and recovery of C, we continue to choose Scheme 1 of Tab. 7-1. Again, we merge the loading variable L_C and recovery variable R_C to one variable named P_C. An application of Proc. 7.1 and Proc. 7.2 rewrites the UREs (2.14) to the following set of equations:

UREs with the specification of the loading, recovery and access of C:

$$\mathsf{A}_C(i,j,k)=\begin{cases} 0<i\leqslant m \wedge 0=j \wedge 0<k\leqslant m & \twoheadrightarrow i \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow \mathsf{A}_C(i,j-1,k) \blacktriangle \blacktriangledown \bullet \end{cases}$$

$$A(i,j,k)=\begin{cases} 0<i\leqslant m \wedge 0=j \wedge 0<k\leqslant m & \twoheadrightarrow a_{i,k} \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow A(i,j-1,k) \blacktriangle \blacktriangledown \bullet \end{cases}$$

$$B(i,j,k)=\begin{cases} 0=i \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow b_{k,j} \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 0<k\leqslant m & \twoheadrightarrow B(i-1,j,k) \blacktriangle \blacktriangledown \bullet \end{cases}$$

$$\mathsf{P}_C(i,j,k)=\begin{cases} i=j-m \wedge k\leqslant i<k+m \wedge 0<j\leqslant m \wedge k\leqslant 0 \twoheadrightarrow 0 \\ (i>j-m \wedge k\leqslant i<k+m \wedge 0<j\leqslant m \wedge k\leqslant 0) \vee \\ (i<j+m \wedge k-m<i\leqslant k \wedge 0<j\leqslant m \wedge k>m) \twoheadrightarrow \mathsf{P}_C(i-1,j,k-1) \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge k=m \twoheadrightarrow C(\mathsf{A}_C(i,j,k-1)) \\ \qquad\qquad +A(i,j-1,k)B(i-1,j,k) \blacktriangledown \end{cases} \quad (7.4) \blacksquare$$

$$C(\mathsf{A}_C(i,j,k))=\begin{cases} 0<i\leqslant m \wedge 0<j\leqslant m \wedge k=1 & \twoheadrightarrow \mathsf{P}_C(\mathsf{A}_C(i-1,j,k-1)) \\ & \qquad +A(i,j-1,k)B(i-1,j,k) \blacktriangle \\ 0<i\leqslant m \wedge 0<j\leqslant m \wedge 1<k<m & \twoheadrightarrow C(\mathsf{A}_C(i,j,k-1)) \\ & \qquad +A(i,j-1,k)B(i-1,j,k) \bullet \end{cases}$$

$$c_{2m-k,j}=\left\{ i=j+m-1 \wedge k-m<i\leqslant k \wedge 0<j\leqslant m \wedge k\geqslant m \twoheadrightarrow \mathsf{P}_C(i,j,k) \right.$$

The corresponding data dependence graph is depicted in Fig. 7–4. The CCUREs for the UREs (7.4) are as specified in (7.3). (The CCUREs for the UREs (7.2) and the CCUREs for the UREs (7.4) are different because the extended index spaces are different.) Because the chosen place function maps some input and output data of A, B and C to internal cells, the PCUREs are required. Their specification follows from Proc. 5.1 and is not presented here. It amounts to defining a characteristic function $\chi(\overline{I})$ of $\overline{\Phi}$ in the space-time diagram.

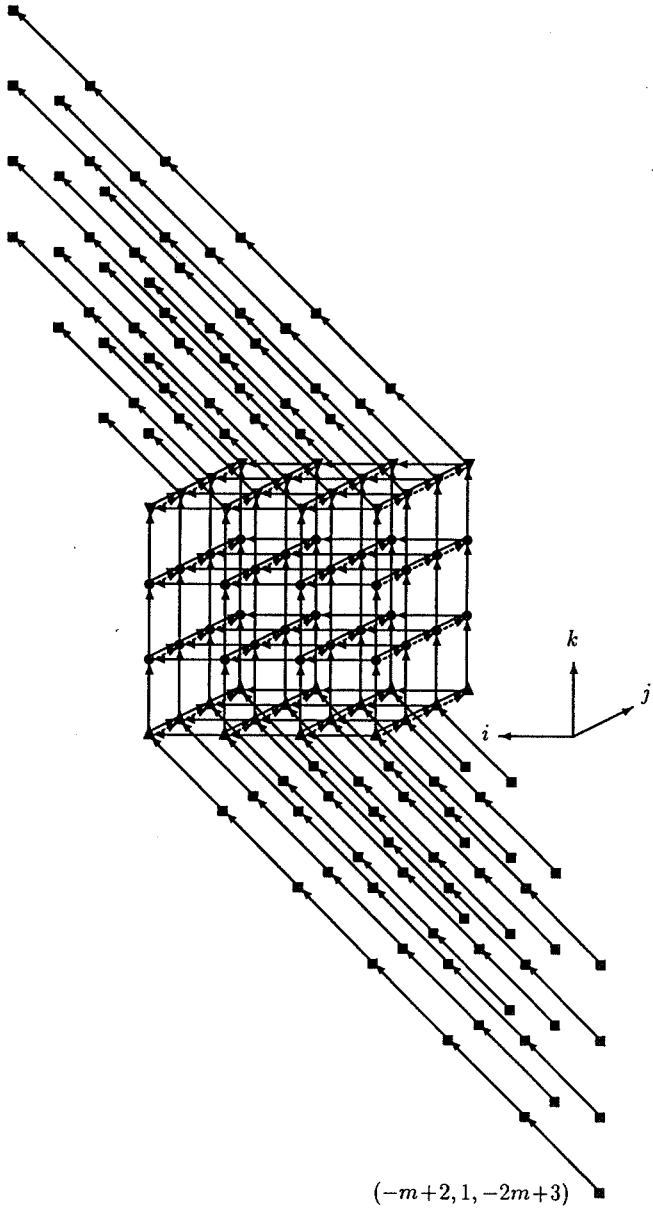


Figure 7–4: The data dependence graph of the UREs (7.4) for matrix product ($m=4$). The allocation vector $\sigma=(1,-1,0)$. The dashed lines depict ϑ_{A_C}.

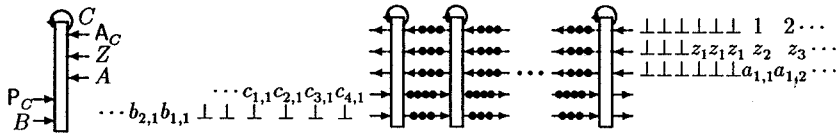


Figure 7–5: The array described by the UREs (7.4), the CCUREs (7.3) with $\vartheta_Z = (0,1,0)$ and the space-time mapping (7.5) ($m = 4$). Propagation control is not shown. $\perp$ denotes the undefined value.

The following program (with the PCUREs excluded) follows from (7.3) and (7.4); it is executed by every cell at every step:

$$
\begin{array}{ll}
 & Z^{\text{out}} = Z^{\text{in}} \\
 & \mathsf{A}_C^{\text{out}} = \mathsf{A}_C^{\text{in}} \\
\textbf{if } \chi_{\overline{\Phi}}^{\text{in}} = \chi_{\text{c}} \wedge Z^{\text{in}} = z_1 \rightarrow & \mathsf{P}_C^{\text{out}} = \mathsf{P}_C^{\text{in}} \\
[] \ \chi_{\overline{\Phi}}^{\text{in}} = \chi_{\text{c}} \wedge Z^{\text{in}} = z_2 \rightarrow & A^{\text{out}} = A^{\text{in}},\ B^{\text{out}} = B^{\text{in}},\ C^{\text{out}}(\mathsf{A}_C^{\text{in}}) = \mathsf{P}_C^{\text{in}} + A^{\text{in}} B^{\text{in}} \\
[] \ \chi_{\overline{\Phi}}^{\text{in}} = \chi_{\text{c}} \wedge Z^{\text{in}} = z_3 \rightarrow & A^{\text{out}} = A^{\text{in}},\ B^{\text{out}} = B^{\text{in}},\ C^{\text{out}}(\mathsf{A}_C^{\text{in}}) = C^{\text{in}}(\mathsf{A}_C^{\text{in}}) + A^{\text{in}} B^{\text{in}} \\
[] \ \chi_{\overline{\Phi}}^{\text{in}} = \chi_{\text{c}} \wedge Z^{\text{in}} = z_4 \rightarrow & A^{\text{out}} = A^{\text{in}},\ B^{\text{out}} = B^{\text{in}},\ \mathsf{P}_C^{\text{out}} = C^{\text{in}}(\mathsf{A}_C^{\text{in}}) + A^{\text{in}} B^{\text{in}} \\
[] \ \chi_{\overline{\Phi}}^{\text{in}} = \chi_{\text{p}} \rightarrow & Z^{\text{out}} = Z^{\text{in}},\ \mathsf{A}_C^{\text{out}} = \mathsf{A}_C^{\text{in}},\ A^{\text{out}} = A^{\text{in}},\ B^{\text{out}} = B^{\text{in}},\ \mathsf{P}_C^{\text{out}} = \mathsf{P}_C^{\text{in}} \\
\textbf{fi} &
\end{array}
$$

Choosing the scheduling vector $\lambda = (m, m, 1)$ gives rise to the the following space-time mapping:

$$
\pi = \begin{bmatrix} m & m & 1 \\ 1 & -1 & 0 \end{bmatrix} \tag{7.5}
$$

The corresponding one-dimensional array is depicted in Fig. 7–5.

To conclude this this section, we consider the one-dimensional array for dynamic programming described in Sect. 5.6.2. Its space-time mapping is:

$$
\pi = \begin{bmatrix} -4 & 2(m+2) & -2 \\ -1 & 1 & 0 \end{bmatrix} \tag{7.6}
$$

Variable C is stationary. The specification of the loading and recovery of the stationary elements of C follows from Proc. 7.1.

A_C is the address variable associated with stationary variable C. We choose $\vartheta_{\mathsf{A}_C} = (-1,0,1)$. $(\pi_{\mathsf{A}_C} = (0,1,0))$. A simple calculation shows that $\ell^C_{\min} = 2$ and $\ell^C_{\max} = m+1$. An application of Proc. 7.2 rewrites the source UREs for dynamic programming, as described in Sect. 3.7.1, to the following set of equations:

UREs with the specification of access of C:

$$\mathsf{A}_C(i,j,k) = \begin{cases} 1<i\leqslant j \wedge 2\leqslant j\leqslant m+1 \wedge k=0 & \twoheadrightarrow j \\ 0<i<j \leqslant m+1 \wedge 0<k\leqslant (j-i)/2+1 & \twoheadrightarrow \mathsf{A}_C(i+1,j,k-1) \\ & \blacktriangleleft \blacktriangleright \blacklozenge \bullet \blacksquare \bigstar \end{cases}$$

$$A(i,j,k) = \begin{cases} j-i=2k & \twoheadrightarrow D(i,j,k) \quad \blacklozenge \blacktriangleleft \\ j-i\neq 2k & \twoheadrightarrow A(i,j-1,k) \quad \blacktriangleright \bullet \blacksquare \bigstar \end{cases}$$

$$B(i,j,k) = \begin{cases} k=1 & \twoheadrightarrow C(\mathsf{A}_C(i+1,j,k)) \quad \blacktriangleleft \blacktriangleright \blacksquare \\ k\neq 1 & \twoheadrightarrow B(i+1,j,k-1) \quad \blacklozenge \bullet \bigstar \end{cases}$$

$$D(i,j,k) = \begin{cases} k=1 & \twoheadrightarrow C(\mathsf{A}_C(i,j-1,k)) \quad \blacktriangleleft \blacktriangleright \blacksquare \\ k\neq 1 & \twoheadrightarrow D(i,j-1,k-1) \quad \blacklozenge \bullet \bigstar \end{cases}$$

$$E(i,j,k) = \begin{cases} j-i=2k & \twoheadrightarrow B(i,j,k) \quad \blacklozenge \blacktriangleleft \\ j-i\neq 2k & \twoheadrightarrow E(i+1,j,k) \quad \blacktriangleright \bullet \blacksquare \bigstar \end{cases}$$

$$c_{1,m+1} = C(\mathsf{A}_C(1,m+1,1))$$

$$C(\mathsf{A}_C(i,j,k)) = \begin{cases} k=1 \wedge j-i=2k-1 & \twoheadrightarrow w_{i,j} \quad \blacksquare \\ k=1 \wedge j-i\geqslant 2k & \twoheadrightarrow \min \begin{pmatrix} A(i,j,k)+B(i,j,k) \\ C(\mathsf{A}_C(i,j,k+1)) \\ D(i,j,k)+E(i,j,k) \end{pmatrix} + w_{i,j} \quad \blacktriangleleft \blacktriangleright \\ k\neq 1 \wedge j-i\geqslant 2k & \twoheadrightarrow \min \begin{pmatrix} A(i,j,k)+B(i,j,k) \\ C(\mathsf{A}_C(i,j,k+1)) \\ D(i,j,k)+E(i,j,k) \end{pmatrix} \quad \blacklozenge \bullet \\ k\neq 1 \wedge j-i<2k \wedge j-i\geqslant 2k-2 & \twoheadrightarrow \infty \quad \bigstar \end{cases}$$

The corresponding data dependence graph is depicted in Fig. 7–6 (compare Fig. 3–4). The CCUREs are specified in Sect. 3.7.1.2. The PCUREs are specified in Sect. 5.6.2. Under the previous space-time mapping (7.6), address variable A_C

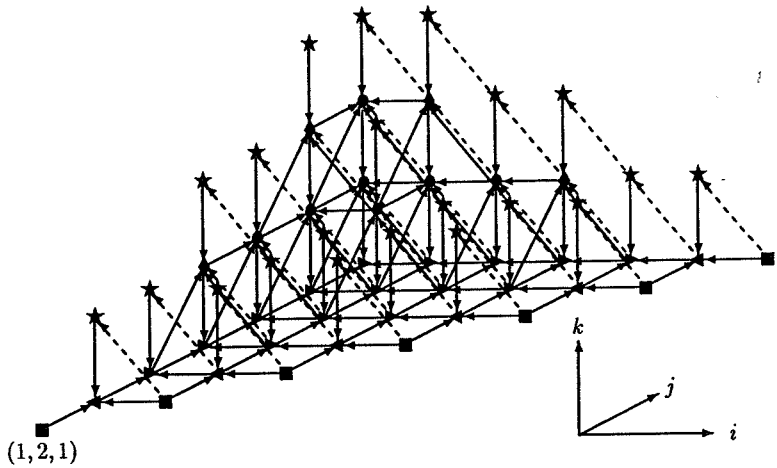


Figure 7–6: The data dependence graph of the UREs with address variable A_C for dynamic programming ($m=7$). The dashed vectors depict ϑ_{A_C}.

encounters a delay of one unit along a neighbouring channel (in the τ-model). This control variable was derived informally in [68].

7.4 Conclusion

In most systolic design methods reported in the literature, the issues of loading, recovering and accessing stationary data are either dealt with in an ad hoc and problem-specific manner or not addressed at all.

We have presented a systematic method for the construction of the specification of loading, recovery and access of stationary variables. The idea is to add loading, recovery and address variables to the source UREs. Their specification is only dependent on the place function and leaves plenty of freedom of choice for dependence vectors of loading, recovery and access of stationary variables and the step function. Once this specification is constructed, the specification of the control signals for systolic arrays is just a matter of constructing the control UREs for the modified source UREs.

8

Conclusion

We have shown how control signals for systolic arrays can be systematically derived by program transformation and construction. The method we propose applies for systolic algorithms defined over a convex set of integer coordinates. The basic idea is to transform the domain predicates in the initial program to UREs that we call control UREs. Then, the search of systolic arrays with a description of both data and control signals reduces to a search of space-time mappings.

In Chap. 2, we have reviewed the basic techniques for the synthesis of data flow for $(n-1)$-dimensional arrays. In Chap. 3, we have presented our method for the construction of control UREs from n-dimensional UREs. We have provided necessary and sufficient conditions for the correctness of the control UREs and a mechanisable procedure that constructs the control UREs from the source UREs. We have provided some heuristics for guiding the user of interactive systolic design systems in the optimisation of the domain predicates (optimised domain predicates lead to a more efficient control flow). We have also shown how redundant control hardware can be completely removed once the space-time mapping has been selected. In Chap. 4, we have reviewed the synthesis of data flow for one-dimensional arrays. We have improved and extended previous results. We have provided unified mapping conditions for the validity of space-time mappings for two one-dimensional array models. We have also presented a variety of equivalent mapping conditions that highlight the properties of one-dimensional arrays from different perspectives. We have concluded that the control UREs constructed for

$(n-1)$-dimensional arrays can be inefficient for one-dimensional arrays. In Chap. 5, we have provided a more appropriate construction. In Chap. 6, we have shown how to eliminate the control signals for a class of systolic arrays by exploiting the algebraic properties of some operators in the source UREs. Finally, in Chap. 7, we have described a systematic method for the construction of UREs that specify loading, recovery and access of stationary elements in the local processor memories of a systolic array.

There are several areas that require further research. In Chap. 3, we have presented a procedure that delivers the control UREs from the source UREs. The control UREs returned by this procedure are completely determined by the specification of the domain predicates of the source UREs. We have shown by example that different specifications of the domain predicates of the source UREs lead to different specifications of the control UREs. It is worthwhile searching for a method for finding the specifications of the domain predicates that lead to (in some sense) optimal control signals.

Once the control UREs are constructed, there are many possible control dependence vectors. The choice of control dependence vector affects the quality of the systolic array. This thesis does not provide a systematic way of making a good choice.

Finally, the control UREs are targeted directly at parameterised systolic arrays. In reality, we need to map both the source and the control UREs to fixed-size arrays (i.e., partitioned arrays). This can be done in three successive steps. First, we obtain a parameterised systolic array by the space-time mapping technique. Second, we partition the array [12,18,59,91]. There is no need for distinguishing which recurrence equation specifies data or control signals. The partitioning demands additional control, which ensures that the right values (data and control signals) as specified by the source and the control UREs arrive at the right cells in the partitioned array at the right steps. This is the task of the third step. The regularity exhibited in the partitioning can be exploited in the derivation of this additional control. This regularity has been recently captured by a more general form of **step** and **place** that contains **mod** or **div** operators [18,82,91].

A

The Normalisation of Domain Predicates

Proc. 3.1 for the construction of separation set $\mathfrak{S}_{\mathrm{src}}$ expects that domain predicates are in normal form. This appendix describes a procedure for the normalisation of certain more general domain predicates. Tab. 9–1 gives a syntactic definition of these domain predicates in terms of a set-theoretic abstract syntax [74]. The domain predicates defined in Tab. 9–1 are distinguished from the domain predicates in normal form in that they may also contain the six operators in the following set:

$$O = \{|\ |, \max, \min, \lceil\ \rceil, \lfloor\ \rfloor, \mathbf{div}\} \tag{9.1}$$

These six operators are the most frequently used in practice. For example, the URE specification for dynamic programming of [15] contains operator $\lfloor\ \rfloor$, the URE specification for transitive closure of [79] contains operators max and min, and the URE specification for LU-decomposition of [49] contains operator min.

We specify a language's syntax by listing its syntax domains and its BNF rules. In Tab. 9–1, the phrase D∈Domain-Predicate indicates that D is the non-terminal that represents an arbitrary member of the syntax domain Domain-Predicate. The structure of domain predicates is given by the BNF rule $\mathrm{D} ::= \mathrm{C}; (\mathrm{D}); \mathrm{D}_1 \wedge \mathrm{D}_2; \mathrm{D}_1 \vee \mathrm{D}_2$ which says that any domain predicate must be either a conditional, or a conditional enclosed in a pair of parentheses, or a conjunction of two conditionals or a disjunction of two conditionals.

An expression E is called a *basic expression* if E is of the form $\pi I + \delta$, i.e., an affine expression of index vector I in $\mathbf{Z}^n$. A conditional E®0 is called a *basic conditional*

$$
\begin{array}{l}
\mathrm{D} \in \text{Domain-Predicate} \\
\mathrm{C} \in \text{Conditional} \\
\mathrm{E} \in \text{Expression} \\
\mathrm{A} \in \{\pi I + \delta \mid \pi \in \mathbf{Z}^n \wedge I \in \mathbf{Z}^n \wedge \delta \in \mathbf{Z}\} \\
\mathrm{n} \in \mathbf{Z}^+ \\
\mathrm{D} ::= \mathrm{C}; (\mathrm{D}); \mathrm{D}_1 \wedge \mathrm{D}_2; \mathrm{D}_1 \vee \mathrm{D}_2 \\
\mathrm{C} ::= \mathrm{E} \circledR 0 \\
\mathrm{E} ::= \mathrm{L}; |\mathrm{E}|; \\
\qquad \mathrm{E} \circledcirc \mathrm{L}; \max(\mathrm{E}, \mathrm{L}); \min(\mathrm{E}, \mathrm{L}); \lceil (\mathrm{E} \circledcirc \mathrm{L})/\mathrm{n} \rceil; \lfloor (\mathrm{E} \circledcirc \mathrm{L})/\mathrm{n} \rfloor; (\mathrm{E} \circledcirc \mathrm{L})\ \mathbf{div}\ \mathrm{n} \\
\qquad \mathrm{L} \circledcirc \mathrm{E}; \max(\mathrm{L}, \mathrm{E}); \min(\mathrm{L}, \mathrm{E}); \lceil (\mathrm{L} \circledcirc \mathrm{E})/\mathrm{n} \rceil; \lfloor (\mathrm{L} \circledcirc \mathrm{E})/\mathrm{n} \rfloor; (\mathrm{E} \circledcirc \mathrm{L})\ \mathbf{div}\ \mathrm{n} \\
\mathrm{L} ::= \mathrm{A}; \mathrm{nL}; |\mathrm{L}|; \max(\mathrm{L}_1, \mathrm{L}_2); \min(\mathrm{L}_1, \mathrm{L}_2) \\
\circledR ::= <; \leqslant; >; \geqslant; =; \neq \\
\circledcirc ::= +; -
\end{array}
$$

Table 9–1: The abstract syntax of the domain predicates.

if E is a basic expression. Conditionals of the from $\mathrm{E}=0$ are called *equalities*, those of the from $\mathrm{E} \neq 0$ *inequalities*. A domain predicate is a *basic domain predicate* if all its constituent conditionals are basic conditionals. A domain predicate is said to be in normal form if it is a basic domain predicate. For convenience, we refer to anything that is not a domain predicate as a *non-domain predicate*.

Domain predicates exhibit a nested parentheses property. To see this, let us make a syntactic substitution in rule E: we replace $(\mathrm{E} \circledcirc \mathrm{L})\ \mathbf{div}\ n$ by $\Gamma\ \mathrm{E} \circledcirc \mathrm{L}, n\ \urcorner$ and $(\mathrm{L} \circledcirc \mathrm{E})\ \mathbf{div}\ n$ by $\Gamma\ \mathrm{L} \circledcirc \mathrm{E}, n\ \urcorner$. If we delete all symbols except those in $\{\lceil, \lfloor, \Gamma, \rceil, \rfloor, \urcorner\}$ from a domain predicate and let n be the length of the resulting string, then the i-th symbol $(0 < i \leqslant n/2)$ from the left forms an operator (either $\lceil\ \rceil$ or $\lfloor\ \rfloor$ or $\Gamma\ \urcorner$) with the i-th symbol from the right.

Example 9.1 These examples illustrate the structure of domain predicates.

Domain predicates:

1. $\lceil i/2 \rceil - j \leqslant 0$
2. $|i+2| - k \neq 0$
3. $(i+j)\ \mathbf{div}\ 2 - k > 0$

4. $\max(i,j)+k=0$
5. $\min(\lfloor i/2\rfloor, k)-j\geqslant 0$

Non-domain predicates:

1. $2\lfloor i/2\rfloor-j=0$
2. $2\lceil(i \operatorname{div} 2+j/3)\rceil+1\geqslant 0$
3. $i \operatorname{div} 2+\lceil(j+k)/2\rceil=0$
4. $\lceil i/2\rceil-\lceil(j+k)/3\rceil<0$
5. $\max(\lceil i/2\rceil, \lceil i/2\rceil+\lfloor j/2\rfloor)\neq 0$ □

The normalisation of domain predicates relies on the notion of rewriting. In general, it is achieved according to a number of pre-defined rewriting rules. A rewriting rule rewrites a domain predicate of a certain form to a domain predicate of a different form. The following lemma states algebraic laws of the six operators in the set O of (9.1). It is the basis of the formulation of the rewriting rules in Lemma 9.2.

Lemma 9.1 *Let* $u,v,w\in\mathsf{R}$, $x,y\in\mathbf{Z}$, $c\in\mathbf{Z}^+$.

$$
\begin{array}{rrcl}
\text{max:} & w\circledR\max(u,v) & \Longleftrightarrow & (w\circledR u\wedge u\geqslant v)\vee(v\geqslant u\wedge w\circledR v)\\
\text{min:} & w\circledR\min(u,v) & \Longleftrightarrow & (w\circledR u\wedge u\leqslant v)\vee(v\leqslant u\wedge w\circledR v)\\
|\ |\text{:} & |u|\circledR v & \Longleftrightarrow & (u\circledR v\wedge u\geqslant 0)\vee(-u\circledR v\wedge u\leqslant 0)\\
\lceil\ \rceil\text{:} & \lceil y/c\rceil\circledR x & \Longleftrightarrow & (y\circledR cx\vee y\circledR cx-1\vee\cdots\vee y\circledR cx-c+1)\\
\lfloor\ \rfloor\text{:} & \lfloor y/c\rfloor\circledR x & \Longleftrightarrow & (y\circledR cx\vee y\circledR cx+1\vee\cdots\vee y\circledR cx+c-1)\\
\textbf{div:} & (y \operatorname{div} c)\circledR x & \Longleftrightarrow & (x\geqslant 0\wedge(y\circledR cx\vee y\circledR cx+1\vee\cdots\vee y\circledR cx+c-1)\vee\\
 & & & (x\leqslant 0\wedge(y\circledR cx\vee y\circledR cx-1\vee\cdots\vee y\circledR cx-c+1))
\end{array}
$$

Proof. Algebraic manipulation. □

The following lemma states the rewriting rules that are used in the normalisation of domain predicates. The notation $\overline{\circledR}$ in rule $|\ |$-elim1 has the following meaning: the overbar denotes the following mapping (whose argument is $\circledR$) from

$\{<, \leqslant, >, \geqslant, =, \neq\}$ onto itself:

$$\overline{®} = \begin{cases} \textbf{if} \quad ® = < \quad \rightarrow \quad > \\ [] \quad ® = \leqslant \quad \rightarrow \quad \geqslant \\ [] \quad ® = > \quad \rightarrow \quad < \\ [] \quad ® = \geqslant \quad \rightarrow \quad \leqslant \\ [] \quad ® = = \quad \rightarrow \quad = \\ [] \quad ® = \neq \quad \rightarrow \quad \neq \\ \textbf{fi} \end{cases}$$

Lemma 9.2

$$\begin{aligned}
\text{0-law:} \quad & \mathrm{E}®0 \iff \mathrm{E}+0®0 \\
\text{max-dist:} \quad & n\max(\mathrm{L}_1,\mathrm{L}_2)®0 \iff \max(n\mathrm{L}_1, n\mathrm{L}_2)®0 \\
\text{min-dist:} \quad & n\min(\mathrm{L}_1,\mathrm{L}_2) \iff \min(n\mathrm{L}_1, n\mathrm{L}_2)®0 \\
\text{| |-law:} \quad & n|\mathrm{L}|®0 \iff |n\mathrm{L}|®0 \\
\text{max-elim1:} \quad & \max(\mathrm{E},\mathrm{L}_1)\pm\mathrm{L}_2®0 \iff (\mathrm{E}\pm\mathrm{L}_2®0 \wedge \mathrm{E}-\mathrm{L}_1 \geqslant 0) \vee (\mathrm{L}_1-\mathrm{E} \geqslant 0 \wedge \mathrm{L}_1 \pm \mathrm{L}_2®0) \\
\text{max-elim2:} \quad & \mathrm{L}_2\pm\max(\mathrm{E},\mathrm{L}_1)®0 \iff (\mathrm{L}_2\pm\mathrm{E}®0 \wedge \mathrm{E}-\mathrm{L}_1 \geqslant 0) \vee (\mathrm{L}_1-\mathrm{E} \geqslant 0 \wedge \mathrm{L}_2 \pm \mathrm{L}_1®0) \\
\text{min-elim1:} \quad & \min(\mathrm{E},\mathrm{L}_1)\pm\mathrm{L}_2®0 \iff (\mathrm{E}\pm\mathrm{L}_2®0 \wedge \mathrm{E}-\mathrm{L}_1 \leqslant 0) \vee (\mathrm{L}_1-\mathrm{E} \leqslant 0 \wedge \mathrm{L}_1 \pm \mathrm{L}_2®0) \\
\text{min-elim2:} \quad & \mathrm{L}_2\pm\min(\mathrm{E},\mathrm{L}_1)®0 \iff (\mathrm{L}_2\pm\mathrm{E}®0 \wedge \mathrm{E}-\mathrm{L}_1 \leqslant 0) \vee (\mathrm{L}_1-\mathrm{E} \leqslant 0 \wedge \mathrm{L}_2 \pm \mathrm{L}_1®0) \\
\text{| |-elim1:} \quad & |\mathrm{E}|\pm\mathrm{L}®0 \iff (\mathrm{E}\pm\mathrm{L}®0 \wedge \mathrm{E} \geqslant 0) \vee (\mathrm{E}\mp\mathrm{L}\overline{®}0 \wedge \mathrm{E} \leqslant 0) \\
\text{| |-elim2:} \quad & \mathrm{L}\pm|\mathrm{E}|®0 \iff (\mathrm{L}\pm\mathrm{E}®0 \wedge \mathrm{E} \geqslant 0) \vee (\mathrm{L}\mp\mathrm{E}®0 \wedge \mathrm{E} \leqslant 0) \\
\text{⌈ ⌉-elim1:} \quad & \lceil \mathrm{E}/n \rceil\pm\mathrm{L}®0 \iff \mathrm{E}\pm n\mathrm{L}®0 \vee \mathrm{E}\pm n\mathrm{L}+1®0 \vee \cdots \vee \mathrm{E}\pm n\mathrm{L}+n-1®0 \\
\text{⌈ ⌉-elim2:} \quad & \mathrm{L}\pm\lceil \mathrm{E}/n \rceil®0 \iff n\mathrm{L} \pm \mathrm{E}®0 \vee n\mathrm{L}+1\pm\mathrm{E}®0 \vee \cdots \vee n\mathrm{L}+n-1\pm\mathrm{E}®0 \\
\text{⌊ ⌋-elim1:} \quad & \lfloor \mathrm{E}/n \rfloor\pm\mathrm{L}®0 \iff \mathrm{E}\pm n\mathrm{L}®0 \vee \mathrm{E}\pm n\mathrm{L}-1®0 \vee \cdots \vee \mathrm{E}\pm n\mathrm{L}-n+1®0 \\
\text{⌊ ⌋-elim2:} \quad & \mathrm{L}\pm\lfloor \mathrm{E}/n \rfloor®0 \iff n\mathrm{L}\pm\mathrm{E}®0 \vee n\mathrm{L}-1\pm\mathrm{E}®0 \vee \cdots \vee n\mathrm{L}-n+1\pm\mathrm{E}®0 \\
\textbf{div}\text{-elim1:} \quad & \mathrm{E}\ \textbf{div}\ n\pm\mathrm{L}®0 \iff \\
& \qquad (\pm\mathrm{L} \geqslant 0 \wedge (\mathrm{E}\pm n\mathrm{L}®0 \vee \mathrm{E}\pm n\mathrm{L}+1®0 \vee \cdots \vee \mathrm{E}\pm n\mathrm{L}+n-1®0)) \vee \\
& \qquad (\pm\mathrm{L} \leqslant 0 \wedge (\mathrm{E}\pm n\mathrm{L}®0 \vee \mathrm{E}\pm n\mathrm{L}-1®0 \vee \cdots \vee \mathrm{E}\pm n\mathrm{L}-n+1®0)) \\
\textbf{div}\text{-elim2:} \quad & \mathrm{L}\pm\mathrm{E}\ \textbf{div}\ n®0 \iff \\
& \qquad (\mathrm{L} \geqslant 0 \wedge (n\mathrm{L}\pm\mathrm{E}®0 \vee n\mathrm{L}+1\pm\mathrm{E}®0 \vee \cdots \vee n\mathrm{L}+n-1\pm\mathrm{E}®0)) \vee \\
& \qquad (\mathrm{L} \leqslant 0 \wedge (n\mathrm{L}\pm\mathrm{E}®0 \vee n\mathrm{L}-1\pm\mathrm{E}®0 \vee \cdots \vee n\mathrm{L}-n+1\pm\mathrm{E}®0))
\end{aligned}$$

Proof. The first rule follows from the fact that 0 is the zero element in $\mathbf{Z}$. The next three rewriting rules follow from the conventional semantics of max, min, and $|\ |$. The remaining rewriting rules follow from Lemma 9.1. □

To illustrate the rewriting rules in Lemma 9.2, we normalise the domain predicates of Ex. 9.1.

1. $\lceil i/2 \rceil - j \leqslant 0$

$\Longleftrightarrow$ {$\lceil\ \rceil$-elim1}

$i-2j \leqslant 0 \vee i-2j+1 \leqslant 0$

2. $|i+2| - k \neq 0$

$\Longleftrightarrow$ {$|\ |$-elim1}

$(i+2-k \neq 0 \wedge i+2 \geqslant 0) \vee (-i-2-k \neq 0 \wedge i+2 \leqslant 0)$

3. $(i+j)\ \mathbf{div}\ 2-k > 0$

$\Longleftrightarrow$ {**div**-elim1}

$(-k \geqslant 0 \wedge (i+j-2k > 0 \vee i+j-2k+1 > 0)) \vee$

$(-k \leqslant 0 \wedge (i+j-2k > 0 \vee i+j-2k-1 > 0))$

4. $\max(i,j)+k=0$

$\Longleftrightarrow$ {max-elim1}

$(i+k=0 \wedge i-j \geqslant 0) \vee (j-i \geqslant 0 \wedge j+k=0)$

5. $\min(\lfloor i/2 \rfloor, k) - j \geqslant 0$

$\Longleftrightarrow$ {min-elim1}

$(\lfloor i/2 \rfloor - j \geqslant 0 \wedge \lfloor i/2 \rfloor - k \leqslant 0) \vee (k - \lfloor i/2 \rfloor \leqslant 0 \wedge k-j \geqslant 0)$

{Apply $\lfloor\ \rfloor$-elim1 for the first two conditionals and $\lfloor\ \rfloor$-elim2 for the third}

$\Longleftrightarrow$ $((i-2j \geqslant 0 \vee i-2j-1 \geqslant 0) \wedge (i-2k \leqslant 0 \vee i-2k-1 \leqslant 0)) \vee$

$(2k-i \leqslant 0 \vee 2k-1-i \leqslant 0) \wedge k-j \geqslant 0)$

Theorem 9.1 *Any domain predicate can be put into normal form.*

Proof. It suffices to show that the theorem holds for conditionals of the form E®0. For reasons of symmetry, we shall not consider expressions that are of the last seven forms of rule E. The proof is conducted by a structural induction on the number of the operators in the set O of (9.1) in a conditional. Assume that the

theorem holds for conditionals that contain less than m such operators. Let E®0 be a conditional that contains m such operators. E can be put into the following forms by rule E:

- max(E, L)®0. Apply 0-law and max-elim1.
- min(E, L)®0. Apply 0-law and min-elim1.
- |E|®0. Apply 0-law and | |-elim1.
- (E⊙L) **div** n®0. Apply 0-law and **div**-elim1.
- ⌈(E⊙L)/n⌉®0. Apply 0-law and ⌈ ⌉-elim1.
- ⌊(E⊙L)/n⌋®0. Apply 0-law and ⌊ ⌋-elim1.
- L®0. L can be put into the following forms by rule L:
  - A®0. It is already in normal form.
  - nL®0. Each domain predicate is finite. We apply rule L recursively until we are in one of the other four cases.
  - n max(L_1, L_2)®0. Apply max-dist, 0-law and max-elim1.
  - n min(L_1, L_2)®0. Apply min-dist, 0-law and min-elim1.
  - n|L|®0. Apply | |-law, 0-law and | |-elim1.
- E⊙L®0. Again, E can be put into the following forms by rule E.
  - max(E, L)⊙L®0. Apply max-elim1.
  - min(E, L)⊙L®0. Apply min-elim1.
  - |E|⊙L®0. Apply | |-elim1.
  - (E⊙L_1) **div** n⊙L®0. Apply **div**-elim1.
  - ⌈(E⊙L_1)/n⌉⊙L®0. Apply ⌈ ⌉-elim1.
  - ⌊(E⊙L_1)/n⌋⊙L®0. Apply ⌊ ⌋-elim1.
  - L_1⊙L®0. L_1 can be put into the following forms by rule L:
    - A⊙L®0. L can be put into the following forms by rule L:
      - A⊙A_1®0. It is already in normal form.
      - A⊙$n$$L_1$®0. Consider rule L for L_1 recursively.

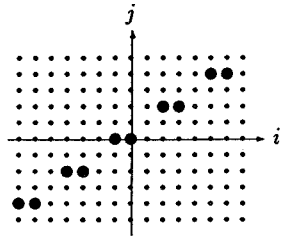


Figure 9–1: The distribution of the points satisfying predicate $2\lfloor i/2\rfloor - j = 0$.

- $A \circledcirc \max(L_2, L_3) \circledR 0$. Apply max-elim2.
- $A \circledcirc \min(L_2, L_3) \circledR 0$. Apply min-elim2.
- $A \circledcirc |L| \circledR 0$. Apply $|\ |$-elim2.

* $nL_2 \circledcirc L \circledR 0$. Consider rule L for L_2 recursively.
* $\max(L_2, L_3) \circledcirc L \circledR 0$. Apply max-elim1.
* $\min(L_2, L_3) \circledcirc L \circledR 0$. Apply min-elim1.
* $|L_1| \circledcirc L \circledR 0$. Apply $|\ |$-elim1.

In each of the above cases, the resulting predicate is still a domain predicate, each of whose constituent conditionals contains less than m operators in the set O of (9.1). The proof is completed by an application of the induction hypothesis. □

Finally, we discuss informally how non-domain predicates can be put in normal form. Let us consider non-domain predicate $2\lfloor i/2\rfloor - j = 0$ given in Ex. 9.1. Rule $\lfloor\ \rfloor$-elim1 does not apply. Fig. 9–1 exposes geometrically the difficulty in normalising this predicate. The points in any of the two lines are not consecutive. They cannot be specified by a finite number of basic conditionals in i and j. Algebraically, predicate $2\lfloor i/2\rfloor - j = 0$ implies the evenness of j.

A non-domain predicate can be put in normal form if it can be rewritten as a domain predicate. The motivation for this transformation is to reuse the indices that do not appear in the non-domain predicate or to introduce extra indices. For example, letting $\lfloor i/2\rfloor = k$ rewrites the previous example predicate to domain predicate $\lfloor i/2\rfloor - k = 0 \wedge j - 2k = 0$, where $j - 2k = 0$ specifies the evenness of j. By the same token, we rewrite the other four non-domain predicates in Ex. 9.1 to the following domain predicates:

2. $2\lceil (i \textbf{ div } 2+j)/3\rceil + 1 \geqslant 0$

$\Longleftrightarrow$ $\{\text{Let } k = \lceil (i \textbf{ div } 2+j)/3\rceil\}$

$2k+1=0 \wedge k - \lceil (i \textbf{ div } 2+j)/3\rceil = 0$

3. $i \textbf{ div } 2 + \lceil (j+k)/2\rceil = 0$

$\Longleftrightarrow$ $\{\text{Let } \ell = \lceil (j+k)/2\rceil\}$

$i \textbf{ div } 2 + \ell = 0 \wedge \ell - \lceil (j+k)/2\rceil = 0$

4. $\lceil i/2\rceil - \lceil (j+k)/3\rceil < 0$

$\Longleftrightarrow$ $\{\text{Let } \ell = \lceil i/2\rceil\}$

$\ell - \lceil (j+k)/3\rceil < 0 \wedge \ell - \lceil i/2\rceil = 0$

5. $\max(\lceil i/2\rceil, \lfloor i/2\rfloor + \lceil j/2\rceil) \neq 0$

$\Longleftrightarrow$ $\{\text{Let } k = \lfloor i/2\rfloor + \lceil j/2\rceil)\}$

$\max(\lceil i/2\rceil, k) \neq 0 \wedge k - \lfloor i/2\rfloor - \lceil j/2\rceil = 0$

$\Longleftrightarrow$ $\{\text{Let } \ell = \lfloor i/2\rfloor\}$

$\max(\ell, k) \neq 0 \wedge k - \ell - \lceil j/2\rceil = 0 \wedge \ell - \lfloor i/2\rfloor = 0$

The five non-domain predicates in Ex. 9.1 become domain predicates if we replace rule E (this implies the deletion of rule L) by

E::=A; |E|;

$E_1 \circledcirc E_2$; $\max(E_1, E_2)$; $\min(E_1, E_2)$; $\lceil (E_1 \circledcirc E_2)/n\rceil$; $\lfloor (E_1 \circledcirc E_2)/n\rfloor$; $(E_1 \circledcirc E_2)$ **div** n

It is not difficult to see that all domain predicates defined this way can be put in normal form if extra indices are introduced as discussed previously. However, the introduction of extra indices increases the dimensionality of the index space and consequently degrades the efficiency of systolic arrays.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] U. Banerjee. *Dependence Analysis for Supercomputing.* The Kluwer Int. Series in Engineering and Computer Science: Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, 1988.

[6] U. Banerjee. Unimodular transformations of double loops. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, chapter 10, pages 192–219. MIT Press, 1991.

[8] M. Barnett and C. Lengauer. The synthesis of systolic programs. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 309–325. Springer-Verlag, 1992.

[9] A. Benaini, P. Quinton, Y. Robert, and B. Tourancheau. Synthesis of a new systolic architecture for the algebraic path problem. *Science of Computer Programming*, 15(2–3):135–158, Dec. 1990.

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

[11] A. Benaini and M. Tchuente. Matrix product on modular linear systolic arrays. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel & Distributed Algorithms*, pages 79–88. North-Holland, 1989.

[12] 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.

[13] P. R. Cappello. A processor-time-minimal systolic array for cubical mesh algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 3(1):4–13, Jan. 1992.

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

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

[16] C. Choffrut and K. Culik. On real-time cellular automata and trellis automata. *Acta Informatica*, 21:393–407, 1984.

[17] P. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. *J. VLSI Signal Processing*, 4(1):27–36, Feb. 1992.

[18] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *Integration*, 12(3):293–304, Dec. 1991.

[19] J.-M. Delosme. A parallel algorithm for the algebraic path problem. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel & Distributed Algorithms*, pages 67–78. North-Holland, 1989.

[20] J. M. Delosme and I. C. F. Ipsen. Systolic array synthesis: Computability and time cones. In M. Cosnard, P. Quinton, Y. Robert, and M. Tchuente, editors, *Parallel Algorithms & Architectures*, pages 295–312. North-Holland, 1986.

[21] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[22] M. L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16(2–3):157–171, Dec. 1990.

[23] B. R. Engstrom and P. R. Cappello. The SDEF programming system. *J. Parallel and Distributed Computing*, pages 201–231, 1989.

[24] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, Feb. 1991.

[25] H. A. Fencl and C.-H. Huang. On the synthesis of programs for various parallel architectures. In *Proc. 1991 Int. Conf. on Parallel Processing, Vol. II*, pages 202–206. Pennsylvania State University Press, 1991.

[26] J. A. B. Fortes and D. I. Moldovan. Parallelism detection and algorithm transformation techniques useful for VLSI architecture design. *J. Parallel and Distributed Computing*, 2(3):277–301, Aug. 1985.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] R. P. Hughey. *Programmable Systolic Arrays.* PhD thesis, Department of Computer Science, Brown University, May 1991. Technical Report CS-91-34.

[33] K. Hwang and Y. H. Chung. Partitioned algorithms and VLSI structures for large-scale matrix computations. In *Proc. Symp. on Computing Arithmetic*, pages 222–232, 1981.

[34] O. H. Ibarra, S. M. Kim, and M. A. Palis. Designing systolic algorithms using sequential machines. *IEEE Trans. on Computers*, C-35:531–542, June 1991.

[35] F. Irigoin and R. Triolet. Dependence approximation and global parallel code generation for nested loops. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel & Distributed Algorithms*, pages 297–308. North-Holland, 1989.

[36] 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.

[37] V. K. Prasanna Kumar and Y.-C. Tsai. Designing linear systolic arrays. *J. Parallel and Distributed Computing*, 7(3):441–463, Nov. 1989.

[38] H. T. Kung. Let's design algorithms for VLSI systems. In *Proc. Caltech Conf. on VLSI*, pages 65–87, 1979.

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

[40] H. T. Kung and M. S. Lam. Wafer-scale integration and two-level pipelined implementations. *J. Parallel and Distributed Computing*, 1(1):33–63, Aug. 1984.

[41] 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.

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

[43] 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.

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

[45] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Processing*, 3:173–182, 1991.

[46] 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.

[47] P. Lee and Z. M. Kedem. On high-speed computing with a programmable linear array. *J. Supercomputing*, 4:223–249, 1990.

[48] C. E. Leiserson and J. B. Saxe. Optimising synchronous circuitry by retiming. In R. Bryant, editor, *Proc. Caltech Conf. on VLSI*, pages 87–116, 1983.

[49] C. Lengauer. Code generation for a systolic computer. *SOFTWARE—Practice and Experience*, 20(3):261–282, Mar. 1990.

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

[51] G. J. Li and B. W. Wah. The design of optimal systolic algorithms. *IEEE Trans. on Computers*, C-34(10):66–77, Jan. 1985.

[52] B. Lisper. *Synthesizing Synchronous Systems by Static Scheduling in Space-Time.* Lecture Notes in Computer Science 362. Springer-Verlag, 1989.

[53] B. Lisper. The Interactive Space-Time Scheduler. *Microprocessing and Microprogramming*, 30(1–5):109–116, Aug. 1990.

[54] B. Lisper. Synthesis of time-optimal systolic arrays with cells with inner structure. *J. Parallel and Distributed Computing*, 10(2):182–187, Oct. 1990.

[55] W. Luk. Specifying and developing regular heterogeneous designs. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, pages 391–409. North-Holland, 1990.

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

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

[58] 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.

[59] 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.

[60] F. J. Núñez and M. Valero. A block algorithm and optimal fixed-size systolic array processor for the algebraic path problem. *J. VLSI Signal Processing*, 1(2):153–162, October 1991.

[61] 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. Swartz-

lander, editors, *Proc. Int. Conf. on Systolic Arrays*, pages 383–391. IEEE Computer Society, 1988.

[62] 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.

[63] 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.

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

[65] 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.

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

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

[68] I. V. Ramakrishnan and P. J. Varman. Dynamic programming and transitive closure on linear pipelines. In *Proc. Int. Conf. on Parallel Processing*, pages 359–364. IEEE Press, 1984.

[69] 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.

[70] S. K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Department of Electrical Engineering, Stanford University, Oct. 1985.

[71] 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.

[72] R. T. Rockafellar. *Convex Analysis.* Princeton University Press, 1970.

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

[74] D. A. Schmidt. *Denotational Semantics.* Wm. C Brown, 1988.

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

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

[77] 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.

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

[79] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[80] M. Valero-García, J. J. Navarro, J. M. Llabería, M. Valeo, and T. Lang. A method for implementation of one-dimensional systolic algorithms with data counterflow using pipelined functional units. *J. VLSI Signal Processing*, 4(1):7–25, Feb. 1992.

[81] V. van Dongen. Quasi-regular arrays: Definition and design methodology. In J. V. McCanny, J. McWhirter, and E. E. Swartzlander, editors, *Systolic Array Processors*, pages 126–135. Prentice Hall, 1989.

[82] V. van Dongen. Mapping uniform recurrences onto small size arrays. 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 191–208. Springer-Verlag, 1991.

[83] 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.

[84] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[85] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[86] 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.

[87] 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.

[88] 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.

[89] Y. Wong and J. M. Delosme. Transformation of broadcasts to propagations in systolic algorithms. *J. Parallel and Distributed Computing*, 14(2):121–145, Feb. 1992.

[90] J. A. Yang and Y.-I. Choo. Parallel-program transformation using a meta-language. In *Proc. Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 11–20. ACM Press, Apr. 1991.

[91] 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.

[92] X. Zhong, S. Rajopadhye, and I. Wong. Systematic generation of linear allocation functions in systolic array design. *J. VLSI Signal Processing.* To appear.