

UT-T00078-SSI
Date: 2015



CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

Laboratorio de Tecnologías de Información

**Síntesis de Arreglo de Procesadores
para Ciclos Anidados con Espacios
de Iteración No-Rectangulares
Usando el Modelo del Politopo**

Tesis que presenta:

José Roberto Pérez Andrade

Para obtener el grado de:

**Doctor en Ciencias
en Computación**

Dr. César Torres Huitzil, Director de la Tesis
Dr. René Armando Cumplido Parra, Co-Director

Cd. Victoria, Tamaulipas, México.

Marzo, 2014

**CINVESTAV
IPN
ADQUISICION
LIBROS**

CLASIF..	UT 00078
ADQUIS..	UT-T00078-001
FECHA:	29-01-2015
PROCED..	Jan. 2015
	\$ _____

U 217789-2001



RESEARCH CENTER FOR ADVANCED STUDY
FROM THE NATIONAL POLYTECHNIC INSTITUTE

Information Technology Laboratory

**Processor Array Synthesis for Loop
Nests with Non-Rectangular
Iteration Spaces Using the
Polytope Model**

Thesis by:

José Roberto Pérez Andrade

as the fulfillment of the
requirement for the degree of:

**Doctor of Science
in Computer Science**

Dr. César Torres Huitzil, Thesis Director
Dr. René Armando Cumplido Parra, Co-Director

Cd. Victoria, Tamaulipas, México.

March, 2014

© Copyright by
José Roberto Pérez Andrade
2014

This work has been carried out with the economic support of the CONACyT, the National Council for Science and Technology from Mexico, through the scholarship number 46577.

The thesis of José Roberto Pérez Andrade is approved by:

Dr. José Luis TecpanecatI Xihuitl

Dr. José Javier Díaz Carmona

Dr. José Juan García Hernández

Dr. Arturo Díaz Pérez

Dr. René Armando Cumplido Parra, Co-Director

Dr. César Torres Huitzil, Thesis Director

Cd. Victoria, Tamaulipas, México., March 7 2014

With all my love to my beloved wife Adriana.
Her patience, support, goodness, and love were the engine
to keep me going on during difficult times.

Acknowledgements

It has been many people which I have met during this stage of my life. In different ways, these people have touched my life, and helped me at different moments of my doctoral studies by bringing me their support, advice, love, and patience. Simply, my words are not enough to show them my deeply gratitude to all of them.

- I am deeply thankful to my wife, Adriana. Her support has been essential from the beginning to the end of this project, her patience has been important to peace my mind, her goodness has been my light in the darkness and her love a warm balsam to my heart. Every second invested during your waiting through these years will be rewarded.
- I am deeply grateful to my parents, Pascual and Amelia, for the support they gave me in every decision I made, for the education they provided, and for all the sacrifices they made. Their advices, support and sacrifices will be possessed forever in my heart. Also, I thank to my brother Isaac and to my sister Saraí, for sharing with me their time when I needed to vent my feelings, and for the support they gave me. I could not be at this point in my life without the help and support of my family and of my wife.
- I would like to express my gratitude, admiration and appreciation to my thesis director Dr. César Torres. His patience, guidance, and confidence during these years helped me to successfully conclude this research. The discussions we had enhance my character for defending the decisions made during the development of this work. His integrity, honesty and teachings will be never forgotten.
- I would like to thank to my thesis co-director Dr. René Cumplido for his advices (the academic and personal ones) from the beginning of this research. Also, I am very thankful for getting me the opportunity of continuing my preparation in another country, and for the financial support that he gave me before the starting the doctoral studies.
- I am thankful for the comments of my thesis reviewers, particularly to Dr. José Juan García whose feedback during the four doctoral seminars enriched this research. Also, I thank to Dr. Manuel Guzmán to encourage me to continue my studies, for his comments about this research, and for his lessons.

- I am especially grateful to Prof. Jürgen Teich, Dr. Frank Hannig, and M. Sc. Srinivas Boppu of the Hardware/Software Co-Desing group at the Erlangen-Nuremberg University. Working with them has enriched this research, my academic experience, and my life. Also, I am thankful to Rafael, Juan, Daniel and Hritam for sharing their time during my stay at Erlangen.
- Thanks to my friends Ana, Erendida, Hugo, and Nelson for sharing to me some of your time when I needed. You made easy my life during my passage through Ciudad Victoria. I will always appreciate it, and never forget your friendship.
- Last but not least, I am deeply thankful to God. Thanks for all the situations that you gave me, the sweet and the bitter ones, since without these situations I would not have become the human being that I am. Thanks God for giving me my wife, my family, my friends and my teachers through my studies at CINVESTAV.

Contents

Contents	i
List of Figures	v
List of Tables	ix
Publications	xiii
Abstract	xv
Resumen	xvii
Nomenclature	xxi
1 Introduction	1
1.1 Parallel Computing in Hardware Architectures	2
1.1.1 High-Level Synthesis	3
1.2 Automatic Parallelization on the Polytope Model	5
1.2.1 Processor Arrays	6
1.2.2 Processor Array Design Methodology	7
1.3 Research Problem	10
1.3.1 Problem Description	10
1.3.2 Research Questions	12
1.3.3 Hypothesis	12
1.3.4 Main Objective and Specific Objectives	12
1.3.5 Contributions	13
1.4 Summary	13
1.5 Document Organization	15
2 Related Work	17
2.1 Automatic Parallelization in Compilers Area	19
2.1.1 Origins of Automatic Parallelization	19
2.1.2 Automatic Parallelization Tools	21
2.2 Automatic Generation of Processor Arrays	23
2.2.1 High-Level Transformations	23
2.2.1.1 Scheduling	24
2.2.1.2 Allocation	25
2.2.1.3 Partitioning	26
2.2.2 Methodologies for Hardware Synthesis	28
2.2.3 Control Generation	30

2.2.4	External Memory	31
2.2.5	Hardware Synthesis Tools	33
2.2.5.1	MMAAlpha	33
2.2.5.2	PARO	35
2.2.5.3	PICO-NPA	38
2.2.5.4	CLooG-VHDL	40
2.2.5.5	Compaan/Laura	41
2.2.5.6	Other Works	42
2.3	Literature Discussion	42
3	Mathematical Background	47
3.1	Algorithmic Modeling	48
3.1.1	Polytope Model	48
3.1.2	Piecewise Regular Algorithm	51
3.1.3	Data Dependences	53
3.2	Space-Time Transformation in the Polytope Model	59
3.2.1	Unimodular Transformation	59
3.2.2	Scheduling Function	61
3.2.3	Allocation Function	64
3.2.4	Iteration Interval	65
3.2.5	Space-Time Mapping	66
3.3	Processor Array Synthesis	70
3.3.1	Full-Size Implementation	71
3.3.1.1	Processor Array Interconnection Topology	71
3.3.1.2	Processor Element Data-Path	72
3.3.2	Partitioning of the Processor Space	76
3.3.2.1	Partitioning as Allocation	77
3.3.2.2	Partitioning with Strip Mining	79
3.3.2.3	Iteration Space of Partitioned Target Polytopes	84
3.3.2.4	Intermediate Memories	85
3.4	Summary	85
4	Control Scheme	87
4.1	Hybrid Control Scheme	88
4.1.1	Motivation	89
4.1.2	Control Scheme Description	91
4.2	Hybrid Control Architecture	93
4.2.1	Sequence Generator	93
4.2.2	Activation-Signal Injector	98
4.2.3	Control Array	100
4.3	Number of Logic Elements	103
4.4	Cholesky Decomposition Case of Study	105
4.5	Summary	111

5	External Memory	113
5.1	Memory Hierarchy	114
5.2	Intermediate Memory	115
5.2.1	Motivation	116
5.2.2	Internal FIFOs	117
5.2.2.1	Rectangular Iteration Space	118
5.2.2.2	Non-Rectangular Iteration Space	120
5.3	External Memory	120
5.3.1	Motivation	122
5.3.2	Architectural Memory Cases	123
5.3.3	External Memory Architectural Scheme	127
5.3.3.1	Memory Banks	128
5.3.3.2	Address Generator Unit	129
5.3.3.3	Input Variable Border Mapping	131
5.3.3.4	Output Variable Border Mapping	133
5.3.3.5	Input Variable Broadcast Mapping	136
5.3.3.6	Output Variable Broadcast Mapping	138
5.4	Number of Logic Elements	140
5.5	Matrix-Matrix Multiplication Case of Study	141
5.6	Summary	146
6	Results	149
6.1	Software Tools and Technological Platform	150
6.1.1	Software Tools	150
6.1.2	FPGA Architecture Overview	151
6.1.3	Motivation of Using the FPGAs	153
6.2	Control Place and Route Results	154
6.2.1	Implementation Results	155
6.2.2	Comparison	161
6.3	External Memory Place and Route Results	162
6.3.1	Implementation Results	163
6.4	Cases of Studies: Place and Route Results	167
6.4.1	MatMul Results	168
6.4.1.1	Place and Route	169
6.4.1.2	Comparison	176
6.4.2	Cholesky Results	178
6.4.2.1	Place and Route	179
6.4.3	Embedded Platform	185
6.5	Evaluation Metrics	191
6.5.1	Acceleration	191
6.5.2	Efficiency	193
6.5.3	Relative Load Imbalance	193
6.6	Processor Arrays Evaluation	194

6.6.1	Implemented Processor Arrays Evaluation	195
6.6.2	Design Space Exploration	199
6.6.2.1	One-Dimensional Arrays	201
6.6.2.2	Two-Dimensional Arrays	202
6.7	Summary	209
7	Conclusions and Future Directions	213
7.1	Conclusions	214
7.1.1	Contributions	216
7.1.2	Revisiting the Research Questions	217
7.2	Future Work	219
A	Appendix	223
A.1	Matrix and FIR Filter	225
A.2	Back and Forward Substitution	227
A.3	Matrix-Matrix Multiplication	229
A.4	Cholesky Decomposition	232
A.5	LU Decomposition	237
	Bibliography	243

List of Figures

1.1	Design flow methodology for deriving systolic arrays.	8
1.2	Design flow methodology followed in the polytope model with partitioning.	10
2.1	Origins and works derived from the polytope model.	18
2.2	Design flow of the MMAAlpha programming environment. Image taken from [36].	34
2.3	Design flow of the PARO framework. Image taken from [64].	36
2.4	PICO-NPA design system components. Image taken from [72].	39
2.5	Design flow of CLoog-VHDL back-end. Image taken from [39].	41
3.1	Matrix-Matrix Multiplication pseudocode.	50
3.2	Matrix-Matrix Multiplication piecewise regular algorithm.	53
3.3	MatMul dependence graph for $N = 4$.	55
3.4	Piecewise regular algorithm for Cholesky decomposition.	56
3.5	Cholesky decomposition dependence graph for $N = 4$.	57
3.6	Reduced dependence graph for MatMul algorithm.	58
3.7	Cholesky reduced dependence graph.	59
3.8	Computation dates assigned to different operations when a linear scheduler is used.	63
3.9	Computation dates assigned to different operations when an affine scheduler is used.	64
3.10	Execution times for each operation in an index point.	69
3.11	Processor space of Cholesky decomposition when $N=4$.	70
3.12	Full-size processor array for MatMult algorithm when $N=5$.	74
3.13	Four processing elements types for MatMul processor array.	75
3.14	Partitioned iteration space for FIR filter with LSGP and LPGS approaches.	77
3.15	The eight different scanning order for a two-dimensional example.	78
3.16	Partitioned MatMul processor space by strips of size four.	82
3.17	Partitioned Cholesky processor space by strips of size four.	83
4.1	Design flow methodology followed in the polytope model.	88
4.2	Example of invalid mapping from the non-rectangular logical array to the physical array.	90
4.3	Processor array hybrid controller block diagram.	92
4.4	Counter-like sub-module internal architecture.	95
4.5	Counter-like sub-modules interconnection when $h = 3$.	96
4.6	Mealy finite state machine for generating <i>Hold</i> and <i>Load</i> signals of <i>Counter₀</i> and <i>Counter₂</i> sub-modules.	97
4.7	Mealy finite state machine for generating <i>Load</i> signal of <i>Counter₁</i> sub-module.	97
4.8	Injection of <i>IndexBus</i> to the processor array from the activation signal injector.	98
4.9	First approach for the activation-signal injector.	99
4.10	Second approach for the activation-signal injector.	100
4.11	Generalization of the control cell architecture.	102

4.12	Mealy finite state for generating the activation pattern.	103
4.13	Activation of a full-size processor array and its mapping to 2x4 physical array.	108
4.14	Internal control cell architecture for Cholesky algorithm.	109
4.15	Interconnection of the control cells forming the control array for Cholesky algorithm	110
5.1	Design flow methodology followed in the polytope model.	114
5.2	Processor array memory hierarchy.	115
5.3	Physical processor array and the connection among two intermediate memory levels.	117
5.4	Interconnection of FIFO memory and its control for rectangular iteration spaces.	119
5.5	Moore finite state machine for generating the FIFOs <i>ReadEnable</i> for rectangular iteration spaces.	120
5.6	Moore finite state machine for generating the FIFOs <i>WriteEnable</i> for non-rectangular iteration spaces.	121
5.7	Moore finite state machine for generating the FIFOs <i>ReadEnable</i> for non-rectangular iteration spaces.	121
5.8	Interconnection of FIFO memory and its control for non-rectangular iteration spaces.	121
5.9	Architectural cases according to the variable type and the mapping possibilities.	125
5.10	Parallel memory accesses according to each architectural cases.	126
5.11	Column-major and row-major order cases for matrix data segmentation.	128
5.12	Examples of data segmentation and distribution for two memory banks.	129
5.13	The two possible AGUs architectures for border and broadcast mapping.	131
5.14	Two-Address generator module in charge of generating two memory bank addresses.	132
5.15	SIPO border module for interfacing two data.	133
5.16	Interconnection of TAGM, dual-port memory and SIPO for input border case.	133
5.17	PISO border module in charge of multiplexing two processor array inputs.	134
5.18	Interconnection of TAGM, dual-port memory and PISO for output border case.	135
5.19	Transporting element.	135
5.20	Interconnection of TAGM, dual-port memory and SIPO for input broadcast case.	136
5.21	Broadcast data array for an 8×8 processor array.	138
5.22	SIPO broadcast module in charge of sending data to the processor array.	139
5.23	PISO broadcast module in charge of sending data to the processor array.	140
5.24	Interconnection of TAGM, dual-port memory and PISO for output broadcast case.	140
5.25	Matrix-Matrix Multiplication piecewise regular algorithm.	142
5.26	Architectural border case for input variable A_{in} .	143
5.27	Architectural broadcast case for input variable B_{in} .	144
5.28	Architectural border case for output variable C_{out} .	145
5.29	Memory system for MatMul algorithm combining the I/O variable cases.	145
6.1	Block diagram for the integration of memory system, control and data-path.	167
6.2	FPGA resource utilization varying the control word for the second MatMul set.	174
6.3	Distribution of the FPGA resources for the second MatMul set.	175
6.4	FPGA resource utilization varying the control word for the third Cholesky set.	184
6.5	Distribution of the FPGA resources for the fourth Cholesky set.	185

6.6	MatMul and Cholesky execution times for their processor arrays and their MicroBlaze implementations.	189
6.7	MatMul and Cholesky throughput per power unit for their processor arrays and their MicroBlaze implementations.	190
6.8	Example of a profile of parallelism.	192
6.9	Cholesky profiles of parallelism for an 8×8 processor array and using the constant-cycle and multi-cycle PEs as baseline.	198
6.10	Efficiency of three MatMul processor arrays.	203
6.11	Relative load imbalance of an 8×8 MatMul processor array.	205
6.12	Relative efficiency and relative load imbalance of three different processor array derived by different allocation functions.	207
6.13	Relative load imbalance of different processor array of size 2×8 and 8×2 derived by different allocation functions.	207

List of Tables

2.1	Comparison of characteristic among automatic parallelization tools.	43
2.2	Comparison among automatic parallelization tools and this research work.	45
3.1	Activation table for the MatMul PE control signals according to the iteration dependent conditions.	75
4.1	Hardware resource utilization of the control architecture in terms of three parameters.	104
4.2	Characterization of the hardware resource utilization for the MatMul control scheme.	104
4.3	Truth table 16:4 decoder	107
4.4	Characterization of the hardware resource utilization for the Cholesky control scheme.	110
5.1	Hardware resource utilization for each memory architectural case.	141
5.2	Characterization of the hardware resource utilization of the MatMul case of study.	146
6.1	Description of the three control implementation sets with their design parameters.	156
6.2	Implementation <i>Ctrlmpl 1</i> PAR results targeted for a XC6VCX240T FPGA device.	157
6.3	Implementation <i>Ctrlmpl 2</i> PAR results targeted for a XC6VCX240T FPGA device.	158
6.4	Implementation <i>Ctrlmpl 3</i> PAR results targeted for a XC6VCX240T FPGA device.	159
6.5	Slices comparison among several control architectures.	162
6.6	Place and route results for the MatMul memory architectural cases.	164
6.7	Place and route results for the Cholesky memory architectural cases.	166
6.8	Description of the first MatMul implementation set with their parameters.	169
6.9	Place and route results for the first MatMul implementation set for a XC6VCX240T FPGA device.	170
6.10	Description of the second MatMul implementation set with their parameters.	171
6.11	Place and route results for the implementations of the 2×2 MatMul system for a XC6VSX475T FPGA device.	173
6.12	Place and route results for the implementations of the 4×4 MatMul system for a XC6VSX475T FPGA device.	173
6.13	Place and route results for the implementations of the 8×8 MatMul system for a XC6VSX475T FPGA device.	173
6.14	Comparison of two 2×2 MatMul processor arrays generated by PARO and by the proposed processor arrays.	177
6.15	Description of the third Cholesky implementation set with their respective parameters. These implementations use the FPGA on-chip memories for storing the input and output matrixes.	179
6.16	Place and route results for the third Cholesky implementation set for a XC6VCX240T FPGA device.	180
6.17	Description of the fourth Cholesky implementation set with their parameters.	181

6.18	Place and route results for the 2×2 Cholesky implementation set for a XC6VSX475T FPGA device.	182
6.19	Place and route results for the 4×4 Cholesky implementation set for a XC6VSX475T FPGA device.	182
6.20	Place and route results for the 8×8 Cholesky implementation set for a XC6VSX475T FPGA device.	182
6.21	Description of the implementation set targeted for an embedded platform.	186
6.22	Place and Route results for a Microblaze and three processor arrays implementations targeted for a XC6SLX45 FPGA device.	187
6.23	Average speed-up and energy consumption per LUT of three processor arrays.	188
6.24	Evaluation metrics for the second MatMul processor array implementation set.	196
6.25	Evaluation metrics for the fourth Cholesky processor array implementation set.	197
6.26	Allocation functions used for two-dimensional algorithms.	199
6.27	Allocation functions used for three-dimensional algorithms.	200
6.28	Acceleration for MatVec/FIR and Back/Foward processor arrays using the constant-cycle PE as baseline.	201
6.29	Average relative efficiency for several MatMul processor arrays and different allocation matrixes.	203
6.30	Average relative load imbalance for several MatMul processor arrays and different allocation matrixes.	204
6.31	Average relative efficiency for several LU processor arrays and different allocation matrixes.	206
6.32	Information for the implementation of an 8×8 Cholesky processor array gathering the metrics and the number of hardware elements required by the data-path, memory and control.	209
A.1	Allocation functions used for two-dimensional algorithms.	224
A.2	Allocation functions used for three-dimensional algorithms.	224
A.3	Average relative acceleration for several MatVec processor arrays using the sequential constant-cycle PE as baseline.	225
A.4	Average relative efficiency for several MatVec processor arrays using the sequential constant-cycle PE as baseline.	225
A.5	Average relative load imbalance for several MatVec processor arrays using the sequential constant-cycle PE as baseline.	226
A.6	Average relative acceleration for several BackForward processor arrays using the sequential constant-cycle PE as baseline.	227
A.7	Average relative efficiency for several BackForward processor arrays using the sequential constant-cycle PE as baseline.	227
A.8	Average relative load imbalance for several BackForward processor arrays using the sequential constant-cycle PE as baseline.	228
A.9	Average relative acceleration for several BackForward processor arrays using the sequential multi-cycle PE as baseline.	228

A.10 Average relative efficiency for several BackForward processor arrays using the sequential multi-cycle PE as baseline.	228
A.11 Average relative acceleration for several MatMul processor arrays using the sequential constant-cycle PE as baseline.	229
A.12 Average relative efficiency for several MatMul processor arrays using the sequential constant-cycle PE as baseline.	230
A.13 Average relative load imbalance for several MatMul processor arrays using the sequential constant-cycle PE as baseline.	231
A.14 Average relative acceleration for several Cholesky processor arrays using the sequential constant-cycle PE as baseline.	232
A.15 Average relative efficiency for several Cholesky processor arrays using the sequential constant-cycle PE as baseline.	233
A.16 Average relative relative load imbalance for several Cholesky processor arrays using the sequential constant-cycle PE as baseline.	234
A.17 Average relative acceleration for several Cholesky processor arrays using the sequential multi-cycle PE as baseline.	235
A.18 Average relative efficiency for several Cholesky processor arrays using the sequential multi-cycle PE as baseline.	236
A.19 Average relative acceleration for several LU processor arrays using the sequential constant-cycle PE as baseline.	237
A.20 Average relative efficiency for several LU processor arrays using the sequential constant-cycle PE as baseline.	238
A.21 Average relative load imbalance for several LU processor arrays using the sequential constant-cycle PE as baseline.	239
A.22 Average relative acceleration for several LU processor arrays using the sequential multi-cycle PE as baseline.	240
A.23 Average relative efficiency for several LU processor arrays using the sequential multi-cycle PE as baseline.	241

Publications

Publications from the Research

Roberto Perez-Andrade, Cesar Torres-Huitzil, Rene Cumplido and Juan M. Campos. *On a Hybrid and General Control Scheme for Algorithms Represented as a Polytope*, in Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPSW), pages 330–333, Anchorage, AK, USA, March, 2011.

Roberto Perez-Andrade, Cesar Torres-Huitzil, Rene Cumplido and Juan M. Campos. *On an External Memory Scheme for Processor Arrays*, IEICE Electronics Express, 10(14), page 20130324, 2013.

Roberto Perez-Andrade, Rene Cumplido, Juan M. Campos, Cesar Torres-Huitzil. *Processor Arrays Generation for Matrix Algorithms Used in Embedded Platforms*, in Proceedings of the International Conference on Reconfigurable Computing and FPGAs, ReConFig, pages 1–6, Cancun, Mexico, December, 2013.

Publications from Collaborations

Srinivas Boppu, Frank Hannig, Jürgen Teich and Roberto Perez-Andrade. *Towards Symbolic Run-Time Reconfiguration in Tightly-Coupled Processor Arrays*, in Proceedings of the International Conference on Reconfigurable Computing and FPGAs, ReConFig, pages 392–397, Cancun, Mexico, December, 2011.

Fernando Martin del Campo, Alicia Morales-Reyes, Rene Cumplido, Roberto Perez-Andrade, Aldo Orozco-Lugo. *A Multi-cycle Fixed Point Square Root Module for FPGAs*, IEICE Electronics Express, 9(11), pages 971-977, 2012.

Processor Array Synthesis for Loop Nests with Non-Rectangular Iteration Spaces Using the Polytope Model

by

José Roberto Pérez Andrade

Doctor of Science in Computer Science, Information Technology Laboratory
Research Center for Advance Study from the National Polytechnic Institute, 2014

Dr. César Torres Huitzil, Thesis Director

Dr. René Armando Cumplido Parra, Co-Director

High-level synthesis methods are concerned with the translation of algorithmic specifications into representations at register transfer level or into a hardware description language. One of the representations used for high-level synthesis is the polytope model, which provides an abstraction to represent loop computations of an algorithmic specification as integer points inside of a polyhedron. As a results, the polytope model is able to derive dedicated hardware parallel architectures in form of processor arrays. Processor arrays consist of a set of processing elements connected in a regular and local way, and able to exploit several levels of parallelism. In order to derive totally functional processor arrays, besides of the processor array data-path, control schemes able to generate the processing elements activation signals, and able to select the required operations are needed. Also, external memory systems capable of providing data from an external source to the array and capable of extracting data produced by the array are required.

Previous research works have focused on the generation of processor arrays able to deal with a unique problem size, and for algorithms whose loop bounds form a rectangular shape. In this dissertation, a control scheme able to generate the control signals for algorithms with rectangular and non-rectauglar loop bounds, and whose problem size is larger than a maximum value provided during synthesis time is proposed. This control scheme uses local and distributed modules in order to

orchestrate the computations of the processor array. On the other hand, also previous works assume that the input data are available when the processor array requires them, and they assume that the output data are extracted when they are produced. In this sense, this dissertation also proposes an external memory system built on four architectural cases which could occur using the polytope model. These architectural cases are based on the use of a multi-clock domain approach, and on the use of dual-port memories.

The proposed control scheme and memory system are integrated into a hardware architecture framework which was validated generating functional processor arrays for two cases of study: matrix-matrix multiplication and Cholesky decomposition algorithms. Each generated processor array has different design parameters, and different processor array sizes. All these processor arrays are targeted for different FPGAs devices. Experimental results exhibit that there is a major impact on increasing the size of the control on the operational frequencies than increasing the problem size that the processor array can solve. Moreover, the external memory results show that the peak I/O bandwidth produced by each of the four architectural cases exceeds the processor array I/O requirements. Also, results demonstrate that one limitation for implementing the processor arrays (including data-path, control and memory) is the amount of memory available inside the FPGAs. Furthermore, the results suggest that solving larger problem sizes comes at the price of dedicating more FPGA silicon to store data than to compute data. Finally, these processor arrays were evaluated with traditional metrics such as acceleration, efficiency and relative load imbalance, showing that algorithms with rectangular loop bounds and low latency operations have a major acceleration, a higher efficiency, and a lower load imbalance than algorithms with non-rectangular loop bounds and high latency hardware operations.

Síntesis de Arreglo de Procesadores para Ciclos Anidados con Espacios de Iteración No-Rectangulares Usando el Modelo del Politopo

por

José Roberto Pérez Andrade

Doctor en Ciencias en Computación, Laboratorio de Tecnologías de Información
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2014

Dr. César Torres Huitzil, Director de la Tesis
Dr. René Armando Cumplido Parra, Co-Director

Los métodos de síntesis de alto nivel (HLS por sus siglas en inglés) están enfocados a la transformación de especificaciones algorítmicas en representaciones a nivel de transferencia de registro (RTL por sus siglas en inglés) o en lenguajes de descripción de hardware. Entre las representaciones usadas de HLS se encuentra el modelo del politopo, el cual provee una abstracción para representar los cálculos anidados de una especificación algorítmica como puntos enteros de un poliedro. Como resultado, el modelo del politopo es capaz de derivar arquitecturas paralelas de hardware dedicado en forma de arreglo de procesadores. Un arreglo de procesadores consiste en un conjunto de elementos de procesamiento conectados localmente y de manera regular, los cuales son capaces de explotar distintos niveles de paralelismo. Con el propósito de derivar arreglos de procesadores totalmente funcionales, además de producir la trayectoria de datos de dichos arreglos, es necesario crear esquemas de control capaces de generar las señales de activación de los elementos de procesamiento, y capaces de seleccionar de manera correcta las operaciones requeridas. Así mismo, sistemas de memoria externos capaces de proveer los datos desde una fuente externa al arreglo de procesadores, y capaces de extraer los datos generados por el arreglo son requeridos.

Trabajos de investigación previos se han enfocado en la generación de arreglos de procesadores capaces de resolver un sólo tamaño de problema y en algoritmos cuyos ciclos anidados tienen formas

rectangulares. En esta tesis, se propone un esquema de control capaz de generar las secuencias de activación para algoritmos cuyos ciclos anidados tengan formas rectangulares o no rectangulares, y cuyo tamaño de problema no sea más grande que un valor máximo dado en tiempo de síntesis. Este esquema de control se basa en el uso de módulos distribuidos y centralizados con el propósito de orquestar los cómputos del arreglo de procesadores. Por otra parte, también en trabajos previos se asume que los datos de entrada están siempre disponibles cuando el arreglo de procesadores los requiere, y además se asume que los datos de salida son extraídos automáticamente cuando éstos son producidos. En dicho sentido, en este trabajo de investigación se propone un sistema de memoria externo construido a partir de cuatro casos arquitecturales que pueden ocurrir cuando el modelo del politopo es usado. Dichos casos están basados en el uso de un enfoque de diseño de múltiples relojes y en el uso memorias con puertos duales.

Tanto el esquema de control propuesto y sistema de memoria están integrados en una arquitectura hardware que fue validada generando arreglos de procesadores para dos casos de estudio: el algoritmo de multiplicación de matrices y el algoritmo para la descomposición matricial de Cholesky. Cada arreglo de procesadores tiene distintos parámetros de diseño y diferentes tamaños. Todos los procesadores fueron sintetizados para diferentes dispositivos FPGAs. Resultados experimentales muestran que las frecuencias de operación del esquema de control son mayormente afectadas cuando el tamaño máximo de problema (para el cual el control puede generar las señales de activación) es incrementado que cuando el tamaño del esquema de control se incrementa. Además, los resultados del sistema externo de memoria muestran que el ancho de banda producido por cada uno de los cuatro casos arquitecturales exceden los requerimientos de entrada/salida del arreglo de procesadores. También, los resultados experimentales muestran que una de limitación para la implementación de los arreglos de procesadores (incluyendo trayectoria de datos, control y memoria) es la cantidad de memoria disponible dentro del FPGA. Además, los resultados muestran que al resolver tamaños de problema más grandes se dedica una mayor cantidad silicio para almacenar datos que para realizar cómputos. Finalmente, los arreglos de procesadores fueron evaluados usando métricas tradicionales como los son la aceleración, eficiencia y desequilibrio de carga relativo, mostrando que algoritmos

cuyos ciclos anidados tienen formas rectangulares y operaciones de baja latencia en hardware tienen una mayor aceleración, una eficiencia alta y un bajo desequilibrio de carga que en el caso de algoritmos con formas no-rectangulares y alta latencia en hardware.

Nomenclature

Acronyms:

- **AGU** : Address Generator Unit
- **ASIC** : Application Specific Integrated Circuit
- **AST** : Abstract-Syntax Tree
- **BRAM** : Block Random Access Memory
- **C.C.** : Constant Cycle
- **CDFG** : Control-Data Flow Graph
- **CDMA** : Code Division Multiple Access
- **CLB** : Configurable Logic Blocks
- **CLooG** : Chunky Loop Generator
- **CPU** : Central Processing Unit
- **DCN** : Data Come as it is Needed
- **DCT** : Discrete Cosine Transform
- **DG** : Dependence Graph
- **DLP** : Data Loop Parallelism
- **DMA** : Direct Memory Access
- **DoP** : Degree of Parallelism
- **DPRA** : Dynamic Piecewise Regular Algorithm
- **DSP** : Digital Signal Processor or Digital Signal Processing
- **FF** : Flip-Flip
- **FF-T** : Flip-Flip T
- **FFT** : Fast Fourier Transform
- **FIFO** : First-In First-Out

- **FIR** : Finite Impulse Response
- **FPGA** : Field Programmable Gate Array
- **FSM** : Finite State Machine
- **GALS** : Globally-Asynchronous Locally-Synchronous
- **GPU** : Graphics Processing Unit
- **HLS** : High-Level Synthesis
- **HTG** : Hierarchical Task Graph
- **I/O** : Input/Output
- **ILP** : Instruction Level Parallelism
- **LDPC** : Low Density Parity Check
- **LLP** : Loop Level Parallelism
- **LLVM** : Low Level Virtual Machine
- **LPGS** : Local Parallel Global Serial
- **LSE** : Least-Square Estimation
- **LSGP** : Local Serial Global Parallel
- **LUT** : Look-Up Table
- **MAC** : Multiplication Accumulation
- **MatMul** : Matrix-Matrix Multiplication
- **MIMO** : Multiple Input Multiple Output
- **MLR** : Multiple-Parameter Linear Regression
- **OFDM** : Orthogonal Frequency Division Multiplexing
- **PA** : Processing Array
- **PAR** : Place-and-Route
- **PE** : Processing Element
- **PIP** : Parametric Integer Programming
- **PISO** : Parallel Input Serial Output
- **PLA** : Piecewise Linear Algorithm
- **PPA** : Pipeline of Processing Arrays
- **RDG** : Reduced Dependence Graph

- **RIA** : Regular Iterative Algorithm
- **ROCCC** : Riverside Optimizing Compiler for Configurable Computing
- **RTL** : Register Transfer Level
- **SARE** : System of Affine Recurrence Equations
- **SIPO** : Serial Input Parallel Output
- **SoC** : System-on-Chip
- **SRAM** : Static Random Access Memory
- **SUIF** : Stanford University Intermediate Format
- **SURE** : System of Uniform Recurrence Equations
- **TAGM** : Two-Address Generator Module
- **TE** : Transporting Element
- **VHDL** : Very High Speed Integrated Circuit Hardware Description Language
- **VLSI** : Very Large Scale Integration
- **WPPA** : Weakly Programmable Processor Arrays
- **WPPE** : Weakly Programmable Processing Element

Symbols and Notation:

- \mathcal{I} Iteration space of the source polytope
- \mathcal{J} Iteration space of the target polytope
- $C^I(\vec{I})$ Iteration dependent condition
- \vec{I} Index or iteration point of the source polytope
- \vec{J} Index or iteration point of the target polytope
- \vec{d}_j Dependence vector
- D Dependence matrix
- T Transformation matrix
- $\vec{\lambda}$ Scheduler function
- $\vec{\lambda}_a$ Affine scheduler function
- $\vec{\lambda}_l$ Linear scheduler function
- \vec{u} Projection vector
- Φ Allocation function
- P Iteration Interval
- \mathcal{T} Time space
- \mathcal{P} Processor space
- \vec{s} Connection vector
- \vec{r} Delay vector
- X Tiling matrix
- L Loop matrix
- SS_{p_0} Strip Size of index p_0
- SS_{p_1} Strip Size of index p_1
- SS_{p_i} Strip Size of index p_0 or p_1
- \mathcal{P}_r Rectangular processor space
- \mathcal{P}_{nr} Non-Rectangular processor space
- W_c Control word

- N_{max} Largest problem size that can be solved with W_c
- $T_1(N)$ Execution time of a sequential implementation for a problem of size N
- $T_p(N)$ Execution time of a parallel system with p processors for a problem of size N
- $S_p(N)$ Acceleration of a parallel system with p processors for a problem of size N
- $E_p(N)$ Efficiency of a parallel system with p processors for a problem of size N
- $L_i(N)$ Load imbalance of a parallel system with p processors for a problem of size N
- $Lr_p(N)$ Relative load imbalance of a parallel system with p processors for a problem of size N
- $W(N)$ Computational work of a problem of size N
- $W_i(N)$ Computational work of a i -th processor for problem of size N
- $W_{max}(N)$ Maximum computational work of a processor for a problem of size N

1

Introduction

Research in hardware architecture of computer systems has always been a central interest of the computer science and computer engineering communities. The investigation goals vary according to the target application, the programmability of the systems, the environment on which the processors will be working on, and many other factors. In high-performance computing systems (HPC) the main goal is focused on obtaining a high parallelism degree and high communication rates regardless the computational resources neither the power consumed. On the other hand, in an embedded scenario, a balance between power consumption and computational performance is desired.

In fact, in last years, the trend to make a trade-off between computational performance and power consumption has been increased. This is specially true in embedded and mobile scenarios, where for example; a mobile phone that supports video playback has a strict power budget, but it still has to meet certain performance criteria [20]. Traditionally, these intensive computational performances were met only by technology advances like smaller transistor area, higher clock rates and other improvements. However, problems like power dissipation and thermal constraints have emerged as dominant design issues; forcing computer designers away from relying on increasing frequency to

improve performance. Thus, using multiple processing units for performing parallel computations and completing a larger volume of work has been the current trend in order to improve performance, specially in signal processing applications [85].

1.1 Parallel Computing in Hardware Architectures

The field of parallel processing is concerned with architectural and algorithmic methods for enhancing the performance or other attributes of computers through exploiting different forms of parallelism. Nowadays, the main trend for achieving greater performance and increasing the computational efficiency is by exploiting different forms of parallelism such as instruction, data or loop level parallelism. With instruction level parallelism (ILP), multiple independent instructions are executed in a processor in the same clock cycle. In data level parallelism (DLP) approach the same instruction is performed on different pieces of data in parallel. Loop level parallelism (LLP) approach is one of the popular parallelization methods in the scientific computing community. In this form of parallelization, independent iterations of the same loop are executed in parallel on different processors [85]. The most important opportunities for both parallelism and locality-based optimizations come from loops that access arrays. These loops tend to have limited dependences among access to array elements and to access arrays in a regular pattern, allowing good data locality, which is important in hardware implementations [6].

Several numeric kernels used in signal processing domain can be computed in forms of nested loops, representing niches of opportunities for being implemented in hardware platforms exploiting the loop level parallelism. In fact, it is commonly accepted that at least an 80% of the execution time is typically spent in computing nested loops, which represent the 20% of program codes [99]. These numeric kernels are required in order to build "complex" algorithms or electronic systems. Algorithms such as matrix multiplication, matrix decomposition, convolution and system equations solvers are used as base for building more complex systems. For example, matrix multiplication is required in some Fast Fourier Transform (FFT) algorithm implementations [92], while algorithms for

MIMO (Multiple Input Multiple Output) systems require matrix decompositions such as Cholesky or QR algorithms [82], [83] in order to meet the MIMO systems requirements. Also, other matrix algorithms such as matrix-vector multiplication, LU decomposition, forward and back substitution, as well as the previous algorithms mentioned are used for a wide variety of applications such as facial feature extraction [67], solving equation systems in LDPC (Low Density Parity Check) codes [114], autocorrelation of signals in sensor arrays environments [42], signatures for CDMA (Code-Division Multiple-Access) communications [60], detection techniques for OFDM (Orthogonal Frequency Division Multiplexing) for MIMO systems [83], classification and target tracking in wireless sensor networks [70], adaptive filtering and two-dimensional finite impulse response (FIR) realization [89], orthogonal matching pursuit for signal reconstruction in compressed sensing theory [122], least-square estimation (LSE) and multiple-parameter linear regression (MLR) [113], among other applications.

1.1.1 High-Level Synthesis

Traditionally, the parallelism of the aforementioned numeric kernels has been exploited by using digital signal processors (DSPs) which are optimized for performing in parallel the most common used operations in signal processing applications like multiplication-accumulation (MAC) operation. Also, highly specialized coprocessor units based on application specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs) platforms have been used for exploiting the LLP by taking advantage of spatial computing paradigm [25]. Independently of the technology employed, computer designers are often directly responsible for crafting these highly specialized application-specific architectures. Despite that these hand-made architectures exploit data and control flow parallelism intrinsic in the algorithm, considerable effort and time are required to develop such hardware implementations. The difficulty of designing these hardware architectures relies on:

- As a first step, hardware designers must understand the algorithm to be implemented into hardware in order to find sources of parallelism.
- Then the designer should explore several ideas for different architectural schemes that completely or partially exploits the algorithm data parallelism.

- When the first architectural drafts are conceived, hardware designers must come up with control mechanisms and memory systems in order to control, to feed and to extract data to/from the hardware data-path. Also, designers should find a balance between hardware resources and speed requirements.
- Finally, the designer must select one or some explored ideas that can be implemented into the hardware architecture.

In summary, these hardware architectures could be very complex to design in a hand-made fashion, leading to spend much time for the design space exploration. Moreover, the design of these parallel architectures by hand is cumbersome and error-prone. A hardware assisted approach to automatically generate dedicated hardware architectures might be beneficial for implementing loop-based algorithms. High-level synthesis (HLS) methods are not error-prone and allow a faster design space exploration than in the case of hand-made architectures [44]. Generally, HLS methods try to extract automatically the parallelism presented in an algorithmic input specification, and at the same time, they derive parallel hardware structures from this algorithmic input. The parallelism extraction and derivation of the hardware structures are performed, by a synthesizer, at a high-level where most of the information needed for deriving the hardware architectures is available. Similarly, in the software context, a compiler is in charge of extracting the parallelism from a sequential program. In both contexts, the process of extracting the parallelism for parallel software or deriving hardware modules is called automatic parallelization.

High-level synthesis is the process of automatic generation of hardware circuits from behavioral descriptions. The target hardware circuit consists of a structural composition of data-path, control and memory elements. The fundamental tasks in HLS methods are decomposed into hardware modeling, scheduling, resource allocation and binding, and control generation [62]. Several HLS methods can be found in the literature, each of them using different hardware models and deriving different kind of hardware architectures. Among these models are the control-data flow graphs (CDFG) [90], the hierarchical task graphs (HTGs) [63], the Kahn process networks [5] and the

polytope model [80]. Although some these HLS approaches use as algorithmic representation loop-based algorithms they do not offer any parallelization, since they generate highly-pipelined mono-processors in order to achieve a higher data throughput. However, the polytope model is able to expose the loop level parallelism of sequential loop-based programs.

1.2 Automatic Parallelization on the Polytope Model

The polytope model is an intermediate abstraction based on parametric integer linear algebra, integer linear programming, affine array accesses, and transformation based on integer matrixes. Formally, a polytope is defined as a geometric object whose volume is defined by an intersection of a finite number of $n - 1$ sub-spaces that divide an n dimensional space. The polytope can be expressed by a system of inequalities that defines each one of its faces. Applying the polytope model to a sequential program, each one of these faces represents a lower or upper loop bound of a nested loop, while the set of integer point belonging to the polytope represents an iteration from a loop program.

The polytope model is recognized as a parallelizing approach for transforming loop programs into parallel architectures with distributed memory. Methods and tools based on it have been used in parallelizing compilers since some years ago for distributed memory parallel machines [80]. Also, this model provides an abstraction to represent loop nest computations and its data dependences using integer points in a polytope. In order to parallelize a loop nest, the polytope model finds data dependences among different dynamic executions of the same statement to determine if they can be executed on different processors simultaneously [6]. Mainly, the polytope model targets counted loops that manipulate array access with affine indexes extracting the parallelism at loop-level (LLP). Note that by using this model no overhead in discovering parallelism is introduced at run time due to the polytope model is a static parallelization method. In fact, compiler transformations can be used to expose the parallelism implicit in the code of loop programs. For example, when parallelism exists in an inner nest, the compiler can exchange the loop with the outermost loop, and thereby maximizes the exploited parallelism [85].

1.2.1 Processor Arrays

Besides of using the polytope model for automatic parallelization in compilers area, it is actually used for the synthesis of hardware architectures in form of processor arrays [23], [36], [44]. The concept of processor arrays has its origin in systolic array designs. A systolic array is a locally connected parallel architecture, whose structure is well-suited to the implementation of many loop-based algorithms in scientific computation, signal and image processing, biological data analysis, etc. The term systolic array was coined by Kung and Leiserson in 1978 to describe application specific VLSI architectures that were regular, locally connected and massively parallel with simple processing elements (PEs) [78]. The systolic array design differs from the conventional von Neumann computational model in its highly pipelined computation, since once a datum item is brought out from the memory, it can be used effectively in each PE while the datum is being pumped from a PE to another PE along the array. This is especially appealing for a wide class of compute-bound computations, where multiple operations are performed on each data item in repetitive manner [76].

Processor arrays are often associated equally with systolic arrays, but the processor array term is far beyond. The computational regularity of these loop-based algorithms is reflected in the processor array, whose structure corresponds to several different potential levels of parallelism that can be exploited [64]. Also, processor arrays are concerned on memory hierarchies and control schemes needed to exploit fine-grain or coarse-grain parallelism level. Consequently a more complex design than systolic arrays is required.

Furthermore, processor arrays assume the necessity of having less amount of processing elements due to hardware constraints, a memory hierarchy for storing data in order to compute several problem sizes, and control mechanisms for synchronizing the PEs activation and their computations. In other words, a processor array needs memory hierarchy and control mechanism which provide functionality for solving size independent problems, contrary to the systolic arrays which are able to solve only a particular problem size [64].

1.2.2 Processor Array Design Methodology

The most common methodology used for automatic parallelization based on the polytope model for the synthesis of processor arrays consists in a set of steps varying from authors [23], [36], [64]. This methodology is shown in figure 1.1 and it consists of the next steps:

1. Specify the original program as a specific input representation by performing some traditional compiler transformations such as single assignment, normalization, localization or broadcast removal, sinking code, dead code elimination among others. The transformed input representation is called *source polytope*.
2. Extract information like the dependence graph, the reduced dependence graph and the computation domain from the source polytope.
3. Define a timing function, or scheduler that assigns a computation date to each task. A scheduler that maps every task to the first possible time step allowed by the dependences is called *free or greedy scheduler* [28].
4. Define an allocation function that assigns the task to PE coordinates so as to avoid conflict, *i.e.* no two tasks with the same computation time are assigned to the same PE [80].
5. Apply the scheduling and allocation functions to the source polytope in order to obtain the *target polytope* which is a space-time representation of the original loop nest.
6. Obtain from the target polytope the systolic array, also called full-size processor array, interconnection topology among PEs

For a parallel execution, the polytope that represents the original program is grouped into time slices, *i.e.* a set of points that can be executed in parallel. The entire process of systolic design in polytope model can be viewed as the application of a series of transformations to the initial specification, until the description of the final array is obtained. In order to perform these transformations it is necessary to transform the original code by a space-time function which is

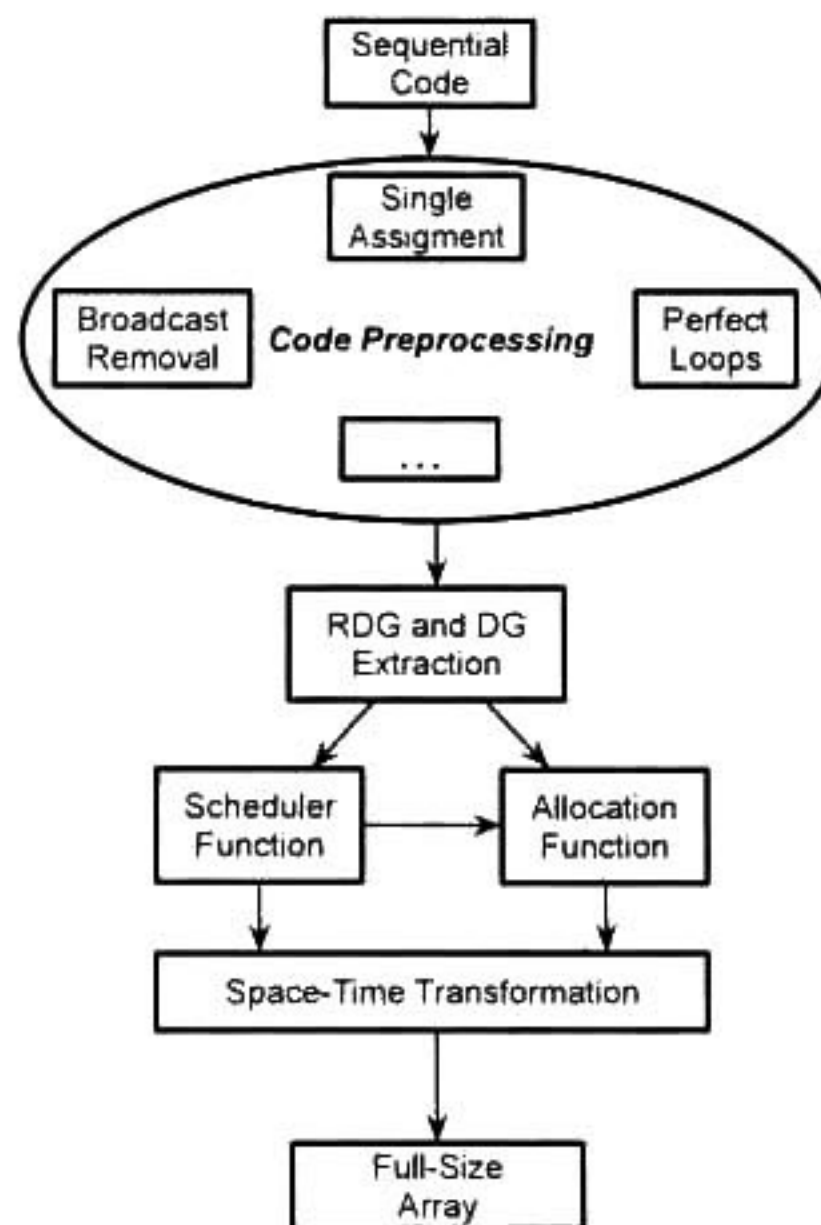


Figure 1.1: Design flow methodology for deriving systolic arrays.

represented by a unimodular matrix. However, if the methodology shown in figure 1.1 is strictly followed some problems are presented:

- The resulting full-size array would be able to solve a unique set of problem instances for a specific size problem.
- Memory management (data feeding for processor array and data recollection from processor array) becomes difficult. This is because the resulting processor array is highly parallel, and it usually requires several chunks of data in order to execute an algorithm and, at the same time, the processor array produces several results that must be extracted from it.
- The total number of PEs could be dependent on the problem size (size of the iteration space). This leads to have several processor arrays that solve the same algorithm but for different sizes.
- The resulting processor array could be non-implementable in a hardware architecture. This is because for large problem instances the processor array could require several PEs. Therefore, area resources could be easily exceeded.

- The PE percentage usage could be low because once a PE has finished its computations, it will be no longer required during the rest of the algorithmic execution

One strategy that helps to deal with these situations is adding a transformation called partitioning. This transformation has been studied in detail in compilers area [6] and it is widely used in processor array synthesis area [36], [65]. In compiler area, partitioning is used to improve data locality through better cache reuse on sequential processors, while in processor array synthesis area it is used to virtualize the full-size array into a smaller array with a fixed number of processors.

The partitioning idea is to divide the computations into chunks such that the corresponding data fits into cache [64]. Partitioning covers the iteration space of computations and divides the original iteration space in several iteration spaces that are subsets of the original one, and then mapping these subsets into a physical processor array. In fact, these subsets can be viewed as a virtualization of the logical processors whose functionality is mapped by a physical array previously given. Adding to figure 1.1 the partitioning transformation, the methodology for the processor arrays generation in the polytope model is shown in figure 1.2. The new steps consist of:

7. After space-time transformations, the target polytope space or time indexes can be partitioned. This index partitioning leads to variable array sizes at compile time, preserving the topology interconnection among PEs.
8. With the partitioned target polytope, it is generated a control scheme, a memory controller and an abstract representation for the processor array topology. The control scheme would indicate when a PE inside the processor array must be activated at certain time instant. The memory controller would be able to feed and extract data to and from the processor array.
9. It is given a functionality for all the PEs in the abstract processor array representation according to the algorithm that it is being treated.

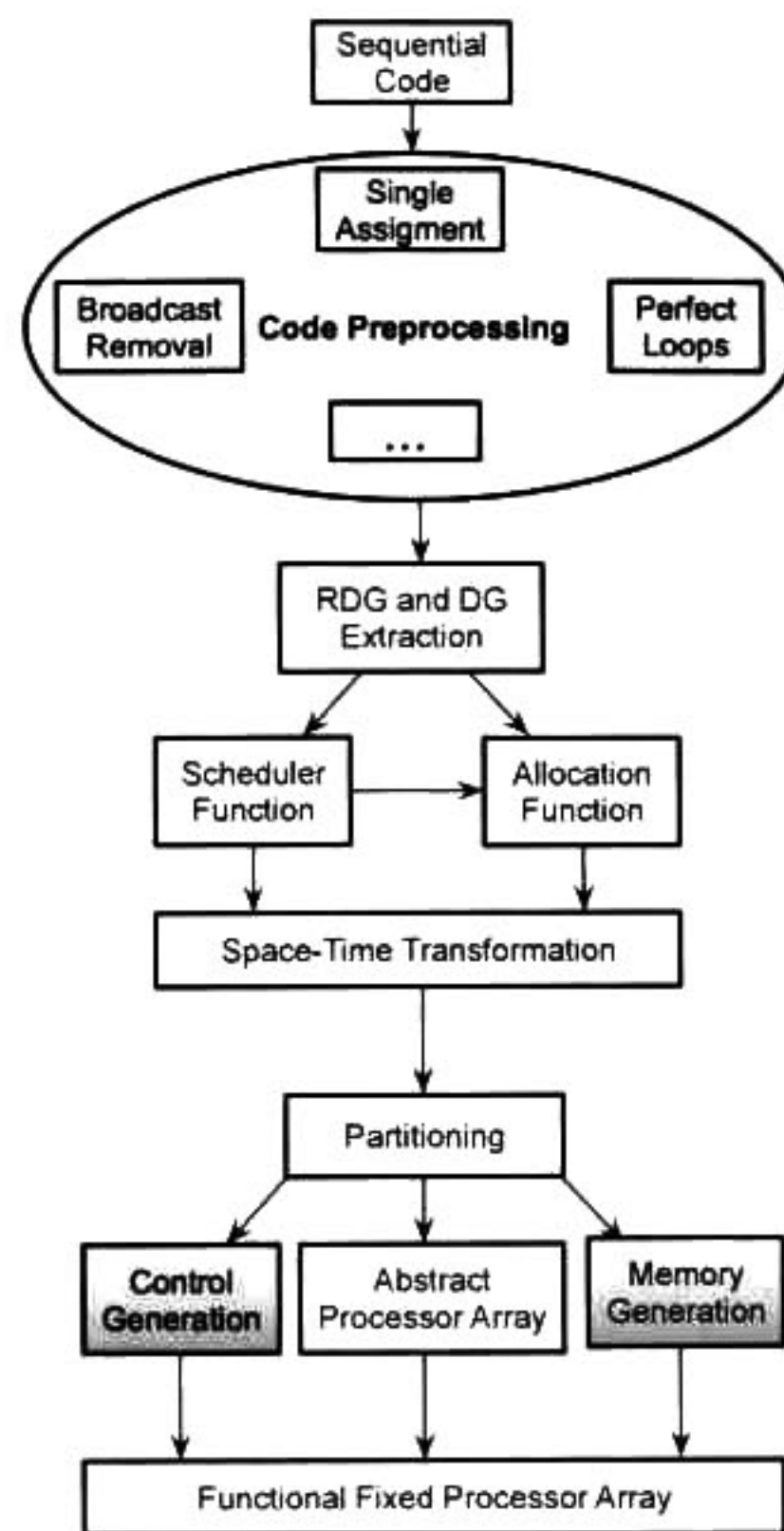


Figure 1.2: Design flow methodology followed in the polytope model with partitioning. The highlighted steps are where this research is focused.

1.3 Research Problem

1.3.1 Problem Description

By partitioning the computations, it emerges a situation that in a full-size processor array is not present: the generation of activation signals for the PEs reutilization. In a full-size processor array a PEs is activated one or more consecutively times, but once the PE stops, the PE remains inactive during the rest of the algorithm computations. On the other hand, in a processor array generated by partitioning, an inactive PE could be reused during an algorithm computation due to the virtualization process from the full-size array to the physical array. Moreover, it could happen that some computations are not mapped to all the PEs when a partition of the original computation

is being executed, and consequently not all the PEs are activated during the partition computation. Such characteristic occurs when the algorithms have non-rectangular loop bounds (concept explained in Chapter 3) like in the case of QR, Cholesky and LU decomposition algorithms. This research work pays special attention on algorithms with non-rectangular loop bounds and the control signals generation for such kind of algorithms.

Furthermore, in theory, a partitioned processor array is able to deal with several problem instances of different problem sizes. However, in practice, some automatic synthesis tools based on the polytope model, are only able to produce processor arrays (full-size or partitioned) able to deal with a unique problem size. One goal of this research work is to derive processor arrays able to deal with several problem instances of several problem sizes.

In order to build complete functional processor arrays it is needed to provide data from an external memory source, and at the same time, it is needed to extract data produced by the processor array and put them into an external memory. Usually, in works covering the processor array synthesis, it is assumed that data are available to the processor array when they are needed, but it is not specified how these data are extracted from external sources. Same ideas apply for the data computed by the processor array which must be recollected from the array. In this sense, it is needed to provide memory systems able to deal with the processor array data demands. The data feeding and data extraction from/to external memories is another focus in this research work.

Finally, this dissertation is focused on seven loop-kernel algorithms which have different characteristics: FIR filter, back and forward substitution, matrix-vector and matrix-matrix multiplication, Cholesky and LU decomposition. Algorithms, like FIR filter, matrix-vector and matrix-matrix multiplication, are widely employed by the HLS community since they are used for exemplifying new synthesis approaches [23], [36], [65], [91]. On the other hand, since this research work is mainly focused algorithms with non-rectangular loop bounds, algorithms like back and forward substitution, and Cholesky and LU decomposition are included into the set of loop-kernel algorithms.

1.3.2 Research Questions

In this dissertation, the following questions are planned to be answered:

1. What are the advantages of using the polytope model for the generation of processor arrays?
2. What control schemes are needed in order to support non-rectangular iteration spaces and to provide problem size independency?
3. What memory schemes or hierarchies are able to provide and extract data from the processor array in order to avoid bottleneck from memory?
4. What are the processor array sizes that have a PE utilization percentage above of 50% for the seven selected loop-kernel algorithms generated by using the polytope model?

1.3.3 Hypothesis

The hypothesis stated in this research work is: "The polytope model, used for the generation of processor arrays, is useful for the construction of control schemes and memory hierarchies which preserve the local connections among PEs, and respect the massively parallel implementations needed in digital signal processing systems based on matrix operations represented as nested loops"

1.3.4 Main Objective and Specific Objectives

The main goal of this research work is: "The design, implementation and validation of an architectural framework for supporting the generation of processor array hardware architectures for loop-based digital signal processing algorithms that can be modeled as a polytope" The specific objectives of this research work are:

1. To design processor array hardware architectures that support the transformations involved in the polytope model (space-time mapping, partitioning, control and memory system generation).

2. To design a control scheme for non-rectangular iteration spaces independently of scheduler used.
3. To design memory schemes able to support the parallel computations extracted by space-time transformations.
4. To provide software tools required in the processor array generation process such as scheduler generation, space-time transformations and automatic partitioning.
5. To provide a set of transformations for the proposed hardware architectures derived by using the polytope model.

1.3.5 Contributions

The list of contributions in this research work are:

1. A novel and general architectural framework able to support space-time transformations in the polytope model, consisting on a control scheme, a memory hierarchy and a processor array data-path.
2. A general control scheme able to perform the selection of operations inside the PE and the activation of PEs inside the processor array regardless of the iteration space shape.
3. A novel external memory system able to perform the data feeding to the processor array and the data extraction from the processor array.
4. A set of software tools that helps in the transformation of sequential loop algorithms, used in complex digital signal processing systems, into a processor array representation.

1.4 Summary

This chapter has stated the problem that it is addressed in this research. The main drawbacks during the systolic hardware architectures design consist of:

1. The complexity of designing these hardware architectures in a hand-made fashion.
2. The cumbersomeness and error-proneness faced during the design.
3. The limitation of the design space exploration, due to the time required during the design process is high.
4. The resulting architectures are fixed to solve a unique problem size.

The polytope model could be used for converting a nested loop program to another version that can be executed in multiple processors. Also, this model could be used for high-level synthesis of processor arrays able to solve problem instances independent of the size problem, unlike architectures based on systolic arrays which are size dependent. However, in order to derive these parallel architectures, it is needed to:

1. Derive memory hierarchies able to provide data to the processor array, and at the same time, extract data from the array.
2. Derive control schemes for the generation the control signals needed to activate, deactivate and reactivate the PEs inside the processor array, as well as the selection of operations performed during the execution of an algorithm
3. Derive mathematical expression that helps to create the memory hierarchies and control schemes.

Much of the information needed for generating these control schemes and memory hierarchies can be extracted from a polytope that represents the sequential loop program, and from the data dependencies presented in the program.

1.5 Document Organization

Chapter 2 is advocated to review the automatic parallelization works in the polytope model presented in the literature. Section 2.1 begins with an introductory overview of the origin of the polytope model, its uses for improving data locality and for parallelizing sequential code. Also, this section lists some tools needed for automatic parallelization. A review of the scheduling, allocation and partitioning, and some methodologies and automatic tools used within the context of the generation of processor arrays are reviewed in section 2.2. A discussion of the related works is presented in section 2.3.

Chapter 3 provides the mathematical background involved in the polytope model. Basic definitions (like iteration space, piecewise regular algorithms, data dependences) needed for modeling source input specifications are presented in section 3.1. Section 3.2 defines the concepts of unimodular transformations, scheduling, allocation and iteration interval which are needed for the space-time transformation. The hardware synthesis for full-size and partitioned implementations is explained in section 3.3.

The control scheme for supporting rectangular and non-rectangular iteration spaces is described in Chapter 4. The motivation behind the control scheme and a description of the functionality desired for the controller are presented in section 4.1. Each control component is described in section 4.2. Section 4.3 shows mathematical expressions for calculating the number of logic elements that the proposed control scheme requires. The generation of the control signals for a case of study with a non-rectangular iteration space is explained in section 4.4.

The memory system derived in internal and external memory is discussed in Chapter 5. The memory hierarchy required by the processor arrays is shown in section 5.1. Section 5.2 briefly explains the abstraction of the processor array internal memory, while section 5.3 describes the external memory composed by four different architectural cases according to the space-time mapping. Also, some mathematical expressions for calculating the number of logic element required by each memory

architectural case are shown in section 5.4. The derivation of external memory as case of study is shown in section 5.5.

The results obtained in this research work are presented in Chapter 6. The technological platform used for validating the control scheme and memory system is presented in section 6.1. The results for the control scheme and memory system are presented in sections 6.2 and 6.3, respectively. Besides, the implementation results of the architectural framework (including the data-path, control and memory) using two cases of study (matrix-matrix multiplication and the Cholesky decomposition algorithms) and several processor arrays, derived by following the design methodology shown in figure 1.2, are shown in section 6.4. Section 6.5 explains three different metrics for evaluating the processor arrays, and section 6.6 presents the evaluation of the implemented processor arrays as well as some design exploration for other algorithms.

Finally, Chapter 7 revisits the contributions and research questions of this research work. Also the conclusions derived from the results, and the possible directions as future work are presented.

2

Related Work

By reviewing the polytope model literature, it is possible to realize that there is a plethora of works concerning several aspects in automatic parallelization, either in compilers or processor array synthesis areas. Also, it is easy to realize that both themes are closely related. Topics like scheduling, allocation and data dependence analysis are required in both compilers and synthesis fields, while other topics like code generation and control signal generation are exclusively used on compilers and synthesis areas, respectively. Figure 2.1 shows that synthesis and compiling fields under the polytope model share an origin and some topics.

In the compilers side, several works cover aspects for automatic parallelization such as task scheduling, task allocation, data locality improvement (memory access or patterns), code generation, partitioning, etc. On the other side, in literature concerning about hardware synthesis based on the polytope model, works focusing in automatic software tools, semi-automatic methodologies or exploration works covering theoretical aspects in hardware generation can be found. Related works about the control signals generation, scheduling and allocating computations with or without resource constraints can be found too. This chapter reviews some works present within the polytope model

context, focusing on hardware generation. The first part of the chapter presents some of the works that provide an historical overview concerning to the main developments made in loop parallelization until reaching the polytope model. Afterwards, some automatic parallelization tools that have been developed in compilers area are presented. Works related to topics like scheduling, allocation and partitioning are reviewed too. The second part resumes some processor array implementations either by being generated in an automatic or semi-automatic way. Also, some main aspect concerning to partitioning, control generation and external memory are addressed. Finally, the third part of this chapter presents a literature discussion about the reviewed works.

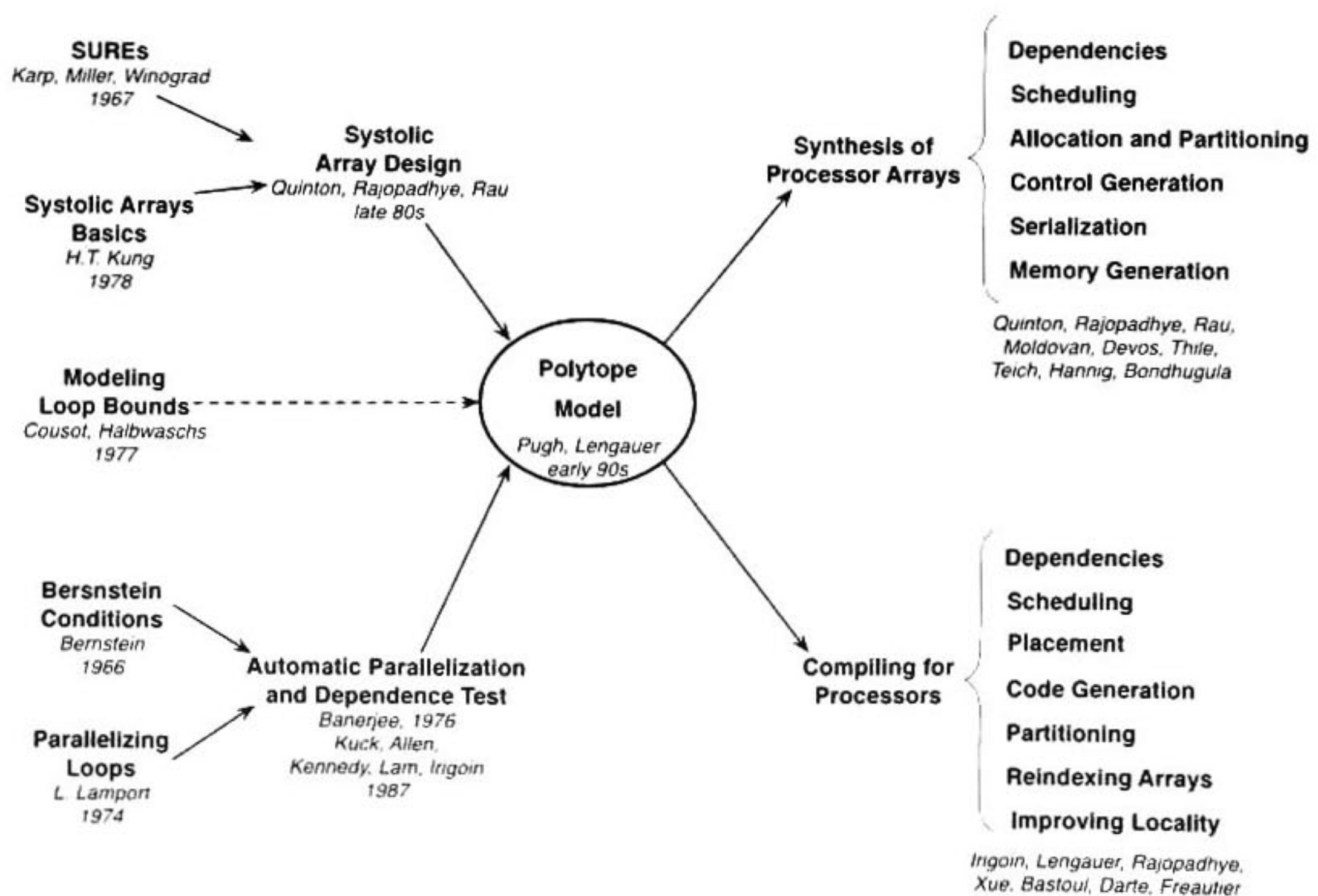


Figure 2.1: Origins and works derived from the polytope model in compilers and synthesis areas including some representative authors in each field. Image taken from [53].

2.1 Automatic Parallelization in Compilers Area

2.1.1 Origins of Automatic Parallelization

The bases for automatic parallelization in both compilers and processor arrays areas using the polytope model were laid in 1967 by Karp, Miller and Winograd in their seminal paper "The organization of Computations for Uniform Recurrence Equations" [71]. In this paper, they introduced the concept of system of uniform recurrence equations (SURE), a mathematical representation of certain types of algorithms without a notion of scheduling. The SURE models the structure of computations in a specific, repetitive, and regular process. Mainly in their paper, Karp *et al.* studied and analyzed three ideas: how the SUREs computations are organized, if the values are well defined (computability problem), and if it is possible to compute these values in a faster way (scheduling problem). To address these problems, they developed a theory based on linear programming and graph theory.

It took some time until Lamport in [77] tackled the problem of loop parallelization by developing the hyperplane method in 1974. This method parallelizes a loop nest in such way that the outermost loop is used for scanning the inner loops computations which are already parallelized. The hyperplane method was the first attempt to parallelize code, but it works well only if some conditions are satisfied. For example, this method requires that inside the loop body there is not any input/output statement neither any control instruction (if-else statement) should be placed inside of the loop body. Although it is not explicitly mentioned, Lamport tackled a special case of SUREs: the "perfectly nested loops". Essentially, a perfectly nested loop is a sequential program described as a set of loops which only contains statements inside its inner loop, *i.e.* none statement is placed among loops.

The first automatic parallelization works took place due to theoretical advances like Bernstein conditions [18], the hyperplane method [77] and the greatest common divisor test [13]. In late eighties, Allen and Kennedy presented an automatic translator which takes as an input a sequential FORTRAN code and transforms it to a High Performance FORTRAN code [9]. They developed the

concept of loop carried and loop independent dependences and proposed one of the classic algorithms used in the automatic parallelization field: the Allen-Kennedy algorithm. This algorithm uses loop distribution for reducing the amount of sequential statements within a loop. Basically, the loop distribution consists of grouping loop independent instances that can be executed in parallel and execute each group sequentially. This process is repeated for each one of the inner loops.

In [112], Lam and Wolf propose a set of algorithms and modifications to the hyperplane method in order to parallelize a nested loop by using the unimodular transformations theory. The goal of their work is to detect permutable loops, and to include in a single transformation simple loop transformations like loop interchange, loop reversal and loop skewing. By applying this unimodular transformations the Lam-Wolf algorithm can expose fine-grain, medium-grain or coarse-grain levels of parallelism. These levels of parallelism are extracted by applying the unimodular transformation on different levels of the loop nest and in the case of medium-grain by using tiling.

In [93], Pugh improves previous methods for performing loop parallelization by taking again ideas presented by Karp *et al.* Previous works to [93] aimed at the parallelization for perfectly nested loops, and they considered the statements inside of the loop body as a single block. However, the improvements proposed by Pugh include the support for non-uniform dependences and scheduling for each statement in a non-perfectly nest loop, *i.e.* without treating the set of statements as a single block. The loop parallelization is performed by using integer programming techniques for representing and evaluating dependences, and by using the Omega-Test (an algorithm designed for dependency analysis).

Although there had been reported works that applied the polytope ideas (like in [9], [71], [77], [93], and [112]), it was not until 1993 when the polytope model was named. In [80] Lengauer provides a walkthrough of the polytope model, including its use for compiling code of parallel architectures and its potentially use for synthesis of processor arrays. By summarizing ideas shown in several previous works, it is exemplified that given an optimization criterion (like the minimization of processor units or

the execution time) an optimal choice can be made automatically or with minimal human interaction. Also, Lengauer recommends the polytope model as the basis for processor array synthesis and as an ingredient in parallelizing compilers.

The contributions made by several authors within the automatic parallelization field led to the development of several tools. Later some of these tools have been used for the generation of processor arrays. The next subsection is dedicated to briefly describe these automatic parallelization tools. It does not try to be exhaustive, but it tries to show that the polytope model has been used for automatic parallelization since some years ago and it is still being as a parallelization tool in compilers, e.g. the Graphite framework in GCC general-purpose compiler [109].

2.1.2 Automatic Parallelization Tools

Paul Feautrier developed an algorithm for obtaining a set of candidates of the lexicographic minimum (or maximum) in a set of integer points belonging to a parametric convex polyhedron [49]. Later, this algorithm would originate the development of a software library named Parametric Integer Programming (PIP or PiPLib). The difference among other integer programming tools is that in PiPLib the polyhedron may be linearly dependent on integral parameters. This library has been used in the context of loop parallelization [22], [58], data flow analysis in sequential programs [50] and processor synthesis in tools like PARO [65] and Compaan/Laura [73].

Another library widely used is the Polyhedral Library (PolyLib) [100] written on C. As PIP, PolyLib is able to operate on objects like vectors, matrixes, lattices, polyhedra and unions of polyhedra. Operations with sets, vectors and matrixes over polyhedra domains, counting integer points, transformations among different representations, and other functionalities are supported too. It was first developed at IRISA, in Rennes, France, in connection with the ALPHA project. PolyLib is used for processor array synthesis tools like PARO [65] and MMAAlpha [36].

Among the tools dedicated to the automatic parallelization is the LooPo project [4] elaborated by Christian Lengauer *et. al.* The purpose of this project is to develop a prototype implementation of loop parallelization methods based on the polytope model. LooPo has been developed since 1994 in the Passau University, Germany, and it has been put under the GNU license. Some research efforts in LooPo include the capability of dealing with “while” loops, the code generation for multicore processors (like modern CPUs and GPUs) and for code generation of low level virtual machine (LLVM). During the devolvement of this project several master and doctoral dissertations have been published *e.g.* [59], [86], and [111].

The Chunky Loop Generator (CLOoG) presented in [14] by Bastoul is a tool for solving the polyhedra scanning code generation problem with non-unitary steps in a loop program. Instead of performing a traditional space-time transformation, CLOoG applies a new scanning order to the source polytope by adding indexes in certain positions in order to change the original loop order. CLOoG is the result of Bastoul doctoral dissertation [15]. In this doctoral dissertation, Bastoul proposes some modifications to previous algorithms in order to improve data locality and program performance. Theses improvements are realized by grouping data in a set of chunks that fits into the size of cache memories. Also, similar to multidimensional scheduling functions developed by Freautier, there are proposed the scattering functions for generation of scanning codes. This tool is used in CLOoG-VHDL synthesis tools [39].

PLuTo is another automatic parallelization tool available [22] which has been developed at the Ohio University. This framework is able to transform C code into OpenMP parallel code, and to perform locality optimizations for coarse-grained parallelization. Also, PLuTo is able to find good transformations for improving communication among processors in multiprocessor environment by finding permutable tilable loops. Similarly to CLOoG, PLuTo is the result of a doctoral dissertation [24] where a set of algorithms and heuristics used in the code optimization processes are described. The work developed by Uday was extended in the hardware synthesis context in [23] by developing a semi-automatic methodology for generating processor arrays implemented in FPGAs.

2.2 Automatic Generation of Processor Arrays

As it was stated in previous sections, the Karp, Miller and Winograd paper [71] laid the basics for automatic parallelization in compilers and processor arrays. In the context of processor array synthesis, the SURE is useful as a behavioral description of the processor array and as an algorithmic high-level specification. From the SURE, several kinds of loop representations like regular iterative algorithms (RIA) [95], system of affine recurrence equations (SARE), piecewise regular algorithms (PRA) [106], piecewise linear algorithms (PLA) [108], dynamic piecewise regular algorithms (DPRA) [64] or perfectly nested loop (in the compiler context) have been proposed. All these classes of algorithmic specifications define special cases of a SURE either with more restrictions or flexibility. Note that the aforementioned automatic parallelization tools take as input the specification of a sequential program and convert it into a parallel version. Depending of the methodology, the tools for processor arrays synthesis differ in the kind of input specification (sequential loop program or a SURE). Furthermore, they differ in the high-level transformations (the scheduling, allocation and partitioning methods) used for assigning a computation date and a place. This section is advocated to summarize different works concerning these high-level transformations as well as methodologies used within the polytope context. Also, some works concerning to the control signal generation and the external memory generation are reviewed too. Finally, tools based on polytope model used in hardware generation (even though some of them do not generate processor arrays) are described.

2.2.1 High-Level Transformations

Among the transformations that have been focus of research in the generation of processors arrays are the space-time mapping transformation (*i.e.* the computation of scheduling and allocation functions) and the partitioning transformation. These high-level transformations are needed in order to derive processor arrays using the polytope model, and consequently some knowledge of such transformations is required. In the following, a briefly review of related works concerning the computation of scheduling and allocation functions as well as the partitioning transformation is preseted.

2.2.1.1 Scheduling

A scheduler is a function which assigns integral time steps to all recurrence equations inside of a SURE in such way that dependences are preserved, *i.e.* it provides the execution order in which the equations could be executed in parallel. Although in the literature there are several works dealing with the scheduling problem in different contexts, this review is only focused on the polytope model. As previously mentioned, the Karp *et al.* paper [71] is one of the first works (if not the first) tackling the scheduling problem. Other work is the one developed by Feautrier, which presents in [51] an efficient algorithm for computing affine and piecewise affine schedules for dependence graphs and affine systems of recurrence equations. In an extension of his previous work Feautrier gives in [52] some evidence for the applicability of multidimensional polynomial schedulers. These methods proposed by Feautrier are implemented in MMAAlpha synthesis tool.

In [28] Darte *et al.* deals with the problem of finding optimal scheduling for uniform dependence algorithms. This paper can be considered as an extension of [71]. It introduces the concept of free or greedy scheduler which is the optimal under the criterion of performing the operations as fast as possible. Another optimal algorithm for detecting fine or medium grain parallelism in nested loops whose dependencies are described by an approximation of distance vectors is presented in [32]. The term optimal is applied in the sense that this algorithm detects the maximal number of parallel loops that can be found. As well as Feautrier in [52], Darte and Vivien in [30] and [31] introduce the concept of multidimensional schedules almost at the same time.

One of the first works concerning the scheduling problem specifically for processor array generation is presented by Thiele in [107]. In this work, a method for scheduling and allocating piecewise regular algorithms (PRA) onto processor arrays is presented based on previous scheduling approaches like in [28]. Concepts like iteration interval, linearly bounded lattice, and linear programming formulations which serve as foundations in the PARO framework are presented too. Thiele explores the problem of scheduling with unlimited and limited amount of computational resources. Since the scheduling

problem with a limited amount of computational resources is an NP-complete problem, Thiele proposes two branch-and-bound strategies for the integer linear programming formulation in order to avoid an exponential complexity.

One recent work proposed by Darte *et al.* is presented in [27]. In this work authors propose a conflict-free methodology for scheduling multidimensional partitioned arrays. This method is called input/output serialization. However, they focus partially on the co-partitioned method (a combination of LSQP and LQGS partitioning approaches) due to some tasks are assumed to be performed by a software host. In their partitioning method they use shift register elements to store and synchronize intermediate data among processors.

An overview of different scheduling techniques used for automatic generation of array processors is presented by Hannig in his doctoral dissertation [64]. Although the main contributions of his dissertation are in the field of modeling the DPRAs and in the development of scheduling techniques for incorporating local and global allocation for the DRPAs; his work also pays special attention in over-viewing scheduling, allocation and partitioning techniques for such algorithm class. Hannig's work is helpful providing an overview of the current state of art in the area of automatic generation of processors array.

2.2.1.2 Allocation

Allocating a SURE consists of assigning the equation instances (recurrences) to processors avoiding that two instances with the same execution date are assigned to be executed in the same processor. The techniques proposed for getting an allocation function have been either linear allocation functions, called projection methods, as well as non-linear allocation functions, namely grouping [29], instruction shifts and partitions. Of these functions, the last three are special cases of piecewise linear allocations functions, where different linear allocation are applied to different regions of the domain. It has been conjectured that the optimal allocation function is always piecewise linear, but this has been not proved [41].

The minimization of the number of processors is quite difficult since there are several criteria determining the cost of a hardware implementation, such as the interconnection topology, the number of registers and the necessary chip area required to implement the processors. In [54], Fimmel and Merker, obtain an allocation function using as objective criteria the chip area. First authors propose an algorithm leading to a small number of processors. Then the algorithm is extended to include the chip area necessary to implement the processor in silicon. This approach tends to be optimal if the index space can be well approximated by parallelepipeds, but even if not, the approach leads to practical results.

Tayou *et al.* in [41] tackles the problem of deriving space-optimal processor arrays from a directed graph given an affine scheduler. They introduce an automatic allocation method based on a preprocessing by reindexing the index space. This reindexing leads a new index domain which allows the derivation of processor arrays using projection vectors. This method transforms an initial dependence graph into a new one that enables the projection method to minimize the number of processors along a number of directions. In addition, this preprocessing allows to improve the potential parallelism by projection of the initial domain. Authors claim that this method could be used within the MMAAlpha context.

2.2.1.3 Partitioning

Partitioning consists of decomposing the SURE computations into sub-blocks of fixed-size that matches the target architecture characteristics for improving data locality or exploiting different parallelism granularities. Partitioning is basically motivated by the simple fact that often, the size of a computational problem exceeds the silicon resources (in form of functional or memory units) of a fixed VLSI structure, like ASICs or FPGAs. In the literature, there are different partitioning approaches derived by different interpretation of the partitioning transformation. The local serial global parallel (LSGP) approach consists of computing all partitions in parallel using a sequential processing element for their execution. The local parallel global serial (LPGS) consists of executing all computations inside of a partition in parallel and traversing sequentially each partition.

One of the first partitioning method called super-node partitioning is presented in [68]. This method is based on a finding a matrix transformation able to schedule the execution order of each partition and the execution order of the iteration instances inside of the partitions. Although it is not proposed an algorithm for finding such matrix, it is shown how some imposed tile conditions are translated to build the matrix.

Moldovan and Fortes [88] present a technique for partitioning and mapping algorithms into VLSI arrays. The partitioning is done such that no extra processor complexity is required. The approach used to partition the problem is dividing the algorithm index space into bands and mapping these bands into the processor space. Intermediate data among bands is stored into first-in-first-out (FIFO) memories. The partitioning and mapping technique developed throughout the paper consist of heuristically finding a time-minimal function, generating a set of transformations, selecting a valid transformation among the previous transformations, which requires the least number of bands, mapping the algorithm indexes to processors and selecting a policy for scheduling the bands.

A partitioning approach able to divide affine dependence algorithms is presented in [105] by Teich and Thiele. They propose the concept of partial localization for embedding intra-tile or inter-tile data dependencies onto new variables. The major advantages of their work are avoiding the unnecessary copy operations created by localization prior partitioning, and avoiding the freedom restrictions of scheduling in early design phases of processor arrays. They suggest to do partitioning as a first step in the automatic parallelization methodology using their partitioning approach.

Dutta *et al.* in [43] and [47] propose a scheduling methodology using partitioning as allocation function. Such work is an example of how the searching of scheduling functions and partitioning transformation used as allocation function could be gathered in an holistic approach. The scheduling methodology proposed by Dutta *et al.* allows the derivation of global and local controllers mapped as synchronous counters which generate the time steps as specified by the space-time transformation. Another methodology for partitioning perfectly nested loops programs with consideration of both

affine and uniform dependences is presented also by Dutta *et al.* in [48]. In this last paper a study of the additional control overhead introduced by the partitioning approach is presented. Besides, an extension of their previous works is reported in [46], where a design methodology for reducing the hardware cost of the global controller and memory address generators by avoiding costly operations (like complex multiplication and division) is also presented. These aforementioned ideas developed by Dutta are implemented in the PARO tool [65].

2.2.2 Methodologies for Hardware Synthesis

One of the first works providing a formal definition for processor array is presented in [95] by Rao. In this work, he also provides a formal definition including the properties of regularity, temporal and spacial locality, and pipelined operation. Besides, he proposes the use of RIA as an algorithmic input for the design of the arrays and provides some procedures for deriving processor array implementations.

Lee and Kedem in [79] present a systematic method for synthesizing linear arrays implementations from nested loops algorithms. This method uses linear functions for transforming the original sequential program of p -nested loops into a form suitable for parallel execution onto a linear array. They propose using some criteria (like minimal execution time or smallest number of processors) in order to derive feasible mapping that satisfies both sequential and parallel programs. The method can be used to derive systolic implementations given the p -nested loop algorithm, or given a processing element (with specific data links, shift registers and other hardware components).

A design methodology for the systematic synthesis of processor array architectures is proposed by Teich in [102]. According to the author, such methodology is implemented in a synthesizer, and is based on three transformations: localization, control generation and hardware matching. These three transformations are applied to a sequential program until it meets the specification and requirements of a target architecture. The final representation can be interpreted as a processor array specification.

Uday *et al.* in [23] present a framework for mapping perfectly nested loops into processor arrays implemented in FPGAs. In their framework, for control signal generation, they do not use linear schedulers but multidimensional ones. Besides, for contraction of the control path they associate each time dimension of the multidimensional scheduler to two global controllers and each controller maintains a time counter, and streams the activation signals from a particular corner of the processor array with a certain delay. Partitioning is supported by using multidimensional scheduler in a LPGS approach, but it lacks of information about any memory support (like FIFO elements) for the partitioning approach used. Moreover, information about an external memory module in charge of providing data is not mentioned in this work. This framework is proved by implementing the matrix-matrix multiplication algorithm (MatMul).

One recent semi-automatic methodology for deriving sequential hardware with a high parallelism level is proposed in [8] by Alias *et al.* This methodology relies on FloPoCo an open-source tool for FPGA floating point arithmetic-core generation [33]. As input specification it takes a C program and the amount of pipeline stages desired, producing VHDL code as result. The input C program is changed by using some transformation within the polytope context like the loop blocking transformation. Although in their work Alias *et al.* obtain a scheduler function, space-time transformations are no performed, thus the scheduling process gives only the computation order considering the floating point operational latencies in order to avoid pipeline bubbles. In other words, this methodology improves data locality but it does not expose the loop level parallelism. This last idea is enforced by the fact that they derive highly-pipelined sequential processor. However, Alias *et al.* show that, from this sequential implementation, obtaining a parallel version by replicating the sequential processor is possible. Although it is not explicitly mentioned, they describe similar partitioning ideas already presented in [36] and [64] like affine and linear scheduling. The control generated by their semi-automatic methodology is composed by a single finite state machine (FSM) that captures the whole loop nest execution sequence. Also, this control generates external memory addresses for where the input/output data is stored.

2.2.3 Control Generation

In order to derive processor arrays, the generation of control signals in charge of activating the processing elements inside of the array is needed. Also, the selection of the operations performed during the algorithm execution is required. Although there are several works for deriving the control structures, these control modules are able to produce the control signals for a specific problem size. One of the first attempts for formalize the control signal generation was presented by Xue and Lengauer in [121]. The basic idea consist of distinguish between different types of computations specified by two kinds of SUREs. One SURE specifies data flow while the second one specifies control flow. Authors restrict themselves to shown the process of generating the control signals, and they do not translate their ideas to hardware structures.

In [61] Guillou *et al.* propose the use of multidimensional schedulers for generating processor arrays within the MMAAlpha context. The main idea for generating the control signals is to scan convex union of all time domains and to provide activation signal for each variable. One way for achieving this scanning is by using a multidimensional counter, which assembles a set of counters connected in a cascade fashion producing the time steps for each scheduler dimension. However, authors propose the implementation of such multidimensional counter like FSMs.

Bednara *et al.* [17] propose a scheme for controlling the bounds of the space-time transformed index space that introduces only two additional signals that indicates the first and the last time step that a PE will have to execute an operation. These two signals are propagated locally to neighbor PEs. Some of this ideas were used for the controller derived by PARO synthesis tool [65] like the combination of global and control facilities. Also, within the PARO context, in [43] Dutta develops a general methodology for control generation when different different partitioning techniques (LSGP, LPGS, co-partitioning) are used. This methodology is based on generating specialized counters for scanning the index space given a space-time mapping and combining global and local control facilities for decoding the scanning codes.

Besides of presenting a framework for the generation of processor arrays, Uday *et al.* in [23] briefly describe a control scheme for generating the control signals obtained from multidimensional schedulers. For deriving the control, they associate each time dimension of the multidimensional scheduler to two global controllers. Each controller maintains a time counter and streams activation signals from a particular corner of the processor array with certain delay.

2.2.4 External Memory

One open problem that has been tried to be tackled is the external memory management needed in order to feed data into and to extract data from the processor array. In [56] Girbal *et al.* show how to design a memory interface for multi-purpose accelerators combining a direct memory access (DMA) ordering predictability. In this work, it is explained the synergy between a special unit called stream and a proposed Stream Table (especial memory element) module which captures most short-distance temporal reuses and increases the apparent bandwidth. They claim that such stream and Stream Table form a generic template for efficiently interfacing multi-purpose loop-based accelerators with the memory. However such assumption is left as a future work and not proof is provided.

In his master thesis Dutta [43], besides of the control generation methodology, briefly discussed two possible memory address generation schemes for external memories. Also, some mathematical expressions for estimating the amount of external and internal memory are presented when the co-partitioning approach is used. This work is within the matrix theory developed in the PARO framework for deriving processor arrays. Also, within the PARO framework, Hannig *et al.* in [65] describes a scheme for the memory controller synthesis based on the use of counters, decoders, address generators, and glue logic for interfacing the processor array to other components integrated in a system-on-a-chip (SoC) environment. However, the input and output data communications are only proposed to be done either by functional simulation, by DMA, or by software running on a host processor.

Plesco in [91] presents a hand-made solution for interfacing an external memory with a processor array generated by MMAAlpha tool as case of study. It is taken the MatMul algorithm for complex numbers using a processor array of 4×4 PEs. Plesco states that besides of the difficulty of designing a memory architecture, it is needed to have some knowledge of the processor array data access patterns and enable an internal data reuse in order to obtain a good performance. Also, within the MMAAlpha context, Derrien in [34] deals with memory aspects involved in the generation processor array. In his work, Derrien proposes a methodology to derive a set of conflict free schedule for input/output data pipelines along the processor array boundaries. These pipelined data are expressed as constraints of the pipeline directions, and of the number of registers between processors. In this work, he used the multidimensional scheduler and the LSPG approach, taking as case of study the MatMul algorithm. A common factor in these two works is the introduction of latency penalties or the implementation of input/output serialization for external data.

A framework for hardware synthesis of memory address sequencers for synchronous dynamic random-access memories (SDRAM) is presented by Bayliss and Constantinides in [16]. This work is one of the first works using the polytope model for deriving memory interfaces for hardware accelerators. More precisely, authors focused on applying traditional loop transformations (like loop interchange) on an augmented polytope representation. This new presentation includes the number of SDRAM row words and the size of the SDRAM burst. Authors restrict themselves to the use of unimodular transformations in order to avoid holes in the transformed iteration space. This framework could be enclosed within the VHDL-CLoG tool since authors use this tool for the control generation of the memory address sequencers [14], and consequently the memory sequencers generated are only for high pipelined processors but not for processor arrays.

The generation of optimized remote accesses for loop accelerators generated by Altera C2H tool is proposed in [7] by Alias *et. al.* In this work, loop tiling is applied on nested loop programs in order to increase the granularity of DDR memory communications. Repetitive remote memory accesses are avoided by improving data reuse among tiles using local memories.

2.2.5 Hardware Synthesis Tools

Although in the literature there are several works concerning the automatic generation of hardware, only few of them are in the scope of the polytope model. This subsection focuses on works, within the polytope context, that automatic or semi-automatically generate hardware putting special attention to those works that generates processor arrays.

2.2.5.1 MMAAlpha

MMAAlpha is a programming environment originally developed by Patrice Quinton *et al.* from an IRISA research team [36], [61]. It is based on the Alpha language, and built as a set of Mathematica toolboxes [94]. This language was originally developed as a functional data-parallel language for an algorithmic description based on SUREs and targeted for the systolic architectures [110]. This programming environment transforms an Alpha program to a VHDL specification, and to a C code for simulation purpose. This VHDL code describes a processor array, thus MMAAlpha is able to generate processor arrays from an algorithmic input.

The MMAAlpha design flow consists of a front-end and a back-end. The front-end is in charge of generating an intermediate program in Alpha language with space-time notions (virtual architecture), while the back-end purpose is the generation of a VHDL code from the space-time Alpha program. Figure 2.2 shows the design flow followed by MMAAlpha. First, the Alpha program is syntactically and semantically analyzed in order to convert the program to an internal representation (a polytope). The next step is to obtain a scheduling function either linear [51] or multidimensional [52]. The one-dimensional scheduling technique provides an execution date for each one of the variables presented in the Alpha program by using an affine scheduling function, whereas the multidimensional scheduling is used for providing an execution date considering a temporal succession. In the MMAAlpha environment, localization, *i.e.* removing long connections and putting all variables with the same dimensionality, is optional and is placed after obtaining a scheduler function. After localization the space-time mapping is applied, providing time and space notions to the indexes of each variable in

the original Alpha program. If localization was not previously applied, then the variable mapped to a signal after space-time mapping is broadcasted to other PEs.

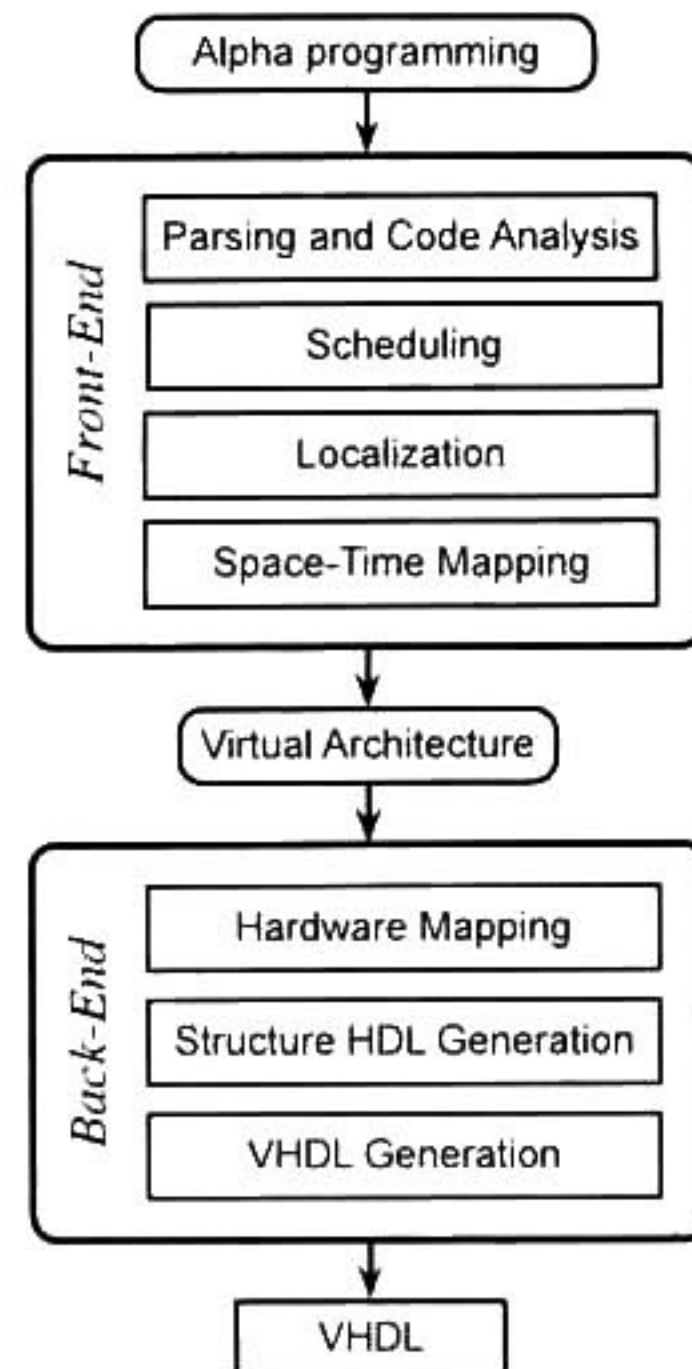


Figure 2.2: Design flow of the MMAAlpha programming environment. Image taken from [36].

The virtual architecture is only an operational parallel description of the initial program, in fact this virtual architecture is still a subset of Alpha program (called Alpha0). In order to generate an architectural description, it is needed to obtain the PE internal description and to generate the control signals at proper times. The hardware-mapping stage is in charge of performing such actions. The PE internal description is obtained from the virtual architecture. The control signal generation consists of replacing linear equalities derived from the space-time mapping by the propagation of simple control signals to other PEs. The structured HDL generation consists of the re-use identical processing elements present in Alpha0 program. This re-use is accomplished by grouping PEs that share the same behavior and region in the processor space, giving as result a modular AlphaD program. The final step is the generation of the VHDL code that is performed by a direct translation of the AlphaD modules to VHDL modules.

In case of the processor array is not physically suitable of being implanted, MMAAlpha uses the aforementioned co-partitioning method proposed by Darte *et al.* in [27]. In the context of MMAAlpha, this means to include the functionality of σ virtual processors into a single physical processor, and adding at least σ delay units in the internal PEs data-path for each variable of each physical processor. Finally, the unique mention about external memory management is shown in [27] where they assume that this task is done by a software host.

MMAAlpha has been targeted for solving linear equation systems using the Jacobi method [87], string matching [36], and computing the score between a hidden Markov model and an observed sequence [35]. Also traditional algorithms like the FIR filter [94], and MatMul algorithm [61] have been targeted in this tool. One characteristic that these algorithms share is that their loop bounds form rectangular shapes. In fact, there are not shown cases where the design methodology followed by this automatic tool has been used for generating processor arrays (full-size or partitioned) for algorithms whose loop bounds form non-rectangular shapes (e.g. back substitution, Cholesky, QR and LU decompositions algorithms). Moreover, although all processor arrays generated in a full-size fashion are size-dependent, a processor arrays synthesized by MMAAlpha using partitioning transformation are unable to solve several problem size instances. If the problem size for which the processor array was derived changes, it will be needed to regenerate the processor array controller, and consequently to regenerate the whole array for the new problem size.

2.2.5.2 PARO

Another automatic tool for processor array generation is the PARO synthesis tool [65]. This tool has been developed mainly by Teich, Hannig, Dutta and other collaborators in Erlangen-Nuremberg University in Germany. PARO is able to map computational intensive nested loop programs into parallel processor arrays architectures that are modeled in VHDL code. Recent efforts have focused on the generation of assembly code for coarse-grained processor array architectures [74] for the invasive computing paradigm [103]. As same as MMAAlpha, PARO uses a special input specification language called PAULA. This language models DPRAs, which are a generalization of the SUREs,

capable of handle run-time dependent conditions. PARO implements lattice functions, and it uses external libraries (like PolyLib [100]) for performing operations over polyhedral objects and several linear programming solvers (like GLPK [1], CPLEX [2] and PIP [49]) required during the scheduling process.

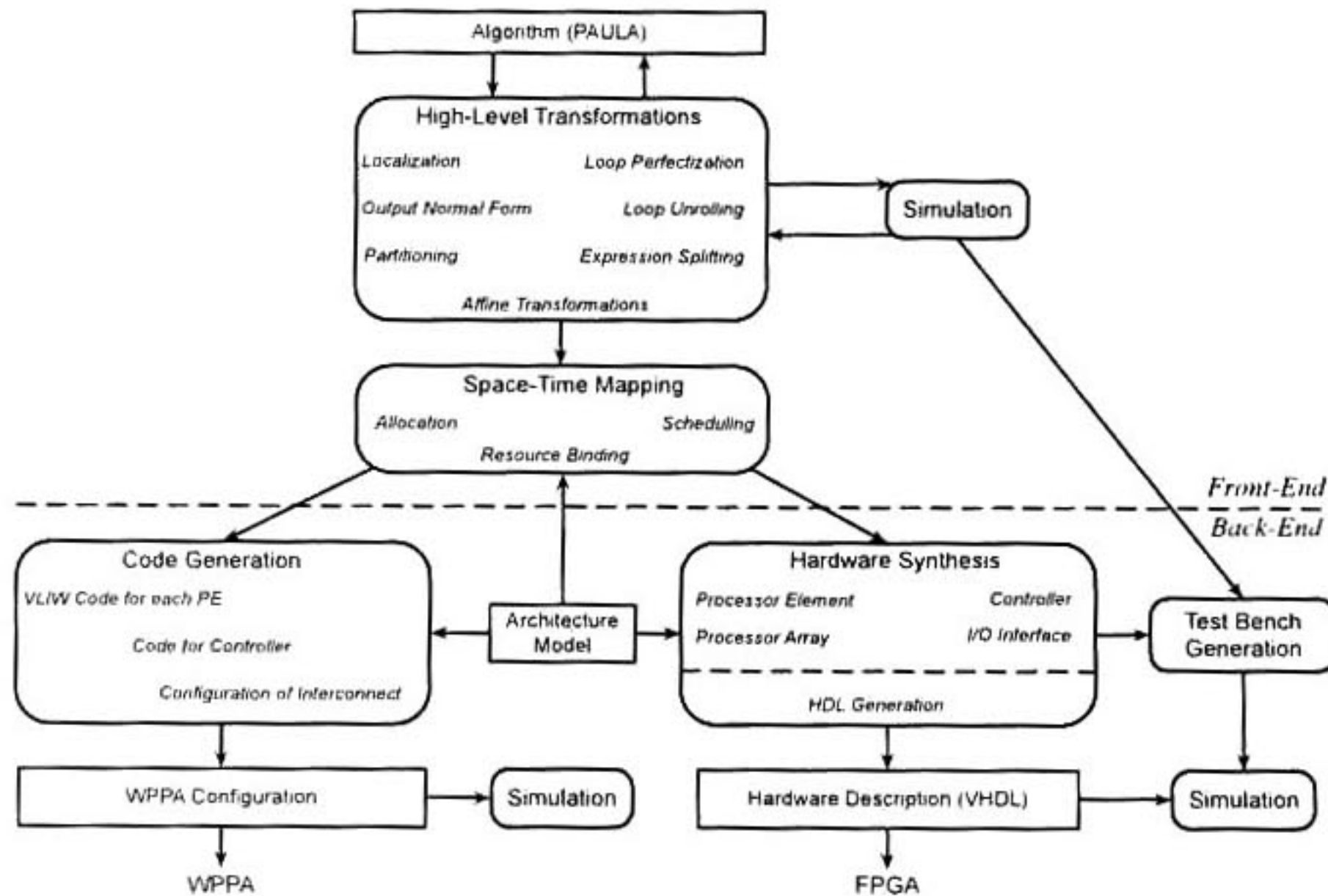


Figure 2.3: Design flow of the PARO framework. Image taken from [64].

The PARO design flow consists of a front-end and a back-end shown in figure figure 2.3. The next design flow description is focused on the generation of VHDL code, since the PARO back-end is able to generate VHDL code or code for coarse-grained processor array architectures. The PARO design flow starts with a PAULA program which describes a specific algorithmic functionality. Transformations like reduction, dead code elimination, serialization and loop unrolling could be applied automatically by PARO tool, whereas transformation like localization and loop perfectization are performed in a semi-automatic way since they require user intervention in order to reduce code redundancy introduced by PARO. Serialization and loop unrolling are helpful when processor arrays for image processing filters are derived. Basically, serialization consists of imposing an input or output sequence for data that come in/out from/to the processor array, while loop unrolling consists of expanding the loop kernel in a factor of n by the coping of $n - 1$ consecutive iterations. With

the purpose of ensuring the correctness of the input specification before performing the space-time mapping, PARO provides a simulator that reproduces the behavior of the PAULA program. Once it has been ensured the input program functionality, PARO is able to extract the information needed to perform the space-time transformation in form of a reduced dependence graph (RDG). This graph is capable of representing the data dependences among statements inside of a loop nest, and if these dependences are taken into account during the scheduling process, a finer parallelism grain could be exploited.

The allocation methods supported by PARO are the projection, LSGP partitioning, LSGP partitioning and co-partitioning. In PARO, the user is responsible for selecting any of these allocation methods in order to obtain a scheduling function. Once the scheduler has been computed, PARO performs the space-time transformation over the PAULA program. The binding transformation is performed by assigning each node of the RDG to hardware functional units capable of performing the node functionality. The processor array synthesis consists of three steps: synthesis of the processor elements, generation of the control structure and derivation of the interconnection topology [64]. The hardware synthesis generates a register transfer level (RTL) description which is later translated and optimized into VHDL code. Also, PARO is able to automatically generate VHDL test-benches in order to perform the verification of the processor array modeled in VHDL.

The controller generated by PARO uses combined global and local control facilities. All control signals that are common for all processor elements are generated by global control units and propagated through the array, whereas local control is only necessary for signals that differ among the processor types [65]. One of the main assumptions in PARO framework is that external data needed to feed the processor is inserted as it is needed, without specifying any operational module which completes such task. Although, the mathematical expressions developed by Dutta in [43] are used for estimating the amount of internal memory (PE internal memory and FIFO memory), the address generator for external memory presented seems to be left behind of the PARO project since there is not information about any implementation concerning the external memory.

Cases of study presented during the development of PARO tool include MatMul algorithm [43], FIR filter [96], discrete cosine transform (DCT), and images filters like edge detection, bilateral [44] and gaussian filters [65]. MatMul, FIR and edge detection algorithms are cases of study that show how PARO could be used for generating partitioned arrays [65]. As the same case of MMAAlpha, algorithms targeted in PARO tool has the rectangular loop bound shape characteristic. Although during the development of PARO, the mathematical theory for deriving processor arrays from algorithms with non-rectangular loop bound shapes has been developed, cases of study where partitioning transformations are applied to algorithms whose loop bounds are not rectangular (like QR, Cholesky and LU decomposition algorithms) have been not shown. Moreover, processor arrays derived by PARO are size dependent due to the assumption of fixed tile sizes [44], and they do not use complex hardware operations used in matrix decompositional algorithm.

Finally, a special mention is required for a kind of hardware architecture that has been developed for supporting polytope concepts within the PARO framework. In [74], Kissler *et al.* present a class of massively parallel embedded processor architectures called weakly programmable processor arrays (WPPAs). The basic building blocks in this architecture are so-called weakly programmable processing elements (WPPEs). These processing elements are called weakly programmable because the limited amount of instruction memory inside of them, and because of the optimized control reduces the overhead. The instruction set of a WPPE is also kept small and specific to instructions commonly needed in digital signal processing domain. In fact, each WPPE can be parameterized at compile time to contain a predefined number of functional units or user-defined functional units.

2.2.5.3 PICO-NPA

PICO-NPA (Program-In Chip-Out and Non-Programmable Accelerator) is a system to automatically synthesize hardware co-processors from perfectly nested loop C programs [72]. The project was developed in the HP Palo Alto research laboratory and now it is being commercialized by Synopsys with the name of PICO Express [3]. This tool generates synthesizable HDL code that defines several RTL algorithmic specifications and HDL test-bench codes for each RTL specification.

The PICO-NPA system has several steps for transforming a loop nest into an RTL architecture (figure 2.4). The first step consists of extracting the array accesses and the data dependences. With this information, the loop iterations are mapped to a fixed amount of processors specified by the user via a spacewalker, while the iterations are mapped to clock cycles. In other words, notion of allocation and scheduling are applied to the input loop program. At this stage information like data-paths and iteration intervals among computations are derived by PICO-NPA. The third stage is to change the loop computation order by using transformations like loop tiling and space-time mapping. In this sense, the partitioning approach used is LSGP. A fourth stage is based on applying traditional loop data dependence analysis and on selecting the tile shapes. Finally, in a fifth stage, the computations are allocated, bound and scheduled with the obtained iteration interval. It is important to emphasize that PICO-NPA does not generate one RTL specification, but several RTL specifications according to area and timing constraints specified by the user. The determination of the best possible solutions is obtained by using the Pareto optimal frontier, leaving to the user the final election among a set of possible implementations each one with different degree of parallelism.

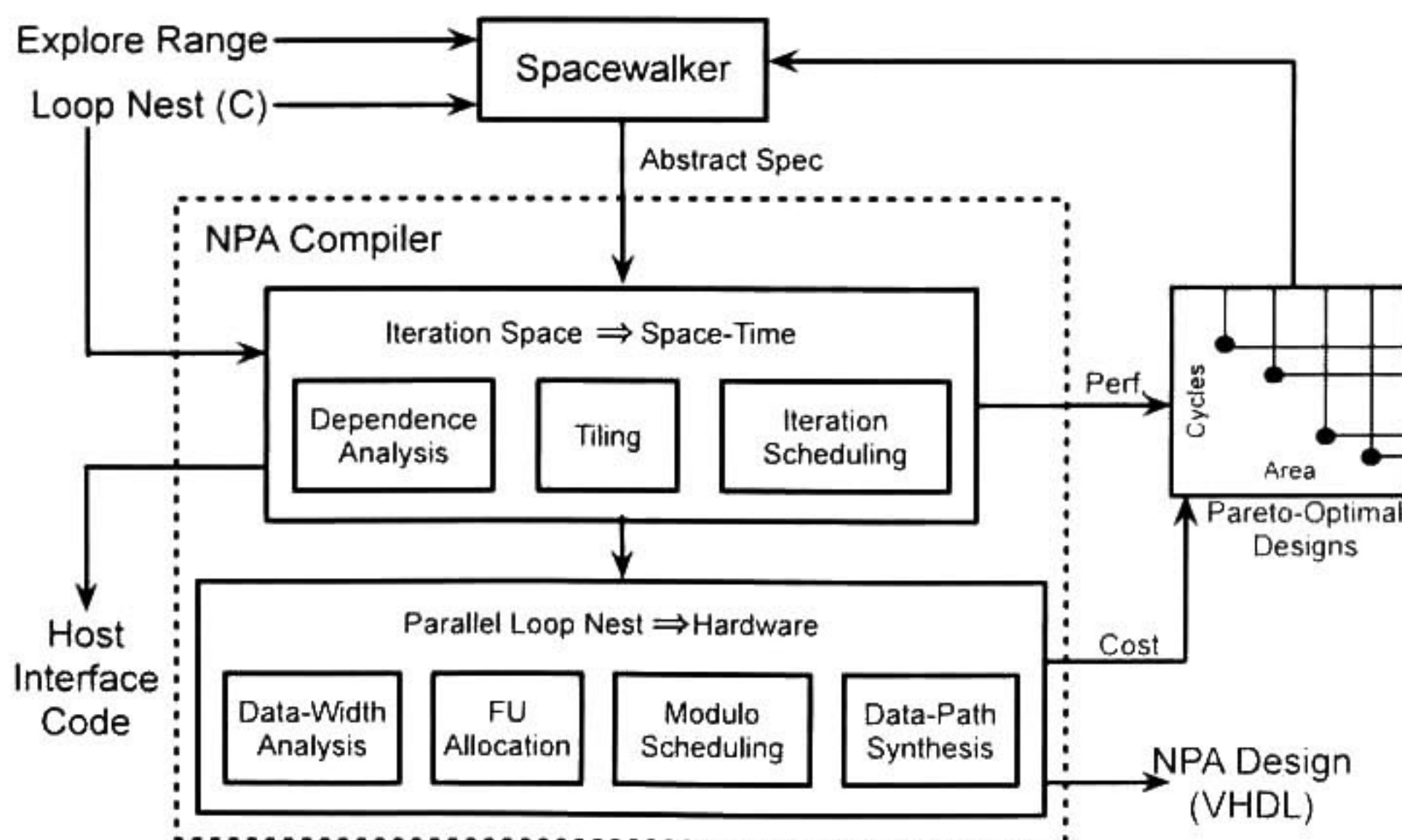


Figure 2.4: PICO-NPA design system components. Image taken from [72].

PICO is able to exploit four different levels of parallelism: loop, instruction, inter-task and intra-task. Besides, PICO is related to the automatic parallelization in the sense that it uses loop unrolling coupled with software pipelining, loop tiling, techniques for load-store elimination, and scheduling methods used for processor array synthesis. Basically, the PICO target architecture is built on three hierarchical levels. The first one consists of simple processing elements containing arithmetical units. Each one of these PEs communicates to other PEs using a data storage structure called ShiftQ. The second level consists of a set of PE locally interconnected called processing array (PA). This PA incorporates local memories in order to reduce the bandwidth needed in external memory accesses. The third is made of a set of PAs connected by FIFO memories called pipeline of processing arrays (PPA). At this final level, a controller is in charge of orchestrating the operations of all PAs, while an interface is used to communicate with a host computer. Data transfers from CPU to the PPA are realized by specialized hardware units. It should be noted that a PA is similar to the processor arrays in the sense that it is composed by simple processing elements interconnected between them in a regular and local fashion. Unfortunately, since PICO is a proprietary technology, several details are omitted.

2.2.5.4 CLooG-VHDL

CLooG-VHDL is a back-end of the chunky loop generator (CLooG) developed by Devos in Leiden University, Netherlands [39]. CLooG-VHDL uses some polytope and automatic parallelization concepts in order to derive hardware accelerators which are not necessarily based on processor arrays. The main purpose of this back-end is the generation of the controllers needed for enabling the action of statements inside of the loop program. The generation of data-path and its controllers is not fully implemented and some parts are left to the user. Due to CLooG-VHDL uses the Bastoul's chunky loop generator [14] it uses C code as algorithmic input specification. Figure 2.5 shows the extension of CLooG to CLooG-VHDL for hardware generation.

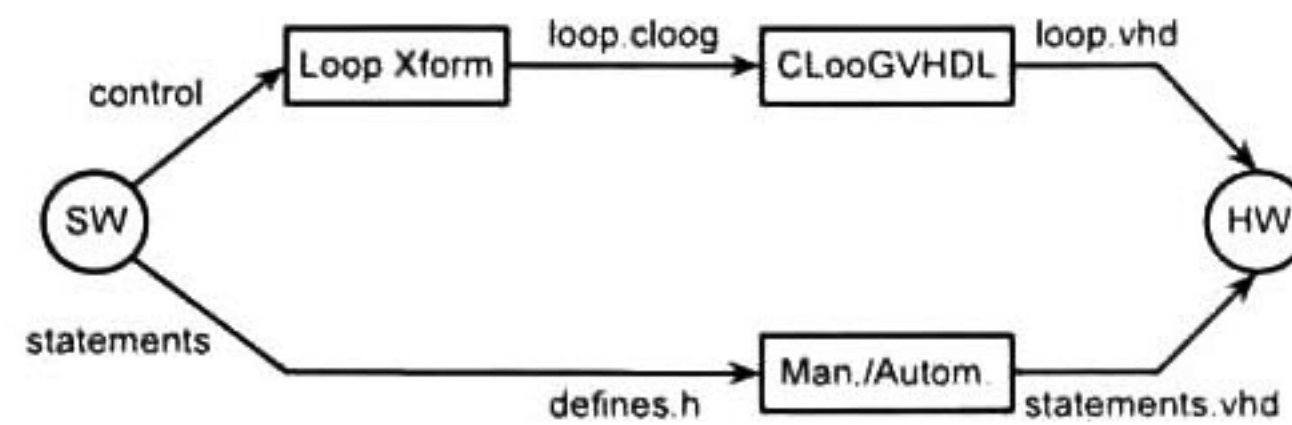


Figure 2.5: Design flow of CLoog-VHDL back-end. Image taken from [39].

The controller generated by this semi-automatic back-end is constructed by using abstract-syntax trees (ASTs) and multidimensional scheduler functions. Basically, the idea is creating separate hardware for different loop statements if possible. The multidimensional scheduler allows to verify which statement and loops instances can be executed in parallel. The control generated by CLoog-VHDL is composed by FSMs that model the abstract-syntax trees and the multidimensional scheduler function. Each FSM corresponds to one dimension of the scheduler vector. Some FSMs calculate the loop bounds and their stride in function of some parameters and the surrounding loops, whereas other FSMs are dedicated to enumerate the statements inside of a loop body. Together, these FSMs generate the activation signals for each loop statement presented in a sequential program. Finally, the construction of data-path for these monolithic processors is partially left to the user.

2.2.5.5 *Compaan/Laura*

Compaan/Laura is a tool chain developed in the University of Leiden, Netherlands [123], for mapping nested loop applications written in Matlab into VHDL. This tool chain is composed by two different tools: *Compaan* which is responsible for translating the sequential nested loop program into Kahn process networks (KPN) and *Laura* which takes as input a KPN specification and generates the VHDL code for such network. Early results of this tool [123] generate a mono-processor based on the statements inside of the loop nest, parallelizing the operations inside of the loops instead of searching a new execution order of the loop instances. In other words, *Compaan/Laura* focuses on improving the performance by applying pipelining techniques. However in [37], the *Compaan/Laura* team uses the polytope (PolyLib library) for generating the control needed to manage the FIFO elements that interconnect different monoproductors derived by their tool chain.

2.2.5.6 Other Works

Within the context of the Stanford University Intermediate Format (SUIF) compiler system, it is the Riverside Optimizing Compiler for Configurable Computing (ROCCC) which is able to synthesize systolic arrays dependent of the problem size. ROCCC is proposed by Betul *et al.* in [26]. ROCCC consists of a front-end in charge of applying loop transformations like partial and full loop unrolling, loop peeling, loop tiling, strip mining and loop fusion. A back-end is responsible for VHDL code generation. Also, within the SUIF context is the DEFACTO tool for hardware synthesis proposed by Diniz *et. al* in [40]. DEFACTO performs compiling transformations like constant propagation, dead code elimination, unroll and jam, loop permutation, and loop tiling in order to expose the parallelism and the data reuse from a C-like input program required for producing VHDL behavioral code.

The single assignment C (SA-C) compiler is proposed by Najjar *et. al.* This compiler takes a single assignment C code in order to derive VHDL code targeted to FPGA platforms. The compiler uses data dependence and control flow graphs in order to represent hierarchically the program structure. Later, these graphs are translated to hardware structures by using compiler transformations like constant folding, operator strength reduction, dead code elimination, loop unrolling, strip mining and temporal common subexpression elimination. SA-C cases of study are limited to digital image processing domain.

2.3 Literature Discussion

There are several works on automatic parallelization for compilers, using dependence analysis, loop transformations and other formalisms. Problems like scheduling, allocation and partitioning are well studied on compilers area for improving data locality [6]. On the processor array synthesis side, some techniques applied on compilers area are used; other ones are taken from this area and modified for deriving processor arrays; meanwhile other techniques are exclusively developed for high-level synthesis. In this section a discussion of the previously reviewed works is presented.

	PARO [65]	CLoogVHDL [39]	MMAAlpha [36]	Compaan-Laura [37]	PICO-NPA [72]	This Work
Input Format	DPRA	C	SURE	Matlab	C	PRA
Architecture Type	Processor Array	Processor	Processor Array	Processor	Hybrid	Processor Array
Scheduling	Several Methods	-	Multidimensional	-	Hierarchical	Linear
Partitioning Tech.	Tiling Matrixes	No	Multidimensional	-	Blocking	Strip-Mining
Controller Style	Counters	FSM	FSM	Counters	Unclear	Counters
Memory Assumption	DCN	None	DCN	-	Done by SW	DCN
Parametric Support	No	No	No	-	No	Yes
Non-Rectangular	Partially	No	No	-	-	Yes
Complex Operations	No	No	No	-	No	Yes
HDL	Yes	Yes	Yes	Yes	Yes	Yes

Table 2.1: Comparison of characteristic among automatic parallelization tools and this research work. DCN is short for "Data Come as it is Needed"

Table 2.1 summarizes some characteristics among five different works previously described. These characteristics include the kind of hardware produced by the tool, the scheduling and partitioning method supported, implementation style of the controller, external memory assumption, support for floating point operations and rectangular loop bound shapes, among other characteristics. Although CLoogVHDL uses the multidimensional scheduler technique, it is not included in table 2.1 due to the scheduler is only used for improving data locality but not for generating processor arrays. In memory management row, the DCN term is short for "Data Come as it is Needed"

Table 2.2 shows a detailed comparison among the three most similar works related to this research. This table specifies the kind of algorithm that each tool has been targeted, the kind of scheduling and allocation technique supported by the tool and the control implementation style. Also, this table shows the three main characteristics of this research compared against the related work: the support for non-rectangular loop bound shape, problem size independency and external memory support.

Several automatic parallelization tools for compilers have been developed focusing on the generation of new code for different parallel computer architectures, often with a strong focus on the completeness of the depth and scope of parallelization achieved. However, a straightforward application from parallelizing compiler techniques to hardware synthesis does not work [63]. Tools like CLoog-VHDL [39], Compaan/Laura [123] and other methodologies [8] fall in this category, mainly because the models used do not detect the loop carried dependences, and therefore the hardware generated does not exploit existing parallelism through loop dependences but through implementing highly pipelined processors. Besides, sometimes it is mandatory to fit compiler techniques to hardware synthesis tools in order to accommodate them to hardware needs. For example, in hardware synthesis, the control costs are not fixed and can vary significantly by little changes in the original program. So, automatic hardware synthesis tools cannot be directly constructed from compilers due to its special necessities, leading to a research topic with several challenges [64] such as the generation of control schemes and external memory interfaces. Another disadvantage found during this review is that automatic synthesis tools provide a hardware solution for problem instances of certain size.

	PARO [65]	MMAAlpha [36]	Uday [23]	This work
Algorithms Supported				
- Matrix-Matrix Based	✓	✓	✓	✓
- FIR and Convolution	✓	✓	×	✓
Image Filtering	✓	×	×	×
Matrix Decompositions	×	×	×	✓
Scheduling Technique				
- Linear	✓	×	×	✓
Affine	✓	×	×	×
Multidimensional	×	✓	✓	×
Allocation Technique				
Projection	✓	✓	×	✓
LPGS	✓	×	✓	✓
LSGP	✓	✓	×	×
Co-partitioning	✓	×	×	×
Control Scheme Implementation				
Centralized	✓	×	✓	✓
Distributed	✓	✓	✓	✓
Non-Rectangular Loop Bounds	×	×	×	✓
Problem Size Independency	×	×	×	✓
External Memory Scheme Support	×	×	×	✓

Table 2.2: Comparison among automatic parallelization tools.

Although tools like PARO and MMAAlpha produce partitioned processor array in different ways, they lack of parametric support. If the problem size for which they were targeted changes, the activation sequence changes too, and consequently a new processor array must be synthesized in order to generate a new activation sequence. In this research work it is derived a general control scheme for control signal generation problem able to provide a semi-independency from the problem size, and able to support a maximal subset of problem sizes given some hardware parameters.

Another missing issue observed during this review is that all the automatic synthesis tools report processor arrays implementations for algorithms whose loops bounds form rectangular shapes, specially in the case of loop bound mapped to processor after space-time mapping. Algorithms

like matrix multiplication, image filtering, FIR filter, string alignment are reported by PARO [65] and MMAAlpha [36] as cases of study that meet the rectangular shape characteristic. Although PARO provides support for any kind of loop bound shape, it does not show cases of study for full-size processor arrays implementations with such characteristic nor partitioned processor arrays. Algorithms like QR, LU and Cholesky decomposition, which are used in digital signal processing domain, share the characteristic of having non-rectangular shapes formed by their loop bounds. In this research work, the generation of processor arrays, control schemes and memory system for such algorithms in a partitioned way is tackled.

Moreover, from this literature review, the lacking of memory interfaces between processor array and external memories comes out. In almost all reviewed works, it is assumed that data comes to the processor array as it is needed. It has been found few works that put some attention to this problem. Although in [43] are briefly discussed two possible schemes for the external memory address generation problem, it is not implemented. Besides, although in [91] it is implemented a hardware interface for a processor array which implements the MatMul algorithm, it is only a specific implementation without any generalization for the other cases of study. This research aims at solving the problem of generating the external memory interfaces for providing/extracting the data required/generated by processor arrays derived using the polytope model.

Finally, a large number of hardware designs without the formalism approach, have been proposed for a wide variety of algorithms, from linear algebra, graph theory, searching, sorting, digital signal processing, etc. For example, the design of the hand-made systolic architecture presented by Qiang *et al.* in [84] is made by the similar polytope's transformations performed in a visual way. Moreover in the digital signal processing domain, some algorithms are built over simpler algorithms such as matrix multiplications, matrix decompositions, system equation solvers, convolutions, linear filters, [60], [67]. [82], [83], [114], [122]. This research explores the generation of processor array for accelerating basic algorithms used in the digital signal processing domain with a mathematical formalism approach, specially decompositional algorithms like QR, Cholesky and LU.

3

Mathematical Background

There is a rich mathematical background underlying the polytope model. Concepts like polytope, piecewise regular algorithm, iteration space, scheduler, allocation, among other concepts are needed to understand the synthesis process. This chapter presents the mathematical background required for the processor array generation. The algorithmic specification required for generating the processor arrays is introduced in section 3.1. Concepts like scheduling, allocation, iteration interval, and space-time mapping are explained in section 3.2. Finally, the methods for obtaining the processing element data-path, the interconnection topology and the how to interpret the partitioning process are explained in section 3.3. Through this chapter two cases of study are developed concurrently: the MatMul and the Cholesky decomposition algorithms. Although the MatMul algorithm has been widely studied, it is very used in the automatic synthesis research community as case of study for exemplifying the modeling concepts. On the other side, the Cholesky decomposition has some interesting characteristics that MatMul algorithm does not have such as complex hardware operations and non-rectangular loop bound shape.

3.1 Algorithmic Modeling

The polytope definition and its link with sequential nested loops programs are introduced in this section. Later, an specific class of SURE is defined as well as definitions concerned with the piecewise regular algorithm term. These definitions are taken from [24], [44], [64], [69] and [80].

3.1.1 Polytope Model

The polytope model provides an abstraction for modeling nested loops programs with regularity restrictions on the loop indexes that represents sequential or parallel programs. It focuses on techniques for optimizing numerical applications that use arrays as data structures and access them with simple and regular patterns. More specifically, this model studies programs that have affine array access with respect to surrounding loop indexes and programs that can be represented or transformed as a set of perfectly loop nest [6]. With the appropriate transformations, the polytope model helps to extract the loop level parallelism presented in a nested loop program. In order to define what a polytope is, it is necessary to introduce some previous geometrical definitions.

Definition 3.1 An m -dimensional function f is affine if and only if it can be expressed in the following form:

$$f(\vec{I}) = A\vec{I} + \vec{b} \quad (3.1)$$

where $\vec{I} = [i_1, \dots, i_n]^T$ and $A \in \mathbb{Z}^{m \times n}$ is a matrix with m rows and n columns and $\vec{b} \in \mathbb{Z}^m$ is an m -dimensional vector. The domain is also a set of integers: $\vec{I} \in \mathbb{Z}^n$

In other words, a function of one or more variables, i_0, i_1, \dots, i_n is affine if it can be expressed as a sum of constants, plus constant multiples of the variables, i.e. $b_0 + b_1 i_1 + b_2 i_2 + \dots + b_m i_n$ where b_0, b_2, \dots, b_n are constants. Affine functions are usually known as linear functions, although strictly speaking linear functions do not have the b_0 term.

Definition 3.2 A set of vectors is an *affine space* if and only if it is closed under affine combinations, i.e. if \vec{x} and \vec{y} are vectors in the space, all points lying on the line joining \vec{x} and \vec{y} belong to the space.

A line in a vector space of any dimensionality is a one-dimensional affine space. In a three-dimensional space, any two-dimensional plane is an example of a two-dimensional affine subspace.

Definition 3.3 An *affine hyperplane* is an $n - 1$ dimensional affine sub-space of an n dimensional space.

Definition 3.4 An *affine half-space* is any of the two parts into which an affine hyperplane divides an affine space.

Definition 3.5 A *polyhedron* is an intersection of a finite number of half-spaces.

Each one of the half-spaces provides a polyhedron face. Hence, the set of affine inequalities, each representing a face, can be used to compactly represent the polyhedron. A polyhedron can be represented by a system of inequalities that defines a half-space following the next form:

$$\begin{aligned} a_{1,1}i_1 + \dots + a_{1,n}i_n &\leq b_1 \\ &\vdots \\ a_{m,1}i_1 + \dots + a_{m,n}i_n &\leq b_m \end{aligned} \tag{3.2}$$

Definition 3.6 A *polytope* is a bounded polyhedron defined as the integer solutions of a system of affine inequalities:

$$\mathcal{I} = \{\vec{I} \in \mathbb{Z}^n \mid A\vec{I} \leq \vec{b}\} \tag{3.3}$$

where $A \in \mathbb{Z}^{m \times n}$, $\vec{I} \in \mathbb{Z}^n$, $\vec{b} \in \mathbb{Z}^m$. The set of integer points within \mathcal{I} is also known as *iteration space* or *computation domain*.

A polytope models the loop bounds of a perfectly nested loop program, and each of these bounds represents a half-space. These loop bound inequalities shown in the expression 3.2 form a polytope, and each one of the loop iterations is an integral point inside of the polytope.

Example 3.1 The matrix-matrix multiplication (MatMul) algorithm is used through this chapter in order to exemplify several concepts. The multiplication of two square matrixes is defined as $C = A \times B$ where A , B and $C \in \mathbb{R}^{N \times N}$ and its mathematical equation is shown in 3.4.

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j} \quad (3.4)$$

A traditional sequential loop algorithm in form of a pseudocode for multiplying two matrixes is shown in the next figure:

```

do i = 0; N-1
  do j = 0; N-1
    do k = 0; N-1
      C(i,j) = C(i,j) + A(i,k) * B(k,j);
    enddo
  enddo
enddo

```

Figure 3.1: Matrix-Matrix Multiplication pseudocode.

The system of inequalities that defines the polytope for the program shown in figure 3.1 represented like expression 3.3 is:

$$\mathcal{I}_{MatMul} = \left\{ \left[\begin{array}{ccc} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{array} \right] \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \left[\begin{array}{c} N-1 \\ 0 \\ N-1 \\ 0 \\ N-1 \\ 0 \end{array} \right] \right\} \quad (3.5)$$

□

3.1.2 Piecewise Regular Algorithm

The SURE concept has originated several cases of recurrence equations, and each one of these cases models different algorithm characteristics. This research work takes the piecewise regular algorithm (PRA) as input specification of an algorithm because the PRA is a more general specification than perfectly nested loop programs, and because it describes the case when conditional statements inside of a loop nest are presented. A piecewise regular algorithm is a specific case of the piecewise linear algorithm defined by Thile and Roychowdhury in [108] and it follows the next definition:

Definition 3.7 A *piecewise regular algorithm* consists of a set of N quantified equations $S_1[I], \dots, S_i[I], \dots, S_N[I]$. Each equation $S_i[I]$ is defined for all $I \in \mathcal{I}_i$ and is of the following form:

$$x_i [P_i \vec{I}] = \mathcal{F}_i \left(\dots, x_j [Q_j \vec{I} - \vec{d}_{ji}], \dots \right) \quad \text{if } \mathcal{C}_i^I(\vec{I}) \quad (3.6)$$

where x_i, x_j are affinely indexed variables. The indexing functions of the variables are defined by the constant indexing identity matrices P_i, Q_j and by the i -th constant integer *dependence vector* \vec{d}_{ji} , of the corresponding dimension. $\mathcal{C}_i^I(\vec{I})$ is called *iteration dependent condition* of an equation. \mathcal{F}_i denotes arbitrary functions and the dots denote similar arguments. \mathcal{I} is an integral subset $\mathcal{I} \subseteq \mathbb{Z}^n$ called *iteration space* of the PRA. The vector \vec{I} represents an iteration point $\vec{I} \in \mathcal{I}$. The set of all points $P_i \vec{I}, \vec{I} \in (\mathcal{I} \cap \mathcal{C}_i^I)$ is called the *indexed space* of variable x_i .

Definition 3.8 An *index space* of a PRA is the subset $\mathcal{I} \subseteq \mathbb{Z}^n$ consisting of all index points defined by expression 3.3. It is also known as *iteration space* or *computation domain*.

Definition 3.9 An *index point* or *iteration point* \vec{I} is an integral element that belongs to the index space \mathcal{I} .

A PRA may seem identical to perfectly nested loop programs, however there are some differences between both terms. In the case of loop programs, there is an explicit order of computations due the lexicographic order of the loop bounds iterations. Also, dependence vectors can be flow-data dependence, anti-dependence or output dependence but, they are always lexicographically positive. On the other hand, in a PRA, the computation order is implicit. In order to calculate the left-hand side of each equation, all the right-side arguments have to be computed. Besides, the dependence vector can be seen as flow dependences since values are used after being computed, but they do not have to be lexicographically positive. The main consequence is that a PRA might be not computable, *i.e.* a computation date for each index point \vec{I} of the index space \mathcal{I} respecting the data dependences might not exist. On the other hand, in a perfectly nested loop program, the worst computation date could be the sequential order imposed by the lexicographical order of the inner loops. The following example shows the MatMul algorithm described as a PRA.

Example 3.2 (MatMul) The MatMul piecewise regular algorithm is shown in figure 3.2. The matrixes A and B are embedded into the iteration space by equations $A_{in}[i, 0, k] = A_{i,k}$ and by $B_{in}[0, j, k] = B_{k,j}$. For purposes of explanation the quantified equations in the PRA are labeled, and the variables A_{in} , B_{in} and C_{out} denote input and output variables. The MatMul index space is already shown in expression 3.5.

Note that there are differences among the sequential code presented in figure 3.1 and the PRA specification shown in figure 3.2. One of these differences is the inclusion of several indexed variables in the PRA because it is only allowed one access to an index variable at an iteration point, unlike in a sequential loop program where several accesses to the same index point (or memory location)

<i>eqn00</i>	:	$y[i, j, k] = A_{in}[i, 0, k]$	if ($j = 0$)
<i>eqn01</i>	:	$x[i, j, k] = B_{in}[0, k, j]$	if ($i = 0$)
<i>eqn02</i>	:	$y[i, j, k] = y[i, j - 1, k]$	if ($j > 0$)
<i>eqn03</i>	:	$x[i, j, k] = x[i - 1, j, k]$	if ($i > 0$)
<i>eqn04</i>	:	$w[i, j, k] = y[i, j, k] \times x[i, j, k]$	
<i>eqn05</i>	:	$z[i, j, k] = z[i, j, k - 1] + w[i, j, k]$	if ($k > 0$)
<i>eqn06</i>	:	$z[i, j, k] = w[i, j, k]$	if ($k = 0$)
<i>eqn07</i>	:	$C_{out}[i, j, k] = z[i, j, k]$	if ($k = N - 1$)

Figure 3.2: Matrix-Matrix Multiplication piecewise regular algorithm.

are allowed. In compilers area, the one-time index point access property is called single assignment. Another characteristic presented in a PRA is the explicitness in which an external memory access, reading or writing, is defined by a set of indexed variables (statements *eqn00*, *eqn01* and *eqn07* in figure 3.2). This is particularly helpful when hardware is synthesized, since after the space-time mapping each of these variables will indicate when a memory access must take place, and which processing element must perform such access. This situation is explained in detail in section 3.3.1.2.

□

3.1.3 Data Dependences

Other concepts needed in the polytope framework are the data dependences, and graphs that describe the dependences in a PRA.

Definition 3.10 A relation between two indexed variables x_i and x_j in a PRA is a *dependence relation* $x_i \delta x_j$ if there is an index point $x_i[\vec{I}]$, and an index point $x_j[\vec{J}]$ and a memory location M such that:

1. Both $x_i[\vec{I}]$ and $x_j[\vec{J}]$ references read or write the memory location M .
2. $x_i[\vec{I}]$ is to be executed before $x_j[\vec{J}]$ in the PRA.
3. During the execution of the PRA, the memory location M is no written in the time period from the end of execution of $x_i[\vec{I}]$ to the beginning of the execution of $x_j[\vec{J}]$.

Since $x_i[\vec{I}]$ must be executed before $x_j[\vec{J}]$, we have $\vec{I} \preceq \vec{J}$, this mean that the index point \vec{J} depends on \vec{I} . The distance $\vec{J} - \vec{I}$ is denoted by the vector \vec{d}_{ji} of dimensionality equal to n . Theses dependence vectors in a PRA can be put together in a matrix D , where each column defines a dependence vector different of zero. Also, the dependences in a PRA can be represented by a graph of dependence relations between index points called the *data dependence graph* or DG. The DG is obtained from the partial order relation between the index points in the iteration space.

Definition 3.11 A *dependence matrix* is the set of the k non-zero dependence vectors \vec{d}_{ji} combined in $D = [\vec{d}_{j_1 i_1}^1, \vec{d}_{j_2 i_2}^2, \dots, \vec{d}_{j_k i_k}^k]$. Each one of these vectors provides the distance between the number of integral points that separates two consecutive variable accesses in the index space.

Definition 3.12 A *dependence graph* is a directed acyclic graph which describes the dependences among the index points of a PRA. Let a PRA be given with a $\mathcal{I} \subseteq \mathbb{Z}^n$, a dependence graph has the following properties:

1. For each iteration point $\vec{I} \in \mathcal{I}$, there is one node.
2. There is an edge from the index point \vec{I} to \vec{J} (where $\vec{I}, \vec{J} \in \mathcal{I}$) if any computation in the index point \vec{J} needs a result computed by index point \vec{I} .

Example 3.3 (MatMul continuation) In the piecewise regular matrix-matrix multiplication algorithm there are several dependences. However, most of these dependences are zero. The non-zero dependences in this example are composed by the vectors $\vec{d}_{yy} = [0, 1, 0]^t$, $\vec{d}_{xx} = [1, 0, 0]^t$, and $\vec{d}_{zz} = [0, 0, 1]^t$. These vectors represent the dependences in the labeled statements *eqn02*, *eqn03*, *eqn05* in the PRA shown in figure 3.2. The matrix D in this case is:

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

The dependence graph of the MatMul algorithm, with the index space bounded by $N = 4$ is shown in figure 3.3. Recall that each one of the vertices on this graph represents an iteration point within the index space and the edges represent the relation between two iteration points.

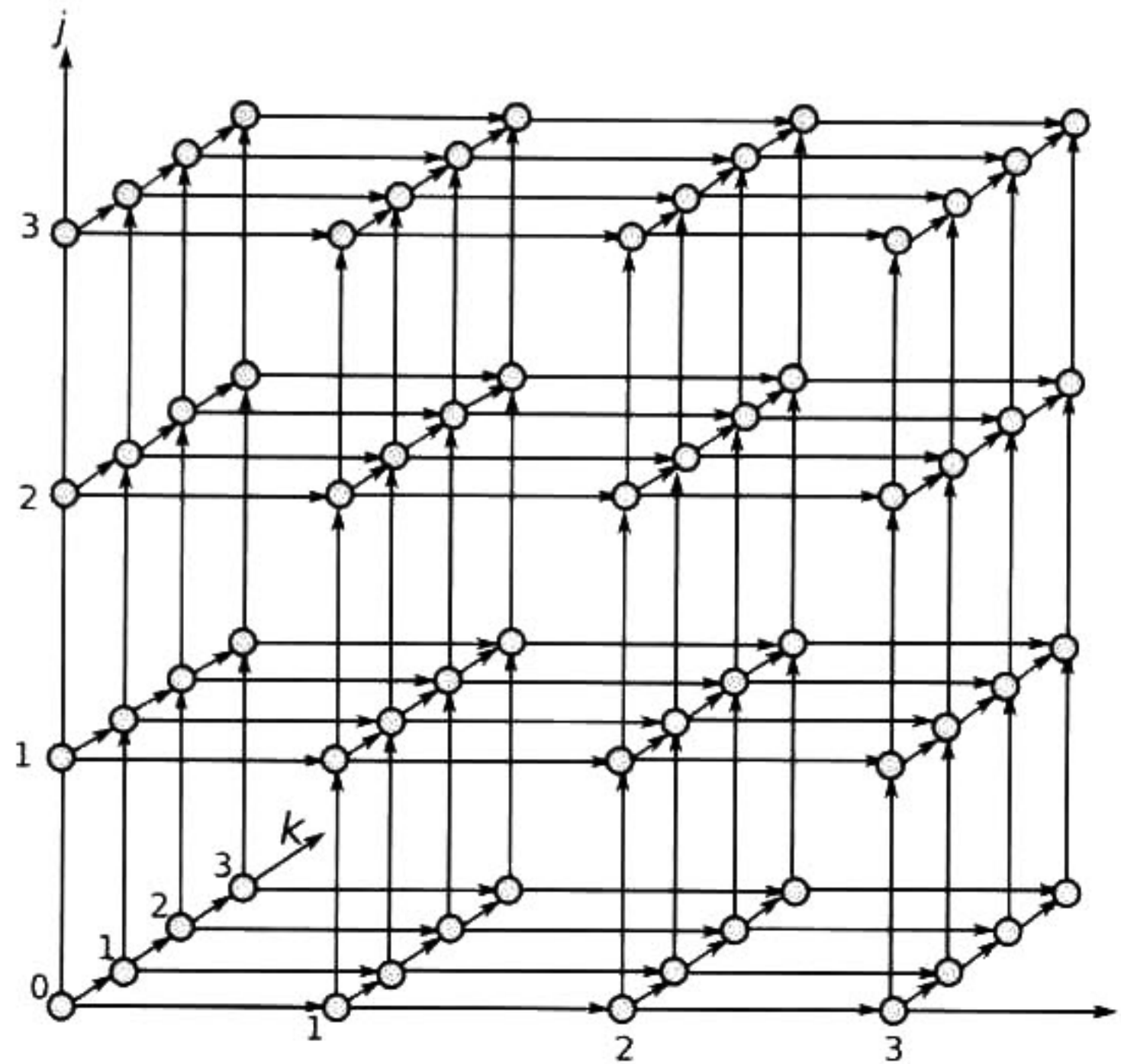


Figure 3.3: MatMul dependence graph for $N = 4$.

□

Example 3.4 (Cholesky) Along with the MatMul example used through this chapter, the Cholesky decomposition is also used in order to exemplify some concepts. The Cholesky decomposition algorithm for a symmetric and positive-definite matrix $W \in \mathbb{C}^{N \times N}$ consists of decomposing W in such way that $W = L \times L^{-1}$, where $L \in \mathbb{C}^{N \times N}$ is a lower triangular matrix. Its form in a picesewise regular algorithm is shown figure 3.4. Variable W_{in} is an input variable where the matrix W is stored and variable L_{out} is an output variable where the resulting matrix L is stored. The Cholesky index space \mathcal{I}_{Chol} is:

$$\mathcal{I}_{Chol} = \left\{ \left[\begin{array}{ccc} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{array} \right] \left[\begin{array}{c} i \\ j \\ k \end{array} \right] \leq \left[\begin{array}{c} 0 \\ N-1 \\ 0 \\ N-1 \\ 0 \\ N-1 \end{array} \right] \right\}$$

eqn00	:	$z[i, j, k] = W_{in}[i, j, k]$	if $(i = 0)$
eqn01	:	$z[i, j, k] = w[i - 1, j, k]$	if $(i > 0)$
eqn02	:	$w[i, j, k] = \text{sqrt}(z[i, j, k])$	if $(j = i \text{ and } k = i)$
eqn03	:	$v[i, j, k] = w[i, j, k]$	if $(j = i \text{ and } k = i)$
eqn04	:	$d[i, j, k] = v[i, j, k - 1]$	if $(k > j)$
eqn05	:	$w[i, j, k] = z[i, j, k] / d[i, j, k]$	if $(j = i \text{ and } k > i)$
eqn06	:	$v[i, j, k] = d[i, j, k]$	if $(j = i \text{ and } k > i)$
eqn07	:	$h[i, j, k] = w[i, j, k]$	if $(j = i \text{ and } k > i)$
eqn08	:	$b[i, j, k] = h[i, j - 1, k]$	if $(j > i)$
eqn09	:	$v[i, j, k] = b[i, j, k]$	if $(j > i \text{ and } k = j)$
eqn10	:	$l[i, j, k] = b[i, j, k]$	if $(j > i \text{ and } k = j)$
eqn11	:	$c[i, j, k] = d[i, j, k]$	if $(k > j)$
eqn12	:	$f[i, j, k] = b[i, j, k] * l[i, j, k]$	if $(j > i)$
eqn13	:	$w[i, j, k] = z[i, j, k] - f[i, j, k]$	if $(j > i)$
eqn14	:	$v[i, j, k] = c[i, j, k]$	if $(j > i \text{ and } k > j)$
eqn15	:	$l[i, j, k] = c[i, j, k]$	if $(j > i \text{ and } k > j)$
eqn16	:	$h[i, j, k] = b[i, j, k]$	if $(j > i)$
eqn17	:	$L_{out}[i, j, k] = v[i, j, k]$	if $(k = N - 1)$

Figure 3.4: Piecewise regular algorithm for Cholesky decomposition.

The DG of Cholesky decomposition is shown in figure 3.5. Note that the index space \mathcal{I}_{Chol} is non-rectangular as the index space \mathcal{I}_{MatMat} due to some inequalities depend on other index points.

□

The size of the DG depends directly on the size of the iteration space, *i.e.* a large iteration space leads to a large dependence graph. A condensed representation of the DG is the reduced dependence graph (RDG). The size of the RDG depends exclusively on the number of indexed variables presented

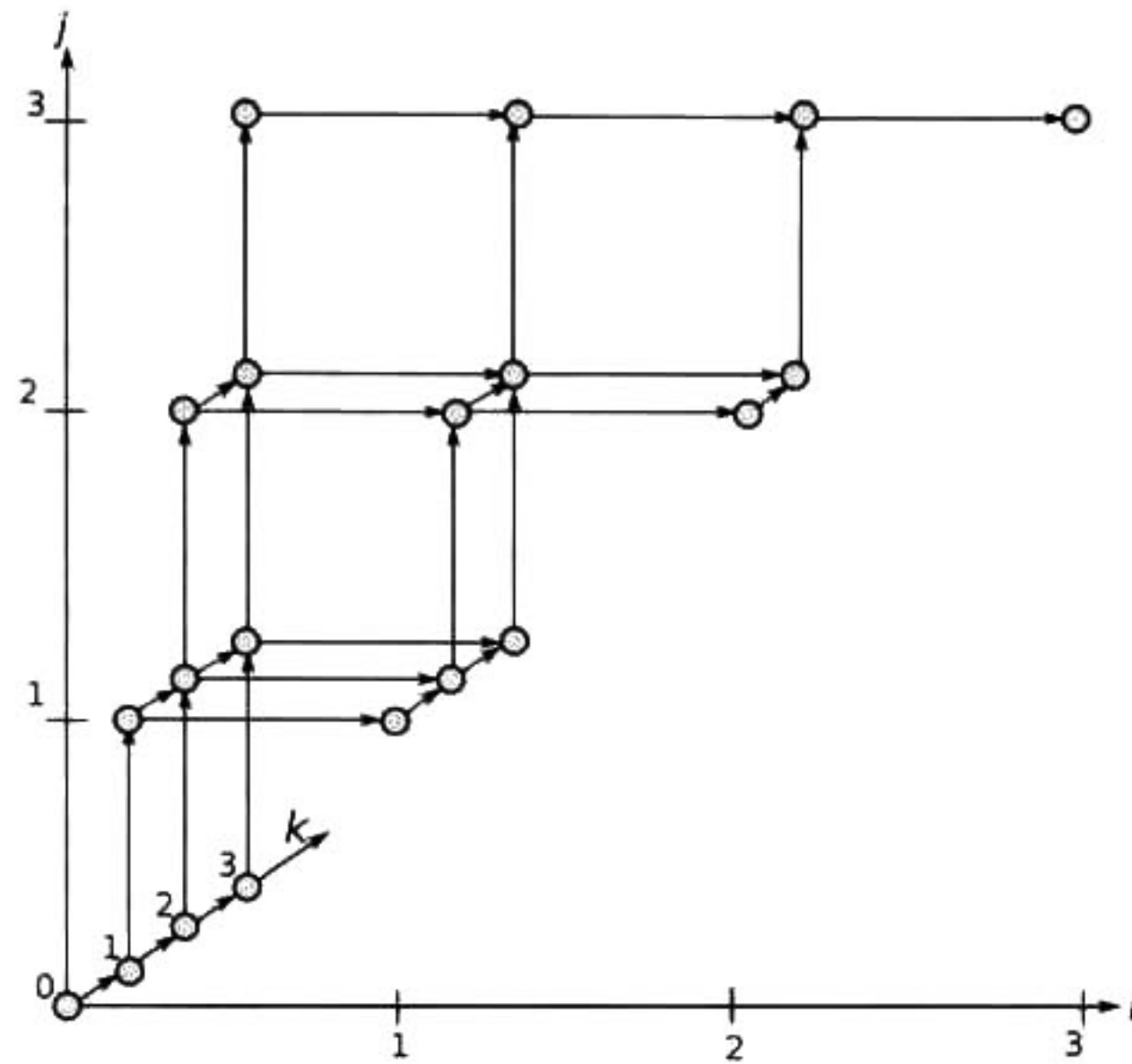


Figure 3.5: Cholesky decomposition dependence graph for $N = 4$.

in a PRA. Mainly, the RDG models dependences among the indexed variables inside of the iteration points as well as dependences among iterations (as DGs). Several different definitions of RDGs could be found in literature [95], [105]. This work focuses on Hannig's RDG definition [64] due to it provides a more general definition than traditional approaches, including the modeling of the latency operations.

Definition 3.13 A *reduced dependence graph* is directed graph which models data dependences among the indexed variables as well as dependences among iterations. Let a PRA be given with $\mathcal{I} \subseteq \mathbb{Z}^n$. The construction of a reduced dependence graph is as follows:

1. There is vertex $v_i \in V$ in the RDG for each indexed variable x_i on the left hand side of the algorithm and for each constant within the algorithm.
2. Each vertex has associated a functionality (input, output, propagation or arithmetical function).
3. For each vertex with an arithmetical functionality has associated an operational latency w_i .
4. For each indexed variable x_i, x_j represented by the vertexes v_i, v_j , exists an edge (v_j, v_i) weighted with the dependence vector \vec{d}_{ji} .

Example 3.5 (MatMul continuation) Assume that the hardware operational latencies for MatMul algorithm are one time unit for both multiplication and addition. These latencies are represented by w_4 , and w_5 . The edge weights different of zero are the represented by the vertexes (eqn_{00}, eqn_{02}) , (eqn_{02}, eqn_{02}) , (eqn_{01}, eqn_{03}) , (eqn_{03}, eqn_{03}) , (eqn_{06}, eqn_{05}) , and (eqn_{05}, eqn_{05}) . The reduced dependence graph for this example is shown in figure 3.6. Here, the edge and vertex whose weight is different of zero are indicated. Labels indicating the operational function of the node are also shown.

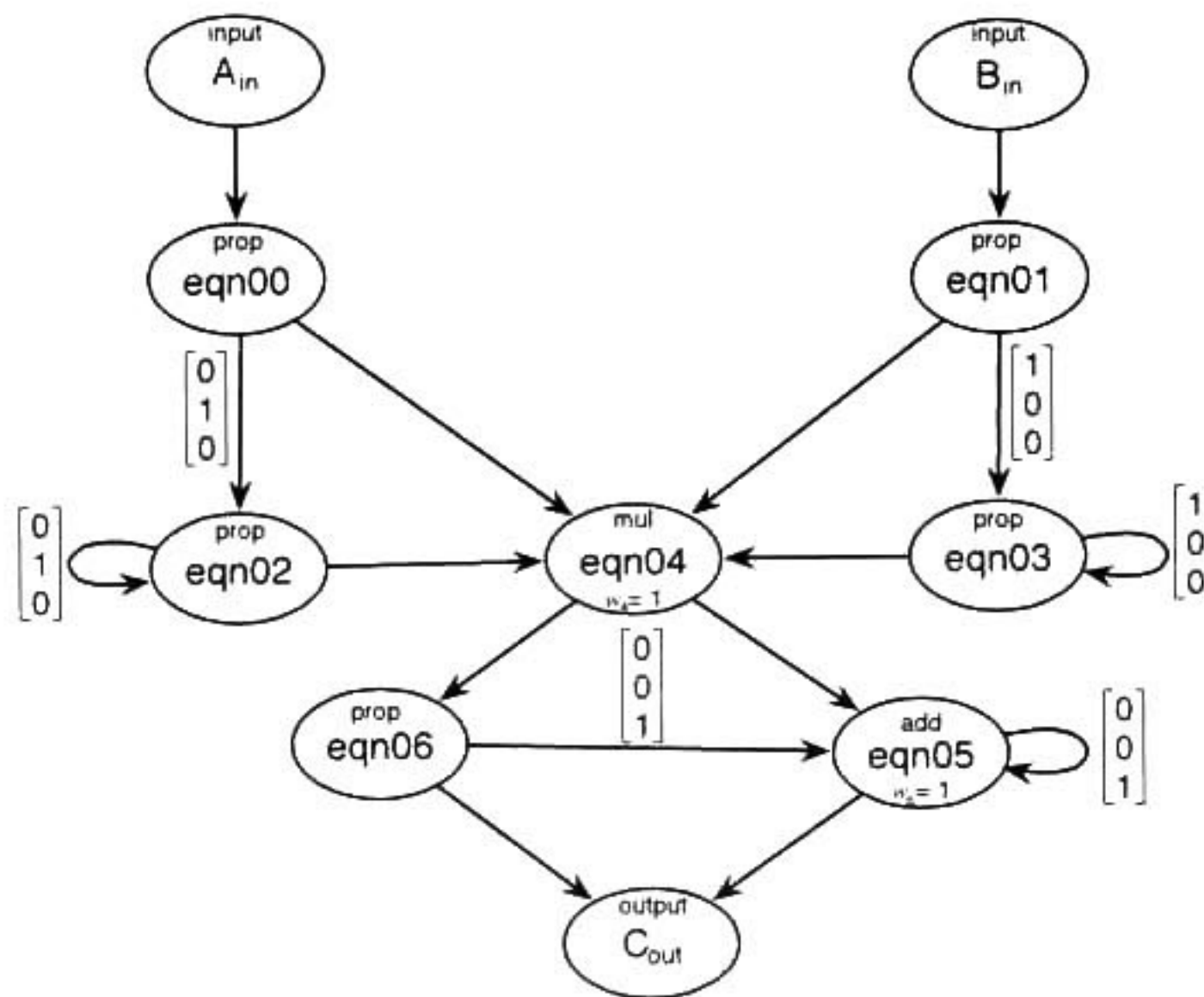


Figure 3.6: Reduced dependence graph for MatMul algorithm.

□

Example 3.6 (Cholesky continuation) The RDG of Cholesky decomposition is shown in figure 3.7. Assume that the latencies for the basic operations are: 12 time units for division, 6 time units for square root, 3 time units for multiplication and one time unit for subtraction. These latencies are represented by w_5 , w_2 , w_{12} , and w_{13} , respectively. The edge weights different of zero are the represented by (eqn_{02}, eqn_{01}) , (eqn_{05}, eqn_{01}) , (eqn_{13}, eqn_{01}) , (eqn_{03}, eqn_{04}) , (eqn_{06}, eqn_{04}) , (eqn_{14}, eqn_{04}) , (eqn_{07}, eqn_{08}) and (eqn_{16}, eqn_{08}) .

□

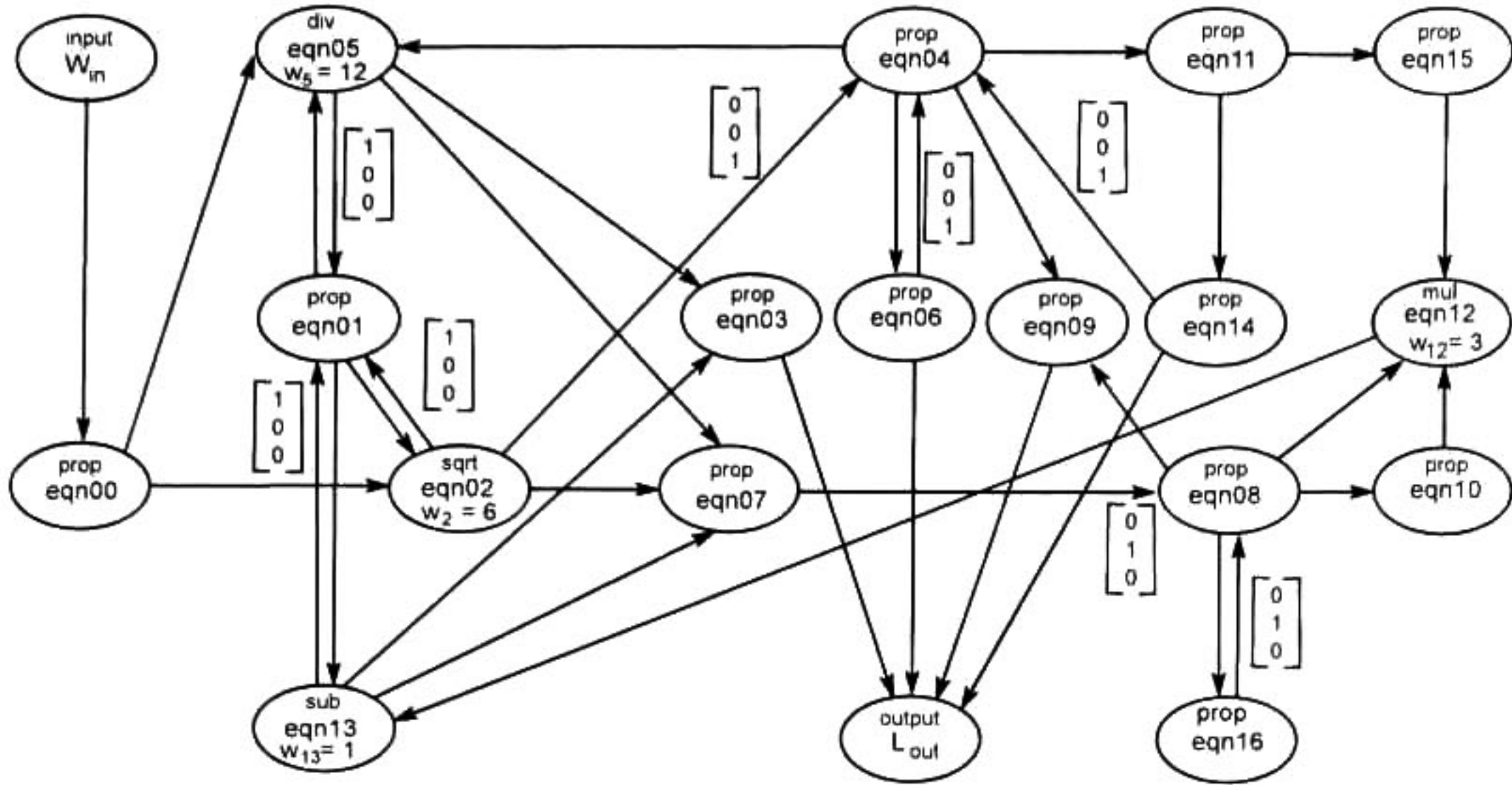


Figure 3.7: Cholesky reduced dependence graph.

3.2 Space-Time Transformation in the Polytope Model

Contrary to a loop program, in a PRA there is not an explicitly execution order. A PRA requires to establish a scheduling for their computations due the lack of an execution order. Besides, in the processor array context, it is required to establish where the computation should be allocated. Together, scheduling and allocating a PRA could be performed by a space-time mapping represented by a unimodular transformation matrix. This section provides the definitions, and background concerning to the unimodular transformations in the polytope model. Also, the concepts of scheduler and allocation functions, and iteration interval within this research work are defined. The last part of this section explain the meaning of the space-time transformation in the context of the processor array PRAs.

3.2.1 Unimodular Transformation

From the loop transformation point of view, the unimodular mapping is seen as a function composed by a set of primitive transformations (loop reversal, loop permutation and loop skewing) that change the ordering of iterations [6]. Basically, loop reversal consists of executing the iterations of a loop

program in a reverse order; loop permutation is an interchange in the order between two or more loops; and loop skewing rearranges the loop iterations such as all dependences are between iterations of the outer loops. In the PRA context, the unimodular transformation provides an execution order and a location where the computation should be placed (loop reversal and loop permutation). Besides, the shape of the index space and the data dependence directions are transformed into a new index space with a different geometrical shape and different data dependence directions (loop skewing). In order to explain the space-time mapping, some previous definitions are presented.

Definition 3.14 A matrix T is unimodular if and only if:

$$\det(T) = \pm 1 \quad (3.7)$$

Definition 3.15 A unimodular transformation is a bijective function represented by a unimodular matrix T . Let $\mathcal{I}, \mathcal{J} \subseteq \mathbb{Z}^n$ be two indexes spaces of dimensionality n . The unimodular transformation maps each index point $\vec{I} \in \mathcal{I}$ to one index point $\vec{J} \in \mathcal{J}$ in the following way:

$$\vec{J} = T\vec{I} \quad (3.8)$$

A unimodular transformation T is legal only if all data dependences are preserved. The loop bound image of the index space \mathcal{I} when applying a transformation T can be put in a matrix form, using the transformation matrix T and the matrix inequality which represents the polytope:

$$\left. \begin{array}{l} A\vec{I} \leq \vec{b} \\ T\vec{I} = \vec{J} \end{array} \right\} \Rightarrow \begin{array}{l} AT^{-1}\vec{J} \leq \vec{b} \\ \hat{A}\vec{J} \leq \vec{b} \end{array}$$

Since the bounds of the index space \mathcal{I} are convex, the above matrix inequality also delimits a new convex space. This space is the minimum convex space which contains all the points of the index space \mathcal{J} whose anti-image belongs to the index space \mathcal{I} . The bounds of the index space \mathcal{J} can be extracted from the inequality $\hat{A}\vec{J} \leq \vec{b}$. For computing the bounds of the index space \mathcal{J} the Fourier-Motzkin elimination algorithm [19] could be used in order to solve the inequalities system

$\hat{A}\vec{J} \leq \vec{b}$. The Fourier-Motzkin algorithm solves the inequalities system by projecting each inequality onto a reduced number of unknowns and then eliminating one by one each unknown. In the processor array context, the unimodular matrix can provide a time notion to one dimension in a PRA index space of dimensionality n , and space notion to the $n - 1$ index space dimensions. Providing such sense is possible if unimodular matrix is constructed by scheduler and allocation functions. Next subsections describe both concepts and how to put them together in the space-time mapping.

3.2.2 Scheduling Function

A scheduler is a timing function that assigns a computation date to each task. Depending on the granularity desired, a scheduler function could assign a computation date to each one of the index points in an index space (coarse granularity), or it could provide a computation date for the computations inside of a index space (fine granularity). The general definition of a scheduler function for a piecewise regular algorithm is as follows:

Definition 3.16 A scheduler $\vec{\lambda}$ is a function $t : \mathcal{I} \rightarrow \mathbb{Z}$ such that for any iteration points $\vec{I}, \vec{J} \in \mathcal{I} : t(\vec{J}) > t(\vec{I})$ if $\vec{J} \succeq \vec{I}$. In other words, a scheduler is a timing function that assigns a computation date to each task such as all dependencies are respected.

Definition 3.17 A scheduler function is called *linear scheduler* of a PRA with a dependence matrix D if it is in the following form:

$$t(\vec{I}) = \left\lfloor \vec{\lambda}_l \vec{I} \right\rfloor \quad \forall \vec{I} \in \mathcal{I} \quad (3.9)$$

where $\vec{\lambda}_l$ is the linear schedule vector $\vec{\lambda}_l \in \mathbb{Q}^{1 \times n}$ such that $\vec{\lambda}_l \vec{d}^k \geq 1$ for all $\vec{d}^k \in D$.

The linear scheduler function concept is closely related to the dependence graph, because this function is obtained by searching the longest path in the iteration space, and then trying to find a rational vector which preserves all dependences minimizing the execution time. Usually, this search is accomplished by using a linear program formulation. The use of floor function with the rational scheduler vector eliminates the necessity of having an integer linear program formulation. For this

research work, it has been selected the linear scheduling method proposed by Darte *et al.* in [28], due this method derives asymptotically scheduler functions equivalent to the best scheduler when all operations of a PRA have an unitary delay. When this assumption is true, the linear scheduler is as good as the fastest existing scheduler. The closeness of the linear scheduler to optimality is useful due to these schedulers are very easy to use in practice, and their simplicity results in a low implementation overhead for control schemes. Geometrically, the linear scheduler produces a set of time hyperplanes which consist of a set of index points that posses the same execution time, *i.e.* each hyperplane is executed sequentially while the index points that forms the hyperplane are executed in parallel.

The concept of linear scheduling considers each iteration point as an atomic unit, *i.e.* all iteration points require the same amount of time to be executed. However, in real-life applications this assumption is not always true; specially in a hardware implementation where different arithmetic operations are completed in different time units. This leads the idea of having an offset for each operation of the iteration point. The kind of schedulers that take into account this idea are called affine schedulers. An affine scheduler is defined as follows:

Definition 3.18 A scheduler function is called *affine scheduler* function of a RDG with a iteration space \mathcal{I} if it is in the following form:

$$t_i(\vec{I}) = \left\lceil \vec{\lambda}_a \vec{I} + \tau(v_i) \right\rceil \quad \forall \vec{I} \in \mathcal{I} \quad (3.10)$$

where $\vec{\lambda}_a$ is the affine schedule vector $\vec{\lambda}_a \in \mathbb{Q}^{1 \times n}$ and $\tau(v_i) \in \mathbb{Q}$ is the time displacement for the start time of the RDG vertex v_i .

Similar to the linear scheduler, the affine scheduler concept is related to the reduced dependence graph because the affine scheduler takes into account the operational latencies attached to the RDG. Also, as the same as the linear scheduler, an affine scheduler could be computed by a linear programming formulation by using the RDG, the latencies of the operations, and the iteration space.

It might happen that a linear and an affine scheduler lead to the same total execution time of a PRA. In the affine scheduler case, the concept of time hyperplanes is not longer used.

Example 3.7 (Cholesky continuation) Assume the linear scheduler function for Cholesky PRA is $\vec{\lambda}_l = [1, 1, 1]^t$ and the size of its iteration space \mathcal{I}_{Chol} is bounded by $N = 4$. Although the four operations required in Cholesky algorithm have different latencies according to example 3.6, the most cost-time operation is taken as reference in order to know the time required for an iteration point to complete its task. In this example the most cost-time operation is the division which takes 12 time units. The total execution time required is given by the number of time hyperplanes, generated by the scheduler function, that crosses the iteration space ($\arg \max t(\vec{I})$) multiplied by the most cost-time operation. In this case the total execution time is 120 time units. The sequential time required in this case would be 240 time units. Figure 3.8 is a Gantt diagram showing the time required for each computation inside of an iteration point and its execution date. It is important to note that each iteration point that belongs to the same time hyperplane starts its computations only when the previous time hyperplane has been completely computed. In other words, there is not allowed any overlapping between iterations.

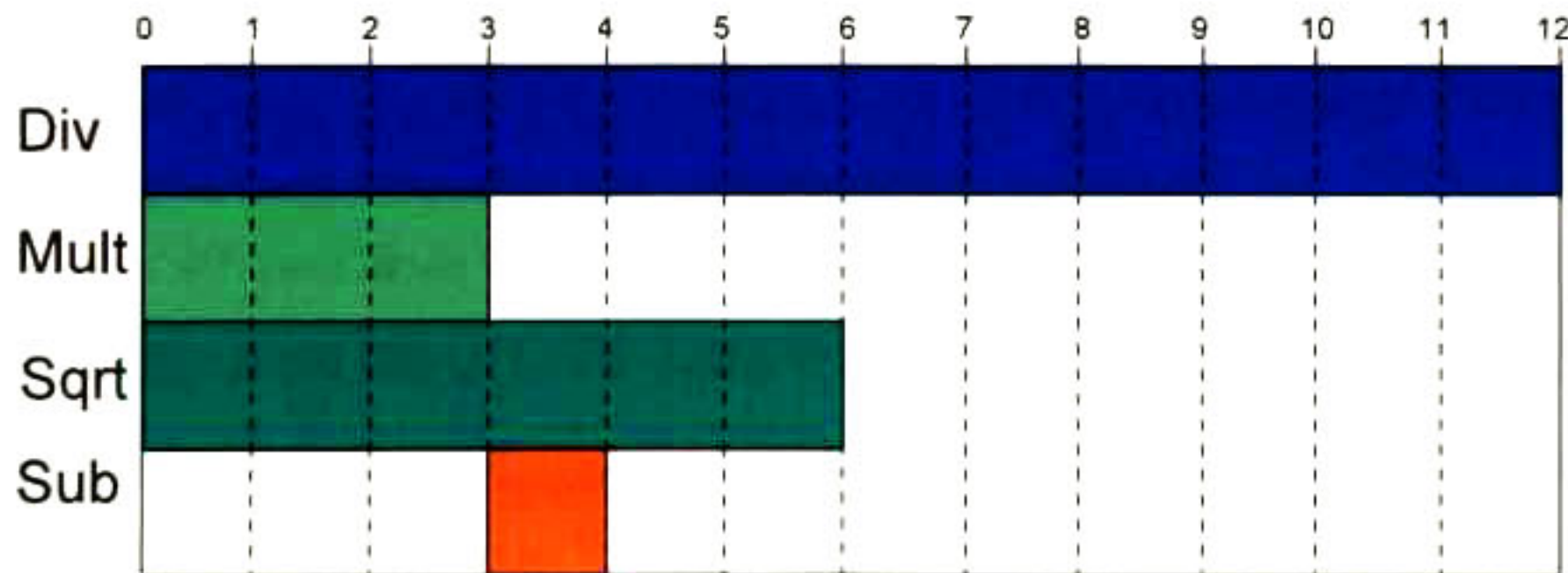


Figure 3.8: Computation dates assigned to different operations when a linear scheduler is used.

Now, assume the affine scheduler $\vec{\lambda}_a = [12, 4, 12]^t$ with the affine part $\tau(v_{12}) = 8$ and $\tau(v_{13}) = 11$. Figure 3.9 is a Gantt diagram showing the scheduling of the computation within an iteration point when the affine scheduler is used. Note that multiplication and subtraction operation are shifted with respect of the same operations shown in figure 3.8 due to the affine part presented in

the affine scheduler. When an affine scheduler is used, it is not needed to wait until all the iteration points in a time hyperplane have totally finished their computations, but once all data are available it is allowed to start new computations belonging to other time hyperplane. This characteristic leads to decrease the total execution time. In affine scheduler case, the total execution time is 90 time units.

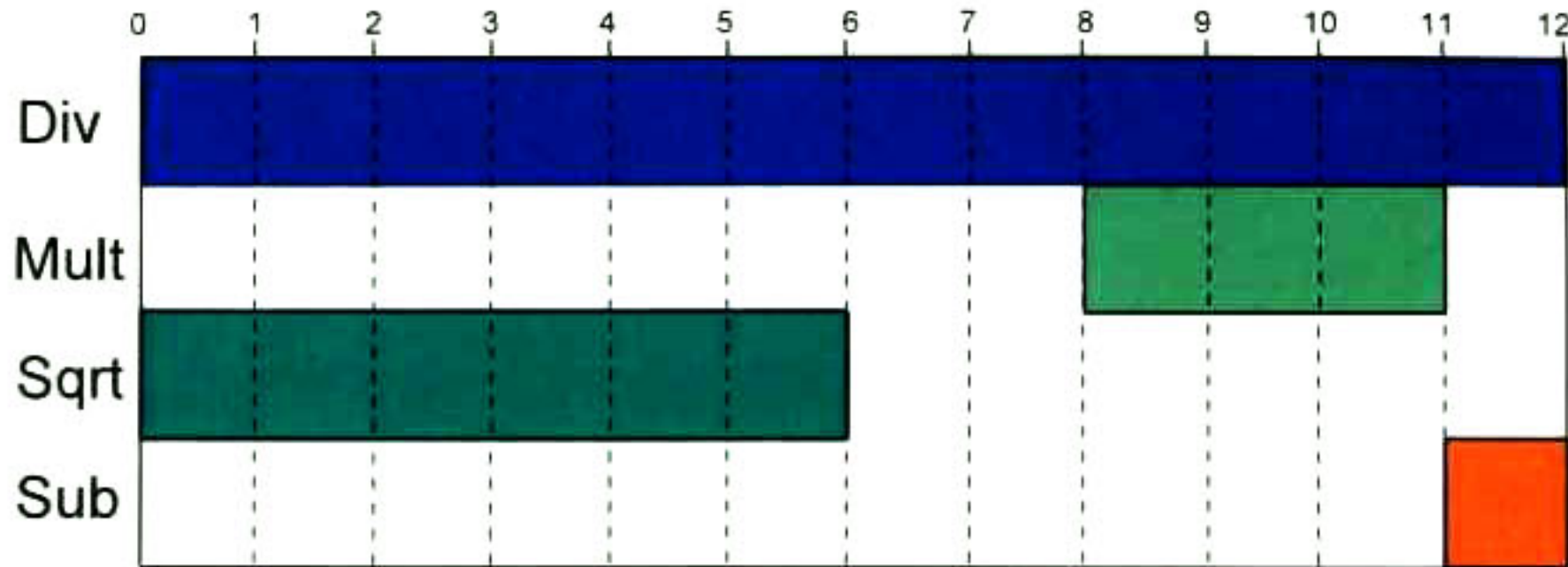


Figure 3.9: Computation dates assigned to different operations when an affine scheduler is used.

□

3.2.3 Allocation Function

The allocation function provides a place (processor) where all index points in an index space are mapped. The general definition of an allocation function for a PRA is as follows:

Definition 3.19 An *allocation* Φ is a function $s : \mathcal{I} \rightarrow \mathbb{Z}^{n-1}$ such that for any pair of iteration points $\vec{I}, \vec{J} \in \mathcal{I} : t(\vec{I}) = t(\vec{J}) \Rightarrow s(\vec{I}) \neq s(\vec{J})$, where t is a scheduler function. In other words an allocation function specifies the spatial distribution of the computations in such way that two or more computations dated at the same time instant do not take place in the same processor.

In the processor array context, the allocation function could be obtained from different ways. Some methodologies, like in PARO, include the allocation function when the scheduler is computed or even they use a partitioning approach as allocation function for computing a scheduler. In this research the selection of an allocation function is after computing the scheduling (in order to respect definition 3.19) and before applying any partitioning technique. The selection of an allocation

function is done by reducing one dimension a n -dimensional index space using a projection vector $\vec{u} \in \mathbb{Z}^n$ as follows:

Definition 3.20 An *allocation function* Φ for a PRA is a $(n - 1) \times n$ matrix constructed from the projection vector \vec{u} as follows:

$$\Phi = \begin{pmatrix} u_i & 0 & \cdots & 0 & -u_1 & 0 & \cdots & 0 \\ 0 & u_i & \cdots & 0 & -u_1 & \vdots & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_i & -u_{i-1} & 0 & \cdots & 0 \\ 0 & \cdots & \cdots & 0 & -u_{i-2} & u_i & \cdots & 0 \\ \vdots & \cdots & \cdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & -u_n & 0 & \cdots & u_i \end{pmatrix} \quad (3.11)$$

where $\vec{u} = (u_1, u_2, \dots, u_n)^t \in \mathbb{Z}^n$ and $u_i \neq 0, 1 \leq i \leq n$.

3.2.4 Iteration Interval

When the scheduling and allocation functions are used for deriving processor arrays, it is required to establish a time interval between the execution of two successive computations on a same processor. Such interval is called iteration interval and its definition is as follows:

Definition 3.21 An *iteration interval* $P \in \mathbb{Z}$ of an allocated and scheduled PRA is the number of time instances between the evaluation of two consecutive instances of an indexed variable within the same processing element. The iteration interval is given by the absolute value of:

$$P = |\vec{\lambda}\vec{u}| \quad (3.12)$$

Obtaining the value of P could be accomplished in two different ways, if equation 3.12 is satisfied. The first possibility is by proposing a value for P , and including it as a constraint into the scheduling linear program formulation. The second possibility is to consider P as a variable in the linear program

formulation, thus the iteration interval will be given as a result of the linear program. The first case forces the linear program formulation to search a scheduler function given a desired iteration interval, whereas the second case allows to the formulation to find a value for P . Such formulations could be found in detail in [64].

Example 3.8 (Cholesky continuation) Although in example 3.7, the iteration interval is not explicitly defined, in the linear scheduler case it is explicitly mentioned. As a result, when the linear scheduler function $\vec{\lambda}_l = [1, 1, 1]^t$ is used, the minimum iteration interval found by the linear programming formulation is $P = 12$ regardless of the used projection vector. This iteration interval means that twelve time steps are needed in order to ensure that all the operations within the same processing element are finished. On the other hand, in the case of the affine scheduler $\vec{\lambda}_a = [12, 4, 12]^t$, the minimum iteration interval will depend on the used projection vector. If the $\vec{u} = [1, 0, 0]^t$ the iteration interval calculated by the linear program formulation will be $P = 12$, but if $\vec{u} = [0, 1, 0]^t$ then the interval will be $P = 6$. Recall that an affine scheduler does not need to wait until all the operations within the same processing element have been completed. Finally, in the linear and affine scheduler cases, the iteration interval could be relaxed to higher values in order to fit a desired throughput by adding P as a constraint in the linear program formulation.

□

3.2.5 Space-Time Mapping

The parallelization based on the polytope model addresses the problem of finding a scheduler function which generates a set of time affine hyperplanes, and selecting an allocation function according to such scheduler. This problem can be formulated as an affine mapping that transforms a piecewise regular algorithmic input specification (*target polytope*) into an output specification (*source polytope*) that contains the same points, but in a new coordinate system in which a dimension is strictly temporal and the others are strictly spatial [80]. The space-time mapping can be accomplished by putting together the scheduler and allocation functions in the transformation matrix T :

Definition 3.22 The transformation matrix T is called a space-time mapping if:

$$T = \begin{bmatrix} \vec{\lambda} \\ \Phi \end{bmatrix} \quad (3.13)$$

where $\vec{\lambda} \in \mathbb{Z}^{1 \times n}$ is the scheduler vector and $\Phi \in \mathbb{Z}^{(n-1) \times n}$ is the allocation matrix.

Definition 3.23 The *iteration space* \mathcal{J} of the target polytope is composed by:

$$\mathcal{J} = \{\vec{J} \in \mathbb{Z}^n \mid AT^{-1}\vec{I} \leq \vec{b}\} \quad (3.14)$$

where $\vec{J} = [t \ p]^t$ is an index point of the target polytope with time and space notions.

The iteration space of the target polytope can be seen as an iteration space \mathcal{J} divided into two subspaces \mathcal{T} and \mathcal{P} which define a time space and a processor space, respectively. The indexed points in the processor space represent the processing elements in the processor array and the index points in the time space are the parallel time steps needed for the PRA execution.

Definition 3.24 A *time space* \mathcal{T} is a one dimensional space which maps the index space \mathcal{I} of a source polytope as follows:

$$\mathcal{T} = \{t \mid t = \vec{\lambda}\vec{I} \wedge \vec{I} \in \mathcal{I}\} \subset \mathbb{Z}^{1 \times n} \quad (3.15)$$

Definition 3.25 A *processor space* \mathcal{P} is a $n - 1$ dimensional space which maps the index space \mathcal{I} of a source polytope as follows:

$$\mathcal{P} = \{p \mid p = \Phi\vec{I} \wedge \vec{I} \in \mathcal{I}\} \subset \mathbb{Z}^{(n-1) \times n} \quad (3.16)$$

Example 3.9 (Cholesky continuation) Assume the same linear scheduler function and iteration space as in example 3.7. Also, assume that the projection vector used is $\vec{u} = [0, 1, 0]^t$ with an iteration interval of $P = 12$. A possible unimodular transformation matrix T for space-time mapping is:

$$T = \begin{bmatrix} \vec{\lambda} \\ \Phi \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.17)$$

After applying the Fourier-Motzkin algorithm to the source polytope iteration space \mathcal{I}_{Chol} , the target polytope index space \mathcal{J}_{Chol} is :

$$\mathcal{J}_{Chol} = \left\{ \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -3 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & -1 \\ 1 & -2 & -1 \\ 0 & 0 & 1 \\ -1 & 1 & 2 \end{bmatrix} \begin{bmatrix} t \\ p_0 \\ p_1 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 3(N-1) \\ 0 \\ 2(N-1) \\ N-1 \\ 0 \\ 0 \\ 0 \\ N-1 \\ 0 \end{bmatrix} \right\}$$

Note that the target index space \mathcal{J}_{Chol} is formed by the indexes t , p_0 and p_1 representing time and processor spaces. Also, note the time space bounds $0 \leq t \leq 3(N-1)$ provide the amount of time steps required to execute the Cholesky PRA. The interpretation of the new target index space is that the index point $[t, p_0, p_1]^t \in \mathcal{J}_{Chol}$ is executed in time t at processor point (p_0, p_1) . In other words, for each index point $(p_0, p_1) \in \mathcal{P}_{Chol}$ it is assigned a subset of index points of the source iteration space \mathcal{I}_{Chol} that will be executed according to the scheduler function which is represented by index t . From compiling point of view, the target polytope is represented as a nested loop of depth three, and the index space is the index loop program where the outer loop is scanned by t , the middle loop by p_0 and the inner loop is scanned by p_1 . Finally, taking into account the iteration

interval, each index point requires P time instances to execute all the operations within the processor point (p_0, p_1) , *i.e.* each PE will take twelve clock cycles to compute a processor point.

Figure 3.10 presents how the computations are assigned for each index point (p_0, p_1) denoted as $PE_{(p_0, p_1)}$. The boxes indicate the computation done in an iteration point. Each one of these operations (square root, division or multiplication/subtraction) is shown in different colors. Some idle times are added for square root and multiplication/subtraction operations, due to it is necessary to synchronize all operations to the division operation. From figure 3.10, it is important to note that index points in the processor space start their computations only when the previous index points have totally ended, *i.e.* there is not overlapping. Also, note that each time unit is divided into time instances (clock cycles) equal to the iteration interval P

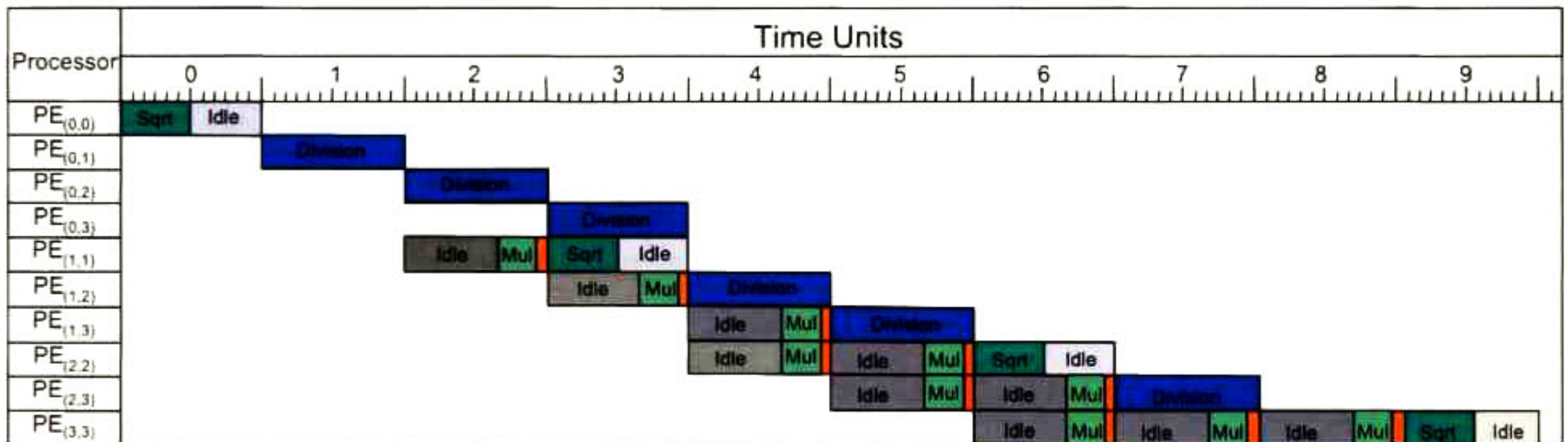


Figure 3.10: Execution times for each operation in an index point.

Finally, the processor space can be represented in a form of DG. Figure 3.11 shows the two-dimensional processor space \mathcal{P}_{Chol} . Intuitively, this figure can be interpreted as an abstraction of the processor array with a topology derived from space-time transformations.

□

The previous example shows how the space-time transformation gives to the PRA iteration space notion of space and time. From the hardware design point of view, these notions provide an activation sequence (given by $\vec{\lambda}$) for each index point of the processor space (given by Φ), and a time interval between two successive computations (given by P). This information is useful when the processor

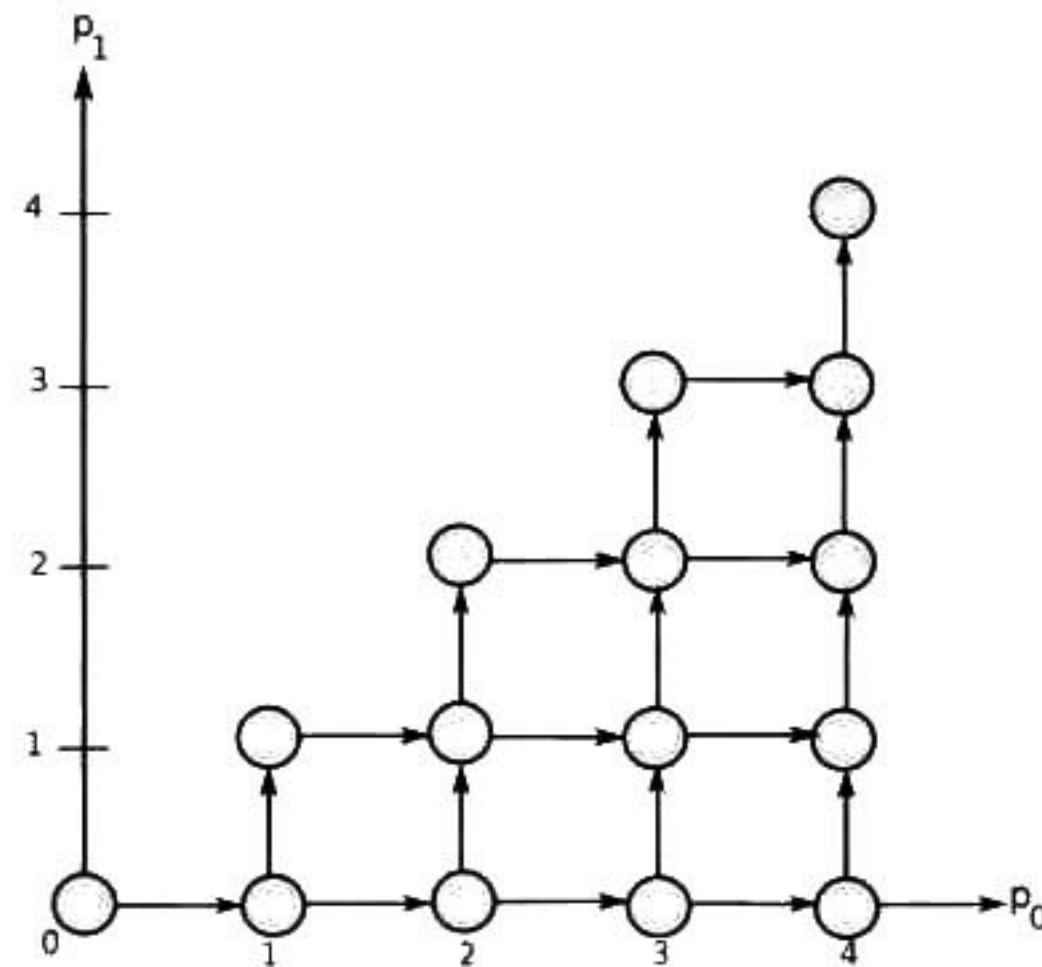


Figure 3.11: Processor space of Cholesky decomposition when $N=4$.

array is designed because it provides an activation pattern for each processing element inside the processor array. Also, this activation pattern facilitates the processor array construction made by synthesis tools. However, this is not all the information that such tools require, since it is also needed to infer the processor array topology and the processing element data-path. Next section provides the concepts needed for the construction of processor array interconnection and the PE data-path.

3.3 Processor Array Synthesis

From an algorithm represented in a piecewise regular form, the construction of the processor array is possible. Information such as dependence vectors, reduced dependence graphs, scheduler and projection functions are used in the derivation of the array interconnection topology, processing element data-path and control structures. By analyzing some of these concepts it is possible to determine if there exists a local connection among processing elements. If a connection exists, it is required to know if the connection will be direct or delayed *i.e.* if the interconnection requires registers or not. Besides of the interconnection, deriving the processing element internal data-path is required too. In this section, the formalisms and ideas needed for synthesizing a processor array from a PRA are described. The first part of this section describes how to construct a full-size processor array dependent of the problem size. Although this array is highly parallel, sometimes it is

unsuitable of being implemented due to it exceeds the computational resources (like functional units or input/output ports), and due to it is high data memory demanding. Furthermore, this full-size array is not able to solve several problem instances, but it only solves an unique problem size. The second part of this section explains how by partitioning the processor space it is possible to reduce the memory demand, making the processor array independent of the problem size and setting the processor array size at synthesis time.

3.3.1 Full-Size Implementation

3.3.1.1 Processor Array Interconnection Topology

The PE interconnections among other PEs are required in order to construct the processor array topology. Such connections are obtained by using the allocation function and the data dependences different of zero. Intuitively, for each data dependence vector $\vec{d}_{ji} \neq 0$ in the PRA, a connection from the indexed variable x_j to x_i should be inferred, *i.e.* the number of dependence vectors different of zero provides the number of PE input/output ports. Formally, the interconnection can be modeled as a connection vector \vec{s} where each element $s_{ji}^k \in \mathbb{Z}$ corresponds with a data dependence vector \vec{d}_{ji}^k in the PRA (indistinct of the vector value). If s_{ji}^k is different of zero, it indicates that a connection should be placed between the indexed variables j and i . If s_{ji}^k is equal to zero (despite that dependence vector \vec{d}_{ji}^k is different of zero), it indicates that a internal PE feedback connection should be placed. The elements of this vector could be obtained as follows:

$$s_{ji}^k = \Phi \vec{d}_{ji}^k \quad (3.18)$$

where Φ is the allocation matrix obtained from the projection vector \vec{u} , and \vec{d}_{ji}^k is the k -th dependence vector between two indexed variables. The m dimensional \vec{s} vector is defined by:

$$\vec{s} = (s_{ji}^1, s_{ji}^2, \dots, s_{ji}^m) \quad (3.19)$$

Besides of the PE interconnection, the amount of delay elements (registers) that should be placed between the PEs interconnections must be determined. The amount of registers is obtained in a similar way as the vector \vec{s} . Given an RDG, a delay vector \vec{r} can be determined by using the start times $\tau(v_i)$ for all vertexes, their operational latencies w_i , their data dependence vector \vec{d} and a scheduler function. If the scheduler is linear, then the start times and the operational latencies are set to zero. Similarly to \vec{s} , the delay connection can be modeled as a vector \vec{r} where each element $r_{ji}^k \in \mathbb{Z}$ corresponds to the k -th dependence vector \vec{d}_{ji}^k in the PRA. The value of r_{ji}^k indicates the amount of time steps that the indexed variable x_i must be stored. It could happen that a delay element should be placed inside of the processor. The elements of the \vec{r} can be obtained as follows:

$$r_{ji}^k = \vec{\lambda} \vec{d}_{ji}^k + \tau(v_j) - \tau(v_i) + w_j \quad (3.20)$$

where w_j is the operational latency of the indexed variable x_j and $\tau(v_j)$ and $\tau(v_i)$ are the computation start times of the indexed variables x_j , and x_i , respectively.

$$\vec{r} = (r_{ji}^1, r_{ji}^2, \dots, r_{ji}^m) \quad (3.21)$$

3.3.1.2 Processor Element Data-Path

Due to the iteration dependent condition $\mathcal{C}^I(\vec{I})$, not all the PEs require the same functional units since they do not perform the entire PRA operations. Recall that one characteristic of the PRA is that the whole algorithm is specified into different pieces, and each one of these pieces are defined by $\mathcal{C}^I(\vec{I})$. As a result of the space-time mapping, these pieces are assigned to different processor points of the processor space \mathcal{P} according to $\mathcal{C}^I(\vec{I})$ of the target polytope. For example, after space-time transformation, only some PEs will perform external I/O memory communication while other PEs will perform a subset of the PRA operations. Therefore, \mathcal{P} can be expressed as a disjoint union of several processor types $\mathcal{P}_i \subseteq \mathcal{P}$ where each set \mathcal{P}_i represents a different processor type.

Taking into account the iteration dependent condition, the data-path for each processor type is synthesized from the reduced dependence graph, since each one of the RDG nodes denotes a different kind of functionality (input, output, propagation, and operational). The operational nodes are directly bound to hardware functional units like adders, multipliers, dividers, etc. Some multiplexers are added in case of there are several nodes corresponding to the same indexed variable. The control lines of such multiplexers are inferred by the $C^I(\vec{I})$ of each indexed variable. Nodes labeled as input or output denote I/O ports from the processor array. These ports are in charge of receiving all data from an external source and of sending data results to an external source. Nodes labeled as propagation are not bound to functional units but to communication buses inside of the PE.

Example 3.10 (MatMul continuation) Following the MatMul example, assume the linear scheduler function $\vec{\lambda}_l = [1, 1, 1]$, and the projection vector $\vec{u} = [0, 1, 0]$. As recapitulation, the non-zero dependences in this example are composed by the vectors $\vec{d}_{yy} = [0, 1, 0]^t$, $\vec{d}_{xx} = [1, 0, 0]^t$, and $\vec{d}_{zz} = [0, 0, 1]^t$. The size of the index space is bound to $N = 5$. Applying the formula 3.18 and 3.20 to each one of the dependence vector \vec{d}_{ji} the connection and delay vector are in the following form:

$$\vec{s} = (s_{xx}^1, s_{yy}^2, s_{zz}^3) = (1, 0, 1)$$

$$\vec{r} = (r_{xx}^1, r_{yy}^2, r_{zz}^3) = (1, 1, 1)$$

These results should be interpreted as follows: each connection element r_{ji}^k in the vector \vec{r} indicates a direct connection between variables j and i . Similarly, each delay element s_{ji}^k in the vector \vec{s} indicates the amount of time steps the indexed variable j should wait for the result of the variable i . In case of s_{yy}^2 an internal feedback connection should be placed with a latency equals to $r_{yy}^2 = 1$. Due to the space-time transformation, the data dependences \vec{d}_{xx} and \vec{d}_{zz} are orthogonal to p_0 and p_1 indexes, respectively; *i.e.* \vec{d}_{xx} is propagated through p_0 direction and \vec{d}_{zz} through p_1 direction.

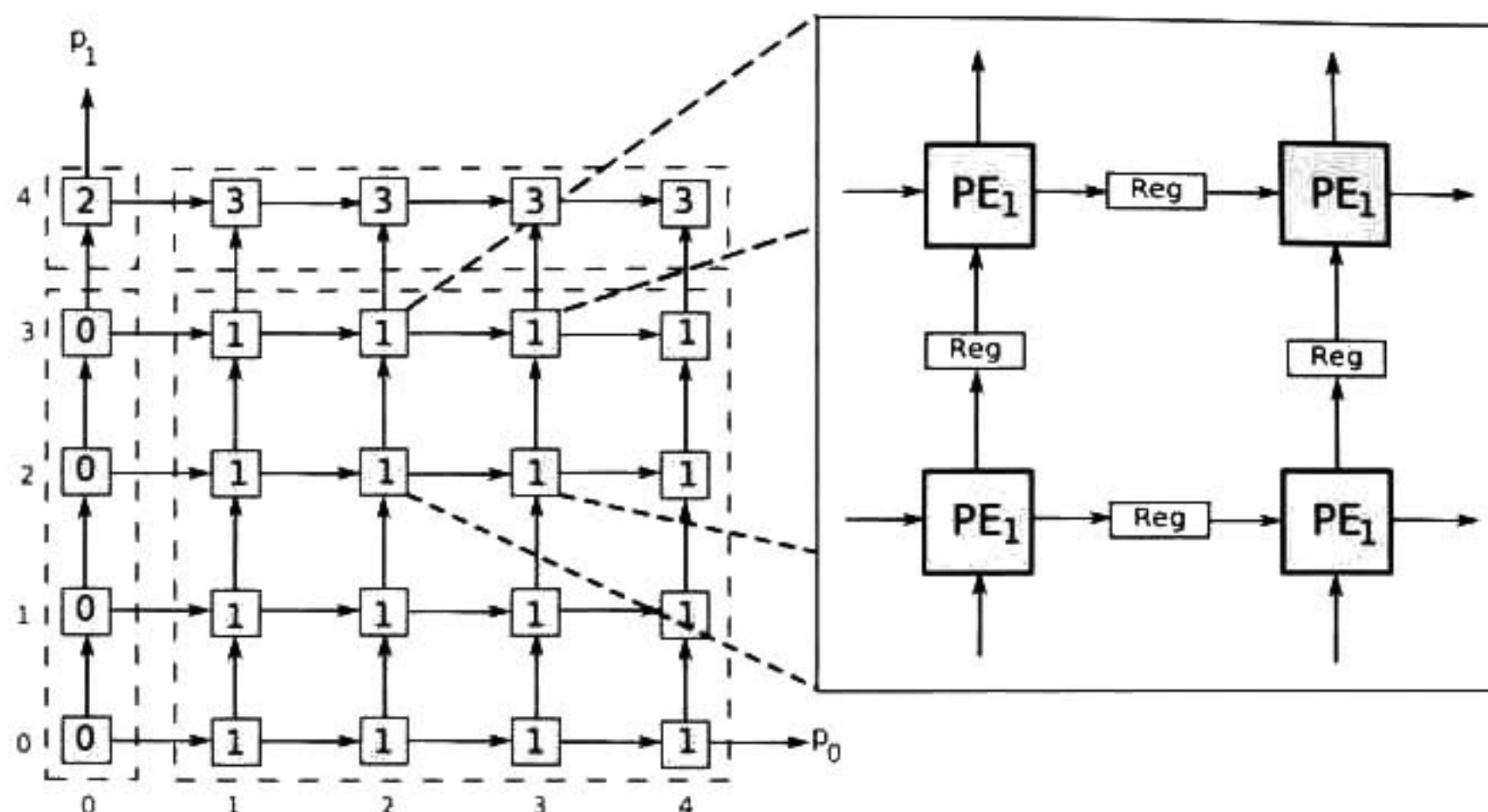


Figure 3.12: Full-size processor array for MatMult algorithm when $N=5$.

Figure 3.12 shows the processor array interconnection resulting from this example. The processing elements are represented by a shaded square box. Some numbers inside of the PEs are added in order to denote the four possible processor types. Note that the four PE types are enclosed in dashed lines and their numbers are assigned in an arbitrary form. The internal data-path of the four processing element types are shown in figure 3.13. Processing element types zero, one, two and three correspond with figures 3.13.a, 3.13.b, 3.13.c and 3.13.d, respectively. The operational nodes in MatMul RDG are bound to a multiplier and to an adder. In the case of indexed variables y , and z , two multiplexers are added for the four processor types, and in the case of variable x a multiplexer is added in for processor types zero and two. Another multiplexer is added in order to put out a final datum denoted by variable C_{out} . The control signal for each multiplexer is generated by a predicate extracted from the iteration dependent condition. The generation of such control signals is in charge of an index controller which maps the predicates shown in table 3.1. Processing element ports for internal connections (inside of processor array) are denoted by x_{in} , z_{in} , x_{out} and z_{out} . Ports for the external connections (outside of processor array) are denoted by A_{in} , B_{out} and C_{out} . The PE ports y_{in} and y_{out} are connected between them as a feedback connection indicated by s_{yy}^2 . Note that the four processor types have the same internal and external ports.

□

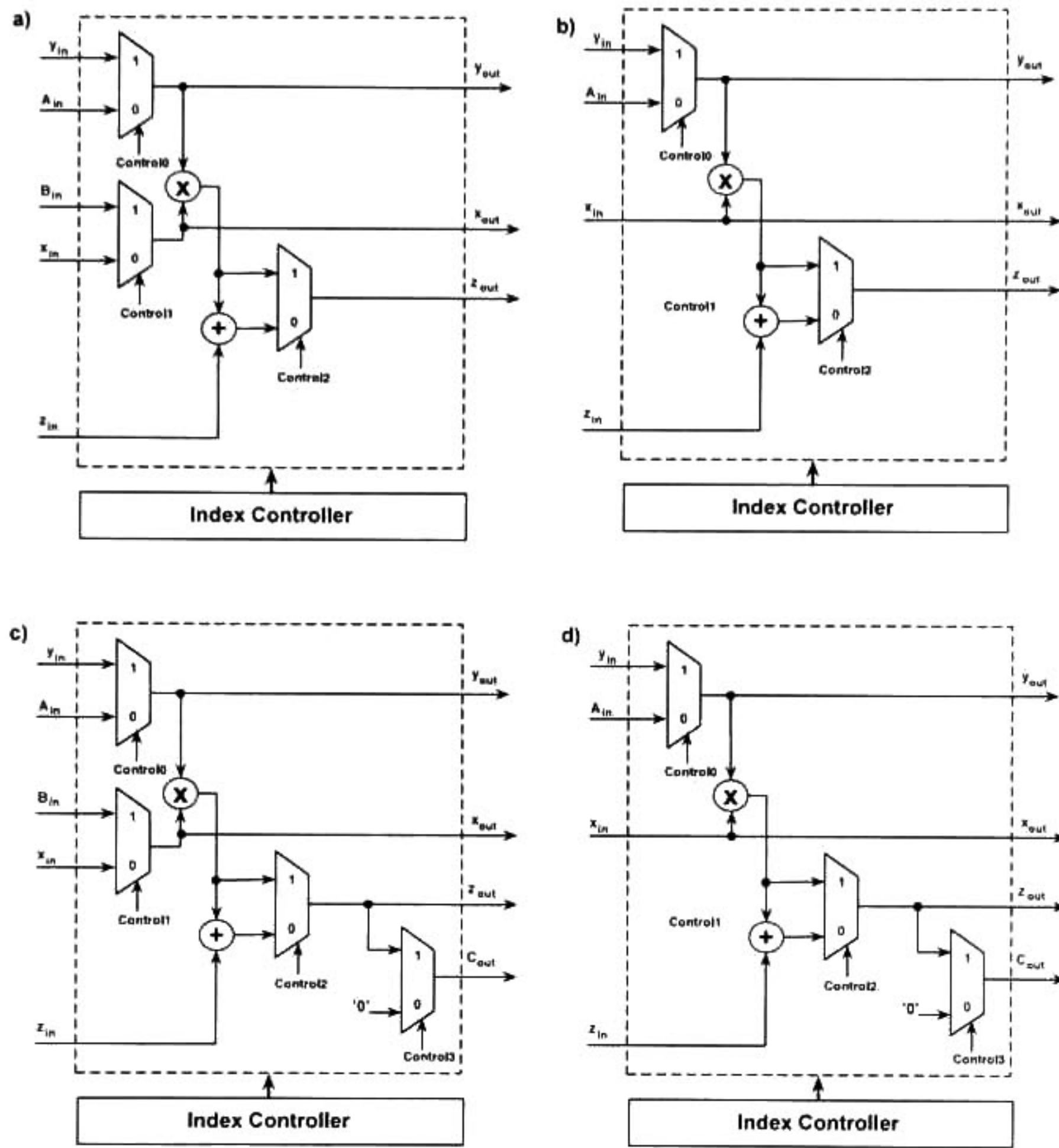


Figure 3.13: Four processing elements types for MatMul processor array.

Predicades	Control Signals
$(j = 0)$	$Control_0 = 1$
$(j > 0)$	$Control_0 = 0$
$(i = 0)$	$Control_1 = 1$
$(i > 0)$	$Control_1 = 0$
$(k = 0)$	$Control_2 = 1$
$(k > 0)$	$Control_2 = 0$
$(k > N-1)$	$Control_3 = 1$

Table 3.1: Activation table for the MatMul PE control signals according to the iteration dependent conditions.

3.3.2 Partitioning of the Processor Space

The space-time transformation provides space and time notions to the indexes of the PRA source polytope, *i.e.* it gives the PEs activation sequence as a function of the i -th time step that is being executed. However, this space-time transformation leads to full-size problem dependent processor arrays unsuitable of being implemented for large iteration spaces. Partitioning techniques help to derive processor arrays independent of the problem size by using a fixed number of processors and mapping the source polytope iteration space to a new processor space. Also, partitioning leads to increment the PEs percentage. Partitioning consists of using congruent tiles for dividing the original iteration space into several iteration spaces that are subsets of the original one. Approaches like LPGS and LSGP are used in order to generate scheduler functions [64]. LPGS approach refers to compute in parallel the iteration points covered by a tile and execute the remaining tiles sequentially. On the other hand, LSGP refers to execute the iteration points inside of the congruent tiles sequentially, but computing the rest of tiles in parallel fashion. With the purpose of exemplifying the LSGP and LPGS partitioning ideas the FIR filter is used in the following example.

Example 3.11 The FIR filter is described by the following equation:

$$Y(i) = \sum_{j=0}^{N-1} A(j) U(i-j) \quad \forall i : 0 \leq i \leq T-1$$

where N denotes the amount of filter taps, $A(j)$ the filter coefficients, $U(i)$ the filter inputs and $Y(i)$ the filter result. The FIR filter iteration space has a rectangular shape and it is defined as:

$$\mathcal{I}_{FIR} = \{[i, j]^t \in \mathbb{Z}^2 \mid 0 \leq i \leq T-1 \wedge 0 \leq j \leq N-1\}$$

The FIR dependence graph is shown in figure 3.14 when $N = 8$ and $T = 8$. In this case, the iteration space is partitioned in eight subsets grouped by the blue boxes. After applying space-time transformation with LSGP partitioning, the blue boxes are mapped as processing elements and each

one of these PEs computes the iteration points which are inside of the blue boxes in a sequential fashion. In contrast, when LPGS approach is applied, each point inside of the blue boxes is mapped to a PE, and these PEs work in parallel while the blue boxes are processed sequentially.

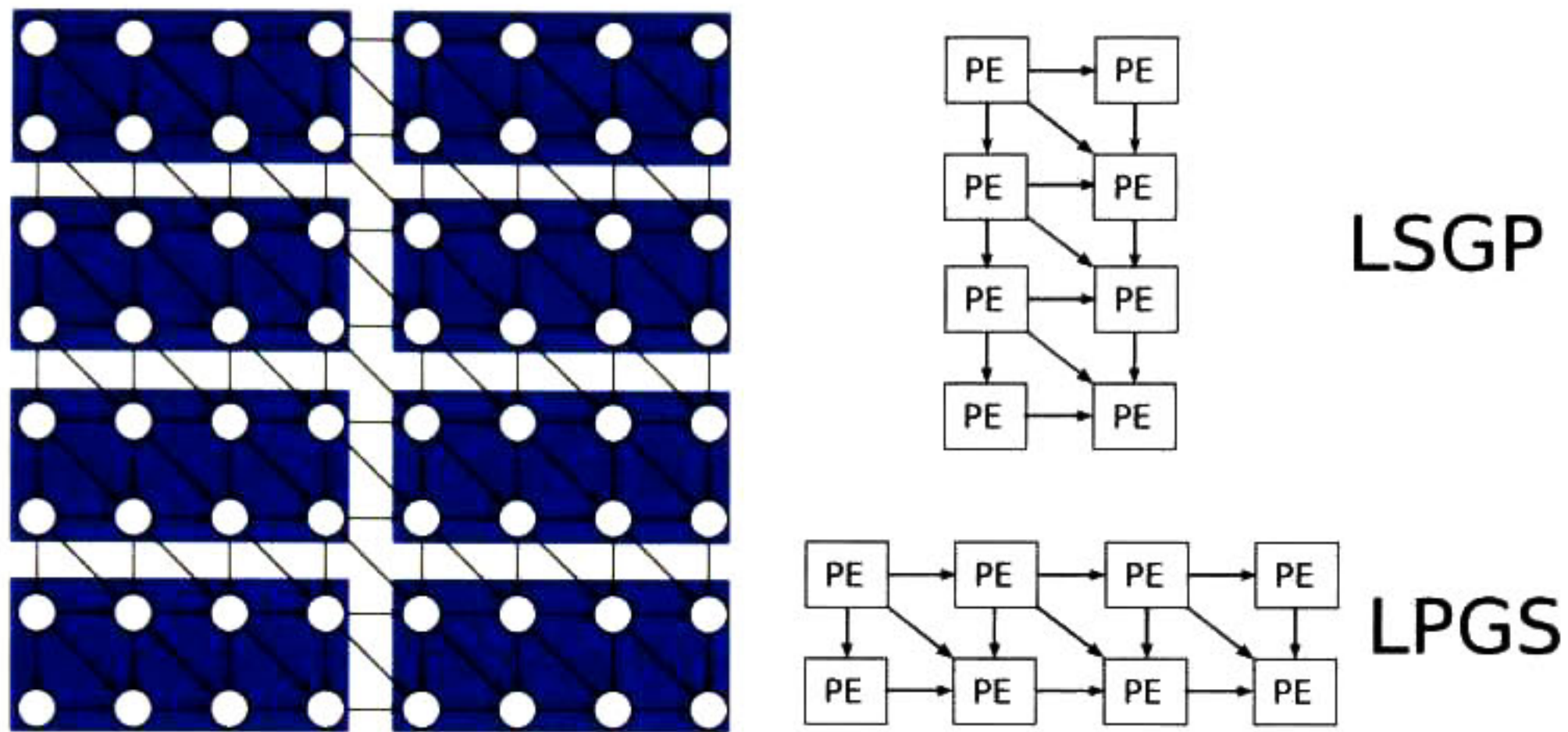


Figure 3.14: Partitioned iteration space for FIR filter. On the right side the LSGP and LPGS approaches.

□

3.3.2.1 Partitioning as Allocation

The LPGS and LSGP could be obtained by a linear programming formulation using partitioning as allocation method. Independently of the partitioning approach applied to a PRA, when partitioning is used as allocation method it happens that:

- The iteration space \mathcal{I} of an algorithm has to be decomposed into two spaces \mathcal{I}_1 and \mathcal{I}_2 , such as $\mathcal{I} = \mathcal{I}_1 + X\mathcal{I}_2$, where $X \in \mathbb{Q}^{n \times n}$ is called tiling matrix. The tiling matrix X defines the tile shape and its size.
- The iteration space dimensionality is increased twice and all indexed variables have to be embedded in higher dimensional iteration space.
- Additional variables have to be added in order to define intra-tile dependences.

By themselves, the iteration space of the sets \mathcal{I}_1 and \mathcal{I}_2 do not provide a particular meaning, *i.e.* it is not specified which one of the partitioning approaches (LPGS or LSGP) are used. It is only by the interpretation of the spaces \mathcal{I}_1 and \mathcal{I}_2 when both acquire a specific meaning. If \mathcal{I}_1 denotes the sequential scanning inside of a tile then a LSGP approach is inferred; but if \mathcal{I}_1 denotes the sequential scanning of the tiles then a LPGS approach should be interpreted. In both cases \mathcal{I}_2 denotes the iteration space that could be executed in parallel. The LPGS and LSGP could be obtained by linear programming formulation [64]. Besides of the operations latencies, and the dependences vector, these both formulations require a loop matrix $L = (l_1, l_2, \dots, l_n) \in \mathbb{Z}^{n \times n}$ which is composed of n loop vectors $l_i \in \mathbb{Z}^n$. This loop matrix determines the sequential scanning of \mathcal{I}_1 . Given a loop matrix, the number of ways for a sequential scanning of \mathcal{I}_1 is $2^n n!$. The 2^n term is the number of the tile corners from where a scheduler may start the computations, whereas the $n!$ term is the number of permutations of the n loop vector [104]. Figure 3.15 shows the eight different sequential scanning order for a 2-dimensional iteration space. Moreover, when the m loop matrixes are explored, the number of different scanning orders is given by $k = m^{2^n n!}$. As a result, exploring all scanning orders demands solving k linear program formulations.

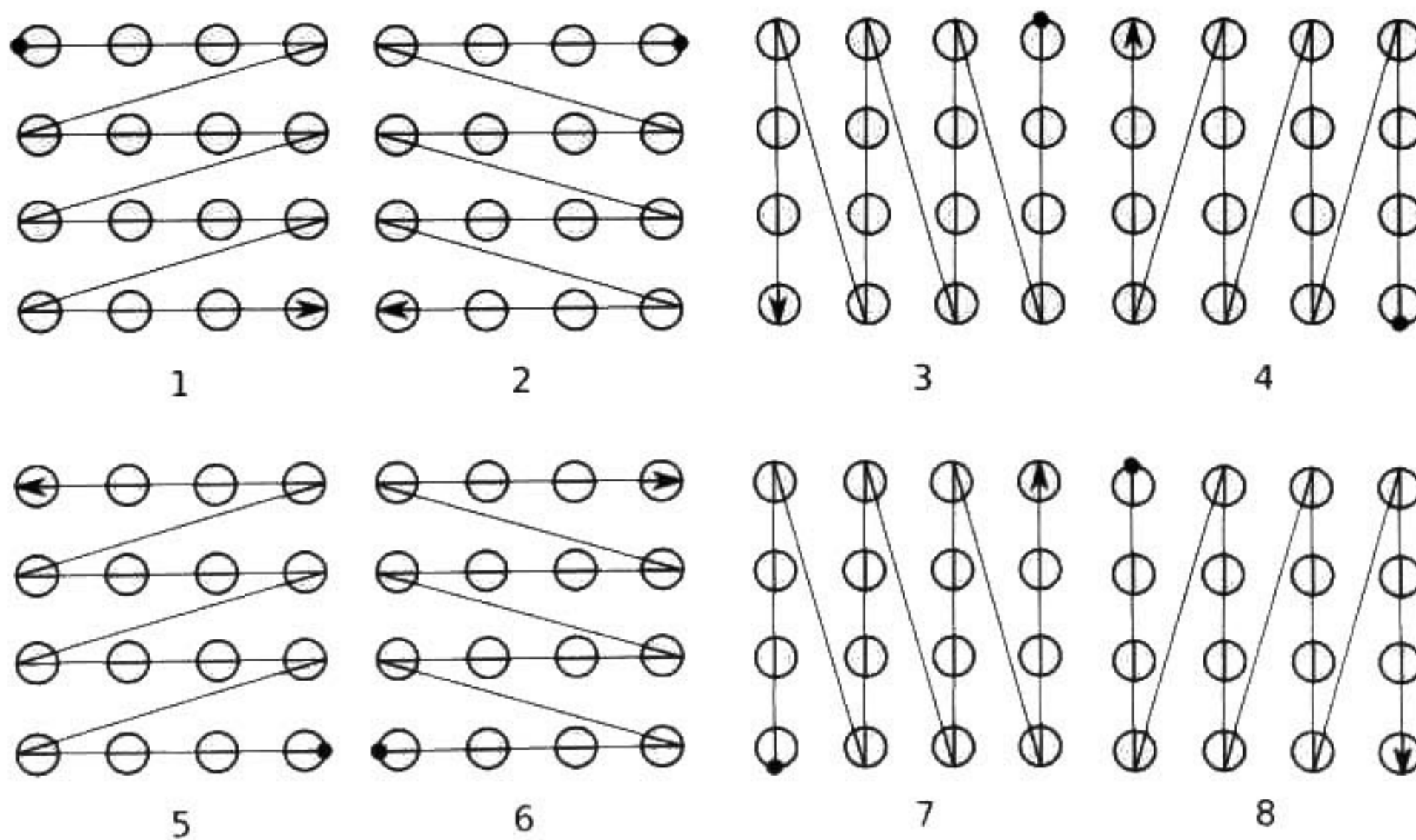


Figure 3.15: The eight different scanning order for a two-dimensional example.

The main difficulty of using linear programming formulation for obtaining LPGS or LSGP schedulers is the number of possible loop matrixes that must be evaluated in order to derive a valid scheduler. For example, it might happen that given a loop matrix L used in a linear programming formulation the solution could be unbounded or unfeasible due the scanning order represented by the matrix L does not respect the PRA data dependences. Although there are some pre-evaluation ways for avoiding the unnecessary formulations [64], they do not eliminate totally the need of evaluation. In this research work, the loop matrixes for partitioning the iteration space are not used. Instead of loop matrixes, strip mining technique is used in order to get the LPGS or LSGP behavior. Besides, the partitioning is performed after space-time transformation instead of performing at the same time partitioning and scheduling as in PARO framework. Using this approach avoids the need for increasing the iteration space dimensionality and consequently embedding the the indexed variables in the higher iteration space. Besides, adding new indexed variables defining intra-tile dependences is avoided.

3.3.2.2 Partitioning with Strip Mining

Strip mining is used to partition one dimension of the iteration space into strips. From the processor arrays point of view, strip mining consists of dividing all dimensions of \mathcal{P} by constant strides, and adding new dimensions for scanning them without the necessity of adding new indexes to the PRA. From the compilers point of view, strip mining divides a single loop into two nested loops; the outer loop steps between consecutive strips, and the inner loop traverses the original iterations within a strip. This process could be repeated as many times as desired for each loop in combination of loop interchange [69]. One dimension of the processor space could be represented by the next inequality:

$$l_k \leq p_k \leq u_k \quad (3.22)$$

where p_k is the k -th index in the processor space, and l_k and u_k are the lower affine bound and upper affine bound, respectively. The bounds of p_k after applying strip mining are obtained by applying the next formula:

$$\left\lfloor \frac{l_k}{SSp_k} \right\rfloor \leq tile_{pk} \leq u_k \quad (3.23)$$

$$\max(l_k, tile_{pk}) \leq p_k \leq \min(u_k, c)$$

where $tile_{pk}$ is the partitioning index whose incremental steps are $SSp_k + tile_{pk}$, p_k is the original index, SSp_k is the size of the strip, and $c = tile_{pk} + SSp_k - 1$. Using this expression for partitioning, the strip boundaries are always parallel to the iteration space axes [69]. In the case of a scheduled PRA, strip mining is applied over the processor space in order to derive a partitioned program version. The scanning sequence of the indexes is changed by the loop interchange transformation. Applying the correct sequence of strip mining and loop interchange the LSGP or LPGS approaches are derived. If after applying strip mining and loop permutation, the time index is left as the outer loop, a LSGP approach is obtained. In contrast, if the new indexes obtained after strip mining are left as the outer loops a LPGS partitioning is obtained. The combination of loop interchange and strip mining is called loop tiling transformation [55]. This dissertation is focused on LPGS partitioning approach.

Example 3.12 (MatMul continuation) Following the MatMul example and using the same parameters ($\vec{\lambda}_l = [1, 1, 1]$, and $\vec{u} = [0, 1, 0]$), the target iteration space after space-time mapping is:

$$\mathcal{J}_{MatMul} = \left\{ \left[\begin{array}{ccc} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & -1 \\ 1 & -1 & -1 \\ 0 & 0 & 1 \\ -1 & 1 & 1 \end{array} \right] \left[\begin{array}{c} t \\ p_0 \\ p_1 \end{array} \right] \leq \left[\begin{array}{c} 0 \\ 3(N-1) \\ 0 \\ 2(N-1) \\ N-1 \\ 0 \\ 0 \\ N-1 \\ N-1 \\ 0 \end{array} \right] \right\}$$

Applying strip mining over the processor space and performing the loop interchange transformation, in order to generate the LPGS approach, the new index space is a five dimensional space whose index points are $[tile_{p_0}, tile_{p_1}, t, p_0, p_1]$ and with the iteration space defined as following:

$$\mathcal{I}_{MatMul_P} = \left\{ \begin{array}{c} \left[\begin{array}{ccccc} -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 1 \end{array} \right] \begin{array}{c} tile_{p_0} \\ tile_{p_1} \\ t \\ p_0 \\ p_1 \end{array} \leq \left[\begin{array}{c} 0 \\ (N-1) \\ 0 \\ (N-1) \\ 0 \\ 3(N-1) \\ 2(N-1) + SS_{p_0} - 1 \\ 2(N-1) + SS_{p_1} - 1 \\ N-1 + SS_{p_0} + SS_{p_1} - 2 \\ 2(N-1) \\ N-1 + SS_{p_1} - 1 \\ 0 \\ 0 \\ 0 \\ N-1 \\ 0 \\ N-1 + SS_{p_0} - 1 \\ N-1 \\ 0 \\ 0 \\ 0 \\ N-1 \\ SS_{p_1} - 1 \end{array} \right] \end{array} \right\}$$

A geometrical interpretation of the new partitioned index space is that the processor space is divided into tiles. Each tile contains a subset of index points of the processor space. Due the new indexes $tile_{p_0}$ and $tile_{p_1}$ are placed before the time space, the set of index points of the processor space inside of a tile can be viewed as physical processors, meanwhile the original processor space can be interpreted as logical processors mapped to the physical processors. Figure 3.16 shows the processor space partitioned when $N = 8$ and $SSp_0 = SSp_1 = 4$ (left size), and the partitioned array of 4×4 PEs (right size). Note that the index points in the processor space are denoted by circles whereas the PEs of the processor array are denoted by boxes. Also, note that it could happen that tile size does not fit exactly in the processor space. In this example, the scanning order of the tiles is done from left to right direction. If a different scanning order is desired, the order to the indexes $tile_{p_0}$ and $tile_{p_1}$ should be changed by loop interchange transformation.

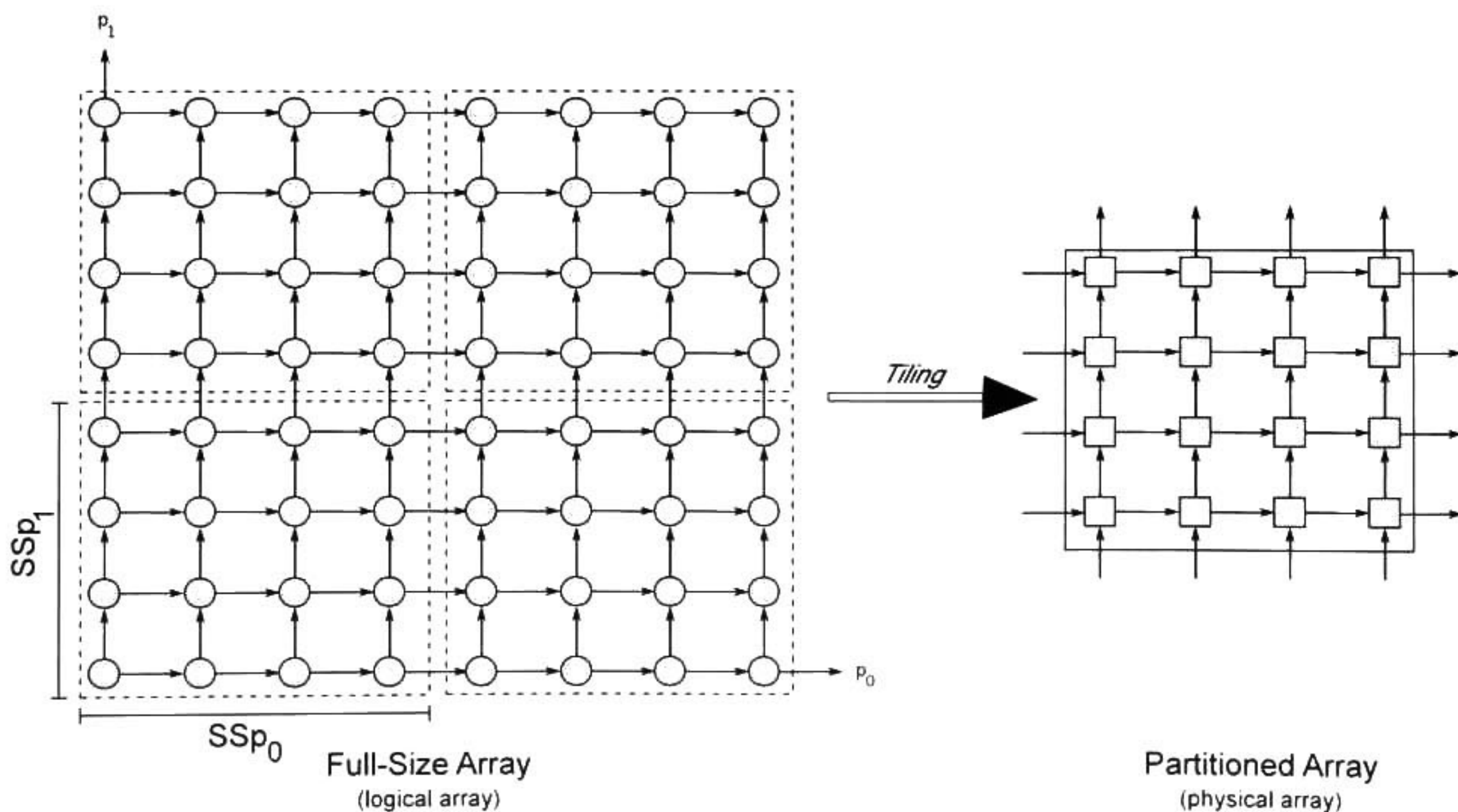


Figure 3.16: Partitioned MatMul processor space by strips of size four and its corresponding physical processor array.

□

Example 3.13 (Cholesky continuation) Continuing the Cholesky example, figure 3.17 shows the processor space partitioned when $N = 8$ and $SSp_1 = SSp_1 = 4$ (left size) and the partitioned array of 4×4 PEs (right size). Similarly to the MatMul example, index points in the processor space are denoted by circles and the PEs of the processor array are denoted by boxes. Note that there are some tiles which are not "full" of index points of the processor space. In such case, the PEs which do not exist in the $tile_{p_0}$ and $tile_{p_1}$ dimensions should not be activated. Moreover, due the Cholesky processor space is not rectangular there are empty tiles which should not be considered when $tile_{p_0}$ and $tile_{p_1}$ dimensions are scanned. In fact, using strip mining leads to ignoring these empty tiles, i.e. they are not scanned.

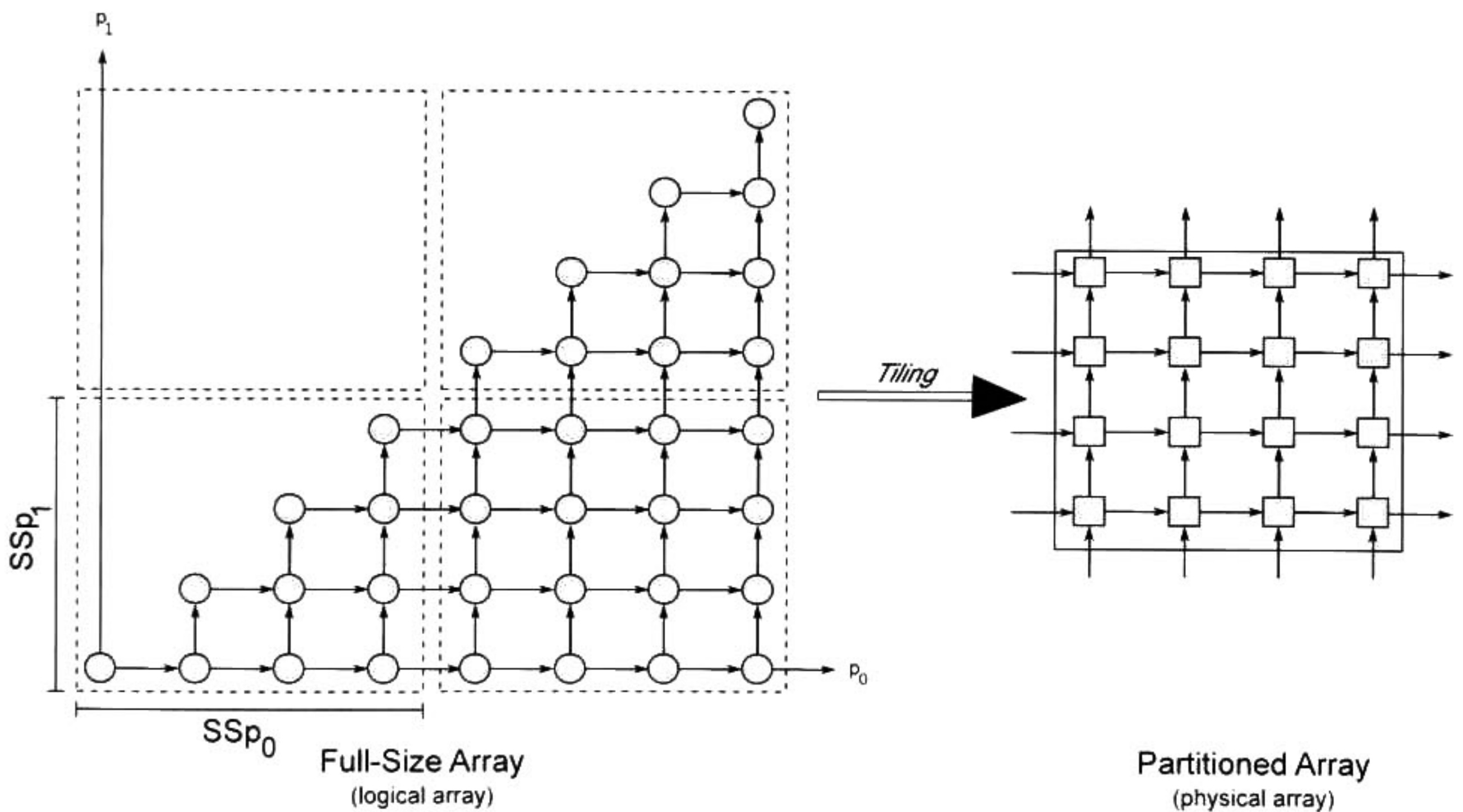


Figure 3.17: Partitioned Cholesky processor space by strips of size four and its corresponding physical processor array.

3.3.2.3 Iteration Space of Partitioned Target Polytopes

The previous examples show how to derive the processor array topology, and the data-path for different types of PEs given a set of data dependences, a scheduler, and an allocation functions. The allocation function is responsible for binding the computations to processor elements and the scheduler function provides the activation pattern of these processors. However, depending on the resulting target polytope iteration space, the processing element activation pattern might be regular or irregular. If the target polytope has a rectangular shape, the number of times that a PE is activated will be the same for all the PEs inside of the array. Algorithms like MatMul, matrix-vector and FIR filter are some examples of iteration space with rectangular shapes after space-time mapping. On the other hand, if the target polytope iteration space is non-rectangular, the number of times that a PE is activated will be different for each PEs inside of the array. Some examples of non-rectangular iteration spaces are Back/Forward substitution, Cholesky, LU and QR algorithms.

In addition, when partitioning is applied, providing the processor array PEs activation pattern for non-rectangular iteration spaces is even more complex, because the number of time steps for which a PE is activated will change while the tiles are being scanned. Besides, recall that when the processor space is partitioned, the original space \mathcal{P} is divided into congruent tiles that are subsets of the original space, and the PEs inside of a tile are viewed as physical processors, meanwhile the original processor space is interpreted as logical array mapped to the physical PEs. In the case of the full-size array, the PEs are activated one or more consecutively times, but once a PE stops it remains inactive during the rest of the processor array computations. On the other hand, in a processor array generated by partitioning, an inactive PE might be reused during an algorithm computation. Such possibility is caused from the non-rectangularity of the processor space, since there are occasions when a physical processor is mapping a non-existing processor index point of \mathcal{P} (like in the Cholesky example). The rectangular and non-rectangular iteration spaces, as well as the changing of the number of the activation time steps should be kept in mind when the control scheme is derived from the target polytope. More details are explained in Chapter 4.

Applying strip mining over the processor space and performing the loop interchange transformation, in order to generate the LPGS approach, the new index space is a five dimensional space whose index points are $[tile_{p_0}, tile_{p_1}, t, p_0, p_1]$ and with the iteration space defined as following:

$$\mathcal{I}_{MatMul_P} = \left\{ \begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} tile_{p_0} \\ tile_{p_1} \\ t \\ p_0 \\ p_1 \end{bmatrix} \leq \begin{bmatrix} 0 \\ (N-1) \\ 0 \\ (N-1) \\ 0 \\ 3(N-1) \\ 2(N-1) + SS_{p_0} - 1 \\ 2(N-1) + SS_{p_1} - 1 \\ N-1 + SS_{p_0} + SS_{p_1} - 2 \\ 2(N-1) \\ N-1 + SS_{p_1} - 1 \\ 0 \\ 0 \\ 0 \\ N-1 \\ 0 \\ N-1 + SS_{p_0} - 1 \\ N-1 \\ 0 \\ 0 \\ 0 \\ N-1 \\ SS_{p_1} - 1 \end{bmatrix} \right\}$$

A geometrical interpretation of the new partitioned index space is that the processor space is divided into tiles. Each tile contains a subset of index points of the processor space. Due the new indexes $tile_{p_0}$ and $tile_{p_1}$ are placed before the time space, the set of index points of the processor space inside of a tile can be viewed as physical processors, meanwhile the original processor space can be interpreted as logical processors mapped to the physical processors. Figure 3.16 shows the processor space partitioned when $N = 8$ and $SSp_0 = SSp_1 = 4$ (left size), and the partitioned array of 4×4 PEs (right size). Note that the index points in the processor space are denoted by circles whereas the PEs of the processor array are denoted by boxes. Also, note that it could happen that tile size does not fit exactly in the processor space. In this example, the scanning order of the tiles is done from left to right direction. If a different scanning order is desired, the order to the indexes $tile_{p_0}$ and $tile_{p_1}$ should be changed by loop interchange transformation.

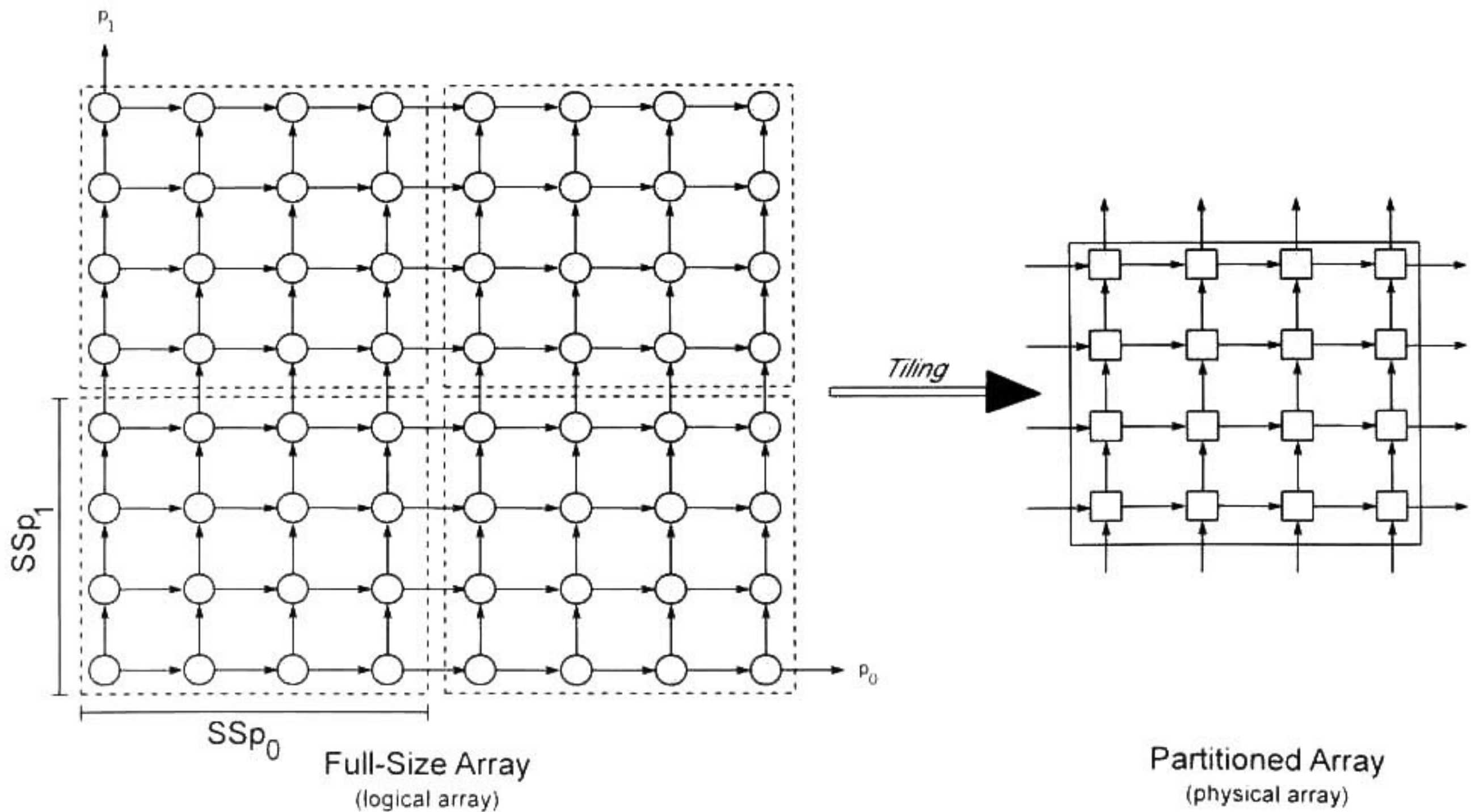


Figure 3.16: Partitioned MatMul processor space by strips of size four and its corresponding physical processor array.

□

Example 3.13 (Cholesky continuation) Continuing the Cholesky example, figure 3.17 shows the processor space partitioned when $N = 8$ and $SSp_1 = SSp_1 = 4$ (left size) and the partitioned array of 4×4 PEs (right size). Similarly to the MatMul example, index points in the processor space are denoted by circles and the PEs of the processor array are denoted by boxes. Note that there are some tiles which are not "full" of index points of the processor space. In such case, the PEs which do not exist in the $tile_{p_0}$ and $tile_{p_1}$ dimensions should not be activated. Moreover, due the Cholesky processor space is not rectangular there are empty tiles which should not be considered when $tile_{p_0}$ and $tile_{p_1}$ dimensions are scanned. In fact, using strip mining leads to ignoring these empty tiles, *i.e.* they are not scanned.

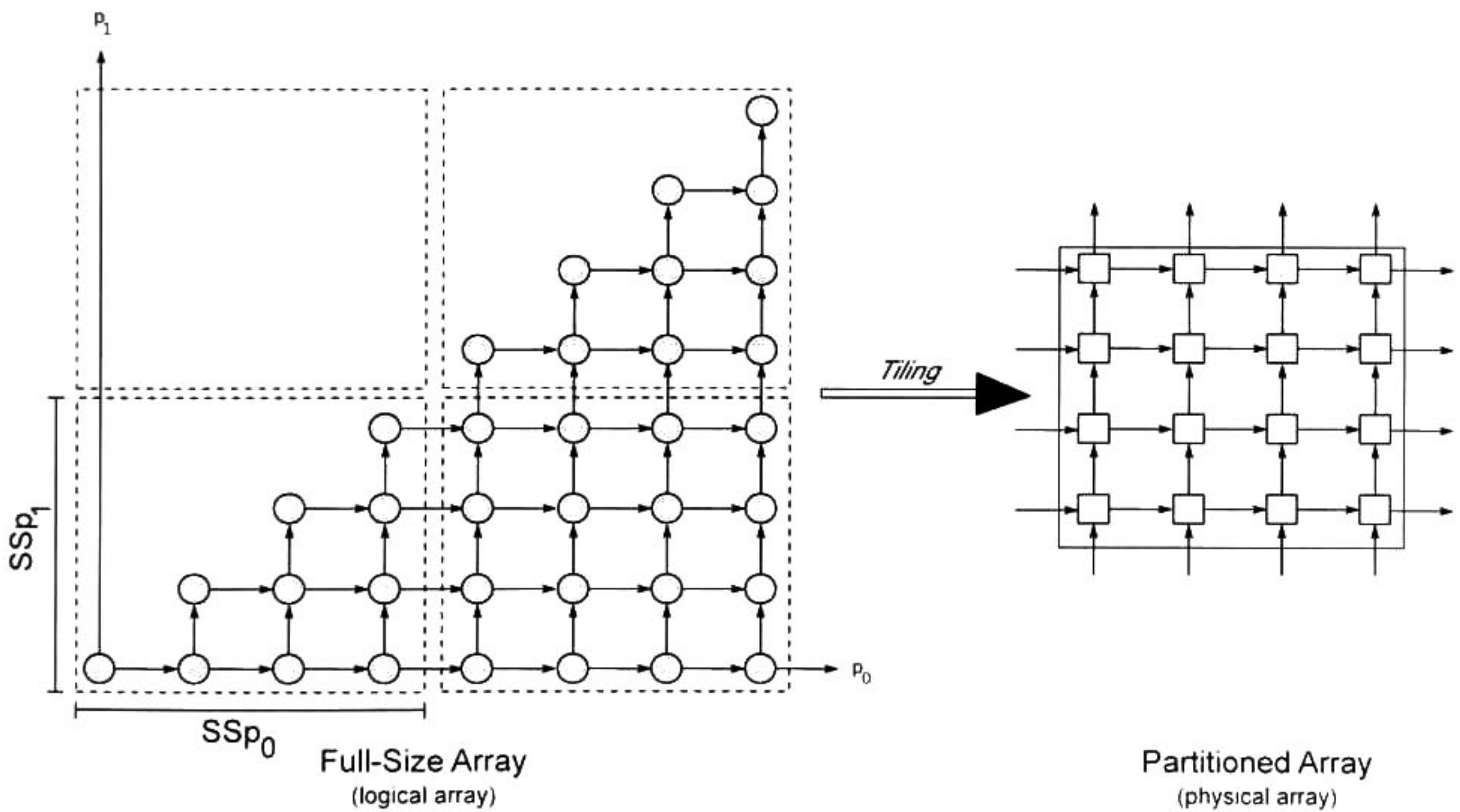


Figure 3.17: Partitioned Cholesky processor space by strips of size four and its corresponding physical processor array.

3.3.2.3 Iteration Space of Partitioned Target Polytopes

The previous examples show how to derive the processor array topology, and the data-path for different types of PEs given a set of data dependences, a scheduler, and an allocation functions. The allocation function is responsible for binding the computations to processor elements and the scheduler function provides the activation pattern of these processors. However, depending on the resulting target polytope iteration space, the processing element activation pattern might be regular or irregular. If the target polytope has a rectangular shape, the number of times that a PE is activated will be the same for all the PEs inside of the array. Algorithms like MatMul, matrix-vector and FIR filter are some examples of iteration space with rectangular shapes after space-time mapping. On the other hand, if the target polytope iteration space is non-rectangular, the number of times that a PE is activated will be different for each PEs inside of the array. Some examples of non-rectangular iteration spaces are Back/Forward substitution, Cholesky, LU and QR algorithms.

In addition, when partitioning is applied, providing the processor array PEs activation pattern for non-rectangular iteration spaces is even more complex, because the number of time steps for which a PE is activated will change while the tiles are being scanned. Besides, recall that when the processor space is partitioned, the original space \mathcal{P} is divided into congruent tiles that are subsets of the original space, and the PEs inside of a tile are viewed as physical processors, meanwhile the original processor space is interpreted as logical array mapped to the physical PEs. In the case of the full-size array, the PEs are activated one or more consecutively times, but once a PE stops it remains inactive during the rest of the processor array computations. On the other hand, in a processor array generated by partitioning, an inactive PE might be reused during an algorithm computation. Such possibility is caused from the non-rectangularity of the processor space, since there are occasions when a physical processor is mapping a non-existing processor index point of \mathcal{P} (like in the Cholesky example). The rectangular and non-rectangular iteration spaces, as well as the changing of the number of the activation time steps should be kept in mind when the control scheme is derived from the target polytope. More details are explained in Chapter 4.

3.3.2.4 *Intermediate Memories*

As a result of partitioning the processor space and using an LPGS approach, the usage of intermediate memories at the processor array borders is required. The purpose of these memories is to store data produced by the PEs placed at the array borders while the tiles are being scanned. Later, during the algorithm execution, the data stored in these memories will be used by the following tiles. The implementation of these memories could be done either by FIFO memories, or by DMA schemes. These intermediate memories could be classified in two different types according to the temporal locality concept. The two types of intermediate memories are called intermediate memory L1 and L2; and their main difference consist of intermediate memories L1 stores data that is likely to be used in a shorter period of time than data stored in intermediate memories L2. Details of these memories are more explained in Chapter 5.

3.4 Summary

This section has covered the background required in the synthesis of processor arrays. Through some examples, it has been exemplified the synthesis process taking as input a piecewise regular algorithm until deriving an hardware architecture specification. Concepts like polytope, piecewise regular algorithm, iteration space, scheduler, iteration interval, allocation, time and processor spaces, and partitioning have been introduced. Also, it has been shown how to generate the processing elements and the interconnection of such PEs in order to construct the processor array by using scheduler and allocation functions. By partitioning the processor space, it could be derived processor arrays that are independent of the problem size. However, it emerges a situation that in a full-size processor array is not present: the generation of activation signals for the PEs reutilization. In a full-size processor array the PE could be activated one or more consecutively times, but once the PE stops, the PE remains inactive during the rest of the algorithm computations. On the other hand, in a processor array generated by partitioning techniques, an inactive PE could be reused during an algorithm computation. In this case, it is needed to generate the control signals that activate the

correct PEs at the proper time and select the operation that the PE should perform. The following chapter is advocated to describe a proposed solution for the control signal generation.

4

Control Scheme

Once the processor array data-path has been derived, the generation of the control signals for enabling the processor and selecting their operations at different time instants is needed. Chapter three has covered all concepts previous to the control generation stage, showing how from an input PRA specification a parallel execution order for each index point of the iteration space could be derived. Later using some algebraic expressions, the processor array interconnection topology is derived. Basically, when a processor array is derived by using LPGS approach the PEs activation patten differs from the full-size implementation. This chapter is focused on describing a hybrid controller, which uses centralized and distributed modules, in charge of providing the control signals for processor arrays derived from the design methodology shown in figure 4.1. The controller described is able to generate control signals for processor spaces which are rectangular and non-rectangular, and at the same time it provides problem size independency. After describing the control scheme, a controller for a Cholesky decomposition is presented as case of study in order to exemplify the control scheme ideas.

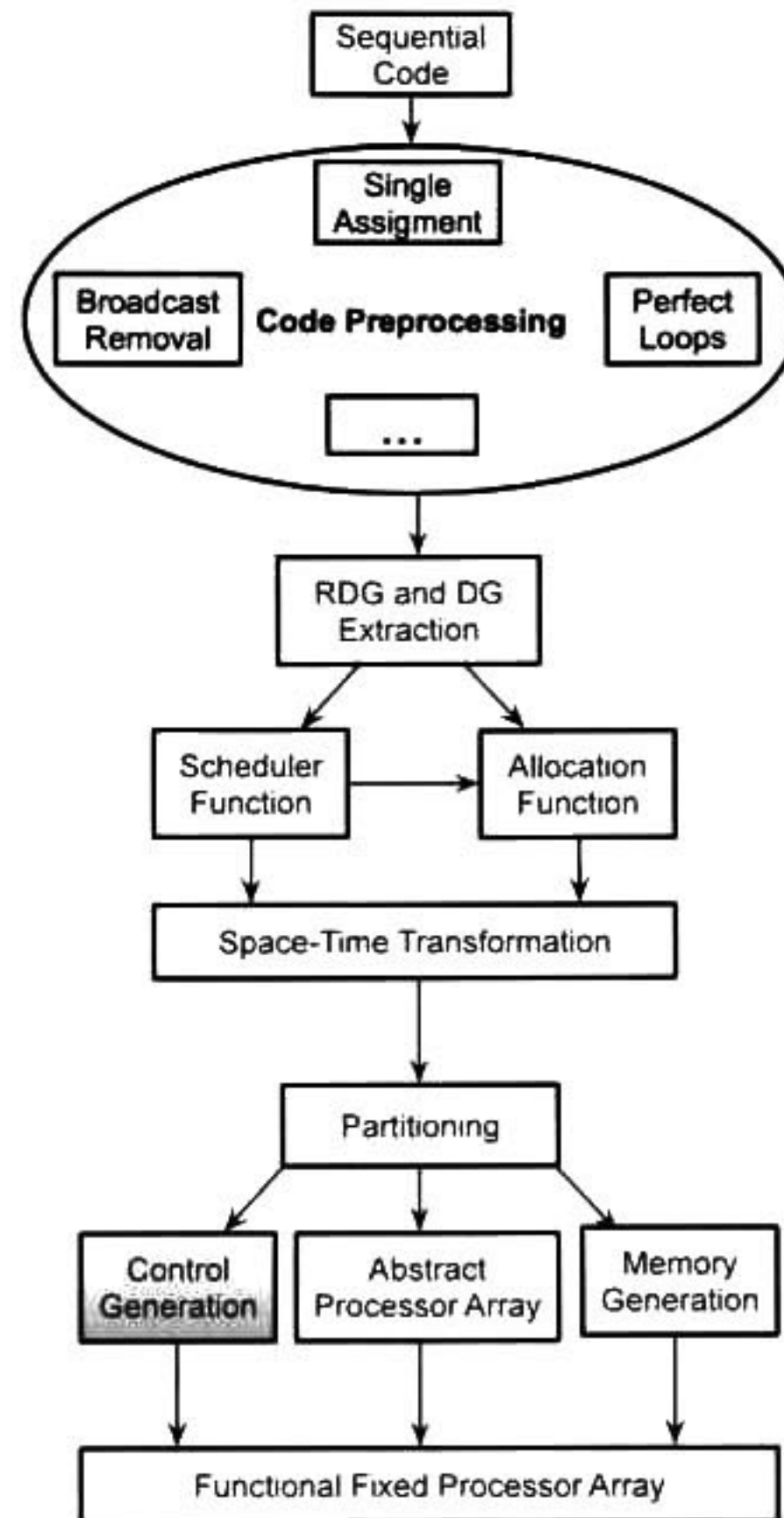


Figure 4.1: Design flow methodology followed in the polytope model, highlighting the control generation.

4.1 Hybrid Control Scheme

The problem of generating the control signals for processor arrays has been previously tackled in different works using distributed, centralized or a combination of both control facilities [17], [23], [43], [61], [102], [121]. Although these works tackle the problem of control signals generation for processor arrays, they are limited in two different aspects: the generation of controllers only for a specific problem size, and for algorithms with rectangular iteration spaces. In this section, the motivation and some additional considerations for the control signal generation are presented.

4.1.1 Motivation

As stated before, the problem of control signal generation for processor arrays has received much attention from the synthesis community using different control models [102]. In a centralized model, all the control signals are generated by a global controller whose control data are broadcast to the entire processor array, like in MMAAlpha synthesis tool [36]. On the other hand, in a distributed control model the control signals are generated by specialized control units distributed exactly as the processor array topology, like in PICO-NPA synthesis tool [72] and in the framework presented by Uday *et. al* [23]. Also, intermediate solutions combining centralized and distributed control models have been used like in PARO [65]. However, these works are limited to generate controllers only for a specific problem size, and consequently their arrays produced are able to solve only a particular problem size. If the problem size for which the arrays were targeted changes, the activation sequence changes too, and as a result a new processor array must be synthesized in order to generate a new activation sequence. This limitation is originated because loop bounds after space-time mapping are statically determined due to the assumption of fixed tile sizes [44]. Therefore these loop bounds are pre-calculated during synthesis time, facilitating the control signal generation for a dedicated processor array in form of counters modules. If the processor array is designed to solve a set of problem sizes, the loop bounds after space-time mapping must be determined dynamically at run-time.

In addition to the loop bounds limitation, the aforementioned works have focused on showing implementations for algorithms whose loop bounds form rectangular shapes. When full-size arrays are derived, the control signal generation for these arrays is straightforward (independently of the algorithm loop bound shape) since a PE could be activated one or more consecutively times, but once the PE stops, it remains inactive during the rest of the array computation. However, when processor arrays using LPGS approach are derived, an inactive PE could be reused when a different tile is being scanned, resulting in a different activation pattern in different tile iterations. As explained in subsection 3.3.2.3, depending on space-time mapping, the processing element activation pattern

might be regular or irregular. If the target polytope has a rectangular shape, the number of times that a PE is activated will be the same for all the PEs inside of the array. But, if the target polytope iteration space is non-rectangular, the number of times that a PE is activated will be different for each PE inside of the array. Furthermore, providing the processor array PEs activation pattern for non-rectangular iteration spaces is more complex than in the case of rectangular iteration spaces, because the number of time steps for which a PE is activated changes while the tiles are being scanned. In this sense, if the processor arrays are required to provide a support for non-rectangular iteration spaces, irregular activation pattern must be considered.

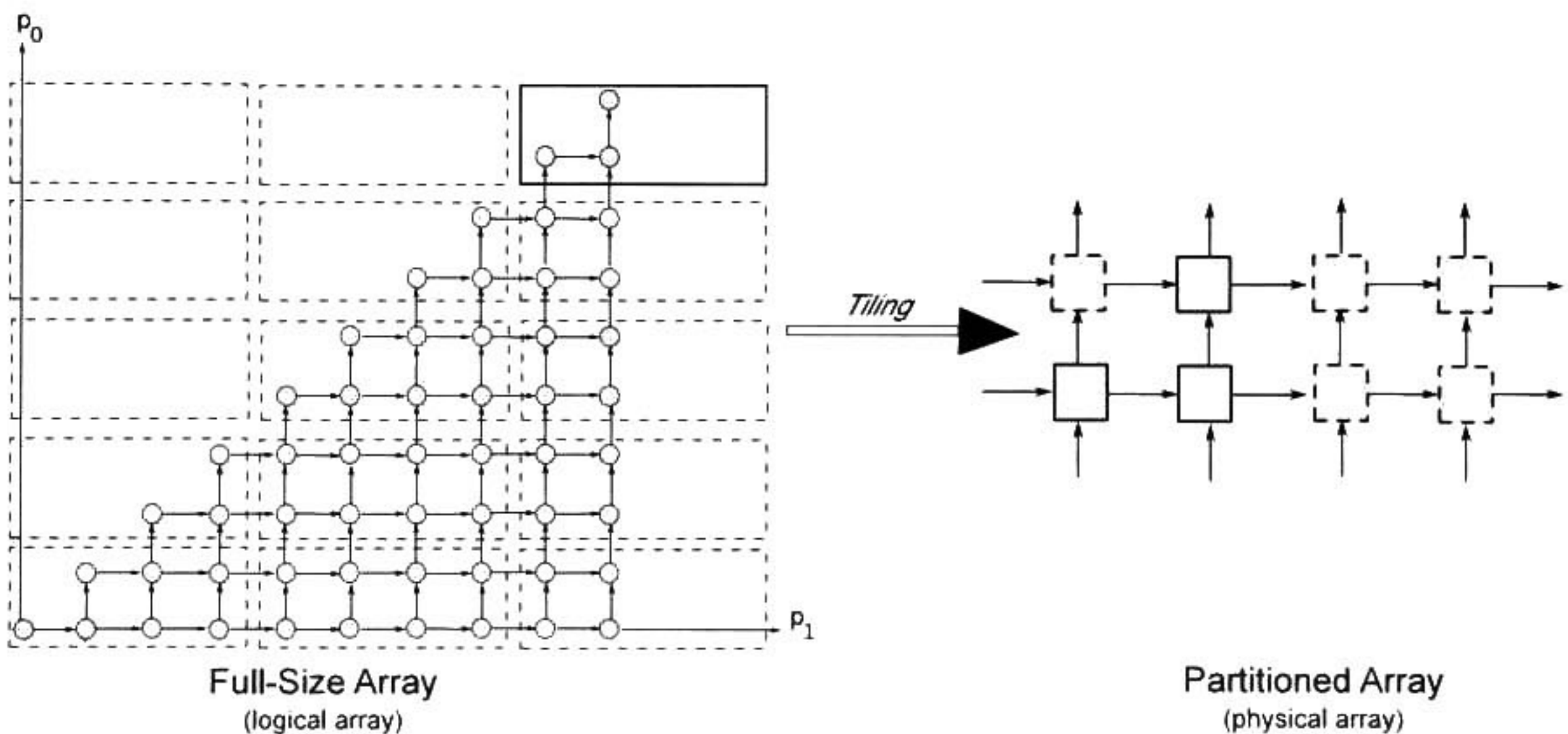


Figure 4.2: Example of invalid mapping from the logical array to the physical array after tiling a non-rectangular processor space. Invalid mapping is denoted by dashed boxes in right size figure.

Besides of the irregularity of the activation pattern in non-rectangular iteration spaces when LPGS partitioning approach is used, it arises another problem which is not presented in rectangular spaces: the correct mapping of processor space index points (logical array) to PEs in the partitioned array (physical array). Figure 4.2 exemplifies this problem showing a non-rectangular processor space grouped into sets of size 2×4 (left size), which are later mapped to a physical processor array of the same size (right size). In this figure, the solid box line in the left size denotes the current tile, whereas solid box lines and dashed box lines in right figure denote a valid mapping from the logical array to the

physical array and an invalid mapping, respectively. In a processor array generated by partitioning, a PE might be or might be not activated during a tile iteration. Such possibility is caused from the non-rectangularity of the processor space, because there are occasions when a physical processor (right size of figure 4.2) tries to map a non-existing processor index point of logical array (left size of figure 4.2). In order to provide a correct processor space mapping to PEs, a run-time mechanism for detecting when a PE maps correctly an iteration point \vec{I} from the processor space \mathcal{P} is required. Together, supporting irregular activation patterns and the correct processor space mapping to PEs must be considered for deriving processor arrays for algorithms with non-rectangular iteration spaces.

Also, generating the control signals (according to the scheduling function and the iteration interval), supporting the non-rectangular processor spaces, and the irregularity of activation patterns should be kept in mind when the processor array control units are derived for algorithms with non-rectangular iteration spaces. In order to provide such support, including hardware modules able to generate the irregular activation pattern and capable of detecting when a valid mapping is occurring during a tile iteration is required. Additionally, if it is desired to provide a problem size independency for these arrays, including the problem size as a parameter in the controller for calculating the loop bounds at run time is required too. In the following subsection, the description of a control scheme for the processor array control signal generation taking into account the aforementioned requirements is presented.

4.1.2 Control Scheme Description

The control scheme presented in this chapter is based on the combination of centralized and distributed control facilities, called hybrid or intermediate control model [43]. In this hybrid model, the most costly hardware and repetitive operations are placed in a central module in order to reduce possible overhead introduced if such operations were placed in a distributed way. In this sense, the proposed centralized control module generates the scanning order of the tile and time indexes obtained after applying strip mining to the processor space \mathcal{P} .

The distributed part of the proposed control scheme is based on two variations of the distributed control model. In the literature, the first variation is called distributed pre-stored control model, and it consists of generating the control signals for each PE by using pre-stored control data in form of look-up-tables (LUTs), FSMs or dedicated counters with comparators, without the necessity of communication among the control units. In the second variation, the control units are in charge of propagating the control signals to their neighbors; and if it is required, these control units modify such signals before of their propagation. This last model is called propagation control model. The proposed control scheme uses both variations in order to provide the activation pattern and a run-time mechanism for detecting valid mappings. Such support is given by intercommunicating locally the control units, and by using counters, FSMs, and comparators for generating the activation pattern, which respects the scheduler function $\bar{\lambda}$ and the iteration interval P .

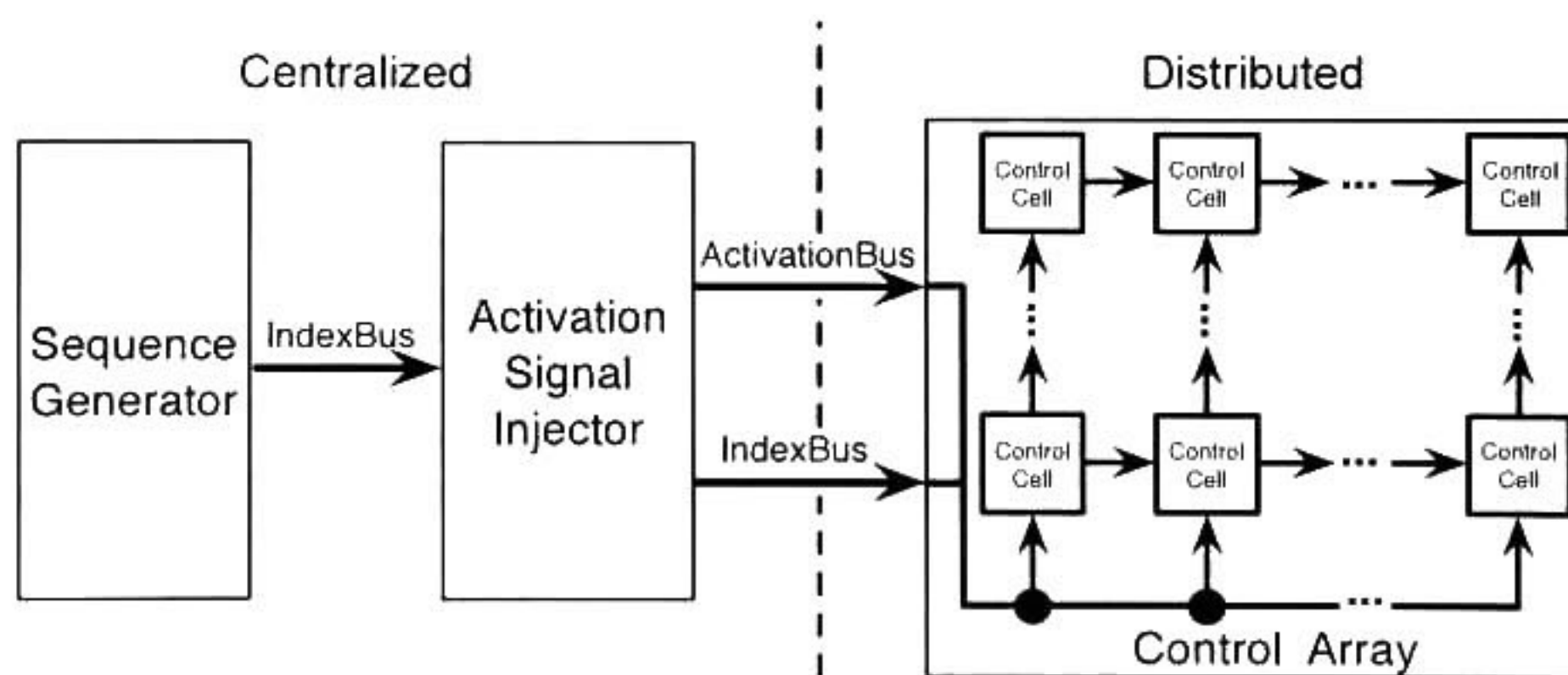


Figure 4.3: Processor array hybrid controller block diagram. Dashed line divides the centralized and distributed modules.

Summarizing the centralized and distributed control modules functionalities, the key idea behind the controller is having a centralized unit in charge of scanning the partitioned processor space \mathcal{P} (represented by the tile indexes), and generating the time index. Later, these indexes are decoded in order to generate an activation signal that is sent to the distributed control units. This activation signal is propagated through the control units, which decide if the signal should be sent to a neighbor PEs in order to generate the activation pattern. The propagation of this signal is done in time steps

equals to the iteration interval P and respecting the scheduler obtained from linear programming formulation. Figure 4.3 shows the block diagram of the control scheme dividing the centralized and distributed parts. In this diagram, the generation of the tile and time indexes (*IndexBus*), and their decoding into an activation signal (*ActivationBus*) are performed by a sequence generator and an activation-signal injector modules, respectively. A set of control cells, which form a control array following the interconnection topology of the processor array, are in charge of propagating the activation signal and generating the activation pattern. In the following section, the description of these hardware modules is presented.

4.2 Hybrid Control Architecture

4.2.1 Sequence Generator

The module in charge of generating the tile and time indexes sequentially for scanning each partition created by strip mining the processor space and generating the time steps inside such partition is the sequence generator. At first glance, this module might be thought as a set of simple counters connected in a cascade fashion. However, it is not true at all, because the bounds of the target polytope after space-time mapping are affine functions of tile and time indexes. Besides, the incremental steps of each counter are not necessarily unitary, since the indexes in charge of scanning the processor space ($tilep_0$ and $tilep_1$) are incremented according to their respective strip size parameters. As a result an straightforward approach for using counters can not be used. For example, the bounds of n -dimensional target polytope could have the following form:

$$\max(0, \left\lceil \frac{tilep_0 - (N - 1)}{2} \right\rceil) \leq t \leq \min(N - 1, \left\lfloor \frac{tilep_1}{2} \right\rfloor)$$

where $tilep_0$ and $tilep_1$ are the indexes introduced after partitioning the processor space \mathcal{P} ,

t is the time index scanning the time space \mathcal{T} , and N is the problem size. Note that the loop bound expressions are affine functions of the indexes $tilep_0$ and $tilep_1$. If the processor array is supposed to provide support for a set of problem sizes, the N parameter must be evaluated at execution time. In order to provide support for the set of problem sizes, adding combinational logic for evaluating affine expressions is required. In this sense, the sequence generator module is composed by a set of counter-like sub-modules connected in a cascade fashion. Between each pair of counters, a combinational logic (Max/Min sub-module) in charge of evaluating the maximum and minimum expressions presented in the target polytope bounds is required. This sub-module has adders, multipliers, combinational logic to evaluate floor and ceiling functions, and comparators for maximum and minimum functions. In case of loop bounds expressions contain division operation, this could be carried out by a right shifter or by a fixed-point multipliers.

From the Max/Min sub-modules, the maximum and minimum expressions for the upper and lower bounds of each tile and time indexes are obtained. The lower bound indicates the starting index value, whereas the upper bound detects when an index has reached the last value. These minimum expressions are helpful for initializing the counter-like sub-modules whereas the maximum expressions help to stop them. Besides of loading the lower limit (as initialization value), and stopping the count, when the upper bound has been reached, the counter-like sub-modules increment their count given the strip size at time steps equal to the iteration interval P . In summary the counter-like sub-module should:

1. Load an input index value in order to start counting (lower bound from the a *max* function).
2. Know when to stop counting (upper bound from the *min* function).
3. Count at different incremental steps according to the strip size parameters SSp_0 and SSp_1 .
4. Enable the counting at time steps equals to the iteration interval P
5. Hold the counting until an inner counter has reached its maximum value.

The internal architecture of the counter-like sub-module is shown in figure 4.4. This sub-module has an asynchronous *Load*, *Reset* and *Hold* signals. Also, the counter-like sub-module has two inputs for the lower and upper bounds, which comes from the Max/Min sub-module. The *STEP* parameter indicates the counting incremental steps, *i.e.* the size of the strips obtained by partitioning, and its value is set at synthesis time. The architecture of the counter-like sub-module consists of an adder, two 2-1 multiplexers, a less-or-equal comparator unit, and a modulo unit. Since the counting incremental steps are different to one, it is necessary to anticipate when a maximum value is reached in order to avoid an overflow. For example, suppose that the next values are present in the counter cell: $STEP = 3$, $LowerLimit = 0$, and $UpperLimit = 10$. The valid sequence of values is $\{0, 3, 6, 9\}$. Note that the $UpperLimit$ value is not reached exactly and it should be avoided the possible overflow to 12. The modulo unit is in charge of controlling a multiplexer, which selects between the actual count value or the next count value avoiding such overflow. In case that $STEP$ parameter is a two-power multiple, the modulo unit is replaced by a set of AND gates. Due to AND gates require less hardware resources than a modulo unit, in the rest of this work there are only considered strip sizes equal to a two-power multiple, *i.e.* processor arrays whose size is a two-power multiple.

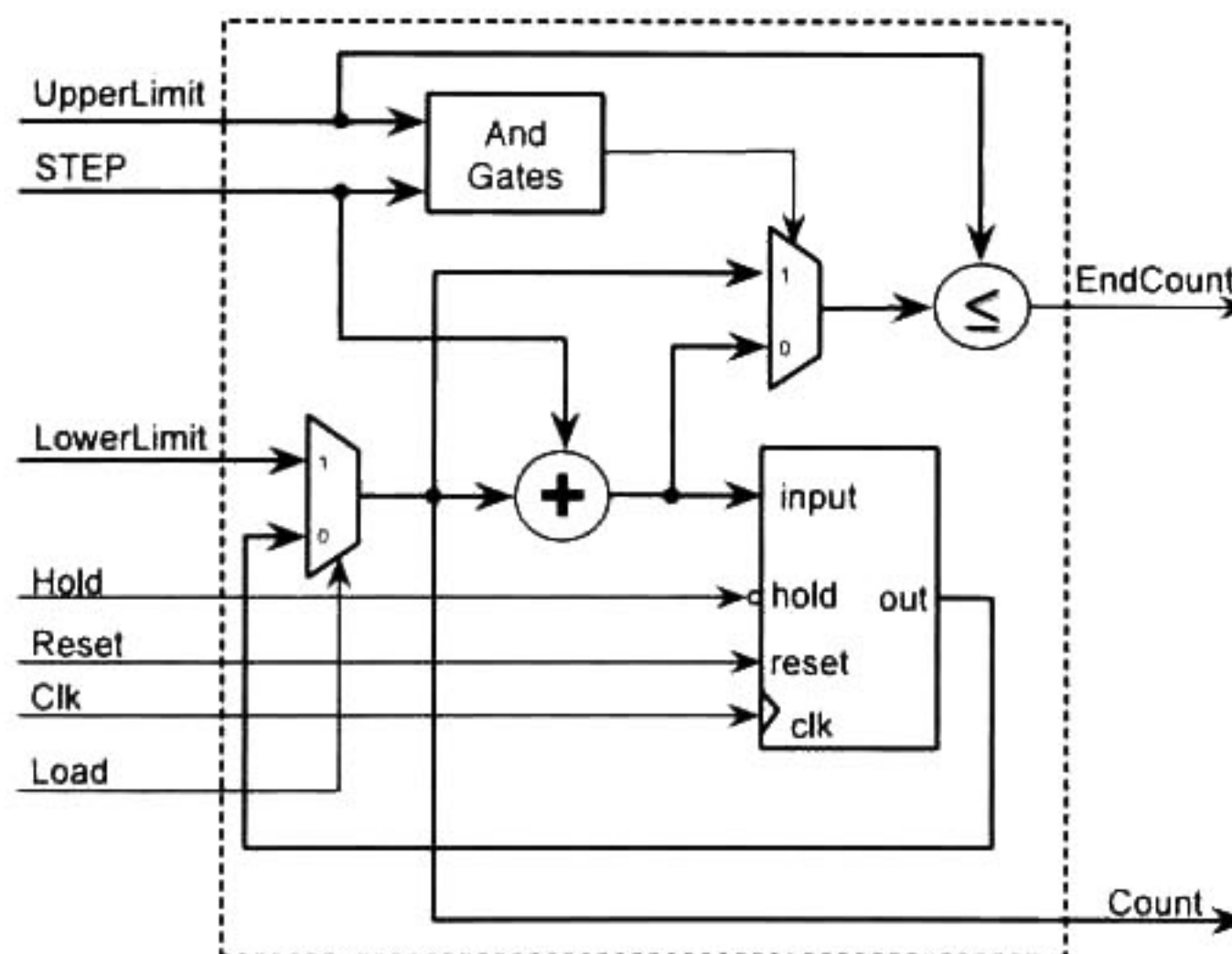


Figure 4.4: Counter-like sub-module internal architecture.

For each non-processor index presented in the partitioned target polytope, a pair of a Max/Min and counter-like sub-module is required. The advantage of using combinational logic in the Max/Min sub-modules is that if the bounds of the partitioned processor space were changed (by altering the scanning order or by a different space-time transformation T), only by changing the Max/Min expression, the sequence generator is able to generate the new tile and time indexes. Moreover, by adding h -pair of counter-like and Max/Min sub-modules the functionality of h non-processor indexes can be achieved. Figure 4.5 presents the sequence generator architecture when $h = 3$. The counter-like sub-modules labeled as $Counter_0$, $Counter_1$, and $Counter_2$ generate the $tilep_0$, $tilep_1$, and t indexes, respectively; with a word width of W_c for all these indexes. Note that the *Hold* signal from the inner most counter is connected to a counter which establishes the iteration interval P . Also, note that the outer counter loads the *LowerLimit* value when it is necessary to compute a new problem instance, therefore the *Load* signal is high-active only when a new computation is required. The *EndCount* signal from this outer counter is high-active only when the generation of the non-processor indexes has ended.

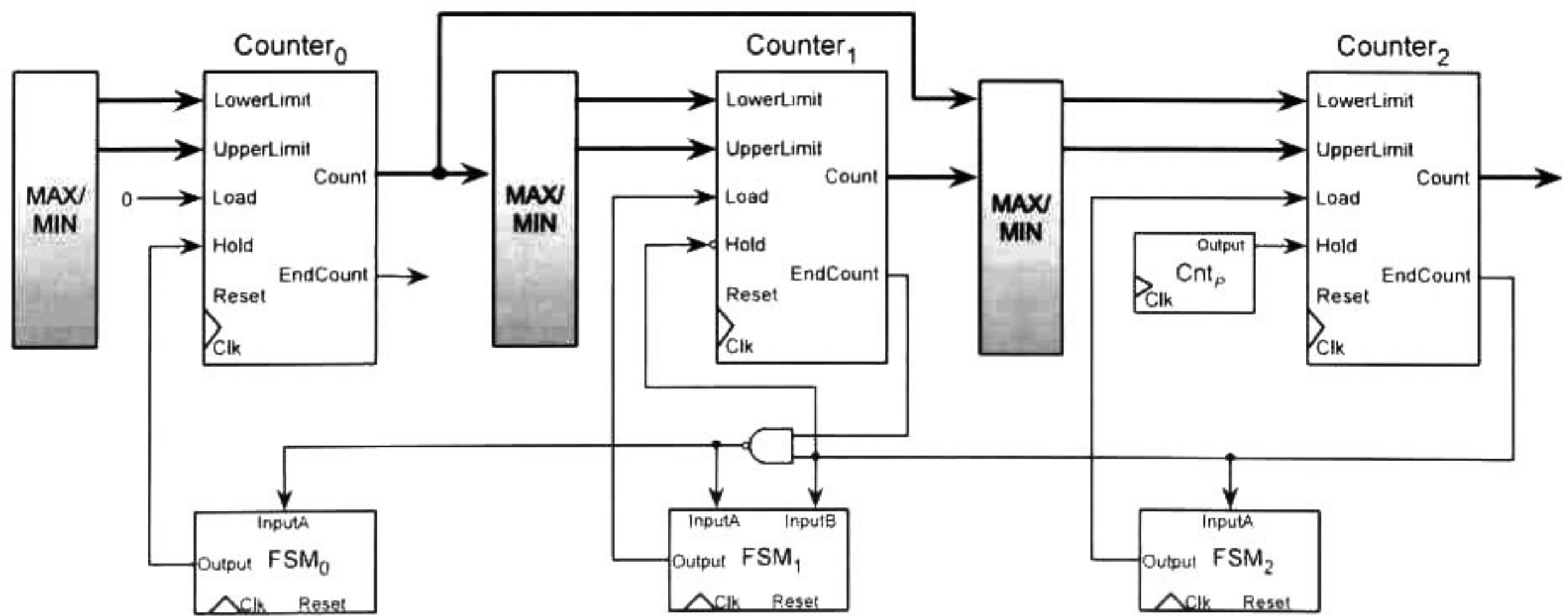


Figure 4.5: Counter-like sub-modules interconnection when $h = 3$. The combinational Max/Min sub-modules are grey-shaded.

The FSM_0 , FSM_1 and FSM_2 presented in figure 4.5 are in charge of activating the *Hold* or *Load* signal for $Counter_0$, $Counter_1$, and $Counter_2$ counter-like submodules, respectively. Although it is

not explicitly shown in figure 4.5, the transitions made by the three FSMs are done according to the iteration interval P , therefore, each FSM includes a P -modulo counter. Basically, the functionality of these FSMs is activating their corresponding control signals (*Hold* or *Load*) by knowing when an inner counter has reached its last value. For this purpose the FSMs use as input the *EndCount* signals from their inner counters. The transition diagram for the Mealy FSM_0 and Mealy FSM_2 is the same as shown in figure 4.6, whereas the transition diagram of the Mealy FSM_1 is shown in figure 4.7. Note that the "don't care" transitions are done according the iteration interval too.

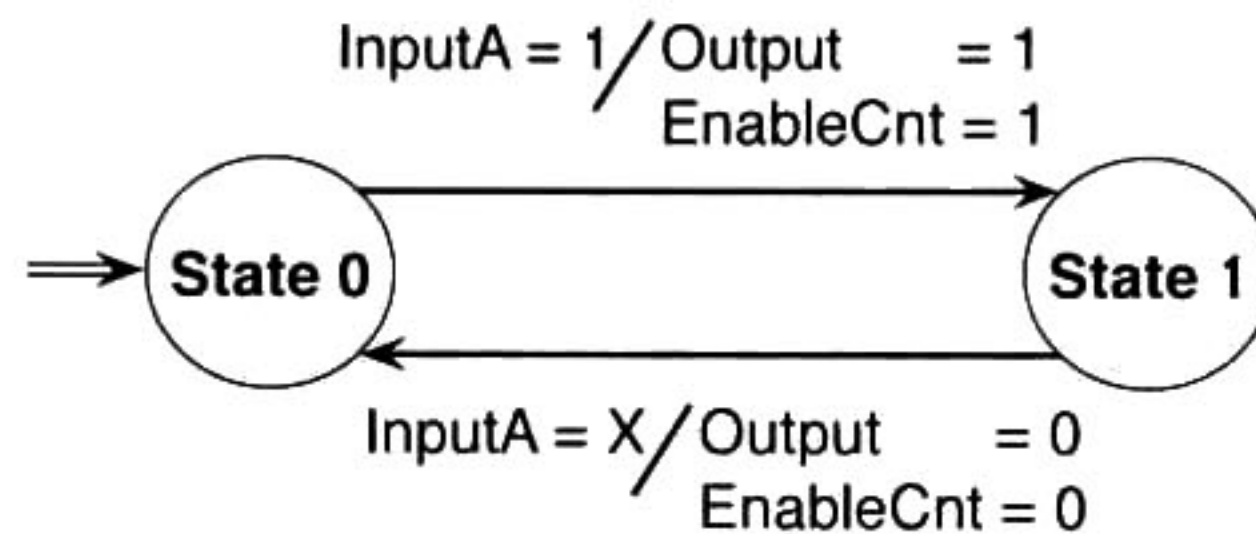


Figure 4.6: Mealy finite state machine transition diagram in charge of activating *Hold* and *Load* signals for the $Counter_0$ and $Counter_2$ sub-modules, respectively.

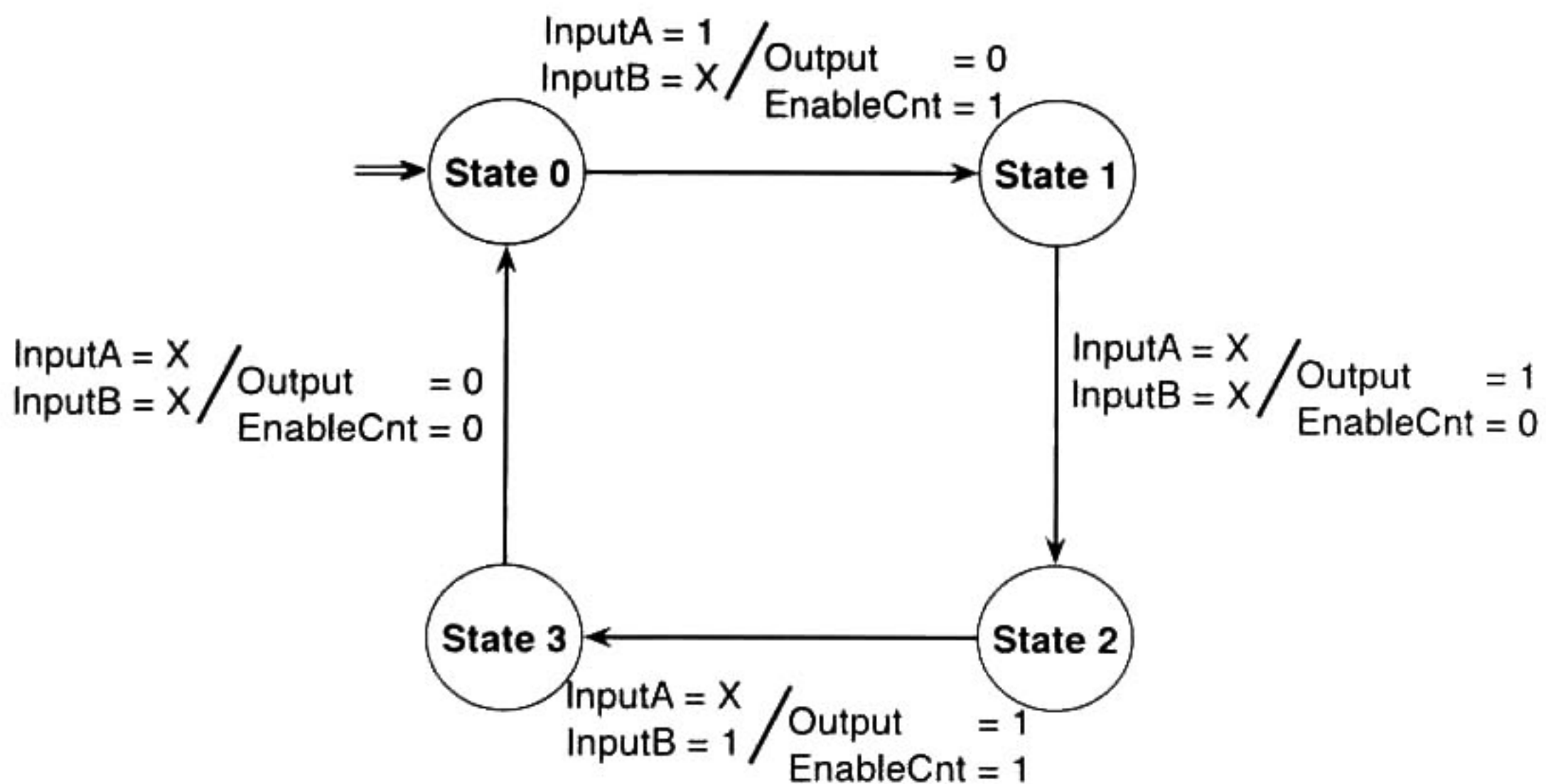


Figure 4.7: Mealy finite state machine transition diagram in charge of activating the *Load* signal for the $Counter_1$ sub-module.

4.2.2 Activation-Signal Injector

The second element of the hybrid control is the activation-signal injector. This module is in charge of selecting which PE, in the bottom row of the control array, must be activated when a new tile is being scanned. This is accomplished by injecting an activation signal to the control array using the *ActivationBus*. Also, an *IndexBus* composed by the indexes $tile_{p0}$, $tile_{p1}$, t , and the size of the problem N is injected to the control array. The reason for injecting the tile and time indexes to the control array is that all PEs must know what tile iteration is being executed at a given time and what is the size of the problem that is being solved. This is helpful for generating the irregular activation patterns, and for detecting correct mappings from the processor space to PEs. Besides, remember that these data provide the information needed for controlling the PEs in the processor array data-path (see section 3.3.1.2). Figure 4.8 depicts the interconnection between the sequence generator, the activation-signal injector and the processor array, showing only the *IndexBus*.

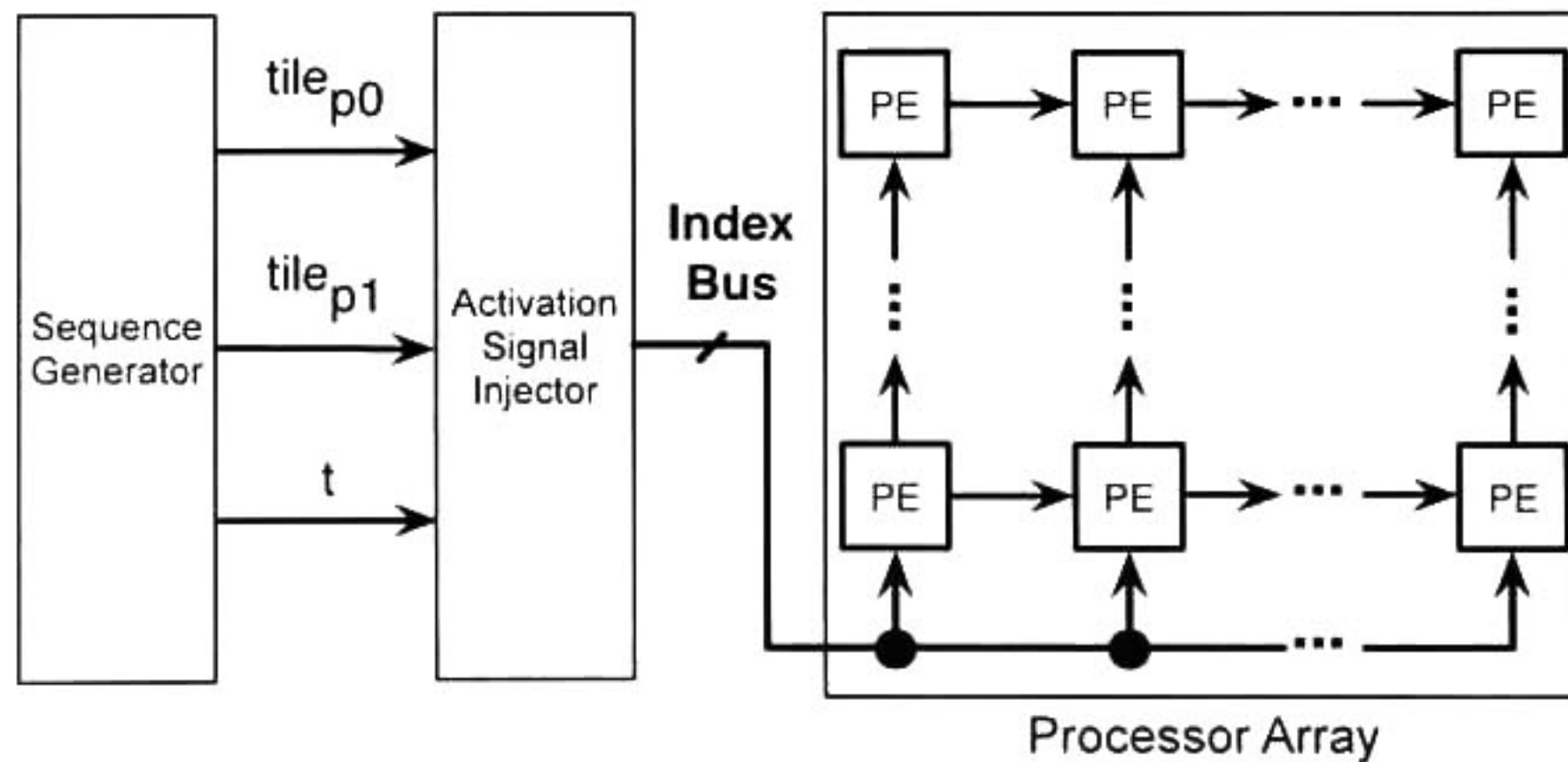


Figure 4.8: Injection of *IndexBus* to the processor array from the activation signal injector.

In order to inject the activation signal (using the *ActivationBus*), and the tile and time indexes (using the *IndexBus*) at the correct PE, it is needed to know which PE will be the first one to be used. Determining the first PE used in a tile iteration helps to start the processor array activation pattern. The first PE activated, when a new tile iteration has started, is obtained by evaluating

the expressions of the lower bounds of the processor space, *i.e.* p_0 and p_1 indexes. Remember that the bounds of processor space are affine functions of the indexes ($tile_{p_0}$, $tile_{p_1}$ and t), therefore these index values are needed in order to evaluate the lower bounds expression of the processor space. Intuitively, by performing such evaluation, it is possible to know the PE where the activation signal should be injected. This leads to the idea of having two sub-modules inside of the activation-signal injector: a set of Max sub-modules in charge of mapping the lower bounds expression of the processor space; and a $W_c : SS_{p_1}$ decoder sub-module for decoding the p_1 index value in order to know the coordinate where the activation signal must be injected (bottom row of the control array). The output width of the $W_c : SS_{p_1}$ decoder sub-module is equal to the SS_{p_1} parameter. Figure 4.9 shows the activation-signal injector module internal architecture, where the combinational Max sub-modules are highlighted. Similarly to the Max/Min sub-modules in the sequence generator, by changing the Max expression, the activation-signal injector is able to provide support for different space-time mappings.

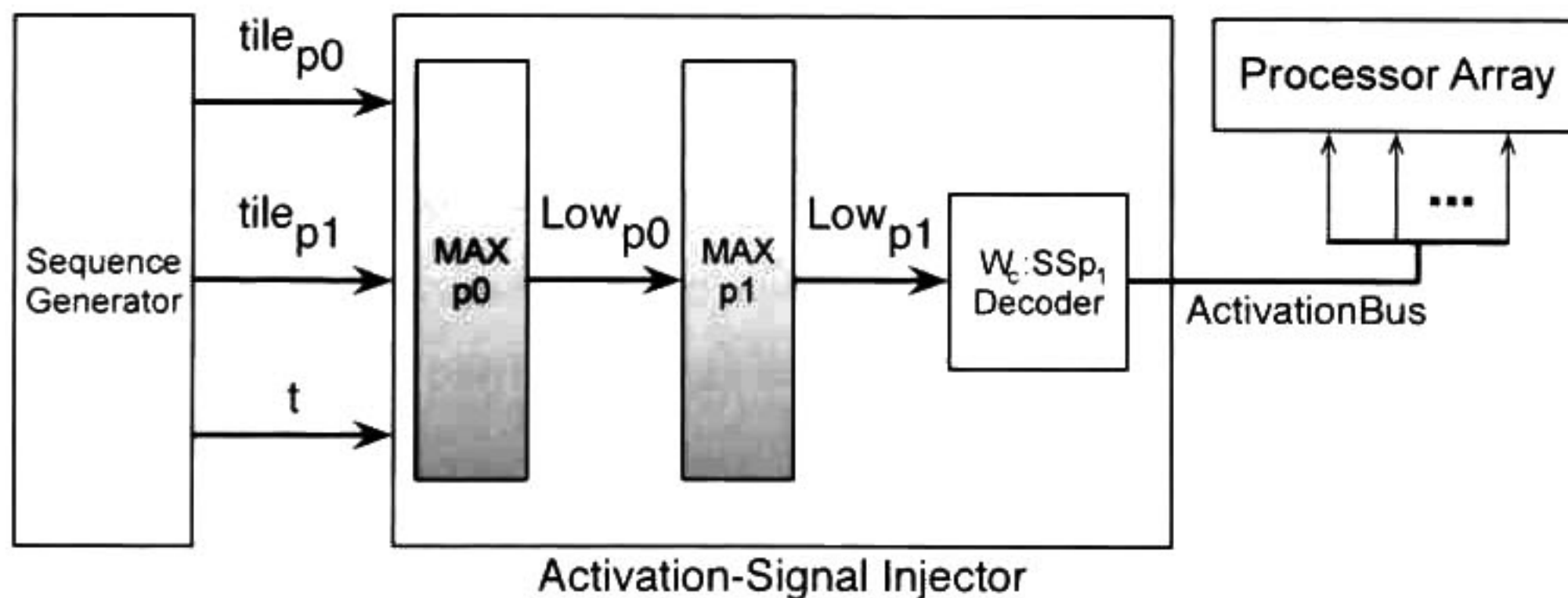


Figure 4.9: First approach for the activation-signal injector. For clarity purpose, it is only shown the *ActivationBus*.

Although this first approach seems to be expensive in terms of combinational logic used, it guarantees that the activation signal is placed at the correct PE at the first time instant of a tile iteration. However, improving this first approach is possible depending on the space processor bounds obtained after space-time transformation. Mainly these bounds depend on the transformation

matrix T , the scanning order of the tiles, and the source polytope index bounds. For example, it might happen that the Max sub-module in charge of evaluating the p_0 lower bound was eliminated; because the lower value of p_0 index could be always the same as the $tile_{p_0}$ index value, thus the p_0 value could be substituted by the $tile_{p_0}$ index value. Figure 4.10 shows this possible modification for the activation-signal injector internal architecture, where the combinational Max sub-module is highlighted. Finally, for injecting the index bus to the processor array, it is only needed to propagate the $tile_{p_0}$, $tile_{p_1}$, t , and the problem size to the bottom row of the control array.

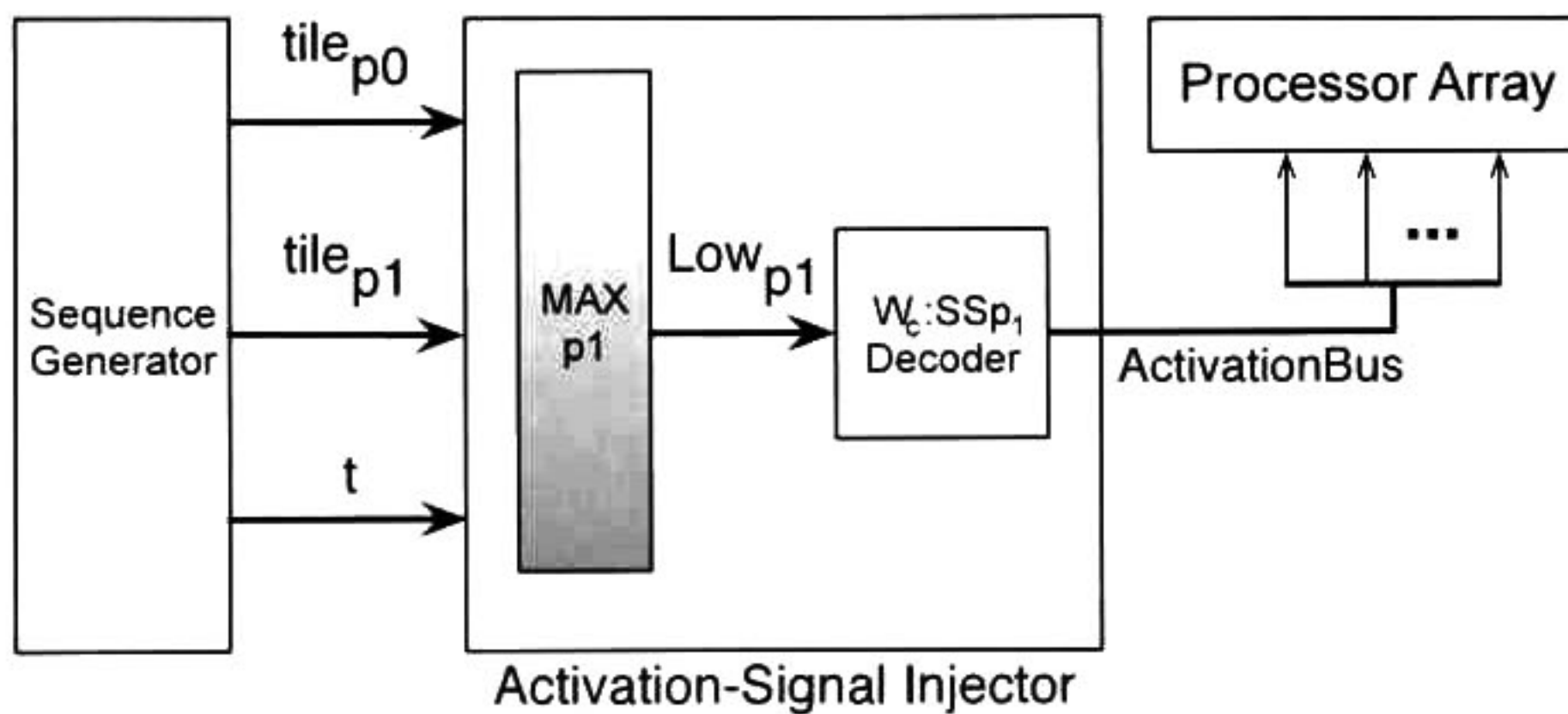


Figure 4.10: A possible second approach for the activation-signal injector. For clarity purpose, it is only shown the *ActivationBus*.

4.2.3 Control Array

The last module of the hybrid approach is the control array which is composed by several control cells that are replicated as many times as PEs has the processor array, following the interconnection topology of the processor array data-path. The control array is in charge of generating the processor array activation pattern (for rectangular and non-rectangular iteration spaces) using the *IndexBus* and *ActivationBus* injected by the activation-signal injector module. The control cells are in charge of circulating the activation signal, and the tile and time indexes through the control array. This data circulation is performed by knowing the processor array activation pattern, and the correct mapping

of the logical array to the PEs in the physical array. By discovering these two characteristics, the control array is able of circulating the *IndexBus* correctly through the processor array, and generating the correct activation pattern, independently of the piecewise regular algorithm and the space-time transformation used.

In a partitioned processor array obtained by strip mining the processor space, knowing when a PE maps a valid index point of \mathcal{P} is needed. This validation is performed while \mathcal{P} is being scanned by the tile indexes, and checking if all the PEs in the physical processor array map a processor space index point that is inside of the processor space boundaries, *i.e.* if logical processor $p \in \mathcal{P}$. If a processor space \mathcal{P}_r has a rectangular shape, then its boundaries depend only on the problem size N ; and if a processor space \mathcal{P}_{nr} has non-rectangular shape, then its boundaries depend on the problem size and on processor space indexes (p_0 or p_1 in the case of $\mathcal{P}_{nr} \in \mathbb{Z}^2$). Therefore, with the purpose of supporting the non-rectangular shapes in a processor space, the control cell includes combinational logic in charge of checking the boundaries of the processor space each time a new tile iteration has started.

Furthermore, in the case of a full-size processor array, the activation pattern of a PE depends only on the number of the index points in the source polytope that are mapped to an index point in the processor space \mathcal{P} . Erhart polynomials can be used to calculate such number [44]. If the index space of the source polytope is only formed by constants values, then the number of index points of the source polytope mapped to an index point $p \in \mathcal{P}$ remains constant. However, if any boundary of the source polytope is an affine function of the index space, the number of index points mapped is not constant, and it will depend on the index components of the index space. These two cases are maintained if the processor space is partitioned. The activation pattern provides an idea of how many clock cycles the incoming activation signal must be kept activated inside of control cell. Thereby, the control cell includes sequential logic for counting how many time steps the activation signal must kept activated (according to the source polytope index points mapped to a PE), and combinational logic for establishing the number of time steps each time a new tile iteration has started.

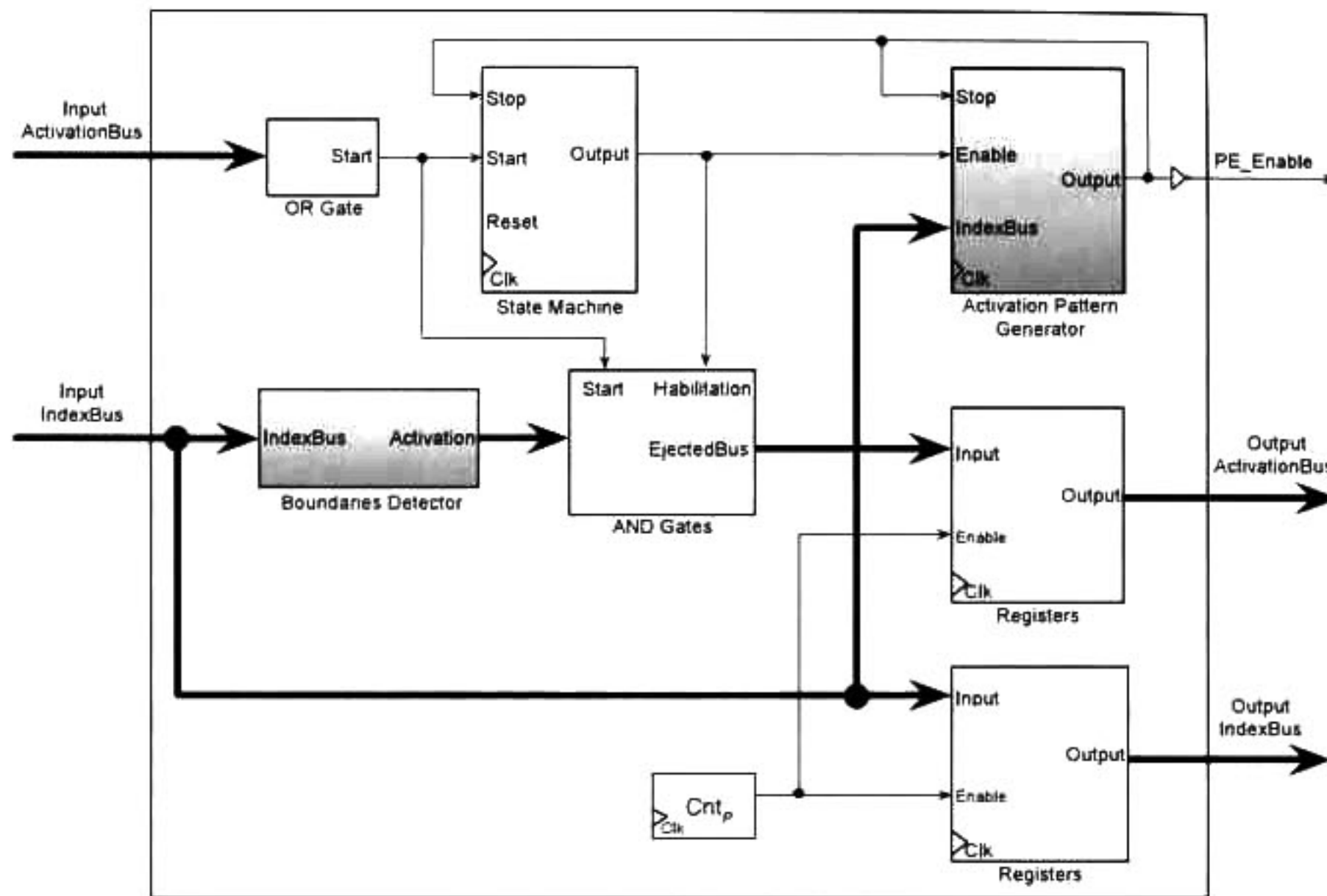


Figure 4.11: Generalization of the control cell architecture.

Figure 4.11 presents the general architecture of the control cell. Coarse lines indicate the *Input IndexBus* injected from the activation-signal injector, and the activation signal (*Input ActivationBus*) coming from the control cell neighbors. The width of the *Input ActivationBus* is equal to the half of neighbors that a control has, *i.e.* it depends on the interconnection topology. The control cell has an activation pattern generator, and a boundaries detector sub-module in charge of generating the activation pattern of a PE and for detecting the boundaries of the logical space processor, respectively. These two sub-modules have combinational and sequential logic which depends on the processor space, and the transformation matrix T ; thus their internal architecture can be only inferred after space-time mapping (see section 4.4). The activation pattern generator controls the enable signal (*PE_Enable*) of the corresponding PE, and it stops a FSM. This FSM is in charge of enabling the activation pattern generator by combining the *Input ActivationBus* in a set of OR gates. Similarly to the FSMs required by the sequence generator, the state transitions of this machine are done according to the iteration interval P , thereby internal modulo counters are included for each control cell. The boundaries detector sub-module and some AND gates are in charge of deciding if

the activation signal must be sent to any of neighboring PEs. Moreover, the control cells have a set of registers for storing the indexes and the activation signal generated by the set of AND gates. These registers are enabled by following at the same rate as the iteration interval and respecting the scheduler. The FSM transition diagram of the control cell is shown in figure 4.12.

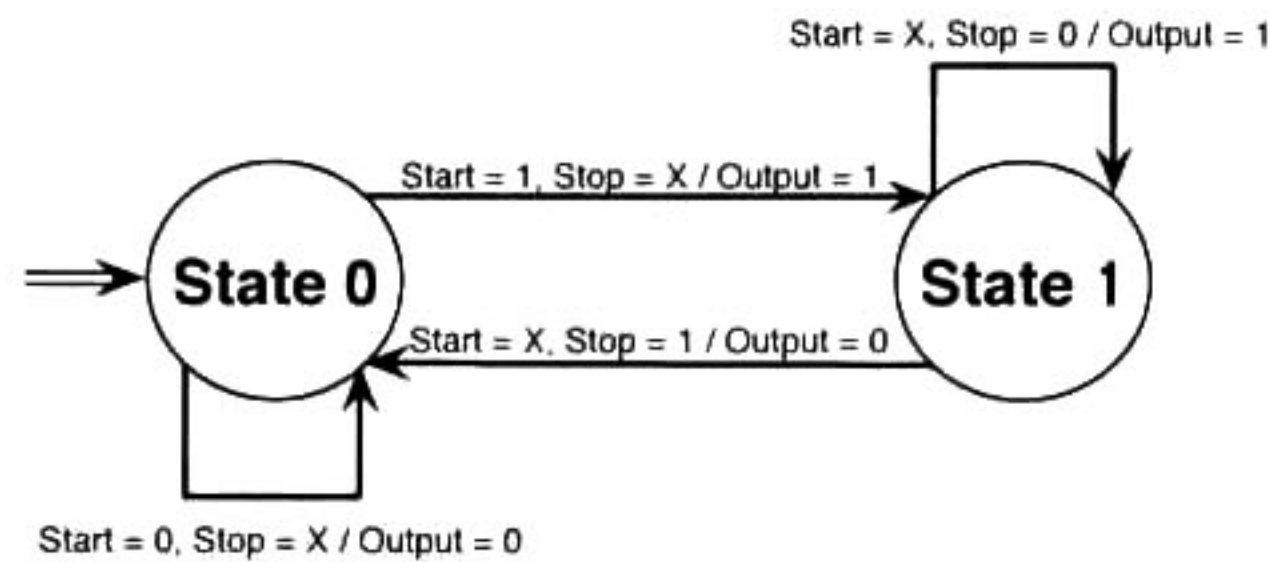


Figure 4.12: Mealy finite state transition diagram in charge of activating the activation pattern generator inside of the control cell.

4.3 Number of Logic Elements

The number of logic elements (adders, multipliers, multiplexers, counters, etc.) required by the sequence generator, the activation-signal injector, and the control array modules can be characterized in terms of some parameters. Table 4.1 shows such characterization, where the strip mining parameters SSp_0 and SSp_1 are the processor array size, h is the number of indexes generated by the sequence generator, and q is the number of data propagated through the control array. Parameters c_0, \dots, c_9 are constant factors that depend directly on two algorithmic aspects: the iteration space of the source polytope, and transformation matrix T . Therefore, h , q and c_0, \dots, c_9 parameters can be set to fixed values when a space-time mapping determined by matrix T is used in a piecewise regular algorithm.

Expressions shown in table 4.1 provide an idea of the minimum number of hardware elements that the control scheme needs without taking into account the algorithm neither the transformation

Number of	Sequence Generator	Signal Injector	Control Cell	Control Array
Adders	$h + c_0$	c_4	c_8	$c_8(SSp_0 \times SSp_1)$
Multipliers	c_1	c_5		
Multiplexers	$2d + c_2$	c_6		
Comparators	$d + c_3$	c_7	c_9	$c_9(SSp_0 \times SSp_1)$
Counters	4		2	$2(SSp_0 \times SSp_1) + 4$
Registers	h	q	q	$q(SSp_0 \times SSp_1)$
1-Bit FFs	1		2	$2(SSp_0 \times SSp_1)$

Table 4.1: Minimum hardware resource utilization of the control architecture in terms of three different parameters.

Number of	Sequence Generator	Signal Injector	Control Array	Control Scheme Total
Adders	6		$2(SSp_0 \times SSp_1)$	$2(SSp_0 \times SSp_1) + 6$
Multipliers	2	-		2
Multiplexers	8	-		8
Comparators	6		$2(SSp_0 \times SSp_1)$	$2(SSp_0 \times SSp_1) + 6$
Counters	4		$2(SSp_0 \times SSp_1)$	$2(SSp_0 \times SSp_1) + 4$
Registers	3	4	$4(SSp_0 \times SSp_1)$	$4(SSp_0 \times SSp_1) + 7$
1-Bit FFs	1		$2(SSp_0 \times SSp_1)$	$2(SSp_0 \times SSp_1) + 1$

Table 4.2: Characterization of the hardware resource utilization of the control scheme for MatMul when the vectors $\vec{\lambda}_l = [1, 1, 1]$, $\vec{u} = [1, 0, 0]$ are used.

matrix T , *i.e.* without considering the constant factors. Also, these expressions show that the sequence generator and signal injector require a number of hardware elements linear, in terms of h and q . However, after partitioning the processor space, the number of hardware elements is constant regardless of the processor array size. On the other hand, although the number of hardware elements that a control cell requires is constant, the number of elements needed for the control array grows in a quadratic way depending directly on the processor array size. Moreover, note that the number of registers required by the control array is proportional to the number of PEs in the array. This characteristic allows to increase the control array size without decreasing the maximum operational frequency due to none critical path is created when more control cell are interconnected. Table

4.2 shows the number of hardware elements required for the control architecture when the MatMul algorithm is used with certain transformation matrix T , and consequently parameters h , q and c_0, \dots, c_9 can be set to fixed values. Note that regardless of the processor array size, the number of hardware elements required by the sequence generator and activation-signal injector remains constant.

4.4 Cholesky Decomposition Case of Study

The hybrid control scheme is designed for supporting different space-time mapping and interval iterations. By changing some mathematical expressions (mapped to combinational logic) present in each one of the three control modules, the generation of the correct activation pattern is possible. Additionally, the control is able to deal with non-rectangular processor spaces in contrast to [36] and [65]. With the purpose of exemplify these characteristics, in this section a case of study using the Cholesky decomposition is presented. The case of study uses a scheduler function $\vec{\lambda}_l = [1, 2, 3]$, the projection vector $\vec{u} = [1, 0, 0]$, the iteration interval $P = 21$, and strip sizes of $SSp_0 = 2$ and $SSp_1 = 4$, resulting in a processor array of 2×4 PEs.

The first step for deriving the mathematical expressions required by the different control modules, is obtaining the tile and time indexes, *i.e.* $tilep_0$, $tilep_1$ and t . These indexes are obtained from the space-time transformation and from partitioning the processor space. Thus, it is needed to construct a unimodular matrix T for the space-time mapping, by using $\vec{\lambda}_l = [1, 2, 3]$ and $\vec{u} = [1, 0, 0]$:

$$T = \begin{bmatrix} \vec{\lambda} \\ \Phi \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.1)$$

Later, applying the Fourier-Motzkin algorithm for transforming the Cholesky PRA (see figure 3.4 in Chapter 3) using the matrix T , the bounds of the target polytope are obtained. After the space-time mapping, the strip mining technique is used over the processor space. The lower and upper bounds of the tile and time indexes indexes are presented in the following set of inequalities:

$$\begin{aligned}
0 &\leq tilep_0 \leq N - 1 \\
\left\lceil \frac{tilep_0}{SSp_1} \right\rceil SSp_1 &\leq tilep_1 \leq N - 1 \\
LowBound_t &\leq t \leq UpBound_t
\end{aligned} \tag{4.2}$$

where $LowBound_t = \max(4tilep_0 + 2a, 9(tilep_0 + a) - 6(N - 1), 9tilep_0 - 6(N - 1))$, $UpBound_t = \min(3(N - 1) + 3tilep_1 + 3b, 6tilep_1 + 3b, 3(N - 1) + 3tilep_1)$, $a = tilep_0 + SSp_0 - 1$ and $b = tilep_1 + SSp_1 - 1$. According to the Fourier-Motzkin algorithm *max* and *min* functions are used for lower and upper bounds, respectively. Similarly, if the bounds expressions have division operations *ceil* and *floor* functions are used in order to avoid non-integer values in the lower and upper bounds, in respective order.

In order to generate the tile and time indexes, it is required to implement the bounds from expression 4.2 to combinational logic in the Min/Max sub-modules. Since the size of the strips in this case of study is a two-power multiple value known at compile time, they can simplify the combination logic needed in the Min/Max sub-modules presented in the sequence generator. For example, the division in the low bound of index $tilep_1$ can be substituted with a constant right shift. Similar improvements could be performed on other expressions. The *STEP* parameters indicating the counting intervals are $tilep_0 = 2$, $tilep_1 = 4$ and $t = 1$. The internal arithmetic for implementing the Min/Max sub-modules in this case of study is assumed to be 16-bits.

In the case of the activation-signal injector, the approach shown in figure 4.9 could be used as first attempt to inject the activation signal along with the index bus. However, this would lead to implement complex lower bound expressions of the processor space. Besides, recall that these bounds are affine functions of other indexes presented in the target polytope. In this first approach the index p_1 depends on p_0 . Nevertheless, it is not necessary to evaluate the p_0 lower bound, because this bound is always equal to the $tilep_0$ value; thus the p_0 value could be substituted by $tilep_0$. Therefore, there is only needed a combinational Max sub-module to implement the lower bounds expression of p_1 (equation 4.3). The division operation needed in equation 4.3 is performed by multiplying a 16-bit

fixed value which represent the constant $1/3$. This assumption does not add any error during the computations of the p_0 low bound due the truncation performed by the *ceil* function.

$$LowBound_{p_0} = \max \left(\left\lceil \frac{t}{3} \right\rceil - tile_{p_0}, tile_{p_0}, tile_{p_1} \right) \quad (4.3)$$

The $W_c : SS_{p_1}$ decoder sub-module decodes the p_1 index value in order to generate the activation signal. The output width of this decoder is equal to the value of SS_{p_1} parameter, which in this case is $SS_{p_1} = 4$. The true table of this decoder using 16-bit for decoding the $LowBound_{p_0}$ value is shown in table 4.3.

$LowBound_{p_0}$ Input	Activation Signal
0000000000000001	0001
0000000000000010	0010
0000000000000011	0100
0000000000000100	1000

Table 4.3: Truth table 16:4 decoder

The control cells require deriving the mathematical expressions for the activation pattern and the boundaries detector. Since the source polytope boundaries in Cholesky PRA case are affine functions of the index space, the number of clock cycles that the activation signal must be kept activated inside of a PE is not constant. The activation pattern can be characterized as a function of the p_0 -th row of the processor array and as function of $tile_{p_0}$ index (equation 4.4).

$$MaxLife_{p_0,p_1} \leq row_i + tile_{p_0} \quad (4.4)$$

Besides of the activation pattern, it is required to know when a physical processor maps a valid logical processor. In this case of study note that while the processor space is being scanned by the tile indexes, there are PEs which are not used. These unused PEs are outside of the boundaries mapped from the logical processor array to the physical one. Figure 4.13 shows a snapshot of the logical array and the physical array when the last partition of the processor space is scanned. Also, in this figure the problem size is $N = 10$. The border lines indicate the boundaries of the logical

array. Note that in this case the size of the processor space (logical array) does not fit exactly in the physical array, thus some PEs in the physical array remains inactive (dashed PEs).

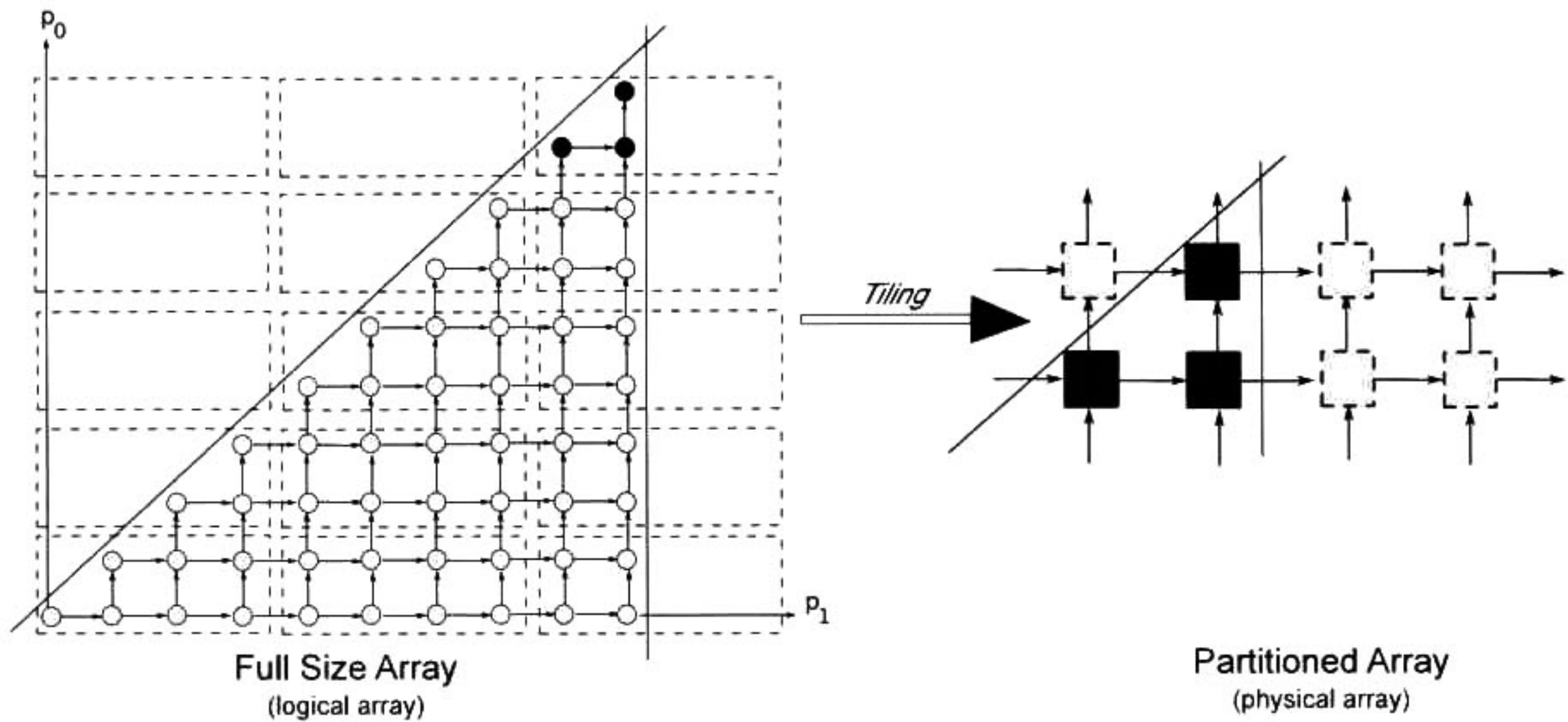


Figure 4.13: Activation of a full-size processor array and its mapping to physical array of size 2x4. Invalid mapping is denoted by dashed boxes in right size figure.

It is possible to detect the PEs inside these boundaries when expression 4.5 and 4.6 are true. Expression 4.5 checks PEs within the main diagonal, and expression 4.6 checks the number PEs required for the problem size.

$$tilep_0 + p_0 + 1 \leq tilep_1 + p_1 \quad (4.5)$$

$$tilep_1 + p_1 + 1 < N \quad (4.6)$$

Together expressions 4.4, 4.5 and 4.6 provide enough information for keeping activated the PEs, and for knowing how to route the indexes, and activation signal through the array. Figure 4.14 shows the control cell internal architecture for this case of study. Due to the interconnection topology, the control cell receives the activation signal from its left and lower neighbors ($Inject_{hb}$ and $Inject_{vd}$ signals, respectively). The boundaries detector, the FSM and the AND gates are in charge of

generating the activation signal for the upper ($Ejected_{hb}$ signal) and righter ($Ejected_{vd}$ signal) PE's neighbors. The activation signal and the indexes must have some clock cycles of delay according to the components $r_{hb}^2 = 2$ and $r_{vd}^3 = 3$ of vector \vec{r} . This delay could be induced by activating periodically the enable signals from the registers following the iteration interval P and the scheduling function. Figure 4.15 shows the interconnection of the control cells forming the control array for the processor array of 2×4 PEs.

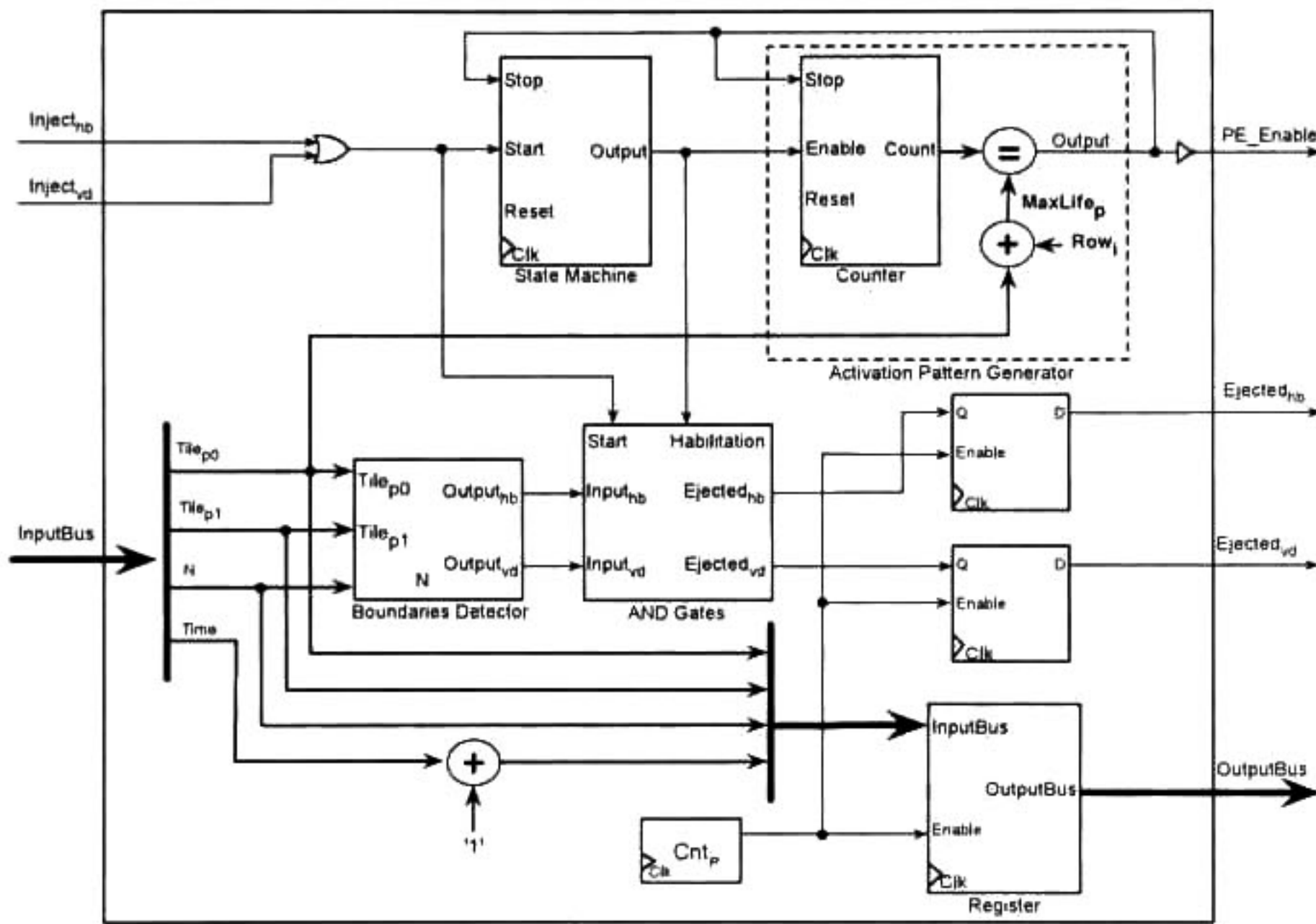


Figure 4.14: Internal control cell architecture for the Cholesky decomposition algorithm case of study.

Finally, table 4.4 shows the number of hardware elements required for the controller developed in this case of study. Recall that for calculating the bounds of the Cholesky partitioned processor space the division required in expression 4.3 is performed by multiplying the fixed constant value of $1/3$. Note that the interconnection of hardware elements in the sequence generator and activation-signal injector modules is more prone to generate larger critical-paths than the control array; since the number of registers in the control array increase at the same rate as the other elements in the control array. This characteristic helps to increase the size of the processor array with a small impact on the operational frequency.

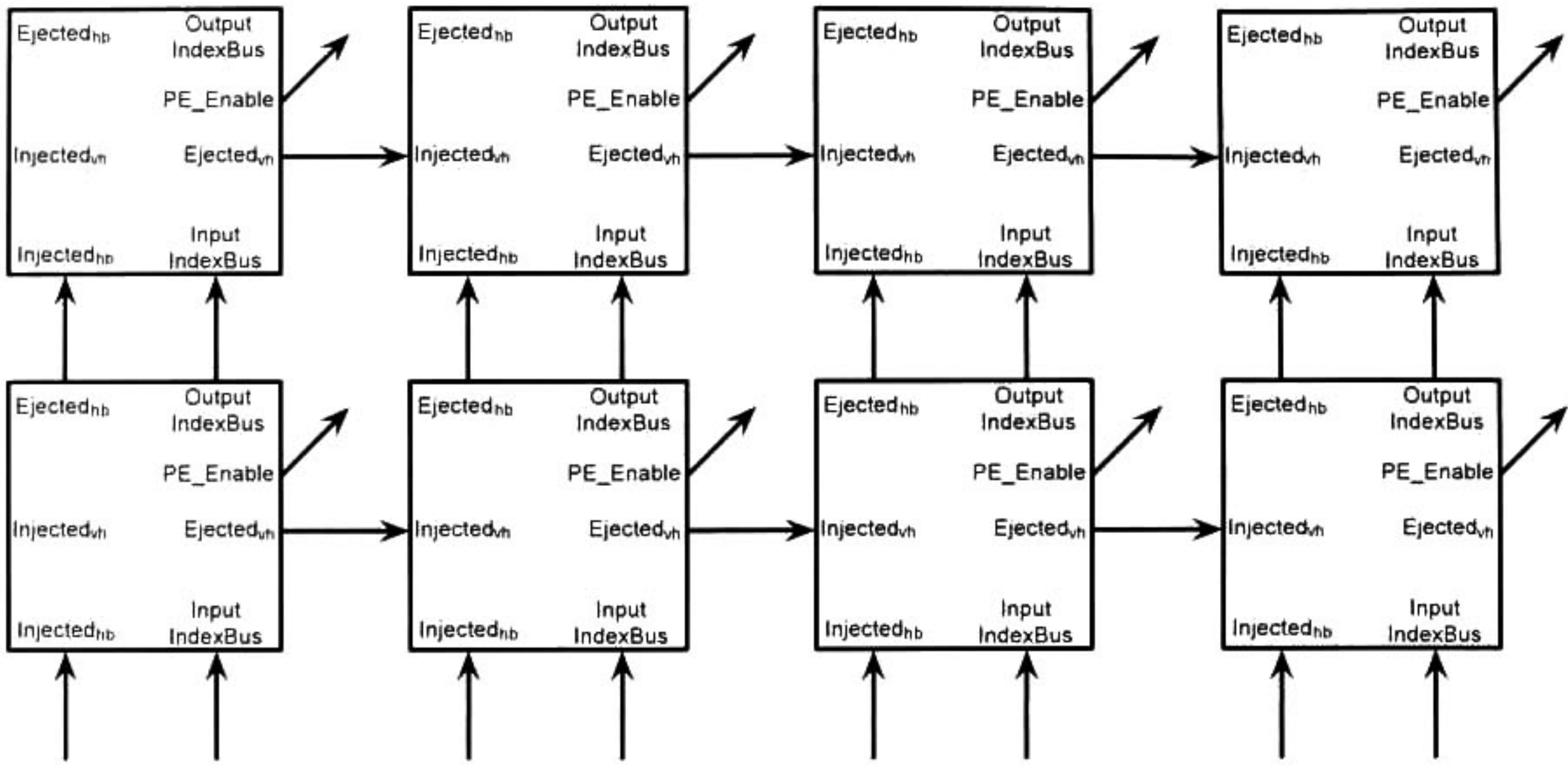


Figure 4.15: Interconnection of the control cells forming the control array for the Cholesky algorithm case of study.

Number of	Sequence Generator	Signal Injector	Control Array	Control Scheme Total
Adders	14	7	32	53
Multipliers	8	8		16
Multiplexers	12	15		27
Comparators	9	15	16	40
Counters	4		16	20
Registers	3	4	32	39
1-Bit FFs	1		16	17

Table 4.4: Characterization of the hardware resource utilization of the control scheme for Cholesky case of study.

4.5 Summary

Although there are works tackling the problem of control generation signals for processor arrays, they are limited in the generation of controllers only for a specific problem size, and for algorithms with rectangular iteration spaces. In this chapter, a control scheme for generating the control signal for processor arrays, independently of the iteration space shape has been presented. This control scheme is able to provide the control signals for a set of problem sizes without the need of regenerating the controller. The proposed controller uses centralized and distributed functionalities, placing the most repetitive and expensive hardware operations to central modules and using several registers in the distributed modules. This helps to increase the control array size, and thereby the processor array size, with a minor impact in the clock frequency.

The main idea behind the controller is having a sequence generator module in charge of scanning tile indexes and generating the time index. These indexes are decoded by the activation-signal injector, which generates an activation signal that is sent to control array. The control array circulates the activation signal and each control cell decides if the signal should be sent to its neighbor, generating the activation pattern.

5

External Memory

One important aspect involved during the processor array derivation is the processor data feeding and the processor data extraction, *i.e.* the external memory aspects. The hybrid control scheme described in last chapter ensures the correct activation of the processing elements within the processor array, when the processor space is being scanned in blocks (due to strip mining). Also, selecting the correct operations during a time instant by inserting the index bus is provided by the control scheme. However, the processor array data-path and the hybrid control are meaningless if data are not provided from an external memory element. Once valid data are being produced by the processor array, extracting the results either for storing them in external devices or for feeding other components inside of a SoC is needed. In this sense, this chapter is mainly focused on describing an external memory interface in charge of inserting/extracting data to/from the processor array (figure 5.1). However, in order to explain completely the design methodology followed by this dissertation work, issues concerning about the processor array memory organization and the intermediate memory for data reuse are briefly and firstly explained.

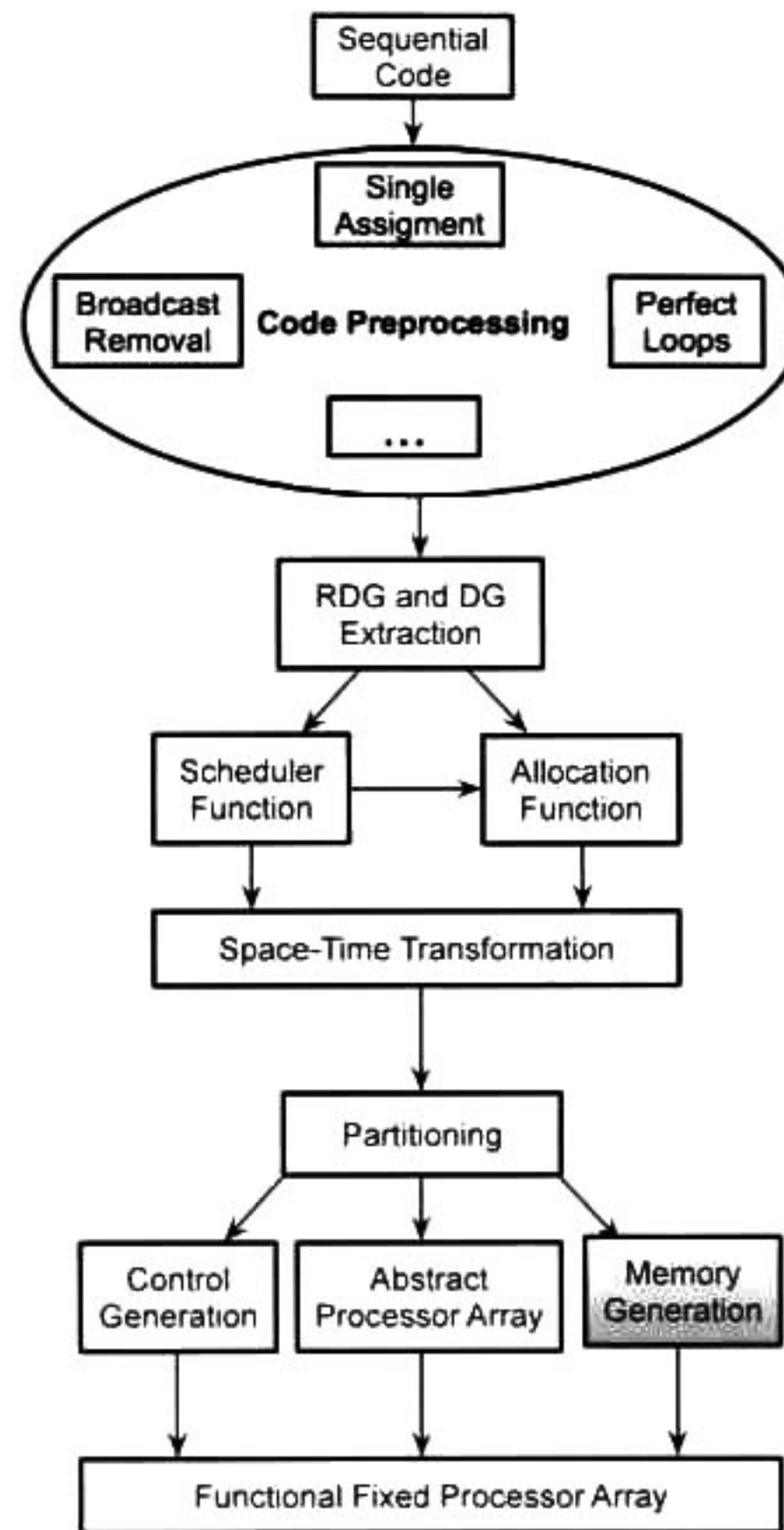


Figure 5.1: Design flow methodology followed in the polytope model, highlighting the memory generation.

5.1 Memory Hierarchy

The internal registers placed inside of each PE, the intermediate memory for data reuse, and the external memory interface, could be organized in a memory hierarchy according to temporal and spatial locality concepts. Temporal locality refers to data that is likely to be reused again within a short period of time; while spatial locality refers to data nearby of the data recently accessed. At the first level of the memory hierarchy, the registers derived from expressions 3.18 and 3.20 are located (see Chapter 3). The next memory levels are composed by intermediate memory divided in two different levels according to data proximity. At the inner intermediate level are memory

elements in charge of storing data with major temporal locality, whereas in the intermediate outer level are memory elements responsible for storing data with less temporal locality. These two levels are called intermediate memory level 1 and intermediate memory level 2, respectively; and they can be compared as traditional cache memories in the sense of their size and their locality. Finally, the outer level is composed by the external memory system in charge of providing/extracting data to/from the processor array. The amount of external memory depends on the problem size that it is being solved, the size of the physical processor array, and the number of input/output variables presented in the source polytope. Together the registers, intermediate memory and external memory conforms a memory hierarchy illustrated in figure 5.2.

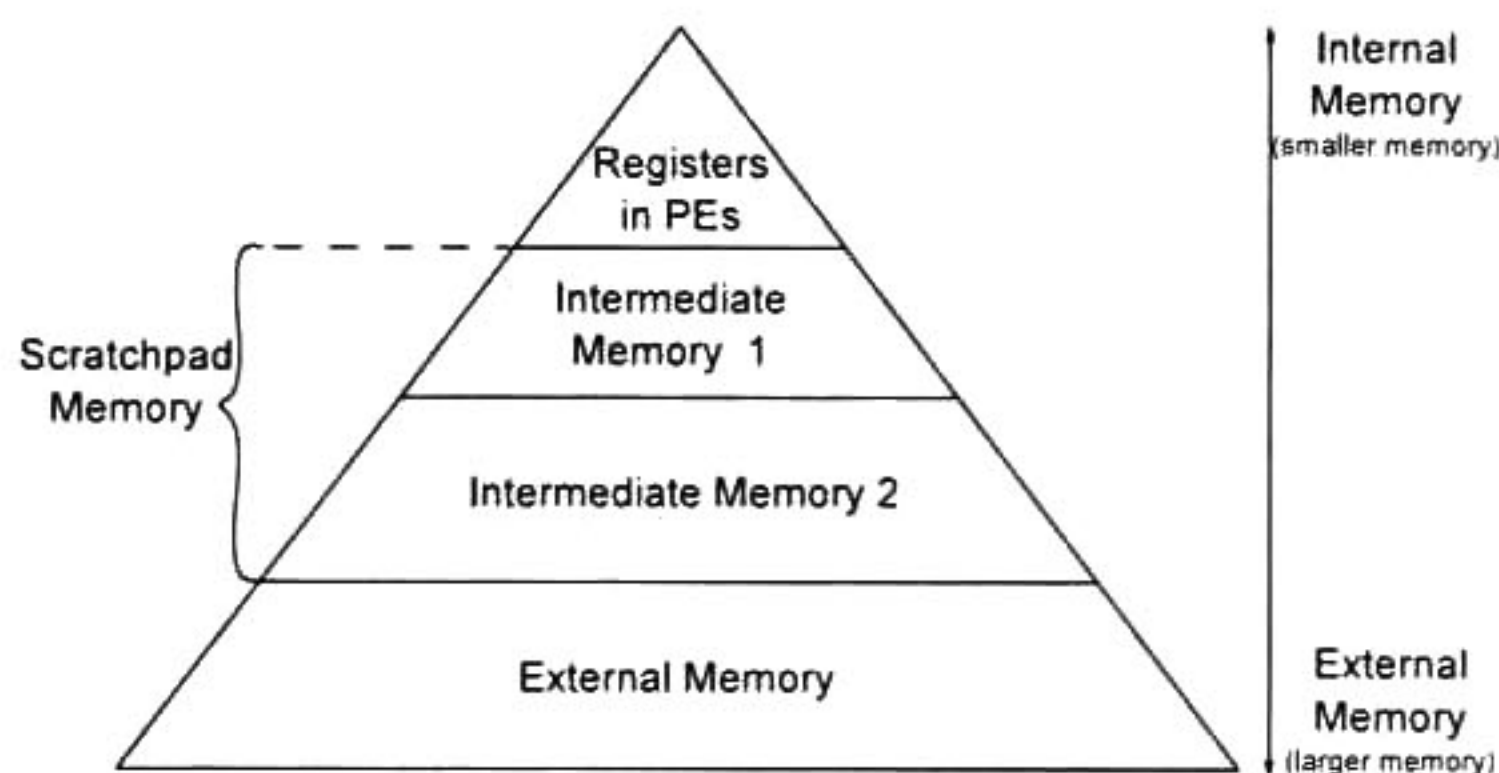


Figure 5.2: Processor array memory hierarchy.

5.2 Intermediate Memory

Intermediate memory is essential for the sake of guarantee the correct algorithmic functionality in partitioned processor arrays. The purpose of this memory is to store intermediary results which have been computed by the processor array for their subsequent usage. The generation of these memories have been considered by previous works for different partitioning approaches. However, it is worth to mention two aspects involved in the generation of such intermediate memories: the size of these memories and their corresponding control. This section is advocated to describe these two aspects; particularly, the control divided into two cases: rectangular and non-rectactgular iteration spaces.

5.2.1 Motivation

In a full-size array implementation the usage of intermediate memory is not required since all temporal values produced by the processor array are stored inside of PEs registers while the computations are being performed. Also, the full-size processor array possesses all the internal memory required for computations in the form of internal registers. However, when partitioning is applied using LPGS approach, an intermediate storage is required for storing data produced by the border PEs, substituting the registers in full-size implementation. Using strip mining with a LPGS approach, the processor space index points are grouped into subsets that are executed in parallel while each subset is executed sequentially. This leads to the idea that tile indexes produce an scanning order for each one of these subsets, and due to such processor space scanning order, some tiles are firstly processed. Figure 5.3 illustrates this last idea by showing a full-size processor array of 16×16 PEs mapped to a physical array of 4×4 PEs. For this figure assume that each PE is activated sixteen consecutive times, and consequently, each PE produces sixteen data. Also, assume the scanning order is done first from left to right ($tilep_1$ index) and down to up ($tilep_0$ index). Due to all PEs produce sixteen data (including the border PEs in the physical array), it should be placed a memory element for each PE able to store the sixteen data produced by the border PEs placed in p_1 direction. When a different tile in the direction of p_1 index is scanned, data produced from the previous tile must be reinserted to the processor array. In this example, the memory elements placed in the p_1 direction are denoted as intermediate memory level 1. Similarly, a second intermediate memory level between the upper and lower processor array borders should be placed.

The intermediate memory could be implemented by using addressable memory or by using FIFO memories. In the case of implementing the intermediate memory as addressable memory, some extra logic for address calculation per each memory element should be added. In this case, the address calculation could be performed using address generation units (AGUs) and counters in order to scan the memory locations while the processor array is doing its computations. From the implementation point of view, this could be impractical, since computing the addresses for each memory location

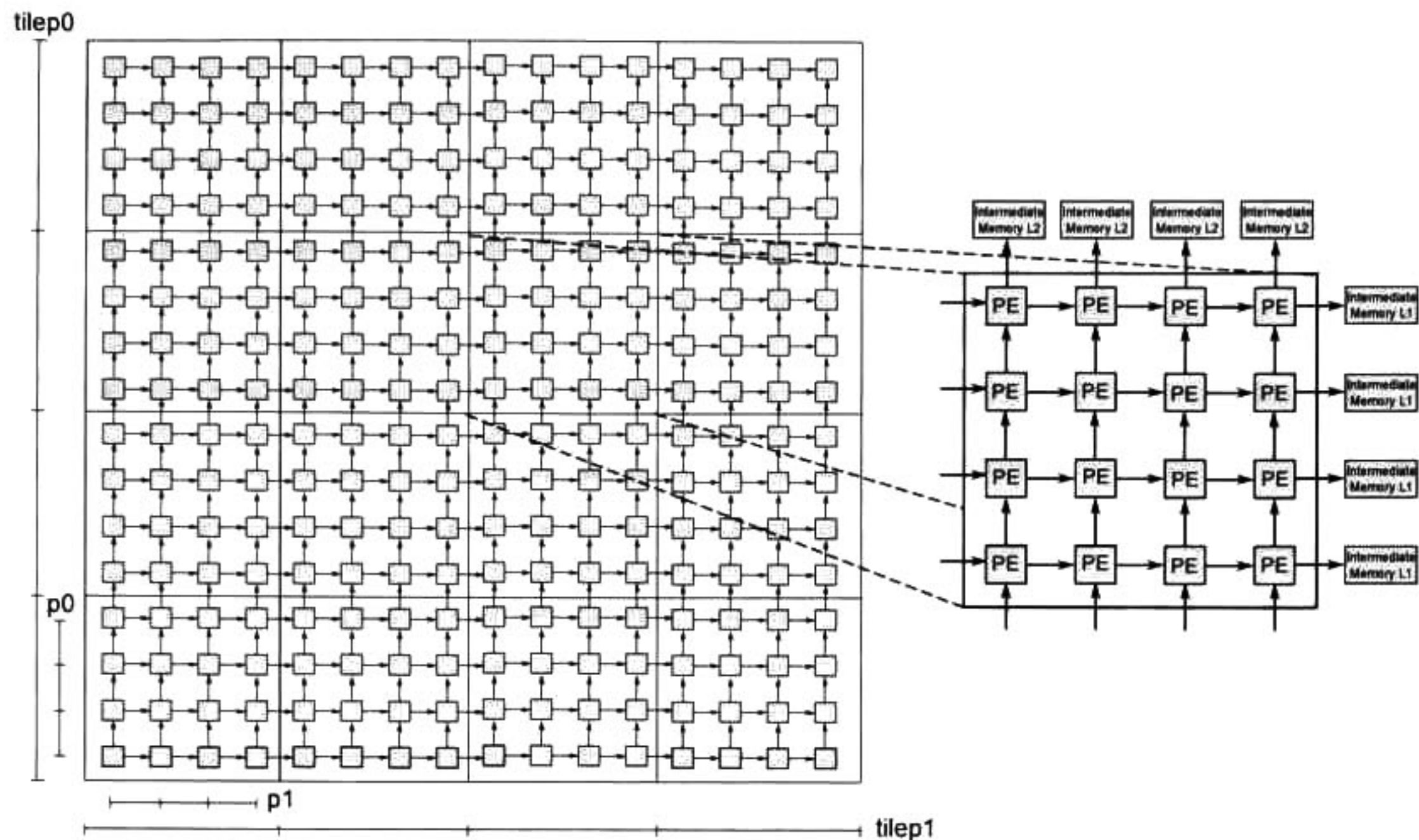


Figure 5.3: Physical processor array of sixteen PEs and its connection among the two intermediate memory levels.

involves n -bit control word operations per intermediate memory module. On the other hand, using FIFO memories, the overhead introduced by n -bit word operations could be reduced by using simple FSMs with two states. Such FSMs would be in charge of generating the write and read signals required in FIFO's implementations. Moreover, the FIFO approach describes more naturally the behavior of the data generated by the border PEs, since the first datum produced by a PE are the first one consumed by the following PEs, *i.e.* the processor array data production/consumption is done in a orderly manner [88]. In the next subsection, more details of the intermediate storage implemented as FIFO memories, called FIFO L1 and FIFO L2, are described.

5.2.2 Internal FIFOs

Independently of using FIFOs or addressable memory as intermediate storage, the number of memory locations required for each intermediate memory must be determined firstly. In general, for rectangular or non-rectangular iteration spaces, the maximum number of memory locations is

related to the size of the problem being solved. For example, when a rectangular iteration space is considered, the amount of data produced by each PE remains constant during the whole algorithm execution, whereas in the non-rectangular case such amount varies. However, in both cases, the maximum number of memory locations is not larger than the size of the problem being solved. Therefore, the problem size could be used as a parameter for calculating the memory size assuming a worst case scenario. In the case of FIFOs L1, the number of memory locations is equal to the amount of data produced by each PE, *i.e.* the problem size. If the example illustrated in figure 5.3 is continued, the number of memory locations required by FIFOs L1 is sixteen. On the other hand, the size of each FIFO L2 memory is estimated with the following equation:

$$FIFOSize_{L2} = \left\lceil \frac{Size_{p_0}}{SS_{p_0}} \right\rceil \times c \quad (5.1)$$

where $Size_{p_0}$ is the size of the processor space in direction p_0 , SS_{p_0} is the size physical processor array in direction of p_0 , and c is the amount of data produced by the border PEs. In other words, the size of the FIFO L2 is a function of the processor space and physical processor size in the same direction (p_0 in this case), and a function of the data produced by the border PEs. In the example shown in figure 5.3 the size of such FIFOs L2 are 64. The design of the control of these FIFOs could be divided into two different cases according to the iteration space shape after space-time mapping.

5.2.2.1 Rectangular Iteration Space

In the case of rectangular shapes, the generation of the FIFOs write/read enable signals is straightforward. The writing signal, for both L1 and L2 FIFOs, is activated only when the border PEs are producing data, *i.e.* when these PEs are activated by the control array. Therefore the activation signal produced by the control cell for each border PE can be used as the FIFO write enable signals. On the other hand, the read enable signals for L1 and L2 FIFOs are produced in a slightly different way compared with the write enables. In fact, two signals are required in order to enable the FIFO readings: the activation signal from left border PE and a signal which indicates when the maximum value of a tile index has been reached. This last signal, called *EndCount*, is

already present in the sequence generator module described in the last chapter. In the case of FIFO L1 the signal which indicates that the $tilep_1$ index has reached its maximum value comes from the $time$ counter, whereas for FIFO L2 the $EndCount$ signal comes from the $tilep_1$ counter. Together, the control array activation and the $EndCount$ signals are able to produce the read enable signals by activating a FSM. The transitions made by the FSM are done according to the iteration interval P , consequently a P -modulo counter for enabling the FSM is added. Figure 5.4 illustrates the idea of the interconnection between two PEs placed on opposite array sides, and the control signals for write/read. One row of PEs is only shown for purpose of clarity, but the interconnection idea can be generalized for any row or column of PEs inside of the physical processor array *i.e.* any two FIFO levels. Signals $Activation\ PE_n$ and $Activation\ PE_0$ are generated by the control cells corresponding to the opposite processing elements PE_n and PE_0 , respectively. The $EndCount$ signal could be either from the $time$ or $tilep_1$ counter, depending on the FIFO level. The FSM is activated at a rate equal to the iteration interval by a P -module counter. Finally, figure 5.5 shows the transition diagram for the Moore FSM in charge of generating the $ReadEnable$ signal.

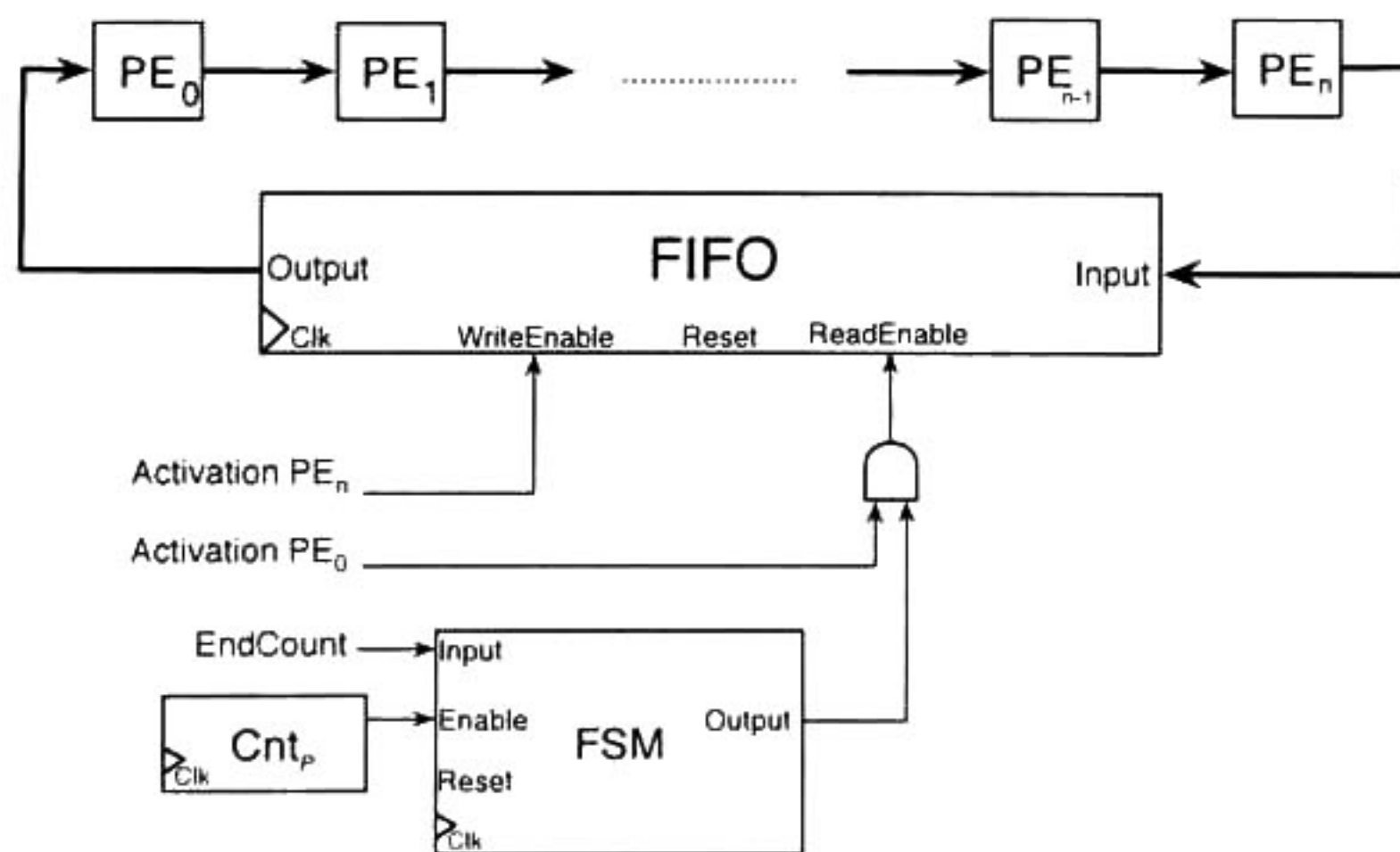


Figure 5.4: Interconnection of FIFO memory and its control at the processor array borders for rectangular iteration spaces.

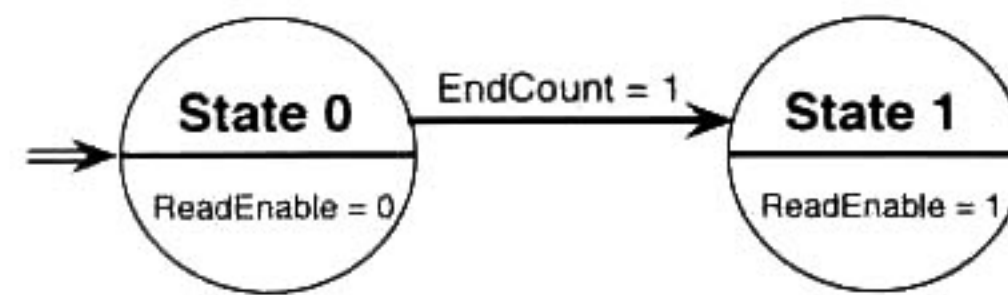


Figure 5.5: Moore finite state machine transition diagram in charge of generating the FIFOs *ReadEnable* signal for rectangular iteration spaces.

5.2.2.2 Non-Rectangular Iteration Space

In the case of non-rectangular iteration spaces, the generation of the FIFOs read enable signal must consider the number of consecutive times a PE is producing data that in future tile iterations will be needed. Such number is represented by the *ReadsAmount* signal, which is related to the concept of activation pattern presented in the control cell (see Chapter 4). Basically, this signal and the activation pattern concept indicate the number of consecutive times a PE is activated, and consequently the amount of data produced by the PE that should be read in further tiles. In addition, similarly to the rectangular case, the *EndCount* is also required for generating the read enable signal, since it indicates when the FIFO reads must start. On the other hand, the FIFO write signal is generated by using the *EndCount* and the activation signal *Activation PE_n*. Write and read enable signals are produced by two different Moore FSMs shown in figures 5.6 and 5.7, respectively. Like in rectangular iteration space case, the FSMs transitions are done at a rate equal to the iteration interval. Therefore, *P*-modulo counter is used in order of generating the *WriteEnable* and *ReadEnable* signals according to the iteration interval *P*. The *ReadsAmount* signal is generated by another counter varying its maximum count during different tile iterations. Figure 5.8 shows the interconnection of these counters, the FSMs and a FIFO memory for one row of PEs.

5.3 External Memory

The derivation of the external memory scheme is closely related to the input and output variables present in the PRA specification, and their corresponding iteration dependent condition $C^I(\vec{I})$ after

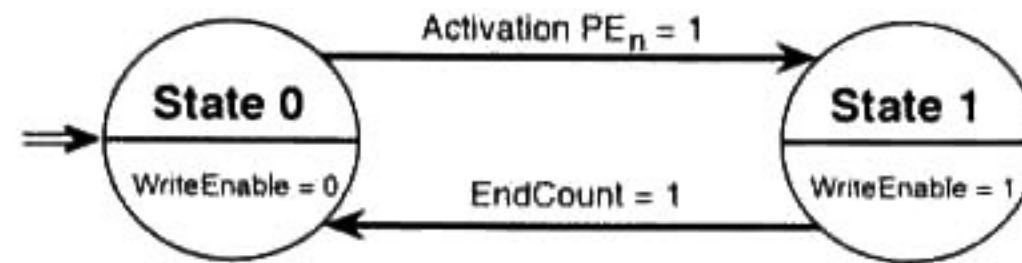


Figure 5.6: Moore Finite state machine transition diagram in charge of controlling the FIFOs *WriteEnable* signal for non-rectangular iteration spaces.

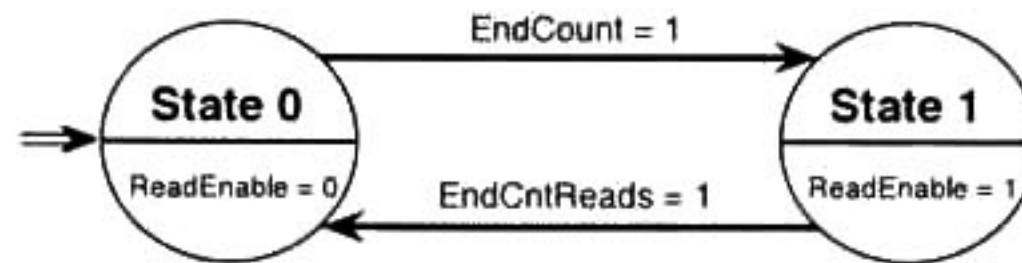


Figure 5.7: Moore Finite state machine transition diagram in charge of controlling the FIFOs *ReadEnable* signal for non-rectangular iteration spaces.

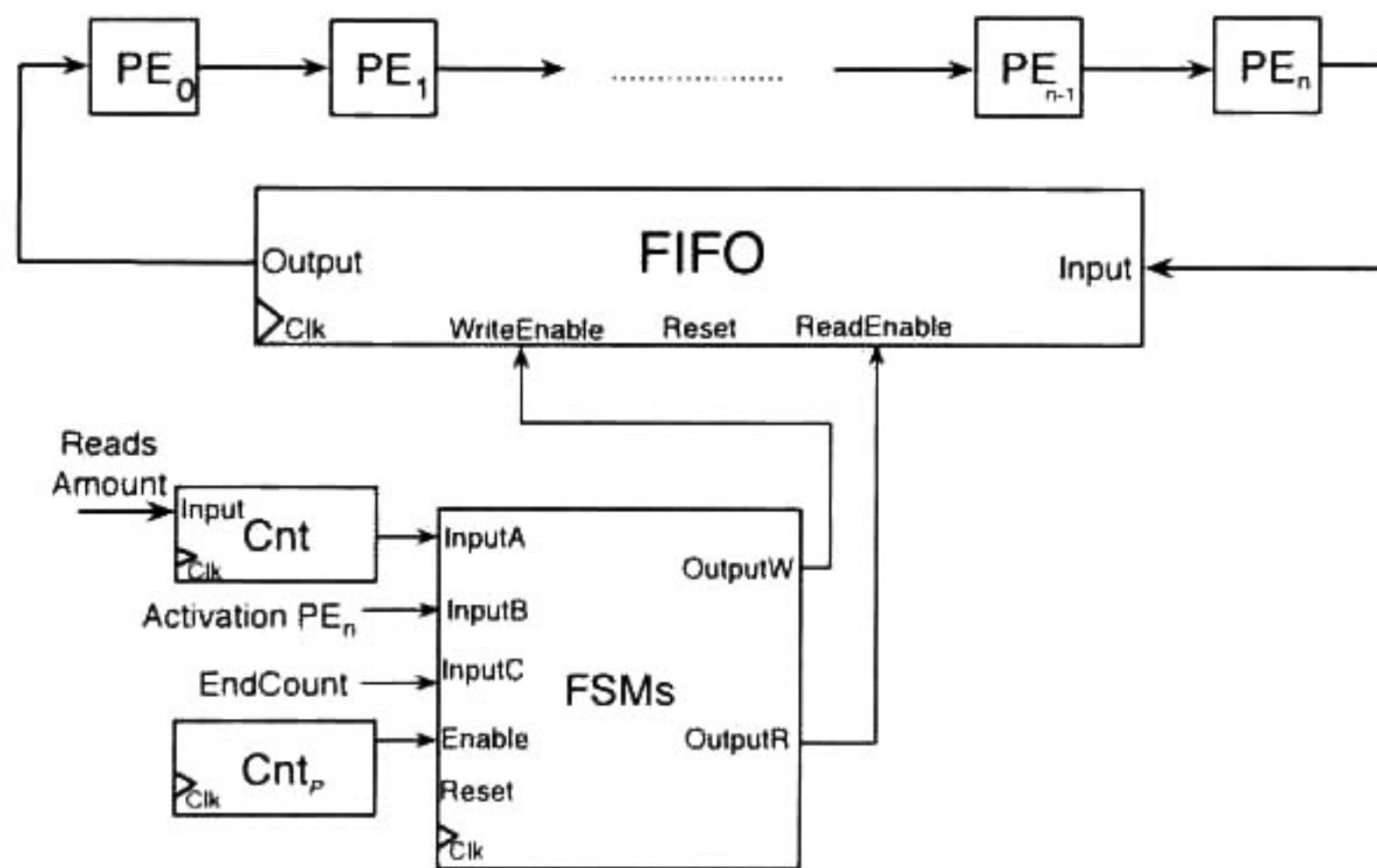


Figure 5.8: Interconnection of FIFO memory and its control at the processor array borders for non-rectangular iteration spaces.

space-time transformation. The transformed input and output variables can be interpreted as a representation of external memory, with a specification of which PE will require a datum at certain time instant. Essentially, after the space-time mapping, and depending on the variable type, four different architectural cases could be derived. These four cases assemble a memory system for inserting/extracting data to/from the processor array. In this section each one of the four architectural cases is explained. The first part of this section establishes the motivation and

considerations presented for the generation of external memory interfaces, mainly focused on the derivation of the number of external communication channels. The second part describes the four cases with their corresponding proposed modular architectures. These architectures are scalable in the sense of by replicating or eliminating the hardware modules, they can be adjusted in order to support changes in the the processor array size, or changes in the iteration interval.

5.3.1 Motivation

Beyond of the fact that there are few attempts for deriving the external memory interfaces for processor arrays using the polytope model, the processor array high data requirements must be taken into account. Processor arrays are highly parallel under the assumption that all data needed for performing their computations are always available. Similarly, if data produced by the array are recollected without stalling the array computations, a high degree of parallelism could be maintained. Thus, the design of memory systems capable of keeping busy the array without stalling its computations is mandatory in order to preserve the processor array performance. Hence, the use of multiple communication channels for inserting and extracting data is required in order to prevent the processor array performance degradation. In this sense, having a highly parallel processor array with an external memory interface able of proving or extracting only a partial amount of data is worthless, since it limits parallelism originally provided by the processor array. An example of this last situation is reported by Plesco in [91] where the total performance obtained by a 4×4 processor array is limited by the use of two communication channels. Consequently, in order to avoid processor array performance degradation multiple communication channels between the memory system and the processor array must be available.

The number of communication channels required for avoiding a processor array performance degradation is related to three aspects: the iteration interval (P), the processor size (denoted by SS_{p_0} and SS_{p_1}), and the number of input/output variables presented in the algorithmic specification. As mentioned in Chapter 3, the iteration interval defines the number of time steps between the evaluation

of two consecutive instances of an indexed variable. Therefore, shorter values of P will lead to increase the number of communication channels, while larger values will decrease the requirement of communication channels. Similarly as the iteration interval, the processor array size impacts directly on the number of communication channels (larger arrays lead to multiple channels while smaller arrays requires few channels). In addition, the number of input/output variables is also proportional to the number of communication channels. Among these three aspects, the input/output variables after space-time mapping is the most important aspect to be taken into account when the external memory is derived; because they classify the communication channels into four possible architectural cases taking into account the processor array size. Moreover, the assumption of $P = 1$ is equivalent to the worst possible scenario for deriving the memory interface. The following subsection describes the relation among the input/output variables after space-time mapping and the derivation of these four architectural cases assuming the worst scenario ($P = 1$).

5.3.2 Architectural Memory Cases

As stated before, the input/output variables can be viewed as a specification of the external memory, and they possess two characteristics that vary depending on the algorithm. The first characteristic is that for each input/output variable, there is an iteration dependent condition $C^I(\vec{I})$ responsible for indicating the index points where the assignation must take place. The second characteristic is a consequence of the first one, and it consists of one dimension of the iteration vector \vec{I} is always set to a constant value due to the iteration dependent condition. Thereby only $n - 1$ dimensions of the n dimensional vector \vec{I} are taken into account for all the input/output variables. In fact, the index variable of the missing dimension is the same index variable present in $C^I(\vec{I})$. Note that in the case of input variables, the iteration condition checks if one iteration index is equal to zero and for output variables the iteration condition checks if one iteration index is equal to the problem size. These two characteristics are useful when an external memory scheme is derived and they should be kept in mind after space-time mapping. After space-time mapping, the iteration dependent condition $C^I(\vec{I})$, and the iteration vector \vec{I} are transformed to time or processor spaces, leading to two mapping

possibilities with different number of communication channels:

1. **Border mapping.** It occurs when the index vector \vec{I} of $C^I(\vec{I})$ is transformed to processor space, and one dimension of the vector \vec{I} in the input/output variable is mapped to the time space. The number of communication channels in this mapping case is equal to SSp_x , where SSp_x could be SSp_0 or SSp_1 depending on the allocation function Φ used.
2. **Broadcast mapping.** It occurs when the index vector \vec{I} of $C^I(\vec{I})$ is transformed to time space, and all dimensions of vector \vec{I} in the input/output variable are mapped to the processor space. In this mapping case, the number of different communication channels is $SSp_0 \times SSp_1$.

For input variables, the first mapping possibility leads to introduce all data stored in external memory from one of the processor array borders, and to extract all produced data from one border in the case of output variables. In contrast, the second possibility leads to introduce and extract data from each PE inside of the processor array for input and output variables, respectively. From the border and broadcast possibilities there are other two possible cases depending on the variable type, *i.e.* if the variable is an input or output type combined with the border or broadcast mapping. Basically, the combination of the two mapping possibilities with the two variable types determine four possible architectural cases. Changing the space-time transformation leads to alter the classification of the I/O variables in any of the four cases. The four architectural cases are:

1. **Input variable border mapping:** the external data are inserted in the processor array borders (figure 5.9.a).
2. **Output variable border mapping:** the data produced by the processor array is placed in the array borders (figure 5.9.b).
3. **Input variable broadcast mapping:** the external data are inserted in each PE inside the processor array regardless of the PE position (figure 5.9.c).
4. **Output variable broadcast mapping** the processor array data results are generated by each PE inside of the processor (figure 5.9.d).

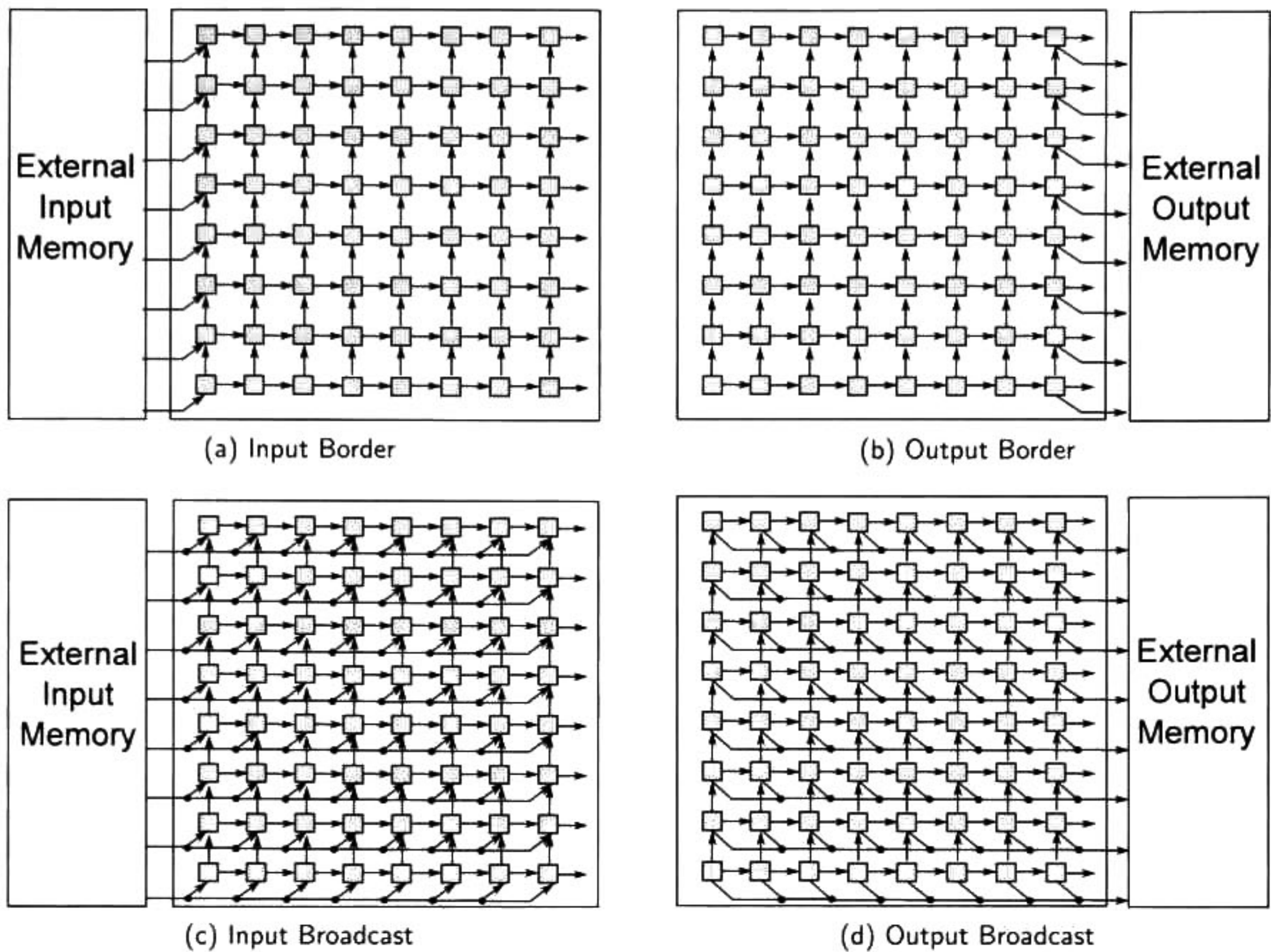


Figure 5.9: Architectural cases according to the variable type and the mapping possibilities.

It should be noted that in the second architectural case (figure 5.9.b), the location of the data results changes depending on the problem size. Recall that when the physical processor is derived, the full-size implementation is interpreted as a logical processor whose PEs are mapped to a physical (partitioned) implementation, and this mapping changes if the problem size is altered. Therefore, for a partitioned array of size SSp_x , there are SSp_x possible places in which data could be produced.

In addition, the number of communication channels required in the border mappings (either for input or output variables) is given by one of the processor array size parameters, *i.e.* SSp_0 or SSp_1 . In the other hand, in the broadcast mapping, the number of different communication channels seems to change in a quadratic way as the processor array size is altered. However all the communication

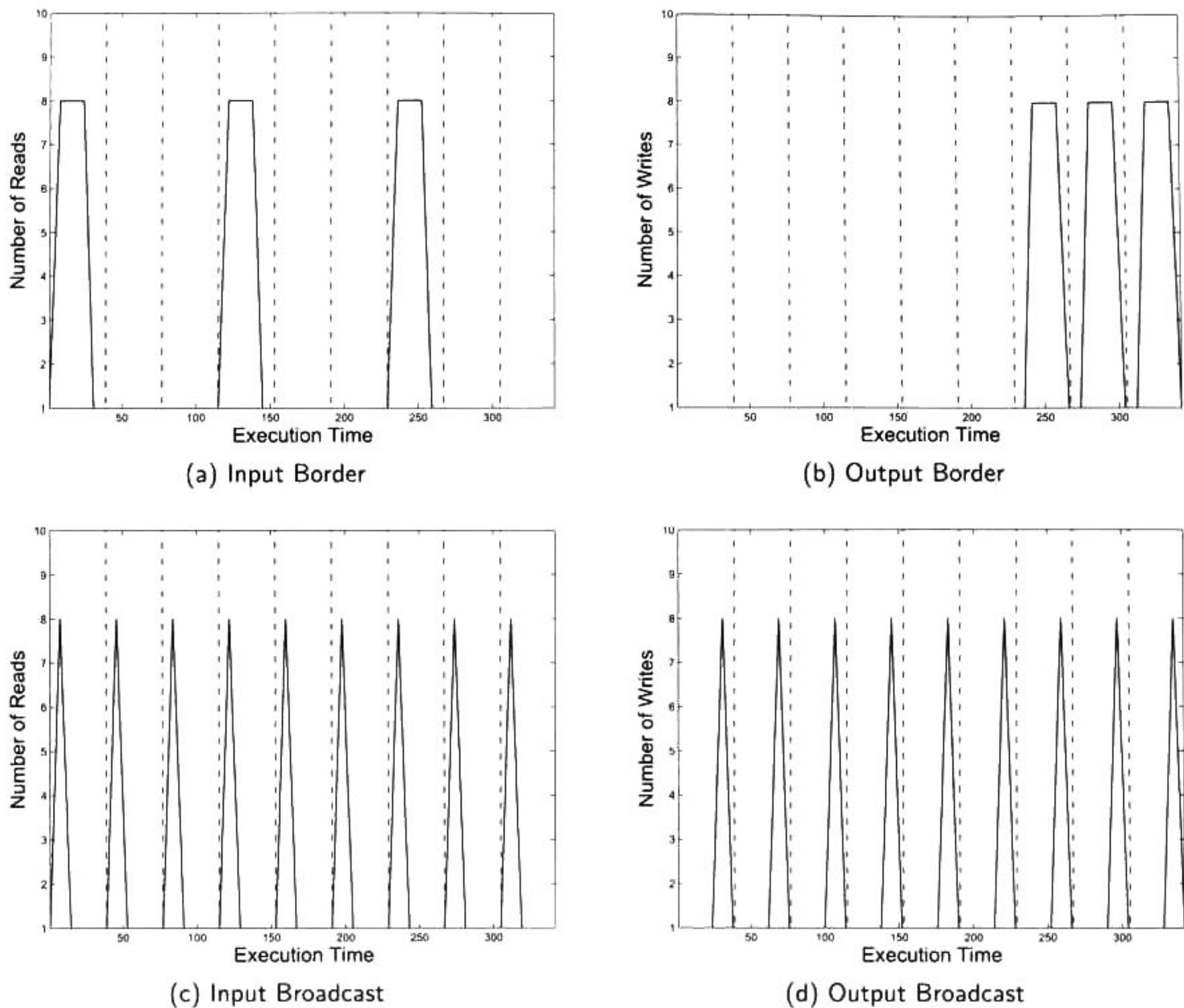


Figure 5.10: Parallel memory accesses according to each architectural cases.

channels are not used at the same time when the broadcast mapping is derived, and consequently the original number of channels ($SSp_0 \times SSp_1$) could be decreased. Figure 5.10 exemplifies the number of parallel external memory accesses required by the four memory cases as a function of the execution time instants. In these graphs, an 8×8 processor array, and a problem size of 24×24 are assumed. The x-axis represents time instants and the y-axis are the number of parallel memory accesses (reads or writes) required at a time instant. Black lines denote the number of memory accesses and the red dashed-lines denote when a new tile iteration has begun. It should be noted that the maximum number of parallel memory accesses is equal to the processor array size in the four architectural

cases. Also, in border mappings, the maximum number of memory accesses is maintained for some time once it has been reached. Contrary, in the broadcast cases, when the maximum number of parallel memory accesses has been reached, the number of memory accesses decreases at the next time instant. Accordingly to these graphs, the maximum number of simultaneous external memory accesses in the broadcast mappings depends on one of the processor array size parameters, like in the border cases. This information is useful since it reduces the communication channels originally required by the broadcast cases. Next subsection describes the internal architecture for each one of the four architectural cases.

5.3.3 External Memory Architectural Scheme

Independently of any of the four architectural cases, the memory scheme is composed by AGUs, memory banks, registers working in serial-input/parallel-output (SIPO) and parallel-input/serial-output (PISO) fashion. The selection of these architectural components, their interconnection, and their internal architecture varies depending on the variable types, the variable index vector \vec{I} , the transformation matrix T , and size of the processor array. In the following subsections the distribution of data among several memory banks, the AGU internal architecture and the memory scheme for each architectural case are explained. The memory scheme satisfies the constraint that all data are required and produced by the processor array during each clock cycle respecting the data dependences. Such constraint can be interpreted as the worst case scenario when the processor array is derived, *i.e.* when $P = 1$. Also, it guarantees that high data demanding algorithms could be supported by the memory scheme without introducing latency. Algorithms like FIR filter, MatMul and matrix-vector multiplication fall inside this category due to their operations can be performed in one clock cycle ($P = 1$). The impact of assuming $P = 1$ in the memory scheme is that external data are distributed into different memory banks working in parallel and in a different clock domain compared to the processor clock. In contrast, for decompositional algorithms like Cholesky, LU and QR the use of memory banks and different clock domains could be relaxed because the operations require several clock cycles to be completed ($P > 1$).

5.3.3.1 Memory Banks

By following the assumption of $P = 1$ and taking into account the maximum number of reads/writes shown in figure 5.10, several simultaneous memory accesses can be required, thus memories with a high bandwidth are required too. In order to overcome such demand, data are stored into different memory banks with a specific organization facilitating their parallel distribution. The data organization takes into account the size of the strips (obtained after strip mining the processor space) and the number of memory banks used. For input/output variables represented as two-dimensional structures, the data are linearized in a row-major or column-major order according to the design necessities [6].

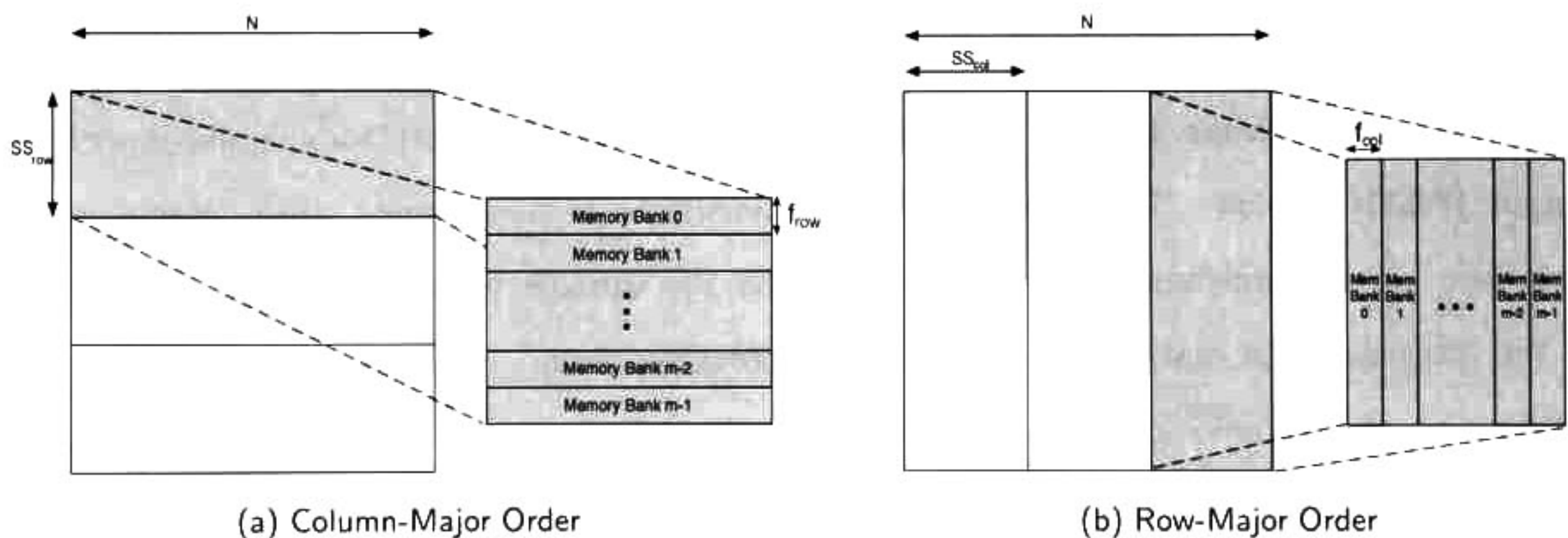


Figure 5.11: Column-major and row-major order cases for matrix data segmentation in different blocks and data distribution of such blocks into different memory banks.

The main idea of the data organization is that a matrix of size $N \times N$ is partitioned into strips of constant size $SSp_x \times N$, where SSp_x is the strip size parameter obtained when strip mining is applied. At the same time, each strip is divided into blocks according to the number of memory banks. The size of these blocks is given by an f factor. Figure 5.11 shows the data distribution for a square matrix of size $N \times N$ in a row-major and column-major order. In both figures, the matrix is partitioned into three blocks of size SSp_{col} and SSp_{row} according to the data distribution. Similarly, each partition is divided into $m - 1$ memory banks by a factor of f_{col} and f_{row} called

memory distribution factors. The number of memory banks (m) for storing a variable is related to the size of the partitioned processor array. Larger processor arrays increase the number of memory banks. Note that the number of strips depends on the matrix size and on the strip size parameters. Also note that a memory bank contains data from different blocks. Figure 5.12 illustrates how the matrix is distributed into two memory banks in row-major and column-major order. Different blocks are represented by different grey shades. The matrix elements are placed contiguously if they belong to the same memory bank despite of they are in different blocks. The memory banks are labeled from zero to $m - 1$.

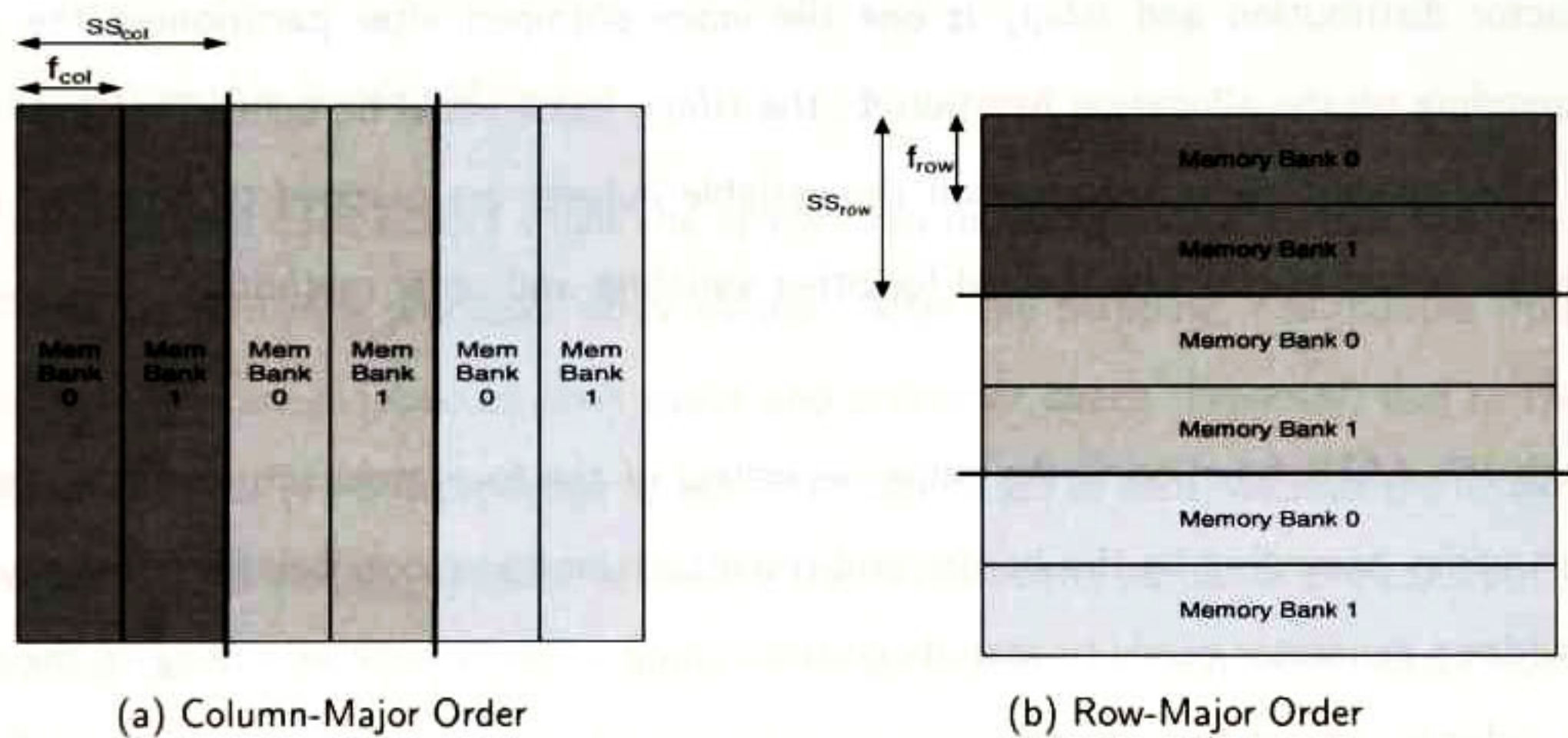


Figure 5.12: Examples of data segmentation and distribution for two memory banks in the column-major and row-major order cases.

5.3.3.2 Address Generator Unit

The memory banks, required by the architectural cases for storing data, require modules in charge of generating their memory addresses. Address generator units are responsible for such task and there are at least as many AGUs as memory banks exist. An AGU contains a combinational module in charge of generating the addresses following a mathematical formula. The formula parameters are in function of the problem size, the indexes of the input/output variable after space-time transformation, the tile index, the memory distribution factor f , and the corresponding memory bank id . The formula

is expressed according to the linear order imposed by row-major or the column-major methods. Formula 5.2 shows an example for the input variable $A_{in}[i, 0, k]$ in the MatMul PRA using a column-major order.

$$MemAddrA_j = [N \times (k + tilep_x)] + (i - tilep_x) - [f_{col} \times N (tilep_x + MemBank_j)] \quad (5.2)$$

where $MemAddressA_j$ is the memory address for the j -th memory bank, i, k are the indexes used for the A_{in} variable, N is the problem size, $MemBank_j$ is the memory bank id, f_{col} is the memory factor distribution and $tilep_x$ is one tile index obtained after partitioning the processor space. Depending on the allocation function Φ , the $tilep_x$ index could be defined a $tilep_0$ or $tilep_1$. Moreover, after space-time transformation the variable indexes are mapped to time and processor spaces. Similar formulas could be derived for other variables and order methods.

Although the AGUs function is the same regardless of the four architectural cases, its internal architecture varies according to the border and broadcast mapping possibilities. Basically, in both cases the address generator needs to scan its corresponding memory bank according to the scheduler function in order to respect the activation sequence imposed by the transformation matrix T . Since in the case of border mapping one of the indexes of the input/output variable is mapped to time space, this index could be used for scanning a memory bank; and the remaining index (mapped to processor space) could be tied to a constant value depending on the position in the processor array border. Therefore, the scanning of the memory bank can be achieved by using the time index value generated by the sequence generator module described in Chapter 4. On the other hand, in the case of the broadcast mapping both variable indexes are mapped to processor space, being only possible to tie one processor index to a constant value (as in the border mapping case). Consequently, the remaining processor index must be scanned by using counters. Figure 5.13 shows the AGU internal architecture for border and broadcast mapping possibilities. Recall that the *IndexBus* contains information like problem size, the tile and time indexes (see Chapter 4).

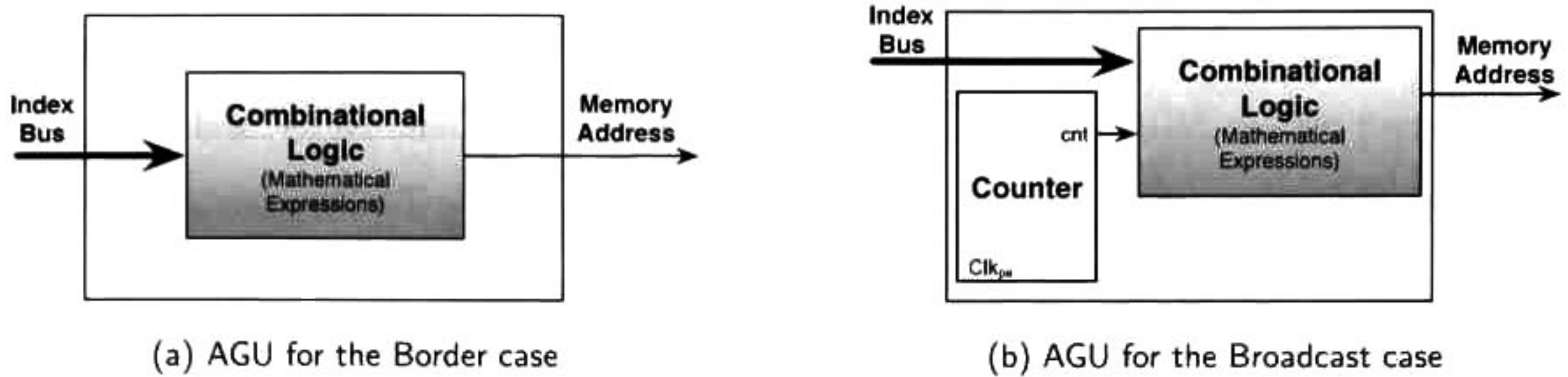


Figure 5.13: The two possible AGUs architectures for border and broadcast mapping.

5.3.3.3 Input Variable Border Mapping

The first architectural case occurs when the space-time mapping indicates that the input variable of the PRA is inserted in a processor array border. After the mapping, the indexes from vector \vec{I} of the input variable are mapped to time space and processor space. However, due to the iteration dependent condition $C^I(\vec{I})$ one processor space dimension is set to zero whereas the other dimension remains variable. This leads to the idea of placing the AGUs at the border of the constant dimension, and generating the remaining processor index. The generation of such index is achieved by using several AGUs. In the case of partitioned arrays the number of AGUs is equal to the processor array size indicated by SSp_x .

A first approach for data extraction is having a memory bank per each AGU. However, this leads to impractical situations where several small memory banks are required. Besides, the distribution of matrix data to these banks could result in a bottleneck problem. A possible solution of this situation is assuming that it is possible to extract two data per memory port in a processor clock cycle. This assumption requires two clock domains with different clock frequencies like a in globally-asynchronous locally-synchronous approach (GALS) [101]. The external memory clock frequency Clk_{mem} is twice faster than the processor array clock frequency Clk_{pa} . As a result of the GALS approach, two AGUs per address port are required in order to generate two different addresses in a

processor cycle. The selection of the addresses generated by these two AGUs is accomplished by a multiplexer and a control unit. The control unit is a single bit flip-flop T whose input is tied to '1' working in the memory clock domain. The purpose of the flip-flop T is to interchange the multiplexer output between the two AGUs. The interconnection of two AGUs, the multiplexer, and the flip-flop T is called two-address generator module (TAGM). Figure 5.14 shows the TAGM internal architecture.

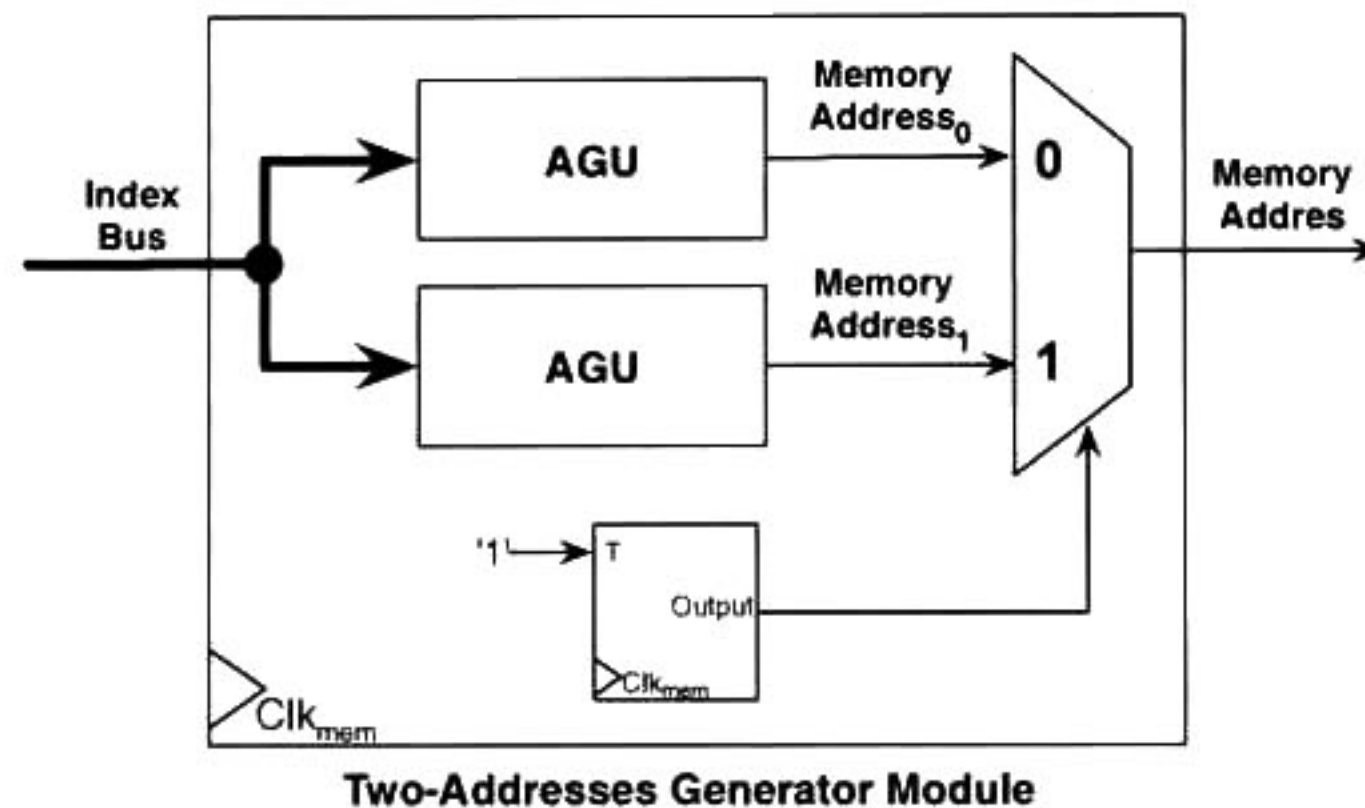


Figure 5.14: Two-Address generator module in charge of generating two memory bank addresses in one processor array clock cycle.

Another possible assumption for data extraction is the use of dual-port memories, leading to increase twice the data extraction rate, and totalizing a 4x data rate compared with the one memory bank approach. Moreover, regardless of the memory bank is multi-port or not, the memory bank produces two data per memory port in a processor clock cycle, thereby at least one of these data must be stored in one memory clock cycle. Besides, it must be coupled both clock domains in order to ensure the setup and hold timing from the memory clock domain to the processor clock domain. Since data flow from the faster domain to the slower one, an interface from a faster to a slower domain is needed. Such interfacing is performed by a SIPO interface which receives two data extracted from the memory bank, and sends both data in the next processor clock cycle. This SIPO interface consists of two-registers pairs controlled by the different clock domains. Each pair of registers works on parallel with respect of the other pair. Figure 5.15 shows the interconnection of the SIPO interface. Finally, the interconnection of the TAGM, the dual-port memory bank and the

SIPO is shown in figure 5.16. Note that in this case the extraction of four data from each memory bank in one processor clock cycle is possible. Also, note that each SIPO output is connected to one PE placed at the processor array border.

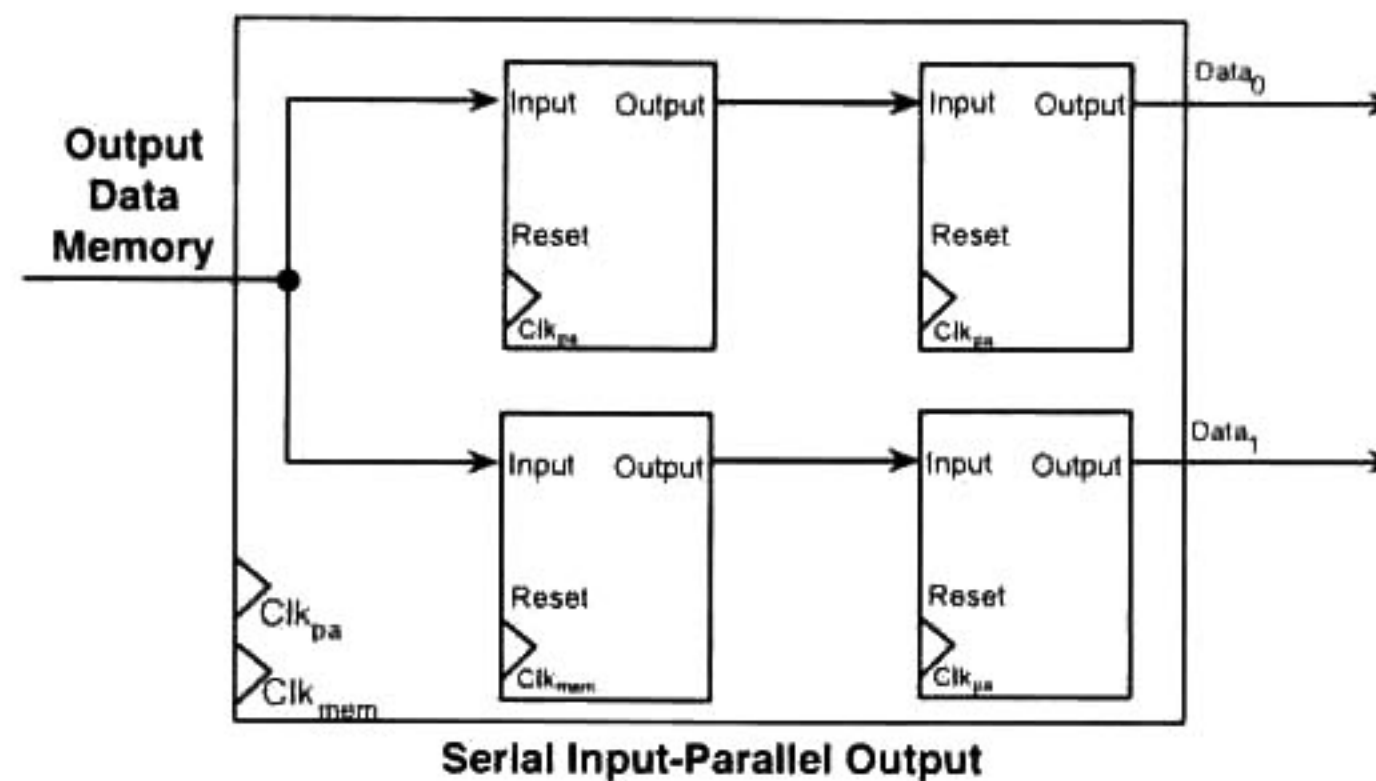


Figure 5.15: SIPO border module in charge of interfacing two data from the faster clock domain to the slower one.

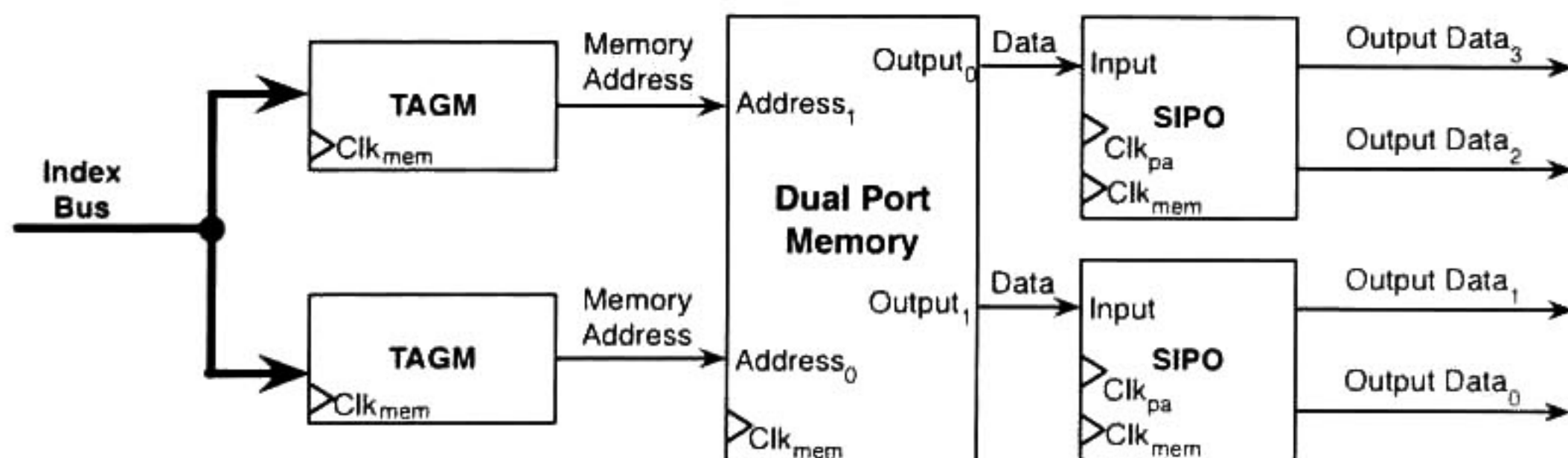


Figure 5.16: Interconnection of the TAGM, dual-port memory and SIPO for the input border case.

5.3.3.4 Output Variable Border Mapping

The second architectural case occurs when the space-time mapping indicates that the output variable of the PRA is extracted from a processor array border. Similarly to the input variable case, after space-time transformation the $C^I(\vec{I})$ indicates that one processor dimension is set to a constant value (problem size) whereas the other dimension remains variable. Therefore, the address generation is

tackled in similar way as the input variable border case, where the processor index is generated by replicating several AGUs at the processor array border. In fact, if the GALS and dual-port memories assumptions are used, the same TAGMs could be used in order to generate the memory addresses similarly to the input variable case. Since each TAGM produces two memory addresses in one processor cycle, it is possible to insert two data produced by the array sharing the same input port. The selection of the two inputs from the processor array is done by a multiplexer and a flip-flop T (like in TAGM) interconnected as shown in figure 5.17. This PISO module is connected to one input port of the memory bank. Figure 5.18 shows the interconnection of the TAGM, PISO and dual-port memory for storing the data results from the processor border. Note that, in the same way as in the input variable case, it is possible to recollect four data from the array in one processor clock cycle.

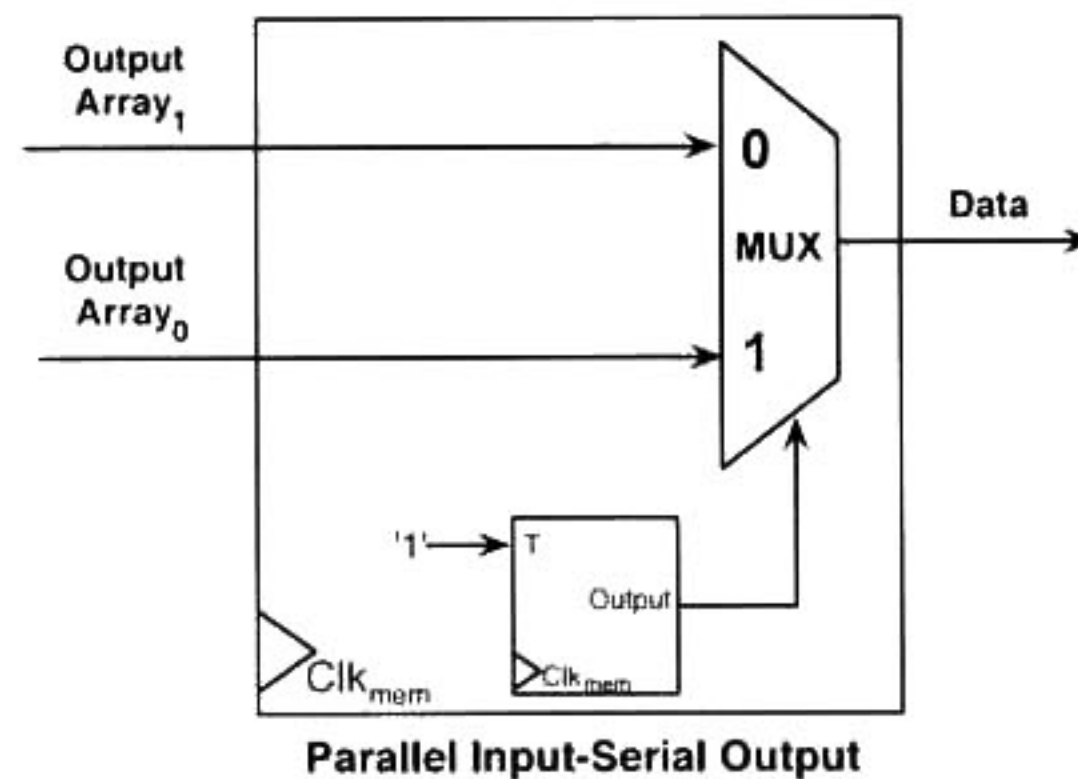


Figure 5.17: PISO border module in charge of multiplexing two processor array inputs.

Although data produced by the processor array is recollected at the processor borders, it is not necessarily produced by the border PEs in a partitioned processor array. This situation occurs when the problem size does not fit exactly in the partitioned array, *i.e.* there is not a valid mapping from the logical to physical processor. In such case data are produced by inner PEs, and these data must be sent to the processor array border. The localization of the column (or row) where the data are produced is given by $N \bmod SS p_x$, where N is the problem size and $SS p_x$ is the strip size parameter. Consequently, if $N \bmod SS p_x = 0$ the output data are produced at the border PEs, otherwise the output could be produced at any column/row of the processor array.

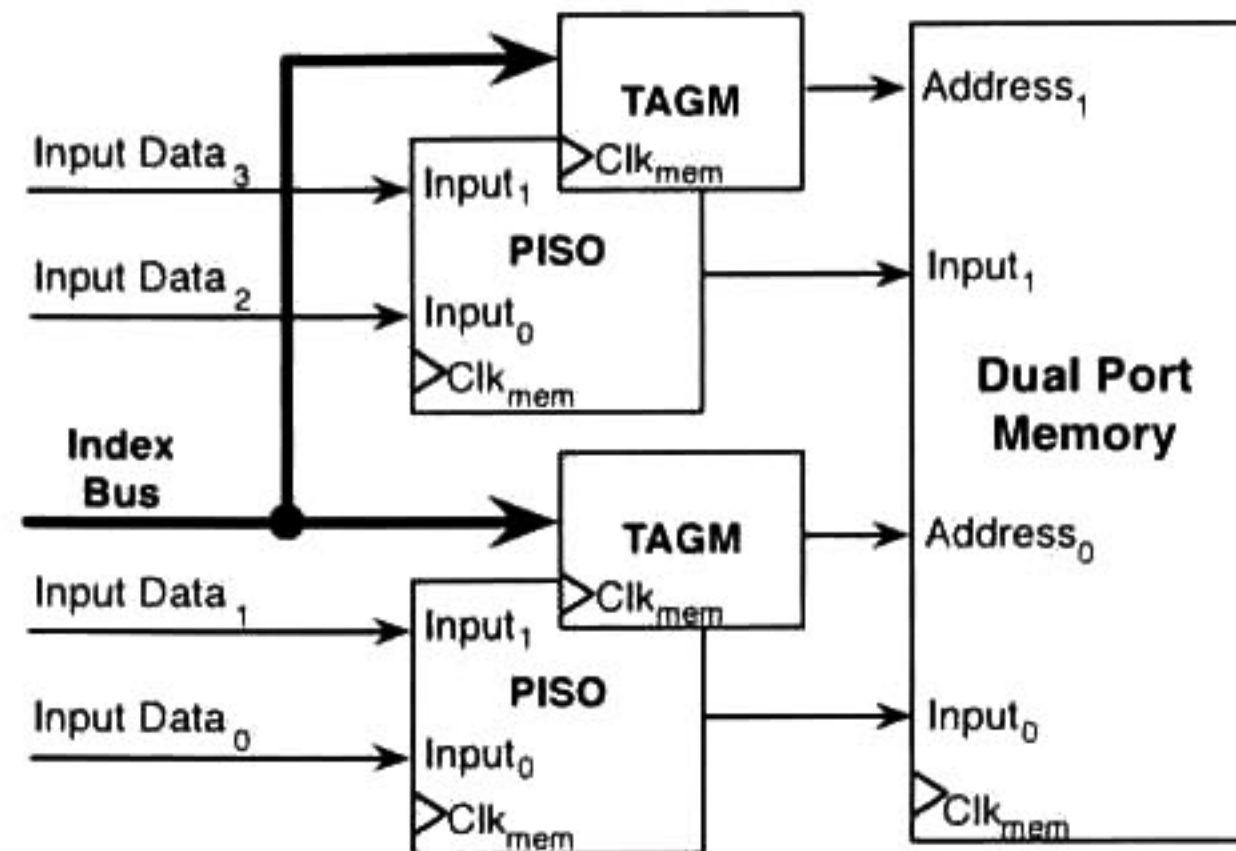


Figure 5.18: Interconnection of the TAGM, dual-port memory and PISO for the output border case.

The sending of output data originated inside the array is accomplished by placing a layer of transporting elements (TE) following the interconnection structure of the processor array. Figure 5.19 depicts the TE internal architecture, which consists of a multiplexer in charge of selecting between the data results produced by the PE or the result produced by its previous neighbor. The selection signal is the *PE_Enable* signal produced by the control cell from the control array (Chapter 4). The register stalls the multiplexer output one clock cycle, thus, data arrive to the array border with a delay of $N - (N \bmod SS_{p_x})$ clock cycles.

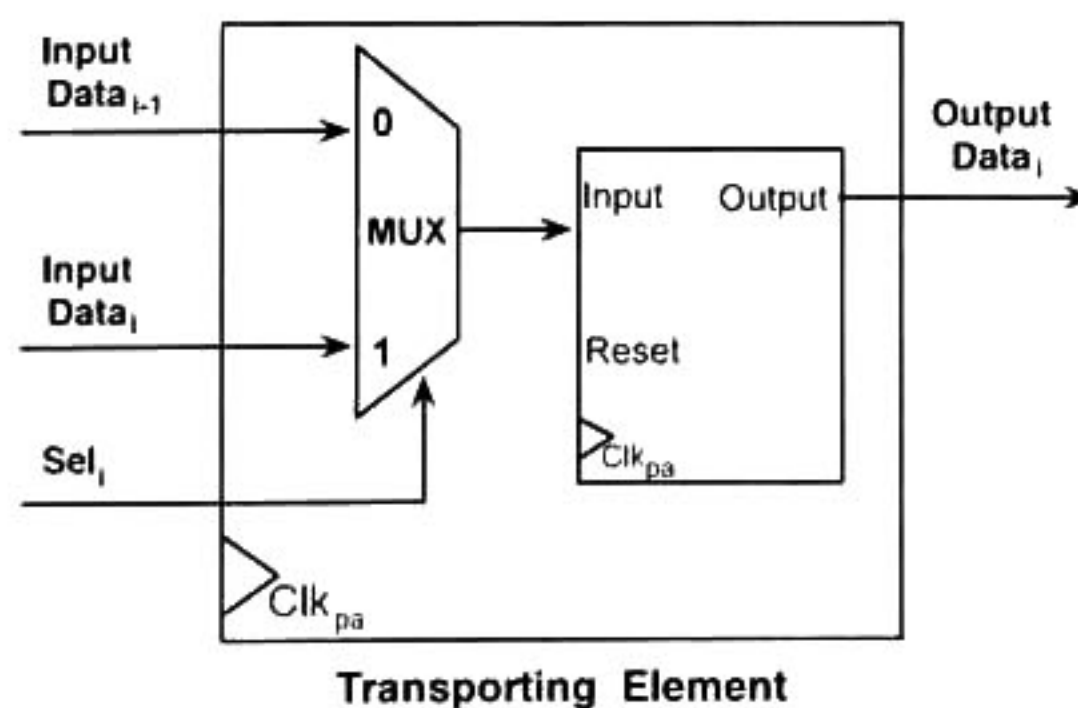


Figure 5.19: Transporting element.

5.3.3.5 Input Variable Broadcast Mapping

The third architectural case happens when the space-time mapping indicates that the input variable of the PRA is inserted in each PE of the processor array. After the mapping, the iteration dependent condition $C^I(\vec{I})$ is mapped to time space while the indexes from vector \vec{I} of the input variable are mapped to processor space. In this case the AGUs should scan one of the processor indexes (as shown in figure 5.13) while the other index is generated by replicating the AGUs at one border of the processor array. Therefore, a similar approach to the input variable border case could be used for generating the memory bank addresses as well as for storing data in dual-port memory banks. Figure 5.20 shows the interconnection of the TAGM, the dual-port memory and the SIPOs interconnection. However, note that there are four SIPO modules instead of two like in the case of input border case. This is because in this architectural case it is required a different kind of storage-interface module compared with the input border case.

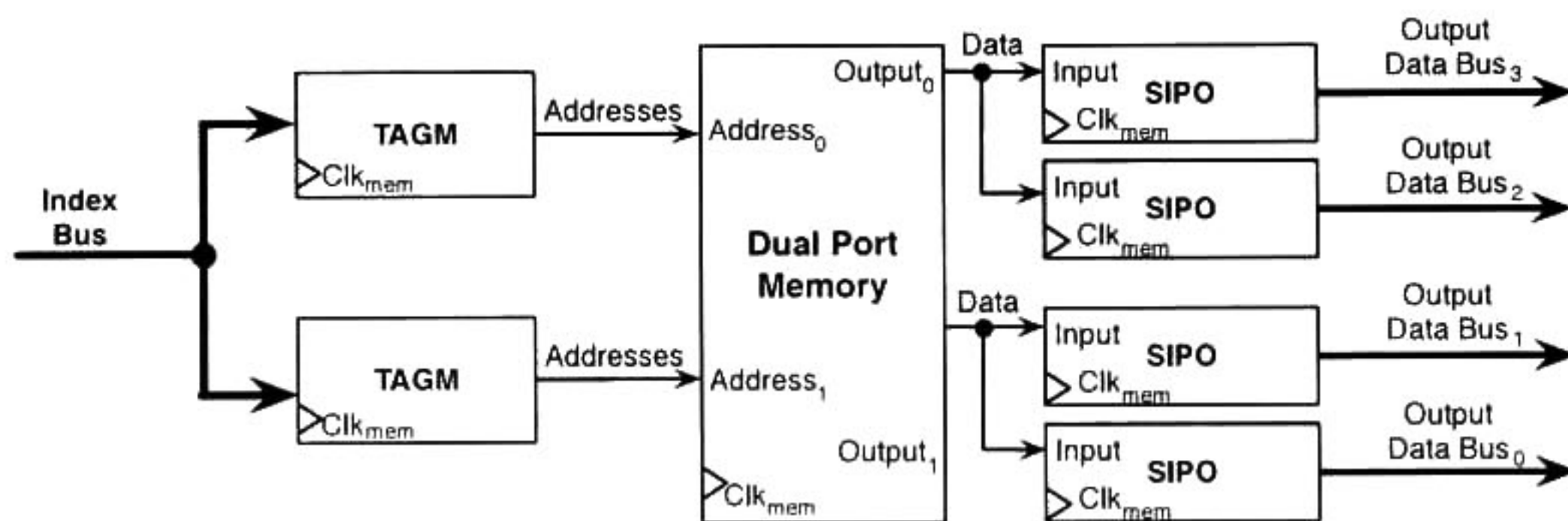


Figure 5.20: Interconnection of TAGM, dual-port memory and SIPO for the input broadcast case.

Contrary to the border case where data are fed continuously at each processor cycle (like a burst mode), the broadcast case only requires a block of data each time a new tile is being scanned. The size of this data block in the worst case scenario is equal to the total number of PEs in the physical array *i.e.* $SSp_0 \times SSp_1$ data. A first approach for inserting such amount of data to the processor array is by sending directly each datum to its corresponding PE in a broadcast approach. Unfortunately, this approach could decrease the maximum operational clock frequency, since there

are inserted long communication lines across the processor array. Another possible way for sending the data into the array is by dispatching the data block in a pipeline fashion. However, the pipelined way leads to synchronization problem, because data extracted from memory banks would not arrive at the proper time at its corresponding PE, *i.e.* the data arrival would not respect the scheduler function. Such situation could be dealt by forwarding the extraction of the whole data block before scanning the current tile, and by sending in advance the data to the PE where each datum is required. Such forwarding requires sending all data contained into sub-blocks from one processor border, and sending these sub-blocks simultaneously. Each time that a sub-block passes through a pipeline stage, one datum of the sub-block is taken by a PE while the remaining data are sent to the next pipeline stage. Therefore, for a processor array of size $SSp_0 \times SSp_1$, if the AGUs are placed in the p_0 direction, then in the first pipeline stage SSp_0 data sub-blocks of size $SSp_1 - 1$ are sent; in the second pipeline stage one datum from the data sub-block of size $SSp_1 - 1$ is used, and SSp_0 data sub-blocks of size $SSp_1 - 2$ are sent, and so on until it is only sent a datum to the processor array border. This approach calls for a high quantity of registers for storing the pipelined data at each pipelining stage. In fact, the set of such registers could be abstracted as an array of registers where the number of registers is decreasing as data get further from the border. This array is called broadcast data array and it is shown in figure 5.21. Grey boxes represent the set of registers and the numbers above the lines indicate the amount of data that it is being pipelined.

As consequence of the first pipeline stage, storing SSp_x data extracted from a memory bank is required. Figure 5.22 shows the internal architecture of the SIPO for the broadcast case. Note that after SSp_x processor clock cycles the SIPO is full and all data are ready to be sent to the broadcast data array. Also, note that the number of processor cycles required to extract an entire data block from m dual-port memory banks is $l_{init} = (SSp_0 \times SSp_1)/(4m)$. Therefore, the data extraction from the memory bank should start l_{init} processor clock cycles before any tile is scanned, *i.e.* it is introduced an initial latency before the processor array starts its computations.

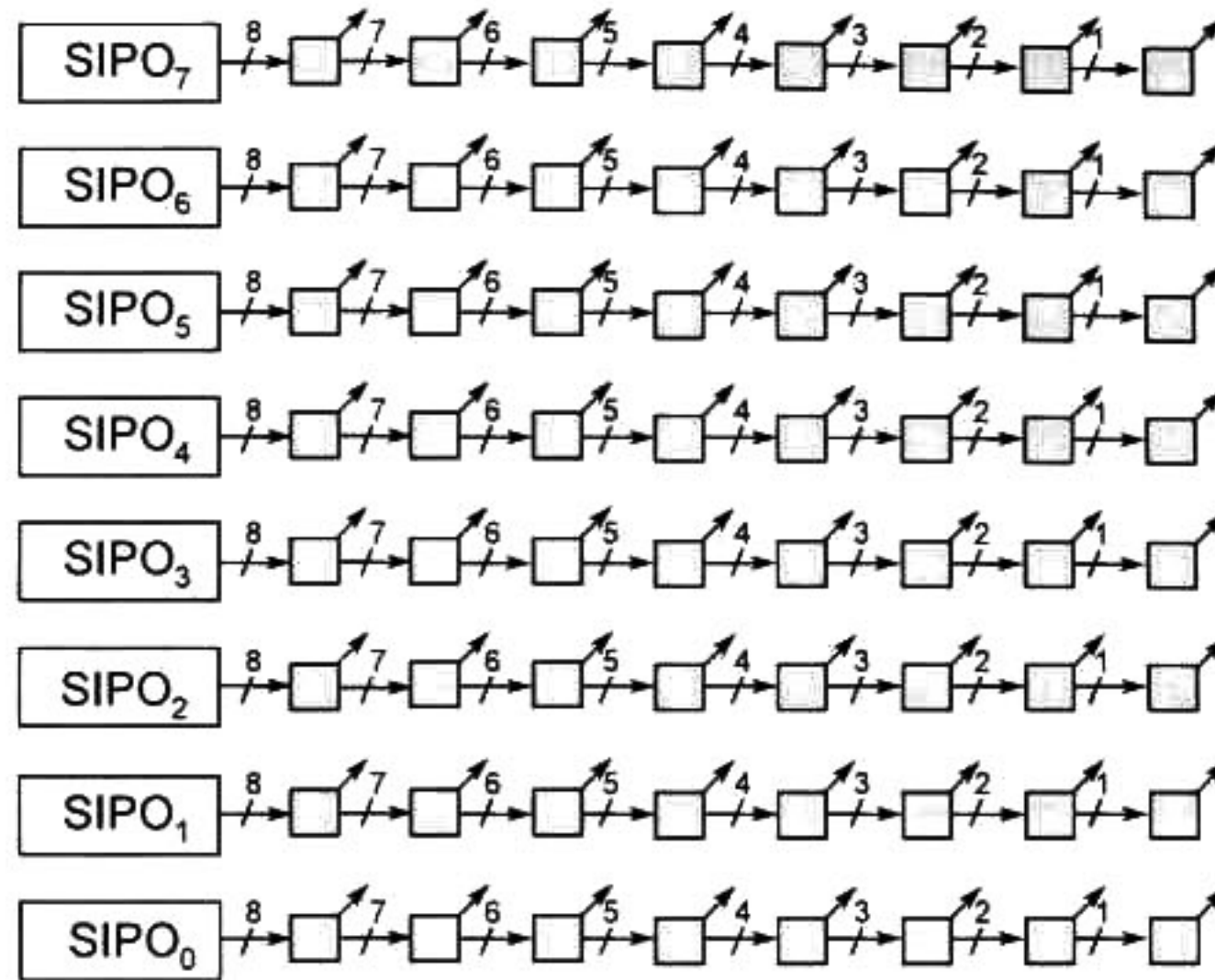


Figure 5.21: Broadcast data array for an 8×8 processor array. From the SIPO side to the other border, the data bus is being decreased.

5.3.3.6 Output Variable Broadcast Mapping

The final case occurs when the space-time mapping indicates that the output variable of the PRA is extracted from each PE of the processor array. Like in the input variable broadcast case, after the mapping, the iteration dependent condition $C^I(\vec{I})$ is mapped to time space while the indexes from vector \vec{I} of the input variable are mapped to processor space. This case shares some similarities compared to the input broadcast case. For instance, in both cases the concepts of data block, pipelining stages, and broadcast array appear too. However, in the output variable case since data are produced by the PEs sending the data through the pipeline stages until data have reached the array border is required. Note that in this case the number of registers is increased as data get closer to the array border, therefore, if the AGUs are placed in the p_0 direction the total amount of data which arrive to the border is $SSp_0 \times SSp_1$ divided in SSp_0 sub-blocks. The PISO broadcast module (figure 5.23) receives a data bus of a sub-block and it selects one datum to be stored in the memory bank by using a $n - 1$ multiplexer. The multiplexer selector signal is generated by a counter which scans all data contained in the data sub-block.

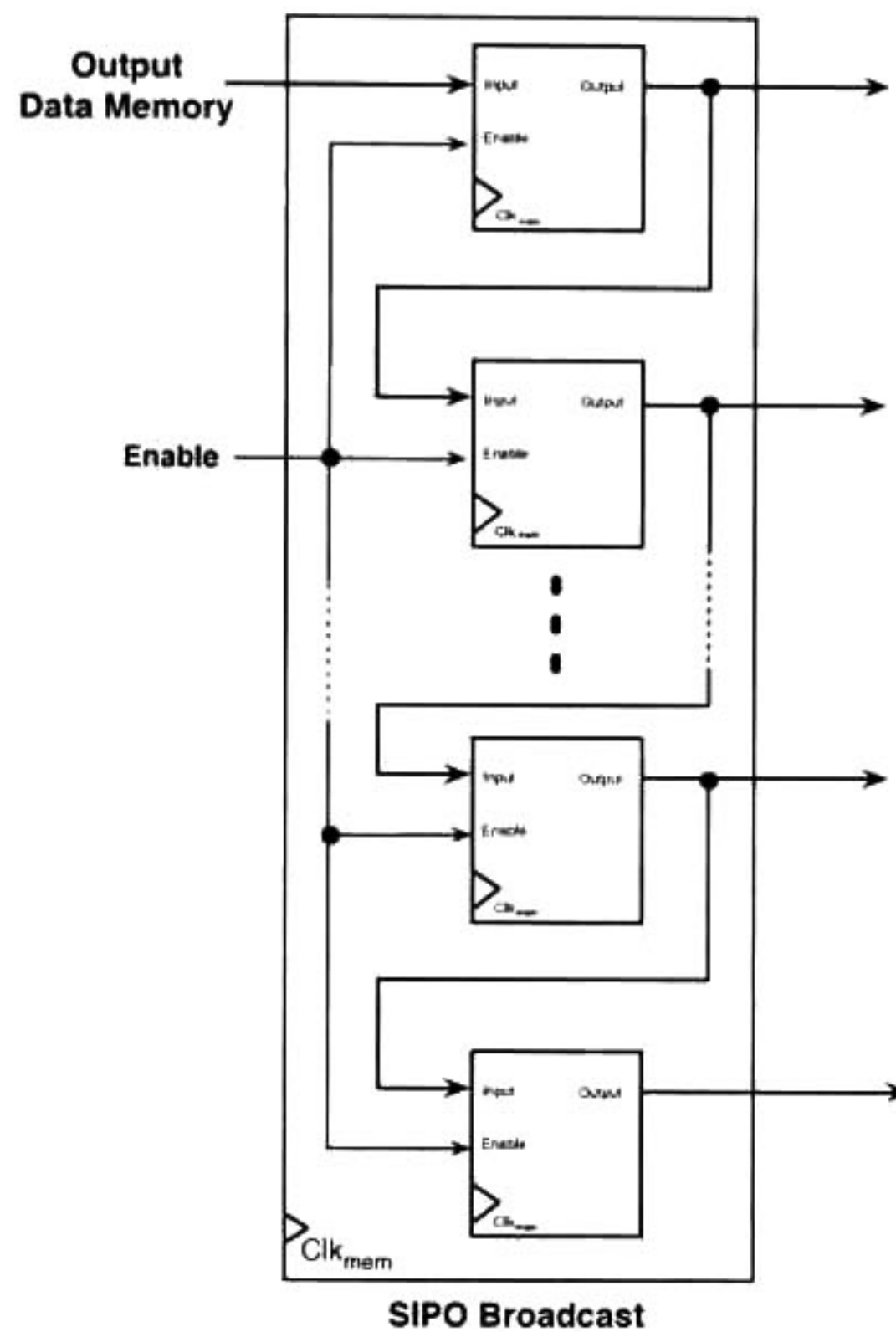


Figure 5.22: SIPO broadcast module in charge of sending data to the processor array.

Finally, figure 5.24 shows the interconnection of the TAGM, the dual-port, and a two-PISO modules placed together. Note that as the previous cases it is possible to store four data to the memory bank in one processor clock cycle. Also, note that as in the case of the input broadcast case, it is required $l_{end} = (SSp_0 \times SSp_1)/(4m)$ processor cycles to store an entire data block in m dual-port memory banks.

The number of logic elements like counters, memories, adders and multiplexers required by each memory architectural case can be characterized in terms of the processor array size. This characterization is useful for providing an idea of the number of computational elements required by each architectural case independently of the implementation technology used. In this sense, the following section presents the characterization of the number of logic elements.

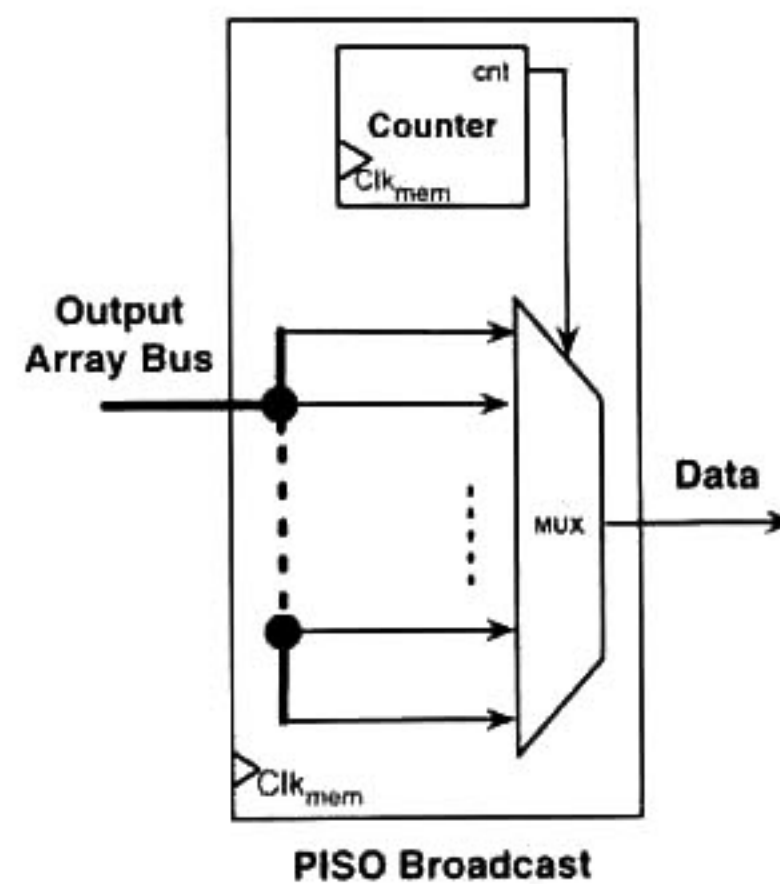


Figure 5.23: SIPO broadcast module in charge of sending several data to the processor array.

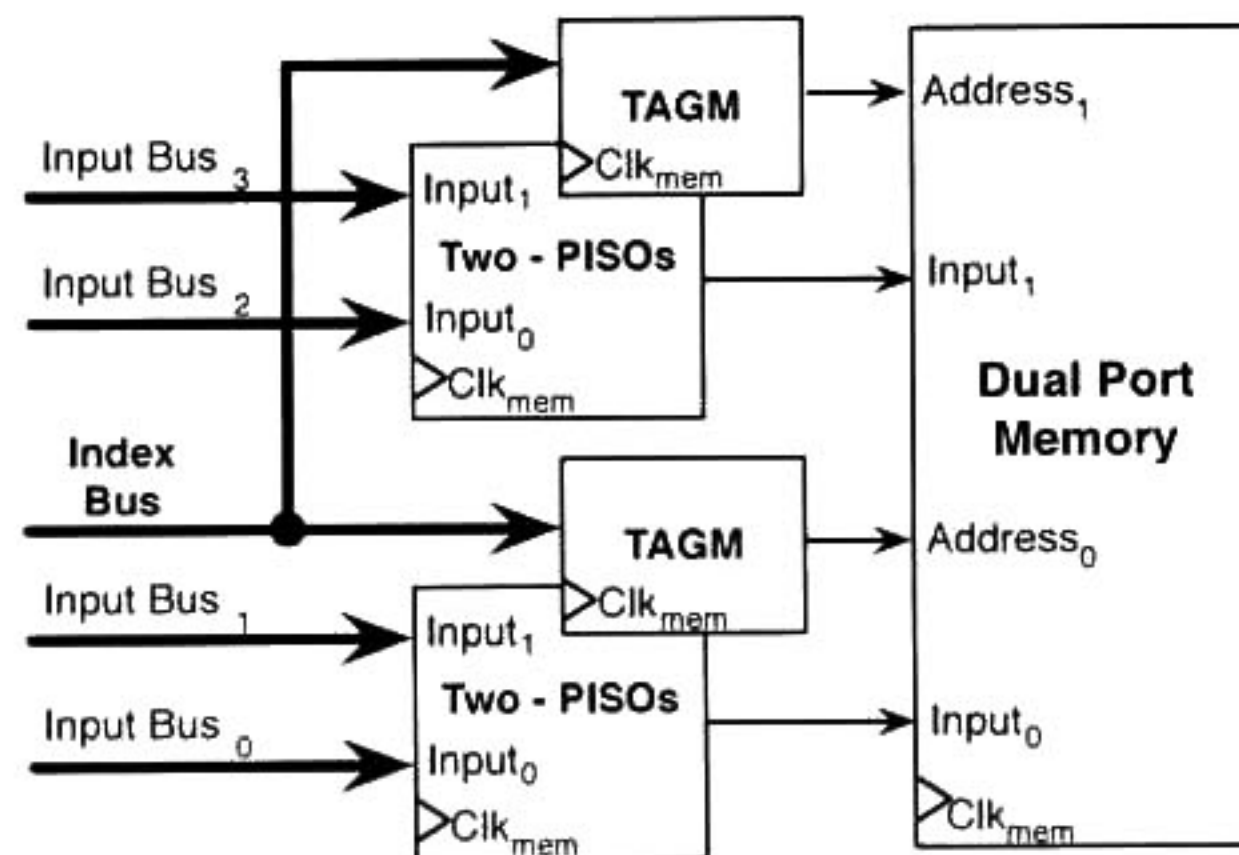


Figure 5.24: Interconnection of TAGM, dual-port memory and PISO for the output broadcast case.

5.4 Number of Logic Elements

The number of hardware elements required by each memory architectural case is shown in table 5.1 in terms of the processor array size (strip size parameters), and the constants c_0 , c_1 , n and f_m . The SSp_x and SSp_y terms refer to any one of the two strip size parameters (SSp_0 or SSp_1), c_0, \dots, c_4 are constant values which depend on the scheduler function, and $g(SSp_x, SSp_y)$ is a function defined as shown in equation 5.3. The use of multi-port memories is represented by n , while f_m is the factor for which the memory system is faster than the processor array.

$$g(SSp_x, SSp_y) = SSp_y \left(\sum_{i=1}^{SSp_x} i \right) + (SSp_x \times SSp_y) \quad (5.3)$$

The assignation of SSp_x and SSp_y variables to SSp_0 and SSp_1 parameters is done according to the processor space direction where the AGUs are placed. If AGUs are placed in p_0 direction, then $SSp_y = SSp_0$ and $SSp_x = SSp_1$; and if the AGUs are placed in p_1 direction, then $SSp_x = SSp_0$ and $SSp_y = SSp_1$. Note that when the input border case is derived, the number of hardware elements grows in a linear way with respect of the strip size parameters. The number of multiplexers and registers in output border case changes at the same rate as the number of PEs in the processor array is altered; but the number of registers for both broadcast cases increase in a cubic factor with respect of the processor array size. Furthermore, if the assumptions of dual-port memory ($n = 2$) and the memory clock frequency doubling the processor array clock frequency ($f_m = 2$) are satisfied, then the number of memory banks is decreased by a factor of four. Besides, if $nf_m > SSp_x$ then any of the two memory assumptions could be relaxed until at least $nf_m = SSp_x$.

Number of	Input Border	Output Border	Input Broadcast	Output Broadcast
Adders	$6(SSp_x)$	$6(SSp_x)$	$6(SSp_x)$	$6(SSp_x)$
Multipliers	$2(SSp_x + c_0)$	$2(SSp_x + c_1)$	$2(SSp_x + c_2)$	$2(SSp_x + c_3)$
Multiplexers	SSp_x/n	$(SSp_0 \times SSp_1) + (SSp_x/n)$		SSp_x
Counters			SSp_x	$2(SSp_x)$
Registers	$2(SSp_x)$	$SSp_0 \times SSp_1$	$g(SSp_x, SSp_y)$	$g(SSp_x, SSp_y)$
1-Bit FFs-T	SSp_x/n	SSp_x/n		
Memories	$SSp_x/(nf_m)$	$SSp_x/(nf_m)$	$SSp_x/(nf_m)$	$SSp_x/(nf_m)$

Table 5.1: Hardware resource utilization for each memory architectural case.

5.5 Matrix-Matrix Multiplication Case of Study

The external memory system is designed to support different transformations and four different architectural cases. In the case of the AGUs, only by changing the mathematical expressions

mapped to combinational logic different transformation matrixes could be supported. Besides if the transformation matrix is changed, it could be also altered the architectural case, and consequently major changes in the memory system should be performed. With the purpose of exemplify such characteristics, this section presents as case of study the memory system for MatMul algorithm when the scheduler function $\vec{\lambda}_l = [1, 1, 1]$, the projection vector $\vec{u} = [1, 0, 0]$, and $P = 1$ are used. The size of the strips for this case are $SSp_0 = 8$ and $SSp_1 = 8$ leading to a partitioned processor array of 8×8 PEs. For this case of study, it is assumed that the hybrid control is already generated and that the memory banks are stored in a column-major order.

<i>eqn00</i>	$y[i, j, k] = A_{in}[i, j, k]$	if ($j = 0$)
<i>eqn01</i>	$x[i, j, k] = B_{in}[i, k, j]$	if ($i = 0$)
<i>eqn02</i>	$y[i, j, k] = y[i, j - 1, k]$	if ($j > 0$)
<i>eqn03</i>	$x[i, j, k] = x[i - 1, j, k]$	if ($i > 0$)
<i>eqn04</i>	$w[i, j, k] = y[i, j, k] \times x[i, j, k]$	
<i>eqn05</i>	$z[i, j, k] = z[i, j, k - 1] + w[i, j, k]$	if ($k > 0$)
<i>eqn06</i>	$z[i, j, k] = w[i, j, k]$	if ($k = 0$)
<i>eqn07</i>	$C_{out}[i, j, k] = z[i, j, k]$	if ($k = N - 1$)

Figure 5.25: Matrix-Matrix Multiplication piecewise regular algorithm.

The MatMul PRA is shown in figure 5.25. The input variables are denoted by variables A_{in} and B_{in} and the output variable is C_{out} . Recall that matrixes A and B used in the multiplication are embedded into the iteration space by the equations $A_{in}[i, j, k] = A_{i,k}$ and $B_{in}[i, j, k] = B_{k,j}$. In the same way, the resulting matrix C is stored in variable C_{out} . In the case of variable A_{in} the iteration dependent condition indicates that a reading from the external memory is only done when $j = 0$, thus the index points for variable A_{in} can be reduced to $A_{in}[i, 0, k]$. Similarly variables B_{in} and C_{out} are reduced to $B_{in}[0, k, j]$ and $C_{out}[i, j, N - 1]$, respectively. After space-time transformation the source polytope index space is mapped as follows:

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} t - p_0 - p_1 \\ p_0 \\ p_1 \end{bmatrix}$$

The space-time mapping indicates that the source polytope index i is mapped to time space whereas indexes j and k represent the processor space. The new index points for variables A_{in} , B_{in} and C_{out} are $A_{in}[t - p_0 - p_1, 0, p_1]$, $B_{in}[0, p_1, p_0]$, and $C_{out}[t - p_0 - p_1, p_0, N - 1]$, respectively. Thereby variables A_{in} and C_{out} are enclosed in the *border mapping* possibility and variable B_{in} in the *broadcast mapping* possibility. In the case of variable A_{in} it falls into the input variable border mapping. Because of the use of a dual-port memory leads to have four memory data for each processor clock cycle, in this case of study two memory banks are required. The AGUs are placed at border p_0 and replicated through the processor index p_1 . The formula mapped to the AGU is shown in expression 5.4. Figure 5.26 shows the memory architecture for variable A_{in} .

$$MemAddr A_j = [N \times (p_1 + tilep_1)] + (t - p_0 - p_1 - tilep_1) - [f_{col} \times N (tilep_1 + MemBank_j)] \quad (5.4)$$

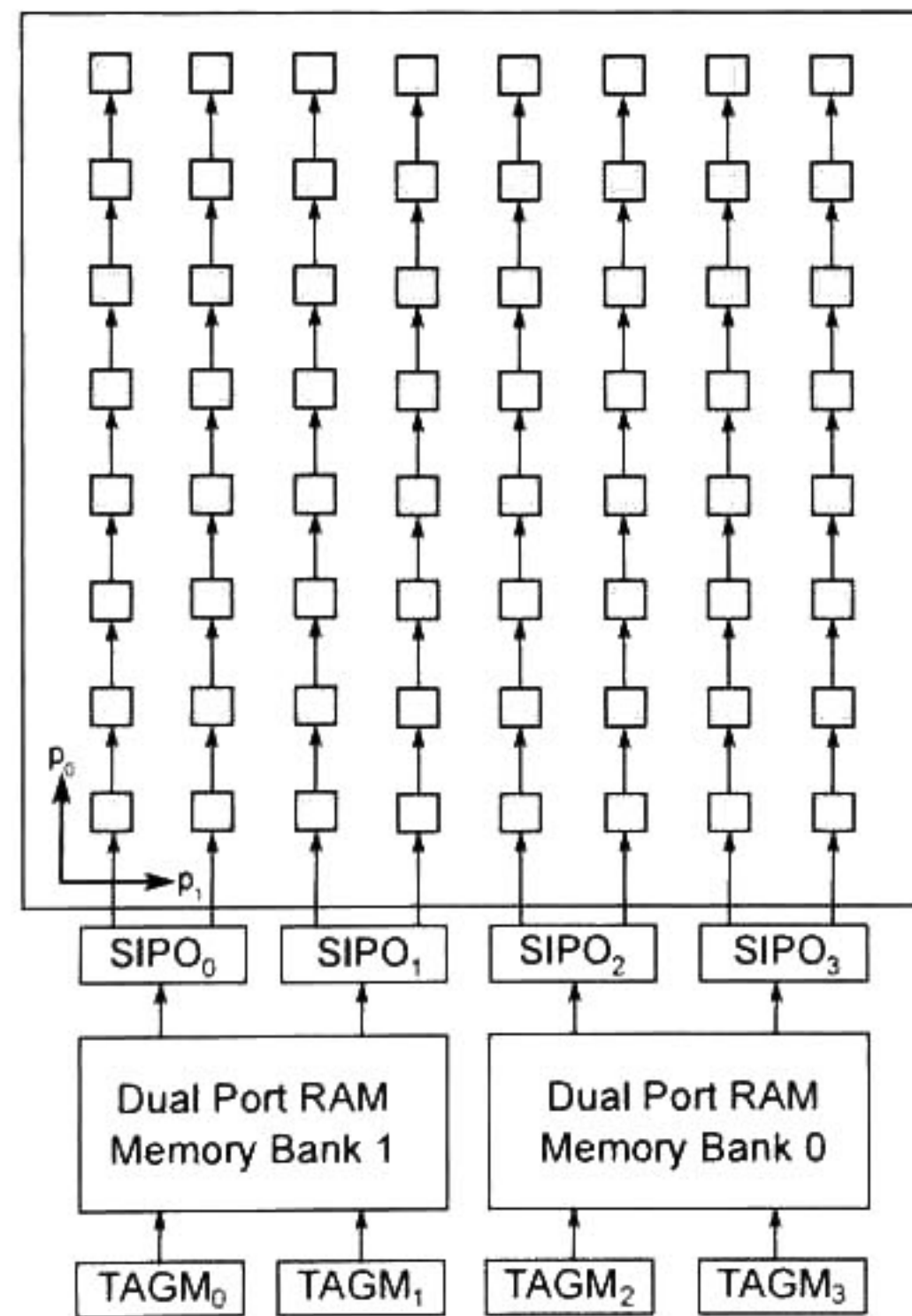


Figure 5.26: Architectural border case for input variable A_{in} .

Variable B_{in} falls into the input variable broadcasting mapping. In this case the AGUs are placed at border p_1 and replicated through the index p_0 . The formula that is mapped to the AGU is shown in expression 5.5. Figure 5.27 shows the memory architecture and the data array for variable B_{in} .

$$MemAddrB_i = [N \times (p_0 + tilep_0)] + (p_1 - tilep_1) - [f_{col} \times N ((p_1 - tilep_1) + MemBank_i)] \quad (5.5)$$

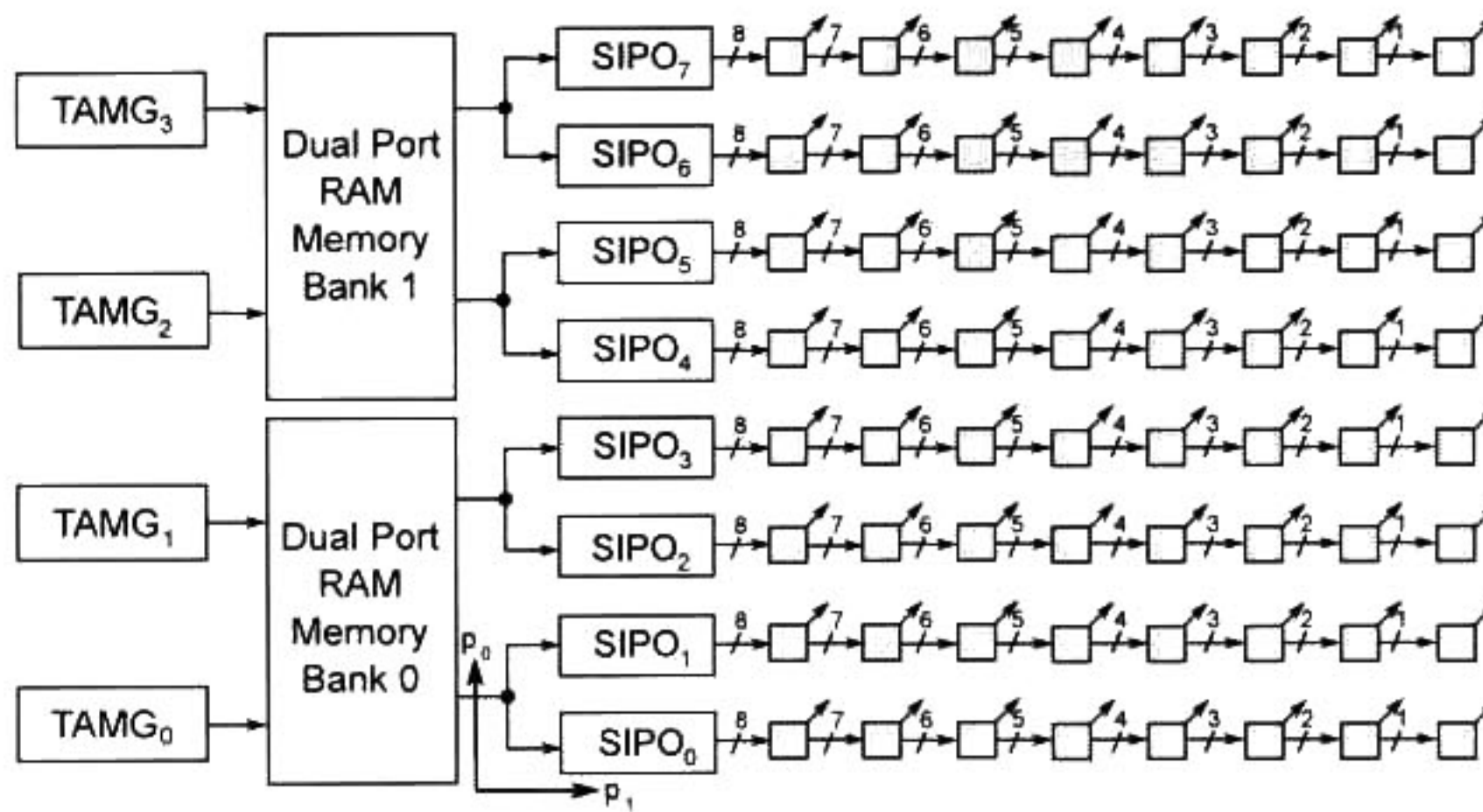


Figure 5.27: Architectural broadcast case for input variable B_{in} .

Variable C_{out} falls into the output variable border mapping. The AGUs are placed at p_1 and replicated through the index p_0 . The formula that is mapped to the AGU is shown in expression 5.6. Figure 5.28 shows the memory architecture and the data array for variable C_{out} .

$$MemAddrC_i = [N \times (t - p_0 - p_1 - tilep_0)] + (p_0 - tilep_0) - [f_{col} \times N (tilep_0 + MemBank_i)] \quad (5.6)$$

The integration of these three memory architectural cases is shown in figure 5.29. Note that the SIPOs on the left side correspond to the broadcast architectural case for input variable B_{in} , the SIPOs from the bottom corresponds to the input variable A_{in} and the PISOs of the right side are from the output variable C_{out} .

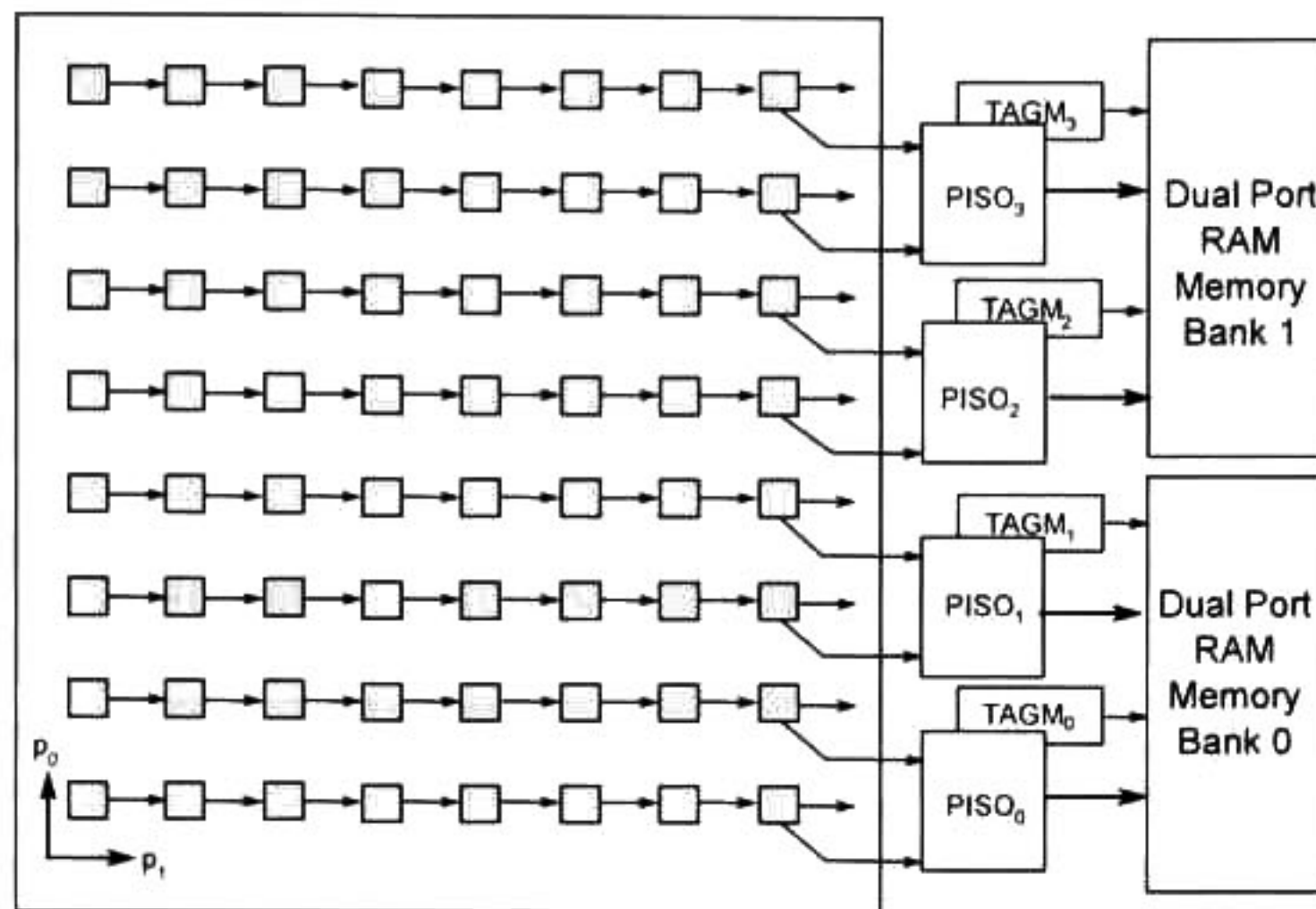


Figure 5.28: Architectural border case for output variable C_{out} .

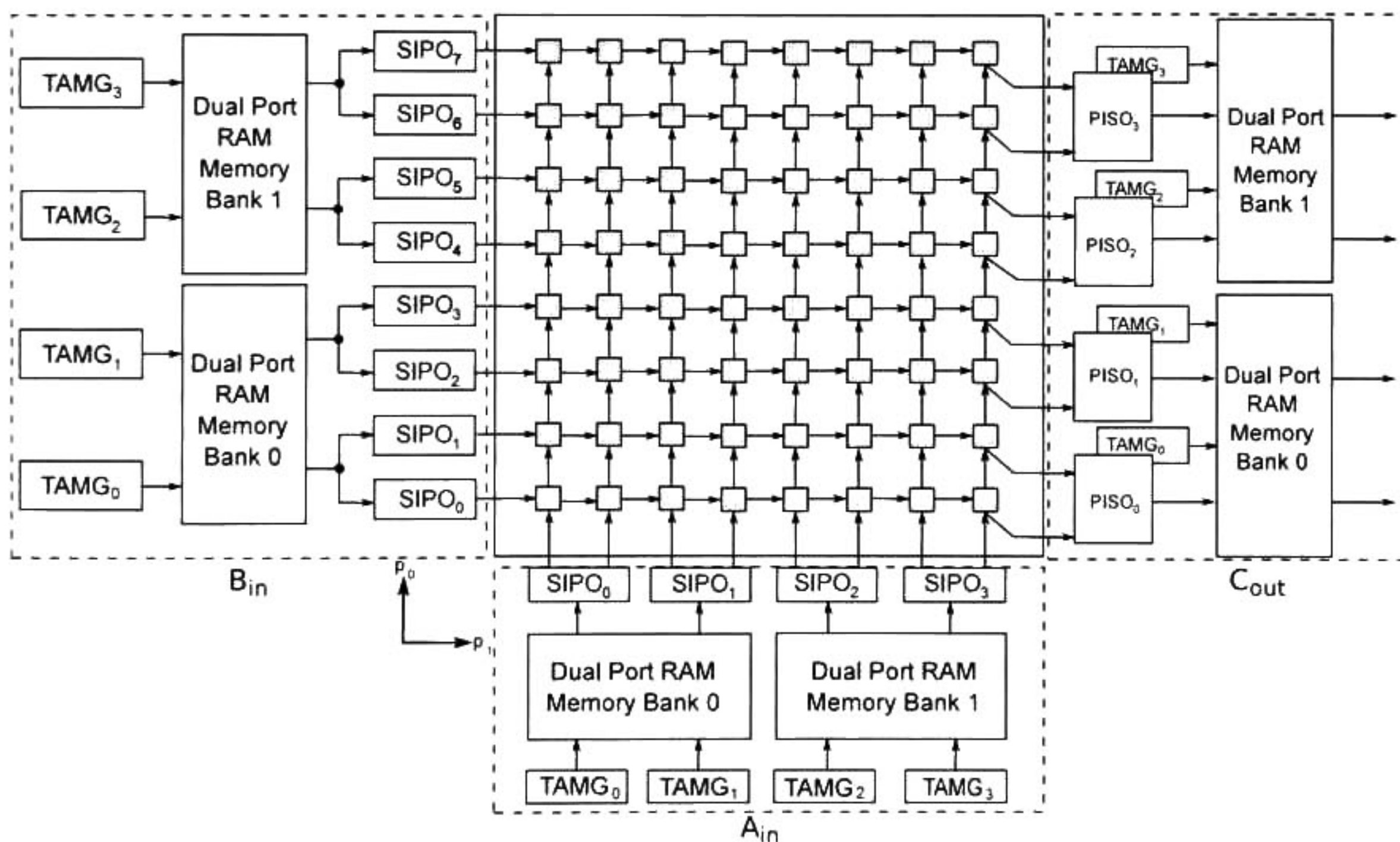


Figure 5.29: Memory system for MatMul algorithm combining the input/output variable broadcast and border cases.

Finally, table 5.2 shows the number of hardware elements required by three memory architectural cases. Note that the constants c_0 , c_1 , and c_2 are set to zero values since the mapping of the scheduler function ($i = t - p_0 - p_1$) does not require multiplications. Also, note that the number of registers required by the Variable B_{in} is major than the other variables since it falls into the broadcasting mapping case.

Number of	Variable A_{in}	Variable B_{in}	Variable C_{out}
Adders	48	48	48
Multipliers	16	16	16
Multiplexers	4		8
Counters		16	
Registers	16	352	64
1-Bit FFs-T	4		4
Memories	2	2	2

Table 5.2: Characterization of the hardware resource utilization of the MatMul case of study.

5.6 Summary

Although there are several works concerning the inner levels of the memory hierarchy, the problem of feeding and extracting data from an external memory source has not received much attention. There are few works concerning such problem, but limited to theoretical issues for synthesis and rarely for practical implementations. In this chapter it has been introduced an approach for four possible architectural cases which could occurs during the space-time transformation. For each case, a hardware architecture has been developed as possible solution to deal with these cases. By using a case of study it has been show how to integrate different architectural cases in a same design.

The proposed architectural memories cases are based on the use of a multi-clock domain approach and the use of dual-port memory banks. Besides it assumes that the external data are required and produced on each clock cycle respecting the scheduler function. If the assumption is relaxed, then

the use of memory banks or different clock domains is not longer needed. Mainly, this assumption could be relaxed if the operations performed by the processor array have high latency times. One advantage of the architectural framework is its scalability, due to only by replicating or eliminating the modules it is possible to fit the design to different processor array sizes. Moreover, by eliminating the use of multi-clock domains and/or replacing the dual-port memory by single ones, it is possible to fit the memory system to smaller or slower processor arrays.

6

Results

This chapter presents the results obtained from the implementation of the control scheme, the memory system and the integration of the processor array data-path with the controller and the memory using the MatMul and Cholesky decomposition algorithm as cases of study. The first section of this chapter, briefly describes the FPGA technology used for validation of the control, memory and integrated system. Later, the implementation results of the control scheme, presented in Chapter 4, and a brief comparison of these results against PARO and MMAAlpha are presented in the second section. The third section is advocated to present implementation results of the memory architectural cases developed in Chapter 5. The fourth section presents the implementation results of the integration of data-path, control and memory using several configurations for two cases of study. Also, section fourth presents the results using an embedded platform, and a comparison against a soft-processor implementation. Finally, the explanation of three different metrics and its application for evaluating the performance of several processor arrays are presented in the fifth and sixth sections, respectively.

6.1 Software Tools and Technological Platform

Before presenting the implementation results of the control, memory and the integration of the processor array data-path, it is important to mention the software tools developed for obtaining the scheduler vector and the target polytope, as well as the validation platform used for implementing the processor arrays. In this section, these topics are briefly described.

6.1.1 Software Tools

Deriving the scheduler function, applying the space-time mapping and obtaining the partitioned version of the PRA are required for deriving the processor array data-path, its control scheme and its external memory system. In order to obtain the scheduler and the target polytope some software tools were implemented. In the case of scheduler function, a program written in C was developed. This program takes as input the polytope specification (or iteration space) in form of the matrix A , vector \vec{b} (see definition 3.6 in Chapter 3), a dependence matrix D or a reduced dependence graph (RDG), and an iteration interval (P). If the input specification uses D , then a GLPK linear program formulation for calculating a linear scheduler will be obtained by the C program; but if the input specification uses a RDG instead of D , then the C program will produce a GLPK linear program formulation for obtaining an affine scheduler function. The linear programs are obtained following the MIP formulations presented in [64], and they are solved by using the GLPK solver [1].

Once the scheduler function has been computed, the user can propose an allocation function (Φ) from a projection vector (\vec{u}). With the scheduler vector and allocation matrix, the transformation matrix T can be constructed (see definition 3.15 in Chapter 3). The space-time transformation is performed by a program written in MATLAB, which performs the Fourier-Motzkin elimination described in [19]. As input this program requires the matrix A and vector \vec{b} and the index vector \vec{l} of the source polytope, and the transformation matrix T . As output, it produces the bounds of new index space of the target polytope in form of a set of inequality system.

After the target polytope has been obtained, strip mining is applied over the processor space indexes in order to obtain a partitioned version of the PRA. In this sense, the processor index to be partitioned needs to be the outer most index of the target polytope index vector \vec{J} , *i.e.* an index interchange must be performed. This interchange is accomplished by using the loop permutation transformation [6]. The loop permutation is done by using the same MATLAB program for space-time transformation, but changing the space-time transformation matrix, for the correct loop permutation matrix. With the partitioned version of the PRA, the control, external memory system and the processor array data-path could be derived. In the next section, the target implementation platform used for implementing the processor arrays is described.

6.1.2 FPGA Architecture Overview

The control scheme, the memory system and the processor arrays were implemented and validated using FPGA technology. FPGAs are electronic devices that blend the benefits of both hardware and software, since they implement circuits just like hardware, but FPGAs can be reprogrammed to implement a wide range of tasks [66], such as cryptographic systems, digital signal processing applications, software defined radio, low power embedded systems. Besides, FPGAs implement computations spatially, computing multiple operations in resources distributed across a silicon chip in a parallel way. Also, FPGAs are suitable of being used as an intermediate solution between software and ASICs, or for a rapid prototyping platform of digital designs. In fact, the FPGA regular internal interconnection architecture assembles the regular connections present in the processor array interconnection topology. Moreover, newer FPGAs allow the implementation of complex arithmetical operations like division, square root and trigonometric functions, as well as they include several small distributed multi-port memories.

Conceptually, FPGAs consist of three main parts: a set of configurable logic blocks (CLBs), a programmable interconnection network and a set of I/O cells. A function to be implemented in an FPGA is partitioned into modules, each of which can be implemented in different CLBs, and

then connected together using the programmable interconnection [21]. The internal architecture of an FPGA varies according to the technology used for its physical implementation, and from the manufacturer. Technologically, the FPGAs could be fabricated in antifuse technology and/or in memory-based technology. Usually manufacturers utilize look-up tables (LUTs) for implementing functions into FPGAs. A K -input LUT (or K -LUT) is a memory-based module that realizes an arbitrary K -variable function containing all the possible results of the arbitrary function for a given set of 2^K input values. The values of the function are stored in such a way that they can be retrieved by the corresponding input value. Therefore, a K -input LUT must provide a 2^K cells for storing the K -variable function. A K -LUT can implement up to 2^{2^K} different functions. In the current FPGA technology, a K -LUT consists of a set of SRAM-cells for storing values, and a decoder that is used to access the correct SRAM location and retrieve the result of the function, which corresponds to the input combination [66], [98]. Several LUTs are usually grouped in large modules along with other functional elements such as flip-flops and multiplexers. The connection between LUTs and other elements inside of such modules is done by using dedicated wires.

Newer FPGA devices include small dual-port memories [120], called Block RAMs (BRAMs), and digital signal processing units (DSPs) in order to provide dedicated units for storing data and performing MAC operations, respectively. In addition, some FPGA manufacturers like Xilinx include embedded hard-core processors as the IBM PowerPC within their FPGAs [119], or as in the case of Altera company, the ARM Cortex-A9 processor [10], [11]. As an alternative of such hard-core processor, the manufacturers have developed processors totally implemented into the FPGAs, taking advantage of the knowledge about their FPGAs architecture. Examples of these soft-processors are the Xilinx MicroBlaze [117] and the Altera Nios processor [12].

The results presented in this research work, are targeted for several Xilinx FPGA devices, focusing on Virtex-6 and Spartan-6 families. These FPGAs families use LUTs which can be configured as one 6-input LUT with one output, or as two 5-input LUTs with separate outputs but common logic inputs. Also, each LUT output can be registered using a flip-flop. In total, four LUTs with

their respective eight flip-flops form a slice along with some multiplexers and carry logic. In the case of Virtex-6 family, each FPGA has dual-port BRAM for storing 36 Kbits, and each BRAM has two completely independent ports; consequently a BRAM can be divided into two independent 18 Kbits BRAMs. Also, each FPGA of the Virtex-6 family, has dedicated multipliers and accumulators modules called DSP48E1, which consist of a dedicated 25×18 bit two's complement multiplier and a 48-bit accumulator. The multiplier can perform logic functions and barrel shifting operations [120]. On the other hand, the Spartan-6 family also has BRAM, and DSPs modules, but they are smaller than their corresponding Virtex-6 modules. In the case of Spartan-6 family, BRAM store 18 Kbits and they can be divided into two 9 Kbits BRAMs, whereas each dedicated multiplier and accumulator module (DSP48A1) consists of a dedicated 18×18 bit two's complement multiplier and a 48-bit accumulator [118].

6.1.3 Motivation of Using the FPGAs

Although one of the main advantage of using FPGAs as processor array implementation platform is the rapid prototyping and validation of complex digital system designs, there are other reasons for which the implementation of processor arrays in FPGA technology might be beneficial. One of these reasons is that the loop-based algorithms require operations which can be computed by using dedicated DSP blocks. One example is the MAC operation which is used in algorithms like MatMul, FIR filter, and LU and Cholesky decomposition algorithms. The use of these DSP blocks decrements the number of LUTs configured for performing arithmetical functions and improves the speed of digital signal applications. Also, with the use of the intellectual property cores designed by the FPGA's manufactures, complex hardware operations like divisions [116], square-roots or trigonometric functions [115] can be realized using the dedicated DSP blocks and BRAM.

Another advantage of using FPGAs for implementing the processor arrays is that FPGAs have several small memories (BRAMs) which can implement the functionality of dual-port or FIFO memories. In the case of the dual-port memories, they are required for storing the input and output

matrixes in the external memory system, and for providing multiple communication channels. On the other hand, in the case of FIFO memories, they are required for storing intermediate data produced by the processor array, avoiding to perform several external memory accesses. Furthermore, the FPGA has regular internal interconnections among its computational elements (LUTs, DSPs and BRAM) which assembles the regular PEs interconnections of the processor array. In the following sections, the implementation results using the FPGAs as prototyping platform are presented in terms of the number of slice LUTs, slice registers, BRAMs and DSPs.

6.2 Control Place and Route Results

Before introducing the control implementation results, it is important to mention a restriction brought by the control word width used for the time and tile indexes. Although the control scheme described in Chapter 4 provides theoretical independency of the problem size, there exists a restriction which imposes a limit to the problem size that can be solved. Such restriction is related to the sequence generator internal control arithmetic in charge of calculating the tile and time indexes while the partitions are being scanned. As a result of arithmetic operations performed by the Max/Min sub-modules, arithmetic overflows might occur. Despite of the sequence generator internal arithmetic deals with such situation, there is a maximum value for which the sequence generator is able to produce the indexes without any overflow risk. Between the tile and time indexes, the time index t is the one which has the greatest value, and therefore, the control bus width W_c must be set to support it. The maximum value of index t is the total execution time that a full-size implementation requires to be completed, that is:

$$t_{max} = \max(\lfloor \vec{\lambda} \vec{I} \rfloor, \vec{I} \in \mathcal{I}) - \min(\lfloor \vec{\lambda} \vec{J} \rfloor, \vec{J} \in \mathcal{I}) + 1 \quad (6.1)$$

where \vec{I} and \vec{J} are two index points of the iteration space \mathcal{I} , and $\vec{\lambda}$ is the scheduler vector. The control bus width is equal to $W_c = \log_2(t_{max})$. Since the iteration space \mathcal{I} depends on the problem size, writing mathematical expressions as function of the control word width W_c is possible.

For example, in the cases of the MatMul and Cholesky algorithms when the scheduler function $\vec{\lambda}_t = [1, 1, 1]$ is used, the maximum value of the time index is $t_{max} = 3(N - 1) + 1$. Note that t_{max} is the upper bound of the time index after space-time transformation. The largest problem size N_{max} that can be solved with a control word width of W_c bits for MatMul and Cholesky algorithms is:

$$N_{max} = \frac{2^{W_c} - 1}{3} + 1 \quad (6.2)$$

Equation 6.2 establishes a connection between the largest problem size and the control word width required in the control-path for solving problem instance of size N_{max} . For instance, with a 12-bit control word the sequence generator is able to generate the tile and time indexes for matrixes no larger than $1,365 \times 1,365$, *i.e.* the processor array could be able to deal with problems of size equal 1,863,225 data. Also, note that by adding an extra bit to the control bus, the matrix size that the control scheme is able to handle is increased twice *i.e.* the amount of data is quadrupled. Formulas 6.1 and 6.2 help to determinate the largest problem size for which the sequence generator is able to produce the tile and time indexes given a scheduler function when the hybrid control scheme is used. It is important to emphasize that the hybrid control scheme is independent of the problem size in the sense that given a control word width W_c , it would be able to provide an activation pattern for the PEs in a processor array of size $StripSizep_0 \times StripSizep_1$ for size of problems less than N_{max} . In this sense, the next subsection presents the hybrid control implementation results for a set of configurations, changing the control array size and their control word width W_c .

6.2.1 Implementation Results

For purpose of validation, the hybrid control scheme was modeled using VHDL Hardware Description Language, placed and routed with Xilinx ISE 13.1, and targeted for a Virtex-6 XC6VCX240T FPGA device. The PAR was done disabling the keep hierarchy option, and the optimization goal was set to speed option. Besides, with the purpose of showing how different algorithms, space-time mappings and iteration intervals affect the control scheme (in terms of FPGA resources and maximum operational frequency), three different implementation sets, named *CtrlImpl 1*, *CtrlImpl 2* and *CtrlImpl*

3, with different control array sizes are shown in this subsection. Each implementation set consists of a controller developed for a different algorithm, using different space-time mappings and different iteration intervals as shown in table 6.1. Furthermore, each set has different configurations with different control word widths and different partitioning parameters, *i.e.* changing W_c , SSp_0 and SSp_1 parameters.

Implementation Code Name	Implemented Algorithm	Scheduler Function	Projection Function	Iteration Interval
<i>CtrlImpl 1</i>	MatMul	$\vec{\lambda}_l = [1, 1, 1]$	$\vec{u} = [1, 0, 0]^t$	$P = 1$
<i>CtrlImpl 2</i>	Cholesky	$\vec{\lambda}_l = [1, 2, 3]$	$\vec{u} = [1, 0, 0]^t$	$P = 1$
<i>CtrlImpl 3</i>	Cholesky	$\vec{\lambda}_l = [1, 1, 1]$	$\vec{u} = [0, 0, 1]^t$	$P = 21$

Table 6.1: Description of the three control implementation sets with their respective design parameters.

In the first set, implementation *CtrlImpl 1* is the MatMul controller derived by using the design parameters shown in table 6.1. PAR results summarized in terms of slice registers, slice LUTs, occupied slices, and maximum operational frequency are shown in table 6.2. Note that for this set, FPGA resources grow by a factor equal to or less than the control array size when the same W_c is considered. However, the FPGA resources scale at different rate when the W_c is changed and the same control array size is considered. Mainly, this is due to the logic required for calculating the mathematical expressions placed in the Max/Min sub-modules. Besides, it should be noted that the operational frequencies of the 8×8 control array are the fastest frequencies in contrast to the 4×4 and 2×2 control arrays. At first glance, it might seem that for small arrays the PAR algorithms, used by the Xilinx ISE 13.1 for the Virtex-6 FPGA family, do not perform well compared against larger arrays. However, a closer look at the routed and placed controllers, using the Xilinx PlanAhead 13.1 software tool, reveals that when the PAR algorithms place the distributed and centralized control modules close to each other, but in a separated way (*i.e.* both modules are not mixed), faster operational frequencies are achieved. For instance, in the case of control array with the slowest frequency (2×2 control cells using a $W_c = 12$), the PAR algorithm divides the control array into

Array Size	Word Size (Bits)	Slice Registers	Slice LUTs	Occupied Slices	Speed (MHz)
2 × 2	8	220	344	137	97.80
2 × 2	9	256	394	176	96.46
2 × 2	10	281	482	197	86.13
2 × 2	11	311	526	200	88.99
2 × 2	12	457	740	273	81.73
4 × 4	8	471	745	293	101.45
4 × 4	9	539	853	358	100.75
4 × 4	10	607	1,146	409	92.55
4 × 4	11	670	1,240	432	90.76
4 × 4	12	1,343	2,047	772	89.49
8 × 8	8	1,273	2,319	945	106.06
8 × 8	9	1,456	2,508	963	101.81
8 × 8	10	1,649	3,730	1,316	91.10
8 × 8	11	1,831	4,128	1,345	92.61
8 × 8	12	4,753	8,036	3,012	93.94

Table 6.2: Implementation *Ctrlmpl 1* PAR results targeted for a XC6VCX240T FPGA device. The number of slice registers, slice LUTs and slices available for this device is 301,440, 150,720 and 75,360, respectively.

two parts and places the centralized modules between the two parts of the divided control array. Contrary, in the case of the control array with the highest frequency (8×8 control cells using a $W_c = 8$), the centralized modules are separated from the control array. Mainly, this could be a consequence of PAR algorithms use without any restriction the FPGA I/O ports.

The second control scheme implementation, called *Ctrlmpl 2*, was developed for the Cholesky decomposition algorithm. In this case, the algorithm and the scheduler function are altered with respect of the *Ctrlmpl 1* implementation as shown in table 6.1. Similarly to the MatMul control results, table 6.3 summarizes the PAR results in terms of the number of slice registers, slice LUTs, and maximum operational frequency. In this implementation the scheduler function leads to complex mathematical expressions for obtaining the lower and upper bounds of the tile and time indexes (as shown expression 4.2 in Chapter 4). Since these expressions are mapped to combinational logic, the Xilinx synthesis tool use the FPGA LUTs resources for mapping these expressions, and consequently

Array Size	Word Size (Bits)	Slice Registers	Slice LUTs	Occupied Slices	Speed (MHz)
2 × 2	8	207	572	221	62.53
2 × 2	9	245	630	271	62.34
2 × 2	10	284	794	307	58.80
2 × 2	11	309	852	314	58.81
2 × 2	12	454	1,103	444	48.76
2 × 2	13	456	1,191	479	47.45
4 × 4	8	422	956	353	66.02
4 × 4	9	497	1,070	423	62.31
4 × 4	10	567	1,411	542	59.17
4 × 4	11	620	1,538	561	59.47
4 × 4	12	1,459	2,552	1,017	53.76
4 × 4	13	1,492	2,766	1,149	50.63
8 × 8	8	1,165	2,496	852	75.55
8 × 8	9	1,350	2,665	998	67.11
8 × 8	10	1,530	3,995	1,374	71.93
8 × 8	11	1,699	4,389	1,436	58.42
8 × 8	12	5,341	9,379	3,407	56.90
8 × 8	13	5,807	10,439	4,220	56.58

Table 6.3: Implementation *Ctrlmpl 2* PAR results targeted for a XC6VCX240T FPGA device. The number of slice registers, slice LUTs and slices available for this device is 301,440, 150,720 and 75,360, respectively.

the usage of slice LUTs is increased with respect of the slice LUTs required in implementation *Ctrlmpl 1*. Moreover, note that the maximum operational frequencies of configurations *Ctrlmpl 1* are faster than the frequencies of configurations *Ctrlmpl 2*. Mainly, this is due to the loop bounds expressions required in implementation *Ctrlmpl 2* are totally mapped to combinational logic and thereby, such logic generate the longest critical path inside the sequence generator. Furthermore, note that table 6.3 includes control words of 13-bit, since in this case of *Ctrlmpl 2*:

$$N_{max} = \frac{2^{W_c} - 1}{6} + 1 \quad (6.3)$$

due to the restriction imposed by expression 6.2. In fact, note that by using W_c bits the implementation *Ctrlmpl 1* is able to solve problem sizes of N_{max} , but *Ctrlmpl 2* implementation is only able to solve problem sizes of $N_{max}/2$. Additionally, the operational frequencies of configurations

Array Size	Word Size (Bits)	Slice Registers	Slice LUTs	Occupied Slices	Speed (MHz)
2 × 2	8	336	477	195	101.78
2 × 2	9	379	561	238	93.11
2 × 2	10	408	605	271	88.44
2 × 2	11	440	660	267	97.95
2 × 2	12	473	819	392	87.95
4 × 4	8	882	1,174	453	108.66
4 × 4	9	1,006	1,357	602	88.30
4 × 4	10	1,176	1,463	586	86.95
4 × 4	11	1,281	1,623	611	83.84
4 × 4	12	1,391	2,038	940	72.77
8 × 8	8	3,273	4,207	1,718	103.90
8 × 8	9	3,496	4,913	2,006	92.45
8 × 8	10	4,224	5,188	2,069	97.96
8 × 8	11	4,639	5,770	2,210	90.00
8 × 8	12	5,093	7,404	3,163	91.21

Table 6.4: Implementation *Ctrlmpl 3* PAR results targeted for a XC6VCX240T FPGA device. The number of slice registers, slice LUTs and slices available for this device is 301,440, 150,720 and 75,360, respectively.

Ctrlmpl 2 for the 8×8 control array are the fastest frequencies in contrast to the 4×4 and 2×2 arrays. Here again a look into the processor arrays placed and routed into the FPGA reveals the same PAR pattern shown in the case of implementations *Ctrlmpl 1*. However, it is worth to mention that in the case of 2×2 control arrays the PlanAhead software shows that the centralized part occupies more FPGA resources than control array. Mainly, this is a consequence of the complex mathematical expressions derived from the space-time mapping, which are mapped to the FPGA LUTs.

Finally, the *Ctrlmpl 3* control implementation was also developed for the Cholesky decomposition algorithm. However, in this case different space-time mapping and iteration interval were used (see table 6.1). Similarly to the previous control implementations, table 6.4 summarizes the PAR results in terms of the number of slice registers, slice LUTs, and maximum operational frequency. In this case, the space-time mapping leads to loop bounds expressions with a minor complexity than in implementation *Ctrlmpl 2*. This is reflected in number of slice LUTs required by implementation

Ctrlmpl 3, which is less than implementation *Ctrlmpl 2*. Also, it should be noted that since implementation *Ctrlmpl 3* uses a different iteration interval than implementations *Ctrlmpl 1* and *Ctrlmpl 2*, the number of slice registers required by implementation *Ctrlmpl 3* is greater than in *Ctrlmpl 1* and *Ctrlmpl 2* implementations. Mainly, this is due the use of counters within each control cell for generating the iteration interval. In fact, the FPGA slice registers grow by a factor almost equal to the control array size for the same W_c . Again, when the PAR algorithms place the distributed and centralized modules close to each other, faster operational frequencies are achieved. Examples of this behavior are obtained for the control arrays of size 2×2 , 4×4 and 8×8 when $W_c = 8$. However, if the centralized module is enclosed by the distributed modules (control cells), then slower frequencies should be expected; as in the case of the 4×4 control array with a $W_c = 8$ where the lowest frequency of the *Ctrlmpl 3* control implementation is achieved.

Summarizing, these implementations with their respective configurations have shown that the control scheme is able to support different space-time transformations with different iteration intervals, and for non-rectangular iteration spaces. Also, the maximum operational frequencies not always decrease when the control arrays are increased. In fact the Xilinx PAR algorithms place the control cells closer among other cells when larger control arrays are synthesized than in the case of smaller control arrays; and in some cases these PAR algorithms place the centralized modules inside the area occupied by the control array resulting in a degradation of the operational frequency. Furthermore, it is worth to mention than in the case of implementations *Ctrlmpl 1* and *Ctrlmpl 2* the faster and slower arrays are the 8×8 with $W_c = 8$ and 2×2 with $W_c = 12$, respectively; and that in the case of *Ctrlmpl 3* the same implementations of 8×8 with $W_c = 8$ and 2×2 with $W_c = 12$ are the second faster and third slower control implementations. In addition, the same implementations were targeted for a Virtex-4 XC4LX80 FPGA device with 4-LUTs in order to observe the behavior of the operational frequencies. In this case of the XC4LX80 device, the PAR algorithms lead to higher operational frequencies in the case of the 4×4 control arrays and the slower frequencies for the 8×8 . This observation might suggest that 4-LUT PAR algorithms behaves better for control arrays of 4×4 than other arrays; while the 6-LUT PAR algorithms behaves better for larger control

arrays from the operational frequency's point of view. However, such hypothesis requires a detailed study and other experimental results targeted for different FPGAs (with different LUT technology) in order to confirm it. Respecting to the FPGA resources, the use of iteration interval greater than one ($P > 1$), leads to increment the number of counters inside of the control cells. This increment is reflected in the number of registers in implementation *CtrlImpl 3*, which exhibits the major quantity of slice registers in almost all the configurations. Finally, complex mathematical expressions obtained after space-time mapping lead to increase the quantity of slice LUTs in the control array, and to degrade the maximum operational frequency. This degradation is due to complex expressions are directly mapped to combinational logic without pipelining their computations.

6.2.2 Comparison

For purpose of comparison against other works, the centralized and distributed components of the control scheme were synthesized with Xilinx ISE 6.1 and targeted for a Virtex XCV800 FPGA device. The number of slices after synthesis for the control generated by MMAAlpha, PARO and the proposed control scheme are shown in table 6.5 using the MatMul algorithm as a case of study. Synthesis results are divided into centralized and distributed parts. Data for PARO and MMAAlpha are obtained from [45] and [61], respectively. In these two works the number of slices that their distributed control units require for a single PE is reported.

Since the control generated by PARO is divided into local and global controller, the number of slices for global controller is also considered [45]. In the case of the proposed control scheme, three different implementations targeted for Virtex XCV800 and Virtex-4 XC4LX200 are considered in table 6.5 with a 12-bit control word width. The Virtex XCV800 implementation is used for comparison purpose against PARO and MMAAlpha controllers, whereas the Virtex-4 XC4LX200 implementations are used for providing synthesis results for a more up to date FPGA device with the same 4-LUT technology as the XCV800 device. Besides, in the case of the Virtex-4, one implementation does not force the synthesizer to use the embedded DSP blocks while the other one uses the DSP blocks.

Despite that a control cell targeted for the same device uses more slices than the controllers derived by MMAAlpha and PARO, the control cell is able to deal with non-rectangular processor spaces as well as it deals with subsets of problem instances no larger than N_{max} , contrary to MMAAlpha and PARO controllers. Moreover, methods for deriving the processor array differ in the scheduling and allocation technique used: multidimensional scheduling for MMAAlpha, and the combination of affine scheduling and tiling matrixes for PARO. In contrast, the proposed control scheme, derived by the design methodology followed in this dissertation, uses linear scheduling combined with strip mining. The combination of both techniques allows to deal with non-rectangular spaces and it allows solving a subset of problem instances instead of solving several instances of a unique problem size. In addition, global controller generated by PARO requires 209 slices while the equivalent global controller of the proposed hybrid control scheme (centralized modules) requires 179 slices. Moreover, if embedded FPGA DSP blocks are used, the number of slices for a Virtex-4 is reduced in a factor of 3.

Control Type	FPGA Resources	MMAAlpha [61] Virtex xcv800	PARO [45] Virtex xcv800	Proposed Control Virtex xcv800	Proposed Control Virtex 4 xc4lx200	Proposed Control Virtex 4 xc4lx200
Distributed	Slices	65	12	86	58	22
	DSP blocks					6
Centralized	Slices		209	179	155	50
	DSP blocks					18

Table 6.5: Slices comparison among several control architectures for a single PE (distributed) and the global controller (centralized).

6.3 External Memory Place and Route Results

This section presents the PAR results of implementing the four memory architectural cases using the MatMul and Cholesky decomposition algorithms. By themselves, the architectural memory cases do not impose a restriction to the processor array implementation as the control scheme does with the W_p parameter; but they follow a restriction imposed by the problem size.

6.3.1 Implementation Results

The memory architectural cases were modeled and validated using VHDL Hardware Description Language. Also, they were synthesized with Xilinx ISE 13.1, and targeted for a Virtex-6 XC6VCX240T FPGA device. The algorithms used for the architectural cases validation were the MatMul and Cholesky decomposition. For both algorithms the partitioning parameters are $SSp_0 = 8$ and $SSp_1 = 8$ leading to a partitioned processor array of 8×8 PEs. Note that these parameters are the same shown as cases of study in Chapters 4 and 5.

The PAR results of the MatMul architectural cases for different problem sizes are shown in table 6.6. These architectural cases were derived by using the scheduler vector $\vec{\lambda}_l = [1, 1, 1]$, the projection vector $\vec{u} = [1, 0, 0]^t$, and the iteration interval $P = 1$. The PAR results show the amount of FPGA resource utilization for a pair of input border, input broadcast, and output border modules (six modules total) using a 32-bit word size, and different control word widths. Recall that according to formula 6.2, the maximum problem size that a processor array is able to solve depends on the number of bits in the control word. In this sense the problem sizes of 86×86 , 171×171 and 342×342 correspond to 8-bit, 9-bit, and 10-bit control words, respectively. Furthermore, table 6.6 shows the PAR results for the MatMul processor arrays of 8×8 PEs. These processor arrays include the data-path, the control scheme, and the broadcast data array required by the input broadcast case. The processor array FIFOs were implemented using built-in FIFOs as in [65]. Besides, each architectural case has a pair of memory banks implemented by the FPGA's BRAMs in form of dual-port memories, and the total size of each pair of memory bank corresponds to the the maximum problem size N_{max} . Although the results are presented for the FPGA BRAMs, it is important to emphasize that off-chip DRAM could be used instead of built-in BRAMs, in order to allow the use of such BRAMs as FIFO memories. The FPGA's DSP blocks are used for calculating the memories directions while the processor array uses them for both the control and the data-path. Note that the memory assumption of $Clk_{mem} \geq 2 \times Clk_{pa}$ is achievable since the processor array operational frequency is three times slower than the worst case for any of the memory cases. Also, table 6.6 shows the theoretical peak

Problem Size	FPGA Resource	Input Border	Input Broadcast	Output Border	Processor Array
86 × 86	Slices Registers	516	2,199	221	24,075
	Slices LUTs	301	1,002	437	20,916
	RAMB36E1	8	8	8	0
	RAMB18E1	0	0	0	0
	FIFO36E1	0	0	0	8
	DSP48E1	16	16	16	192
	Speed (MHz) Bandwidth (GB/s)	296.64 9.49	251.82 8.06	319.79 10.23	56.48 5.42
171 × 171	Slices Registers	532	2,215	265	24,582
	Slices LUTs	575	1,071	671	21,944
	RAMB36E1	24	24	24	0
	RAMB18E1	8	8	8	0
	FIFO36E1	0	0	0	32
	DSP48E1	16	16	16	192
	Speed (MHz) Bandwidth (GB/s)	205.42 6.57	165.72 5.30	286.94 9.18	57.23 5.49
342 × 342	Slices Registers	536	2,219	297	25,209
	Slices LUTs	1,249	1,725	1,402	25,050
	RAMB36E1	98	98	98	0
	RAMB18E1	16	16	16	0
	FIFO36E1	0	0	0	128
	DSP48E1	16	16	16	192
	Speed (MHz) Bandwidth (GB/s)	165.70 5.30	132.83 4.24	193.38 6.19	55.31 5.31

Table 6.6: Place and route results for three memory architectural cases and three processor array implementations for MatMul algorithm targeted for a XC6VCX240T FPGA device

bandwidth obtained by each architectural case, and the peak bandwidth required by the processor array. For the three problem sizes, the processor array I/O bandwidths requirements are exceeded by the total I/O bandwidth provided by the memory architectural cases due to the use of several communication channels (eight channels per architectural case). In the case of the implementation for problem size of 342×342 , the total I/O bandwidth provided is almost three times faster than the I/O bandwidth required by the processor array. Such total I/O bandwidth is obtained by adding each architectural case bandwidth as shown in table 6.6, and each one of these bandwidths is calculated by multiplying the word width times the amount of communication channels.

A direct comparison with other works like PARO and MMAAlpha is not possible, since authors do not show results for memory implementations. However, a recently effort for implementing a hand-made memory system for a MatMul processor array derived by MMAAlpha tool was developed by Plesco in [91]. In this case, the input bandwidth required by a processor array of 4×4 PEs is not satisfied by the Plesco's solution, since a CPU (PowerPC) is responsible for bringing the external data by using two communication channels. In this last case, the bandwidth required by the 4×4 array is 3.2 GB/s, while the input data transfer rate achieved is 32 MB/s due to latency in the memory controller.

The PAR results for Cholesky architectural cases for different problem sizes and using the scheduler vector $\vec{\lambda}_l = [1, 1, 1]$, projection vector $\vec{u} = [0, 0, 1]^t$, and iteration interval $P = 21$ are shown in table 6.7. Similarly to 6.6, these results show the FPGA resource utilization for a pair of input border, and output broadcast modules (four modules total) using a data word width of 32-bit, and different control word widths. Also, table 6.7 shows the PAR results for the Cholesky processor array of 8×8 PEs. The square root and division operations required by the Cholesky decomposition are implemented using the Xilinx IP cores. In the case of division, the IP core uses BRAMs (in form of RAMB18E1 modules) for each division module required in the array. As well as in the MatMul case, the processor array FIFOs were implemented by using the FPGA's BRAMs. It should be noted that in MatMul and Cholesky architectural cases the number of DSP blocks are the same, since the AGUs in both implementations contain similar mathematical expressions for calculating the memory addresses derived from the indexes after space-time mapping. Moreover, note that the output broadcast case requires a small number of slice registers, because the PISO broadcast modules needed in this architectural case consist of a unique 8-1 multiplexer per row/column of the array (eight 8-1 multiplexers in this implementation). Besides, the broadcast data array is not included as part of the PAR results in the broadcast architectural case, but in the processor array. As a consequence of the use of 8-1 multiplexers, the number of slice LUTs is increased in this architectural case. Similarly to the MatMul, note that the memory assumption of $Clk_{mem} \geq 2 \times Clk_{pa}$ is also achievable for the Cholesky processor array. Furthermore, table 6.7 shows the peak bandwidth obtained by the

Problem Size	FPGA Resource	Input Border	Output Broadcast	Processor Array
85 × 85	Slices Registers	531	68	23,885
	Slices LUTs	331	2,630	22,137
	RAMB36E1	8	8	0
	RAMB18E1	0	0	8
	FIFO36E1	0	0	8
	DSP48E1	16	16	296
	Speed (MHz) Bandwidth (GB/s)	293.08 9.38	328.08 10.50	65.29 4.18
171 × 171	Slices Registers	529	84	24,314
	Slices LUTs	608	2,906	24,080
	RAMB36E1	24	24	0
	RAMB18E1	8	8	8
	FIFO36E1	0	0	32
	DSP48E1	16	16	296
	Speed (MHz) Bandwidth (GB/s)	200.20 6.41	310.46 9.39	64.24 4.11
342 × 342	Slices Registers	589	88	25,142
	Slices LUTs	1,260	3,339	26,970
	RAMB36E1	98	98	0
	RAMB18E1	16	16	8
	FIFO36E1	0	0	128
	DSP48E1	16	16	296
	Speed (MHz) Bandwidth (GB/s)	157.97 5.06	186.43 5.97	59.85 3.83

Table 6.7: Place and route results for three memory architectural cases and a processor array implementation for Cholesky algorithm targeted for a XC6VCX240T FPGA device

two Cholesky architectural cases, and the peak bandwidth required by the processor array. In the last case, the peak bandwidth is calculated assuming that the Cholesky processor array requires new data on each clock cycle. However, due to the iteration interval is set $P = 21$ the I/O requirements of Cholesky processor array are fewer than the I/O bandwidth shown in table 6.7. Nevertheless the memory architectural cases overcome the Cholesky I/O data demands. In summary, the four memory architectural cases achieve the memory assumption of $Clk_{mem} \geq 2 \times Clk_{pa}$. Also, the PAR results show that the broadcast memory cases require major amount of FPGA resources than the border cases, as stated by expression presented in table 5.1.

6.4 Cases of Studies: Place and Route Results

In previous sections, FPGA implementation results for both control scheme and memory architectural cases were presented separately. In this section, the cases of study developed in Chapters 4 and 5 are presented as complete systems (data-path, control scheme and memory system). Figure 6.1 depicts the block diagram of this integrated system, showing in different colors the two clock domains. In this integrated system the tile and time indexes are generated by the sequence generator, later these indexes are decoded by the activation-signal injector and by the AGUs in order to generate the processor array activation sequence and the input memory addresses, respectively. Data extracted from the input memory is inserted inside the processor array by the SIPO elements and at the same time the activation signal is injected to the processor array. All intermediate data produced by the array are stored in the FIFO memories and reused without the need of accessing an external memory. Once results are being produced by the processor array they are recollected by PISO registers and they are stored in an output memory.

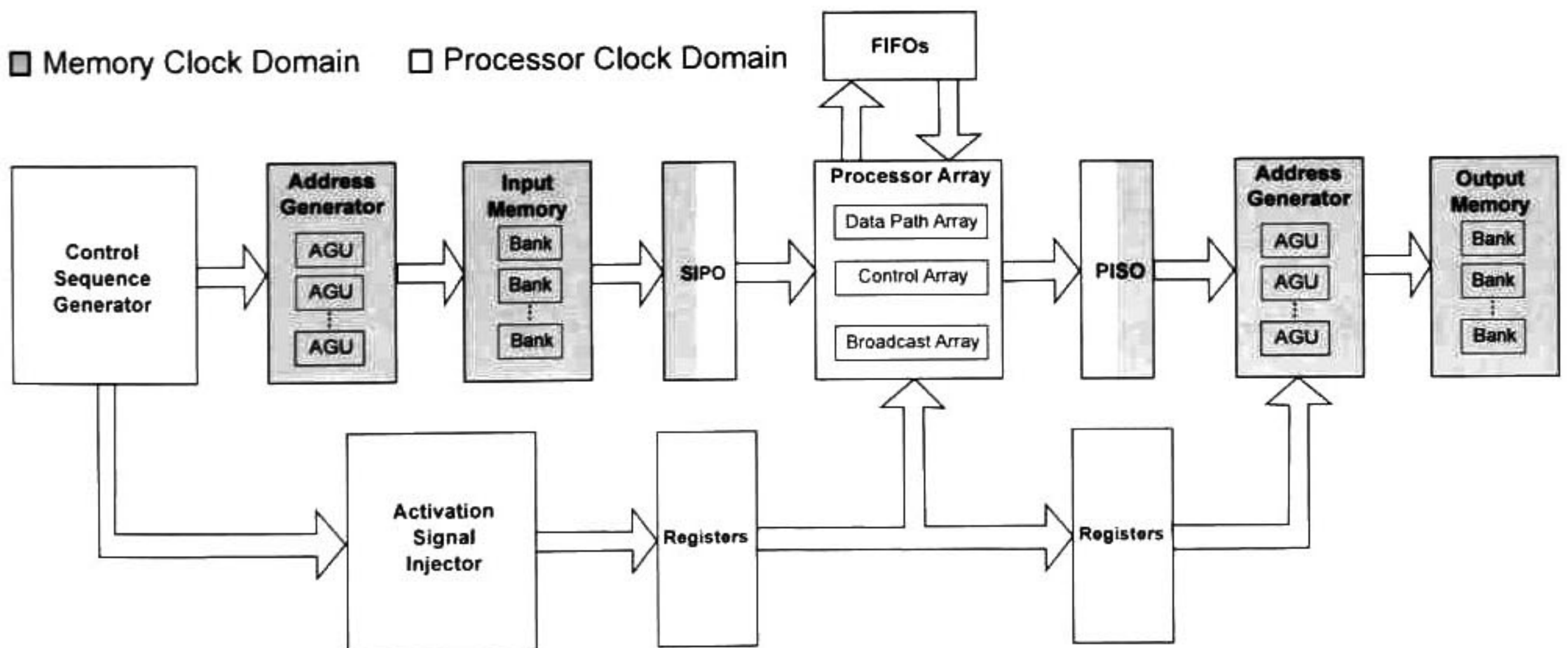


Figure 6.1: Block diagram for the integration of memory system, control and processor array data-path.

It is important to emphasize that the integrated system is able to support different scheduler functions by changing the mathematical expressions mapped to combinational logic in:

- The Max/Min sub-modules located in the sequence generator (control scheme).
- The Max sub-modules placed in the activation-signal injector (control scheme).
- The boundaries detector and the activation pattern generator sub-modules in the control control cells (control scheme).
- The AGU sub-modules required by each architectural case (external memory).

Besides, if the projection vector is also changed, by the correct selection of the memory architectural cases, the support of different space-time transformations given the matrix T is possible. Furthermore, similarly to PARO, the integrated system supports the use of different iteration intervals by using P -modulo counters in the control scheme. Recall that the combination of the scheduler function and the projection vector derive the transformation matrix T , *i.e.* the space-time mapping. In the next subsections, the PAR results for the integrated system implementing two cases of study (MatMul and Cholesky) are presented targeted for different Virtex-6 FPGA devices (subsections 6.4.1 and 6.4.2). Also, the PAR results for an embedded platform using a Spartan-6 FPGA are presented in subsection 6.4.3. All the PAR results were obtained using the Xilinx ISE 13.1 tool.

6.4.1 MatMul Results

For purpose of showing different aspects involved when processor arrays are implemented in FPGA technology, the PAR results are grouped into two different sets and targeted for different FPGA devices. The first set consists of the MatMul processor array including control, data-path and memory system fully implemented in an FPGA, with the purpose of showing that the processor array integration is possible with the current technology for a set of problem sizes. However, due to the limited FPGA's memory, since they are not designed for storing large amount of data, the second set of implementations consists of assuming the memory banks are placed outside of the FPGA, leaving

6.4 Cases of Studies: Place and Route Results

In previous sections, FPGA implementation results for both control scheme and memory architectural cases were presented separately. In this section, the cases of study developed in Chapters 4 and 5 are presented as complete systems (data-path, control scheme and memory system). Figure 6.1 depicts the block diagram of this integrated system, showing in different colors the two clock domains. In this integrated system the tile and time indexes are generated by the sequence generator, later these indexes are decoded by the activation-signal injector and by the AGUs in order to generate the processor array activation sequence and the input memory addresses, respectively. Data extracted from the input memory is inserted inside the processor array by the SIPO elements and at the same time the activation signal is injected to the processor array. All intermediate data produced by the array are stored in the FIFO memories and reused without the need of accessing an external memory. Once results are being produced by the processor array they are recollected by PISO registers and they are stored in an output memory.

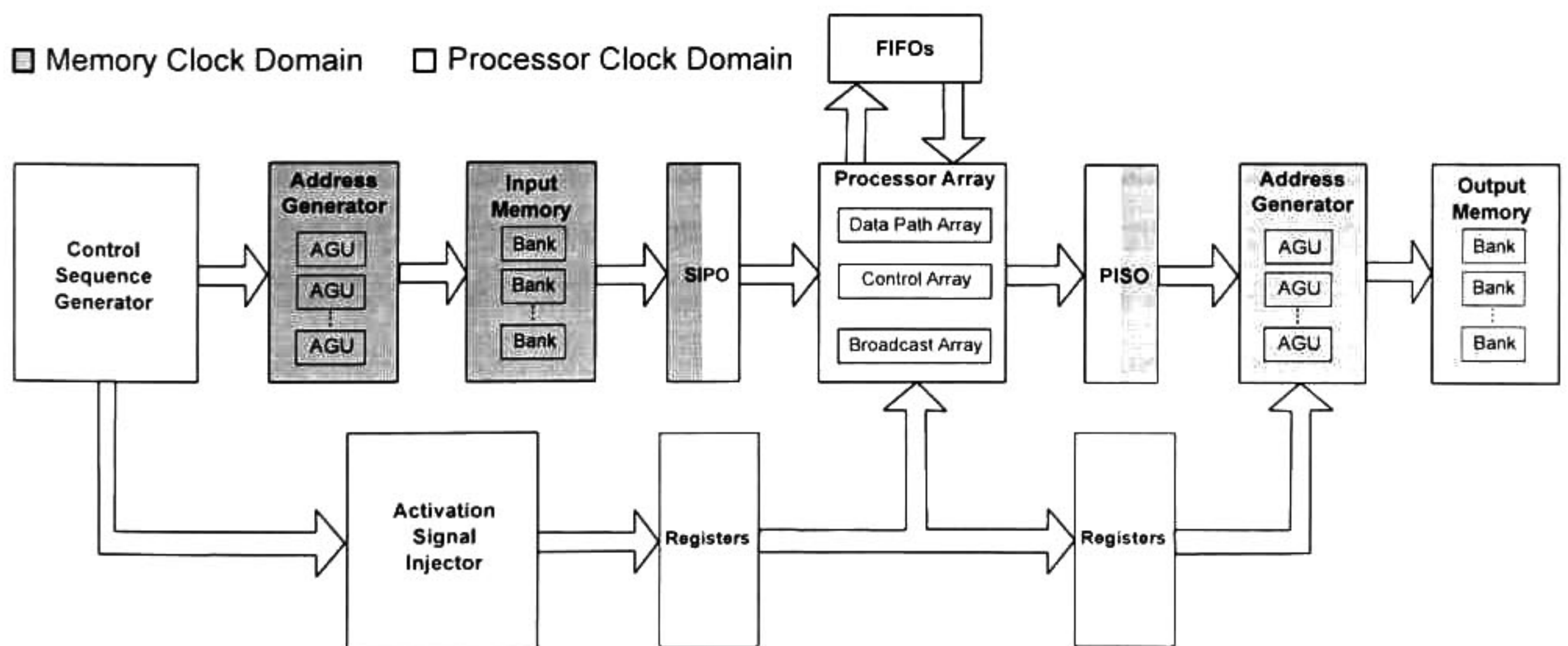


Figure 6.1: Block diagram for the integration of memory system, control and processor array data-path.

It is important to emphasize that the integrated system is able to support different scheduler functions by changing the mathematical expressions mapped to combinational logic in:

- The Max/Min sub-modules located in the sequence generator (control scheme).
- The Max sub-modules placed in the activation-signal injector (control scheme).
- The boundaries detector and the activation pattern generator sub-modules in the control control cells (control scheme).
- The AGU sub-modules required by each architectural case (external memory).

Besides, if the projection vector is also changed, by the correct selection of the memory architectural cases, the support of different space-time transformations given the matrix T is possible. Furthermore, similarly to PARO, the integrated system supports the use of different iteration intervals by using P -modulo counters in the control scheme. Recall that the combination of the scheduler function and the projection vector derive the transformation matrix T , *i.e.* the space-time mapping. In the next subsections, the PAR results for the integrated system implementing two cases of study (MatMul and Cholesky) are presented targeted for different Virtex-6 FPGA devices (subsections 6.4.1 and 6.4.2). Also, the PAR results for an embedded platform using a Spartan-6 FPGA are presented in subsection 6.4.3. All the PAR results were obtained using the Xilinx ISE 13.1 tool.

6.4.1 MatMul Results

For purpose of showing different aspects involved when processor arrays are implemented in FPGA technology, the PAR results are grouped into two different sets and targeted for different FPGA devices. The first set consists of the MatMul processor array including control, data-path and memory system fully implemented in an FPGA, with the purpose of showing that the processor array integration is possible with the current technology for a set of problem sizes. However, due to the limited FPGA's memory, since they are not designed for storing large amount of data, the second set of implementations consists of assuming the memory banks are placed outside of the FPGA, leaving

only the intermediate memory inside the FPGA and allowing to solve a larger set of problem sizes than in first the implementation set.

These two sets of implementations use the same configuration design parameters, *i.e.* the same scheduler, projection, and iteration interval. The values of these parameters are $\vec{\lambda}_l = [1, 1, 1]$, $\vec{u} = [1, 0, 0]^t$, and $P = 1$; while the partitioning parameters (processor array size) vary for each implementation. The data word width is 32-bit for all implementations. The control word varies for each implementation, mainly with the purpose of each implementation supports a different problem size. Finally, the FIFOs required for the processor arrays were implemented by using the Xilinx's IP-Core as shift registers and as built-in FIFOs for FIFOs L1 and FIFOs L2, respectively; whereas memory banks were implemented using the FPGA's built-in BRAMs.

6.4.1.1 Place and Route

Table 6.8 shows the description of the first implementations set with their corresponding code names. This set is formed by three different MatMul processor array implementations. These implementations use the partitioning parameters of $SSp_0 = 8$ and $SSp_1 = 8$, leading to processor arrays of 8×8 PEs. Note that these implementations differ in their control word width W_c , thus each implementation leads to different N_{max} values. Also, table 6.8 shows the type of memory banks, required by the architectural cases, were implemented using the FPGAs BRAMs, *i.e.* as on-chip memories. The targeted FPGA device used for implementing the first set was a Virtex-6 XC6VCX240T.

Implementation Code Name	Processor Array Size	Max. Prob. Size (N_{max})	Control Word (W_c)
<i>MatMul-On-8x8-086</i>	8×8 PEs	86×86	8-bit
<i>MatMul-On-8x8-171</i>	8×8 PEs	171×171	9-bit
<i>MatMul-On-8x8-320</i>	8×8 PEs	320×320	10-bit

Table 6.8: Description of the first MatMul implementation set with their respective parameters. These implementations use the FPGA on-chip memories for storing the input and output matrixes.

FPGA Resources		MatMul-On	MatMul-On	MatMul-On
Resource	Available	8x8-086	8x8-171	8x8-320
Slice Registers	301,440	20,201	20,478	20,922
Slice LUTs	150,720	14,263	15,106	21,221
Slices	37,680	5,926	6,900	8,612
RAMB36E1	416	24	72	258
RAMB18E1	832	0	24	36
FIFO36E1	416	8	32	128
DSP48E1	768	240	240	240
Max. Frequency (MHz)		58.01	56.99	55.42

Table 6.9: Place and route results for the first MatMul implementation set for a XC6VCX240T FPGA device. The second left-side column shows the number of FPGA available resources.

The first set of PAR results is shown in table 6.9 in terms of the slice registers, slice LUTs, slices, DPS48E1 and BRAMs. Recall that BRAMs could be used as: FIFO elements (FIFO36E1) or as 36 Kbit dual-port memories (RAMB32E1) or as two 18 Kbit dual-port memories (RAMB18E1). Results exposed in table 6.9 show that at least an 80% of FPGA logic resources are unused. However, the percentage of the FPGA memory resources used for implementations *MatMul-On-8x8-086* and *MatMul-On-8x8-171* are 7% and 27%, respectively; showing an increment of the FPGA memory resources. In fact, note that the number of BRAMs required for the *MatMul-On-8x8-171* implementation is almost four times the memory resources required by the *MatMul-On-8x8-086* implementation due to the problem size grows in a quadratic way. Also, these results suggest that an 8×8 processor array system implementation varying the control word width to $W_c = 10$ is possible from the FPGA logic resources point of view. However, the largest problem size supported by $W_c = 10$ (i.e. a problem size of 342×342) is not fully achievable because the number of BRAMs available in the XC6VCX240T FPGA device are exceeded. In this sense, the number of BRAMs available an FPGA device become a technological limitation. Thus, table 6.9 shows the PAR results of a third implementation (*MatMul-On-8x8-320*), which is able to solve problems of size 320×320 though the control is able to deal with a larger problem size (342×342). Implementation *MatMul-On-8x8-320* uses the 97% of the BRAMs available in the XC6VCX240T FPGA, whereas the requirement of LUTs and registers resources do not exceed the 15%. At last, note that the

operational frequencies degradation is 5% when the control word is increased from $W_c = 8$ to $W_c = 10$, *i.e.* an increment of 25% in the word width.

The results shown in table 6.9 suggest that in order to solve larger problem sizes, selecting an FPGA device with more BRAMs, reducing the data word width, placing the memory banks or intermediate memories outside of the FPGA are possible solutions to deal with the limitation of FPGA memory elements. In this sense, a second set of implementations, placing the memory banks outside of the FPGA, was targeted for an FPGA device with a major number of BRAMs than the XC6VCX240T FPGA device. Thus, the second implementation set was targeted for an XC6VSX475T FPGA device and it consists of twelve different implementations varying the control word (W_c) and the processor array size (SSp_0 and SSp_1). Table 6.10 describes each implementation with their respective name and parameters. In this second set of implementations, it should be pointed out that processor arrays are limited only to square array sizes (*i.e.* $SSp_0 = SSp_1$), and to control words whose N_{max} values are totally supported by the BRAMs available in the selected FPGA.

Implementation Code Name	Processor Array Size	Max. Prob. Size (N_{max})	Control Word (W_c)
<i>MatMul-Off-2x2-086</i>	2 × 2 PEs	86 × 86	8-bit
<i>MatMul-Off-2x2-171</i>	2 × 2 PEs	171 × 171	9-bit
<i>MatMul-Off-2x2-342</i>	2 × 2 PEs	342 × 342	10-bit
<i>MatMul-Off-2x2-683</i>	2 × 2 PEs	683 × 683	11-bit
<i>MatMul-Off-4x4-086</i>	4 × 4 PEs	86 × 86	8-bit
<i>MatMul-Off-4x4-171</i>	4 × 4 PEs	171 × 171	9-bit
<i>MatMul-Off-4x4-342</i>	4 × 4 PEs	342 × 342	10-bit
<i>MatMul-Off-4x4-683</i>	4 × 4 PEs	683 × 683	11-bit
<i>MatMul-Off-8x8-086</i>	8 × 8 PEs	86 × 86	8-bit
<i>MatMul-Off-8x8-171</i>	8 × 8 PEs	171 × 171	9-bit
<i>MatMul-Off-8x8-342</i>	8 × 8 PEs	342 × 342	10-bit
<i>MatMul-Off-8x8-683</i>	8 × 8 PEs	683 × 683	11-bit

Table 6.10: Description of the second MatMul implementation set with their respective parameters. These implementations assumes off-chip memories for storing the input and output matrixes.

The PAR results for the second implementation set are shown in tables 6.11, 6.12 and 6.13 for processor arrays of size 2×2 , 4×4 , and 8×8 PEs, respectively. In the same way as previous PAR results, these tables show the number of FPGA resources (slice registers, slice LUTs, slices, DPS48E1 and BRAMs) required by each implementation. Since this set does not implement memory banks, the reported number of BRAMs are limited to implement the FIFO functionality (FIFO36E1). When comparing the number FPGA resources occupied among the different array processor sizes, it should be noted that despite that the sizes are incremented in a factor of 4x, the number of slice registers is incremented by an average factor of 3.4x, whereas the number of slice LUTs is incremented on an average factor of 2.7x. The worst case, for slice registers and slice LUTs, is when implementations *MatMul-Off-4x4-086* and *MatMul-Off-8x8-086* are compared, with an increment of 3.8x in the case of slice registers and 3.9x for slice LUTs.

Furthermore, note that incrementing the control word W_c has a major impact on the number of slice registers and slice LUTs resources for small processor arrays than for larger arrays, as discussed in subsection 6.2.1. In general, the operational frequencies are above of 59 MHz. These relative slow frequencies are originated from the control scheme, since the Max/Min sub-modules (placed at the sequence generator) generate the longest critical path due to their combinational logic for calculating the loop bound during execution time. Also, it should be noted that an outlier value in the operational frequency in the case of the *MatMul-Off-4x4-342* implementation. Viewing the placed and routed processor arrays (using the Xilinx's PlanAhead), it is observed that the *MatMul-Off-4x4-342* critical path is physically the longest path of the MatMul arrays.

In addition, it should be noted from tables 6.11, 6.12 and 6.13 that the number of DSP blocks (DSP36E1) for a same processor array size remains constant, since the DSP blocks are only required in the processor array PEs and for the AGUs modules. Besides, note that the number of BRAMs (FIFO36E1) is variable for the same processor array size, but number of BRAMs remains constant when the problem size has a same value.

FPGA Resources		MatMul-Off	MatMul-Off	MatMul-Off	MatMul-Off
Resource	Available	2x2-086	2x2-171	2x2-342	2x2-683
Slice Registers	595,200	1,406	1,475	1,730	2,539
Slice LUTs	297,600	1,351	1,769	2,748	4,969
Slices	74,400	560	722	1,021	2,124
FIFO36E1	1,064	8	32	128	512
DSP48E1	2,016	24	24	24	24
Max. Frequency (MHz)		72.02	78.10	73.76	71.75

Table 6.11: Place and route results for the implementations of the 2×2 MatMul system for a XC6VSX475T FPGA device. The second left-side column shows the number of FPGA available resources.

FPGA Resources		MatMul-Off	MatMul-Off	MatMul-Off	MatMul-Off
Resource	Available	4x4-086	4x4-171	4x4-342	4x4-683
Slice Registers	595,200	5,260	5,341	5,657	6,506
Slice LUTs	297,600	3,756	4,789	6,274	10,143
Slices	74,400	1,611	1,937	2,552	4,231
FIFO36E1	1,064	8	32	128	512
DSP48E1	2,016	72	72	72	72
Max. Frequency (MHz)		78.00	75.41	59.10	69.17

Table 6.12: Place and route results for the implementations of the 4×4 MatMul system for a XC6VSX475T FPGA device. The second left-side column shows the number of FPGA available resources.

FPGA Resources		MatMul-Off	MatMul-Off	MatMul-Off	MatMul-Off
Resource	Available	8x8-086	8x8-171	8x8-342	8x8-683
Slice Registers	595,200	20,064	20,275	20,685	21,673
Slice LUTs	297,600	15,020	15,629	19,876	25,391
Slices	74,400	5,655	5,965	7,341	10,402
FIFO36E1	1,064	8	32	128	512
DSP48E1	2,016	240	240	240	240
Max. Frequency (MHz)		73.88	74.70	65.26	63.64

Table 6.13: Place and route results for the implementations of the 8×8 MatMul system for a XC6VSX475T FPGA device. The second left-side column shows the number of FPGA available resources.

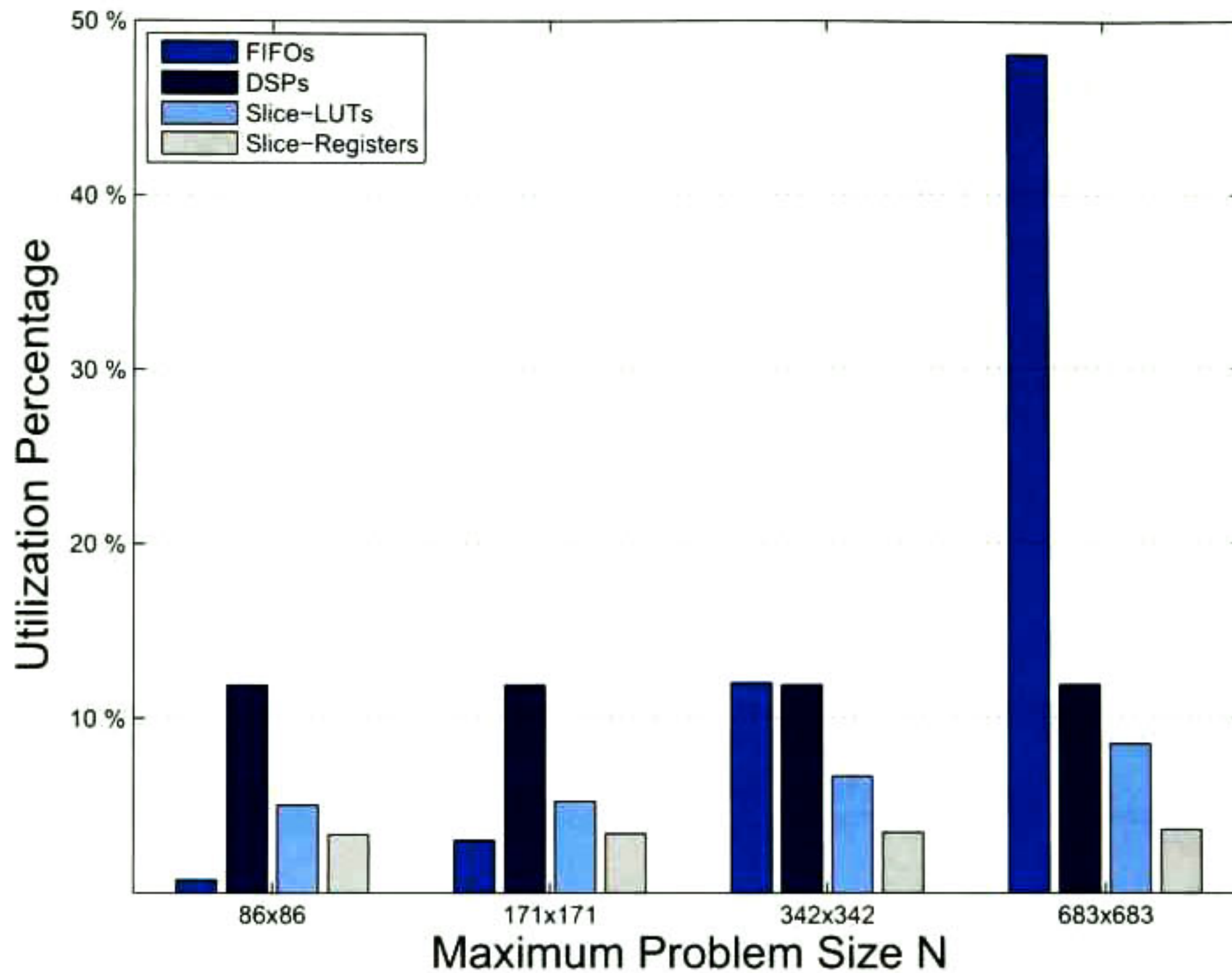


Figure 6.2: FPGA resource utilization percentage varying the maximum problem size (control word) for implementations shown in table 6.13.

With the purpose of showing the FPGA resource utilization percentages, figure 6.2 exemplifies the FPGA utilization percentage for table 6.13, *i.e.* the processor arrays of 8×8 PEs varying the control word. In this figure, note that the number of BRAMs is duplicated when the maximum problem size changes, while the number of DSPs remains constant. Also, note that the number of slice registers and slice LUTs do not exceed the 10% of the FPGA resources. This suggests that larger arrays could be implemented in FPGA technology considering only the FPGA computational resources. However, increasing the processor array size would lead to increase the interconnections among PEs, and consequently the FPGA dedicated interconnection structure might be not enough for mapping the processor array topology, leading to use the FPGA slices as interconnections. Similar graphs for tables 6.11 and table 6.12 are omitted since their percentage of the required slice LUTs and slice registers are low.

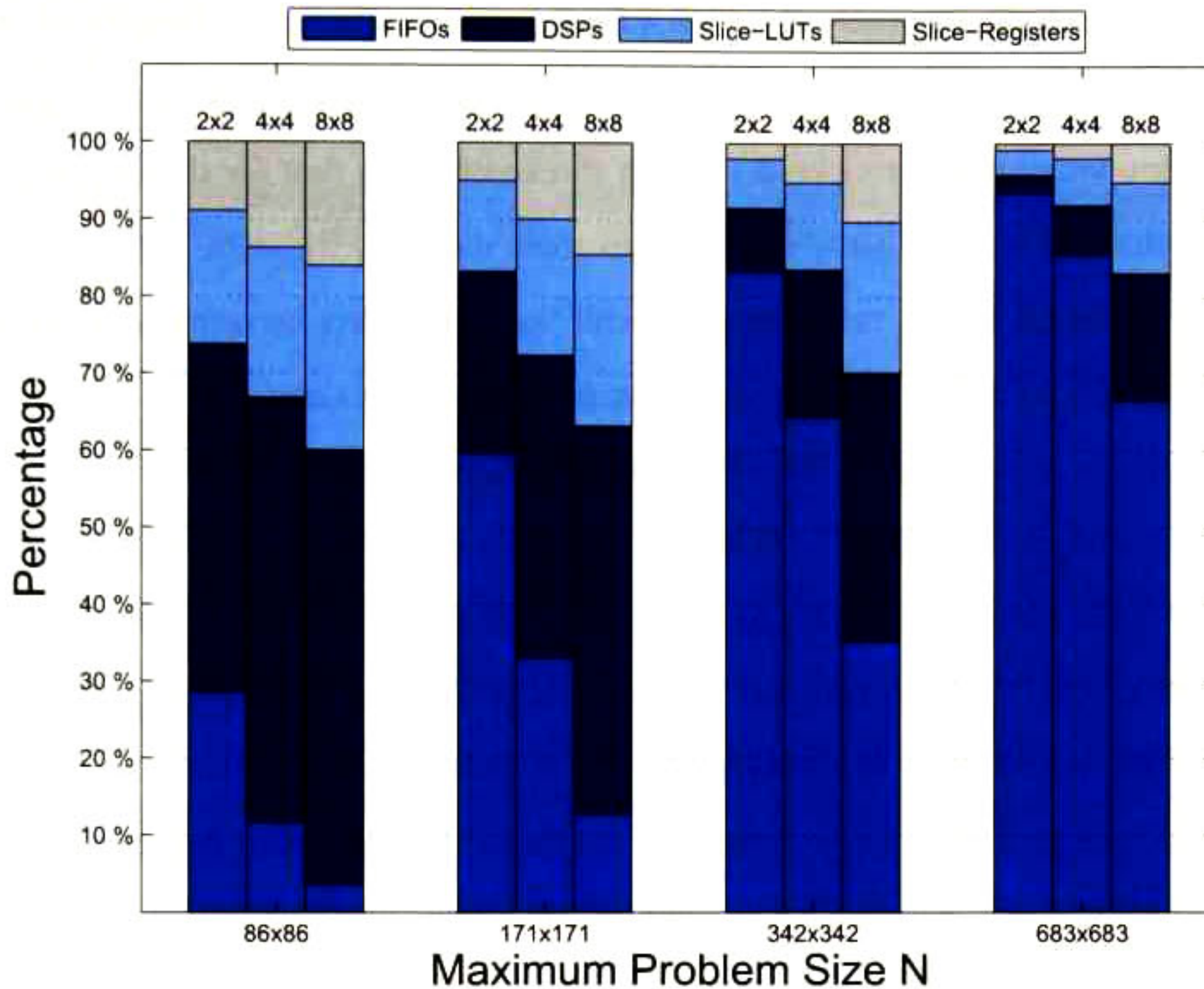


Figure 6.3: Distribution of the FPGA resources for data of tables 6.11, 6.12 and 6.13. Data are grouped according to the maximum problem size, and each member of a group corresponds to different processor array size.

Figure 6.3 shows the FPGA resource distribution considering each implementation presented in table 6.10. Each bar corresponds to a different processor array size, and each set of grouped bars corresponds to the maximum problem size. This figure shows the percentage of the implementation that is mapped to slice registers, slice LUTs, BRAMs and DSPs, *i.e.* it considers each implementation as a complete design (100%) and it assigns a unitary percentage value for each FPGA resource. The main purpose of this graphic is to show that increasing the maximum problem size leads to increase the percentage that an implementation dedicates to memory resources instead of computing elements such as slice LUTs or DSPs. It is important to emphasize that since this graphic assigns a unitary weight for each FPGA resource, it is only an optimistic and general approximation of the FPGA resources distribution given a processor array. A more realistic silicon distribution would assign a different weight for each FPGA resource according to its computational density or storage

capacity in order to weight equitably the FPGA heterogeneous elements. Despite of the limitation of this general approximation, the graphic helps to visualize the percentage that an implementation dedicates to memory and computing resources. In this sense, note that for the largest problem sizes, the percentage that a processor array dedicates to store data is 93%, 85% and 66% for the 2×2 , 4×4 and 8×8 processor arrays, respectively. Recall that these data correspond to the intermediate memories, *i.e.* FIFOs L2. The percentages that processor arrays dedicate to store data indicate that for solving larger problem size, the FPGA resources are more dedicated to store data than for performing computations; despite of FPGAs are mainly designed for performing computations rather than for storing data. In other words, although the FPGA has a large number of computational elements in form of LUTs, these elements are not totally used as in the case of the BRAMs when larger problem sizes are solved. On the other hand, for the smallest problem size, the DSPs require above of 45% of the processor array design. With respect of the slice registers and slice LUTs, the highest percentages are achieved for implementation *MatMul-Off-8x8-086* with 15% and 23%, respectively; *i.e.* for smaller problem sizes.

6.4.1.2 Comparison

A direct comparison against other works is complex, but still possible in the case of PARO [64]. However, a comparison against MMAAlpha is harder since in [61] authors show synthesis results for the multiplication of two small non-square matrixes (resulting in a matrix of 10×4), and limiting themselves to show only the number of slices required for a XCV800 FPGA device. The synthesis results shown in [64] are targeted for a Virtex-II 8000 FPGA device, implementing two 2×2 processor arrays named as *MatMul-PARO-2x2-100* and *MatMul-PARO-2x2-6* for solving problems of size of 6×6 and 100×100 , respectively. These implementations use a 16-bit integer arithmetic, and they are able to solve only the problem size for which they were derived. It should be noted that these results are targeted for an FPGA with a 4-LUT technology, while the results presented in the previous subsection are targeted for a 6-LUT technology. Despite that this technological difference, some comparisons could be done using implementation *MatMul-Off-2x2-171*, and using a new implementation called *MatMul-Off-2x2-6*.

FPGA Resource	PARO [64] XC2V8000		Proposed Approach XC6V5X475T	
	MatMul-PARO 2x2-100	MatMul-PARO 2x2-6	MatMul-Off 2x2-171	MatMul-Off 2x2-6
Slice 4-LUTs	1,736	829	-	-
Slice 6-LUTs	-	-	1,769	967
Equivalent 6-LUTs	578	276	1,769	967
Equivalent 4-1 MUXs	868	415	1,769	967
Slice Registers	931	795	1,475	1,367
18 Kbits BRAMs	505	0	64	0
18 × 18 DSP Block	4	4	-	-
18 × 25 DSP Block	-	-	24	24
Operational Frequency (MHz)	97	115	78	79
Execution Time (cycles)	260,000	72	255,000	72
Matrix Problem Size	100	6	[1, ..., 171]	[1, ..., 6]

Table 6.14: Comparison of two 2×2 MatMul processor arrays generated by PARO and by the proposed processor arrays.

Table 6.14 shows the comparison of the synthesis results reported in [64] and the PAR results of the proposed approach. Such comparison is in terms of LUTs, registers, BRAMs, multipliers, operational frequency, and number of clock cycles to complete the multiplication of two matrixes of different sizes. Since the LUTs for both cases are different, they are normalized in terms of the number of 4-1 multiplexers [98], and in terms of equivalent 6-LUTs [66]. It should be noted that the results from [64] are limited to synthesis, due to the Virtex-II 8000 FPGA device has only 168 BRAMs (each storing 18 Kbits), making the PAR not possible. Although the processor arrays generated by PARO require a smaller number of equivalent 4-1 multiplexers and 6-LUTs, it should be noted that processor arrays *MatMul-Off-2x2-171* and *MatMul-Off-2x2-6* have a 32-bit data word, contrary to PARO implementations of 16-bit. In this sense, the number of equivalent 4-1 multiplexers and 6-LUTs shown in table 6.14 could be halved in the case of the proposed implementations, resulting in a similar number of multiplexers and a 34% more of equivalent LUTs compared to PARO implementations.

Furthermore, using a 32-bit data word for performing the PE's MAC operation requires three 18×25 DSP Blocks; while for computing the 16-bit MAC operation, one 18×18 DSP Block is required.

Therefore, half of the 24 DSP Blocks required by the proposed arrays are used for computing the MatMul operation, while the other 12 DSP Blocks are employed by the AGUs. With respect to the number of slice registers, implementations *MatMul-PARO-2x2-100* and *MatMul-PARO-2x2-6* require fewer elements than the proposed arrays. Mainly, this is due to in the proposed processor arrays, the control cells propagate more information (using the *Index Bus*) to the processor array than the processor generated by PARO, where only single signals are propagated through the array. Recall that the *Index Bus* is a set of signals composed by the tile and time indexes, and the problem size, each composed of W_c bits.

Additionally, despite that the proposed processor arrays require more computational elements than PARO implementations, it must be pointed out that the number of cycles required for multiplying two matrixes of size 100×100 , in the case of the proposed *MatMul-Off-2x2-171* implementation is 5,000 cycles faster than the results reported in [64]. In the case of *MatMul-Off-2x2-6* and *MatMul-PARO-2x2-6* implementations both require the same number of clock cycles for multiplying two matrixes of size 6×6 . Besides, both proposed implementations are able to deal with problem sizes within the rage of values instead of solving a unique problem size like the PARO processor arrays. Finally, note that the proposed arrays have a minor operational frequency degradation if the problem size is increased compared to the processor arrays derived by PARO.

6.4.2 Cholesky Results

In the same way as in the MatMul case, the PAR results are grouped into two different sets and targeted for the same FPGA devices. These two sets are named as third and fourth sets in order to continue the previous count. The third set of implementations consists of three Cholesky processor array including control, data-path and memory fully implemented in the FPGA, whereas the fourth set of implementations consists of assuming that memory banks are outside of the FPGA.

These two sets use the same configuration design parameters, *i.e.* the same scheduler, projection, and iteration interval. The values of these parameters are $\vec{\lambda}_l = [1, 1, 1]$, $\vec{u} = [0, 0, 1]^t$, and $P = 21$; while the partitioning parameters vary for each implementation. The data word width is 32-bit and the control word varies for each implementation. The FIFOs required for the processor arrays were implemented by using the Xilinx's IP-Core as shift registers and as built-in FIFOs for FIFOs L1 and FIFOs L2, respectively; whereas memory banks were implemented using the FPGA's built-in RAMs. Finally, division and square root operations were implemented by using Xilinx's IP-Core.

6.4.2.1 Place and Route

Table 6.15 shows the description of the third implementations set. This set is formed by three different Cholesky processor array implementations. Similarly to the first set of MatMul implementations, the Cholesky implementations use the same partitioning parameters of $SSp_0 = 8$ and $SSp_1 = 8$ (an 8×8 processor array), and they differ in their control word width W_c (leading to different N_{max} values). Also, table 6.15 shows that the memory banks, required by the architectural cases, were implemented using the FPGAs BRAMs. The targeted FPGA device used for implementing this set was a Virtex-6 XC6VCX240T.

Implementation Code Name	Processor Array Size	Max. Prob. Size (N_{max})	Control Word (W_c)
<i>Chol-On-8x8-086</i>	8×8 PEs	86×86	8-bit
<i>Chol-On-8x8-171</i>	8×8 PEs	171×171	9-bit
<i>Chol-On-8x8-342</i>	8×8 PEs	342×342	10-bit

Table 6.15: Description of the third Cholesky implementation set with their respective parameters. These implementations use the FPGA on-chip memories for storing the input and output matrixes.

The third set of PAR results is shown in table 6.16 in terms of the slice registers, slice LUTs, slices, DPS48E1 and BRAMs. Again, recall that BRAMs could be used as FIFO36E1, or as RAMB32E1 or as two RAMB18E1 memories. In these PAR results, some RAMB18E1 modules are used for implementing a divisor functionality. In fact, the projection vector $\vec{u} = [0, 0, 1]^t$ was chosen since it

derivates processor arrays whose divisions and square root operations are only performed in the PEs placed at the main diagonal of the array. In other words, eight RAMB18E1 modules are dedicated to implement the division operation while the remaining RAMB18E1 modules are dedicated to store input and output data.

FPGA Resources		Chol-On	Chol-On	Chol-On
Resource	Available	8x8-086	8x8-171	8x8-342
Slice Registers	301,440	24,246	24,709	25,588
Slice LUTs	150,720	22,671	25,310	29,276
Slices	37,680	8,405	9,107	10,585
RAMB36E1	416	16	48	196
RAMB18E1	832	8	24	40
FIFO36E1	416	8	32	128
DSP48E1	768	328	328	328
Max. Frequency (MHz)		66.19	64.28	64.77

Table 6.16: Place and route results for the third Cholesky implementation set for a XC6VCX240T FPGA device. The second left-side column shows the number of FPGA available resources.

The results exposed in table 6.16 show that there is at least 80% of unused FPGA logic resources, whereas the percentage of the FPGA memory resources used for implementations *Chol-On-8x8-086*, *Chol-On-8x8-171* and *Chol-On-8x8-342* are 7%, 22% and 83%, respectively. Contrary to the MatMul case, in the Cholesky decomposition the largest problem size supported by $W_c = 10$ is totally achievable, since the Cholesky algorithm requires a less number of memory banks than the MatMul algorithm. Recall that according to the PRA specification, for computing the matrix decomposition only one input matrix is required, whereas the multiplication requires two input matrixes; and in both algorithms, the result is only one matrix. In relation to the operational frequencies, note that there is a degradation of 3% when the control word is increased from $W_c = 8$ to $W_c = 10$, *i.e.* an increment of 25% in the word width.

Although the memory bank requirements for Cholesky implementations are less than in the case of MatMul processor array, placing the memory banks outside of the FPGA is also helpful for supporting

Implementation Code Name	Processor Array Size	Max. Prob. Size (N_{max})	Control Word (W_c)
<i>Chol-Off-2x2-086</i>	2 × 2 PEs	86 × 86	8-bit
<i>Chol-Off-2x2-171</i>	2 × 2 PEs	171 × 171	9-bit
<i>Chol-Off-2x2-342</i>	2 × 2 PEs	342 × 342	10-bit
<i>Chol-Off-2x2-683</i>	2 × 2 PEs	683 × 683	11-bit
<i>Chol-Off-4x4-086</i>	4 × 4 PEs	86 × 86	8-bit
<i>Chol-Off-4x4-171</i>	4 × 4 PEs	171 × 171	9-bit
<i>Chol-Off-4x4-342</i>	4 × 4 PEs	342 × 342	10-bit
<i>Chol-Off-4x4-683</i>	4 × 4 PEs	683 × 683	11-bit
<i>Chol-Off-8x8-086</i>	8 × 8 PEs	86 × 86	8-bit
<i>Chol-Off-8x8-171</i>	8 × 8 PEs	171 × 171	9-bit
<i>Chol-Off-8x8-342</i>	8 × 8 PEs	342 × 342	10-bit
<i>Chol-Off-8x8-683</i>	8 × 8 PEs	683 × 683	11-bit

Table 6.17: Description of the fourth Cholesky implementation set with their respective parameters. These implementations assumes off-chip memories for storing the input and output matrixes.

larger problem sizes. In this sense and analogously to the second set of implementations, a fourth set of Cholesky processor arrays was targeted for a Virtex-6 XC6V5X475T FPGA device placing the memory banks outside of the FPGA. This fourth set consists of twelve different implementations varying the control word (W_c) and the processor array size (SSp_0 and SSp_1). Table 6.17 describes each implementation with its respective name and parameters.

The PAR results for the fourth implementation set are shown in tables 6.18, 6.19 and 6.20 for processor arrays of size 2 × 2, 4 × 4, and 8 × 8 PEs, respectively. In the same way as previous PAR results, these tables show the number of FPGA resources required by each implementation. As in the case of the second implementation set, the fourth set does not implement memory banks. The reported number of BRAMs are reported as FIFO36E1 elements and as RAMB18E1. Recall that in the case of Cholesky processor arrays the RAMB18E1 modules are dedicated to implement the division operation.

FPGA Resources		Chol-Off	Chol-Off	Chol-Off	Chol-Off
Resource	Available	2x2-086	2x2-171	2x2-342	2x2-683
Slice Registers	595,200	2,835	2,912	3,176	3,990
Slice LUTs	297,600	3,504	3,911	4,908	7,112
Slices	74,400	1,260	1,545	1,735	2,124
FIFO36E1	1,064	8	32	128	512
RAMB18E1	2,128	2	2	2	2
DSP48E1	2,016	46	46	46	46
Max. Frequency (MHz)		80.86	76.63	82.21	77.07

Table 6.18: Place and route results for the 2×2 Cholesky implementation set for a XC6VSX475T FPGA device. The second left-side column shows the number of FPGA available resources.

FPGA Resources		Chol-Off	Chol-Off	Chol-Off	Chol-Off
Resource	Available	4x4-086	4x4-171	4x4-342	4x4-683
Slice Registers	595,200	7,864	7,996	8,411	9,316
Slice LUTs	297,600	8,050	8,863	10,334	14,155
Slices	74,400	3,052	3,372	3,863	5,811
FIFO36E1	1,064	8	32	128	512
RAMB18E1	2,128	4	4	4	4
DSP48E1	2,016	116	116	116	116
Max. Frequency (MHz)		81.51	79.51	76.62	71.06

Table 6.19: Place and route results for the 4×4 Cholesky implementation set for a XC6VSX475T FPGA device. The second left-side column shows the number of FPGA available resources.

FPGA Resources		Chol-Off	Chol-Off	Chol-Off	Chol-Off
Resource	Available	8x8-086	8x8-171	8x8-342	8x8-683
Slice Registers	595,200	24,246	24,677	25,548	26,803
Slice LUTs	297,600	22,360	24,375	27,469	34,256
Slices	74,400	8,534	9,374	10,182	13,326
FIFO36E1	1,064	8	32	128	512
RAMB18E1	2,128	8	8	8	8
DSP48E1	2,016	328	328	328	328
Max. Frequency (MHz)		75.76	73.65	72.92	68.86

Table 6.20: Place and route results for the 8×8 Cholesky implementation set for a XC6VSX475T FPGA device. The second left-side column shows the number of FPGA available resources.

Note that in this set of implementations, the processor array size is also incremented by a factor of 4x. However, the number of slice registers and slice LUTs are incremented by an average factor of 2.8x and 2.4x, respectively. The major increment is when implementations *Chol-Off-4x4-086* and *Chol-Off-8x8-086* are compared, with an increment of 3x on the number of slice registers and 2.7x in the case of slice LUTs. Besides, note that the average increments in Cholesky implementations are lower than in the case of MatMul implementations, whereas the major increment is exhibited in the same case as the MatMul implementations, *i.e.* when $W_c = 8$ and the processor array is increased from 4×4 to 8×8 PEs. Also, similarly to the MatMul set, incrementing the control word W_c has a major impact on the number of slice registers and slice LUTs resources for small processor arrays than for larger arrays (see subsection 6.2.1).

In the case of the operational frequencies, note that they are above of 68 MHz, and that these frequencies are faster than in the MatMult implementation. In fact, for implementation *Chol-Off-4x4-086* the maximum operational frequency achieved is 81 MHz, resulting in the fastest processor array implemented. One possible reason for achieving greater frequencies in the case of Cholesky implementations might come from the use of specialized IP cores for division and square root operations. It is worth to mention that the *Chol-Off-2x2-342* implementation is the fastest of the 2×2 processor array, and that if the maximum problem size required to be solved is $N_{max} \leq 342$ and there is not restriction about the FPGA resources, then *Chol-Off-2x2-342* implementation should be considered. Also, note that the number of BRAMs for each implementation corresponds to the processor array size and the number of DSPs blocks remains constant for the same processor array size. In addition, as in the MatMul implementation sets, the number of BRAMs is variable for the same processor array size, but it remains constant if the problem size is set to a fixed value.

Figure 6.4 shows the FPGA resource utilization percentage for implementations shown in table 6.20. Similarly to the MatMul case (figure 6.2), the number of BRAMs is duplicated when the maximum problem size is duplicated, while the number of DSPs remains constant. Also, the number of slice registers and slice LUTs do not exceed the 10% of the FPGA resources.

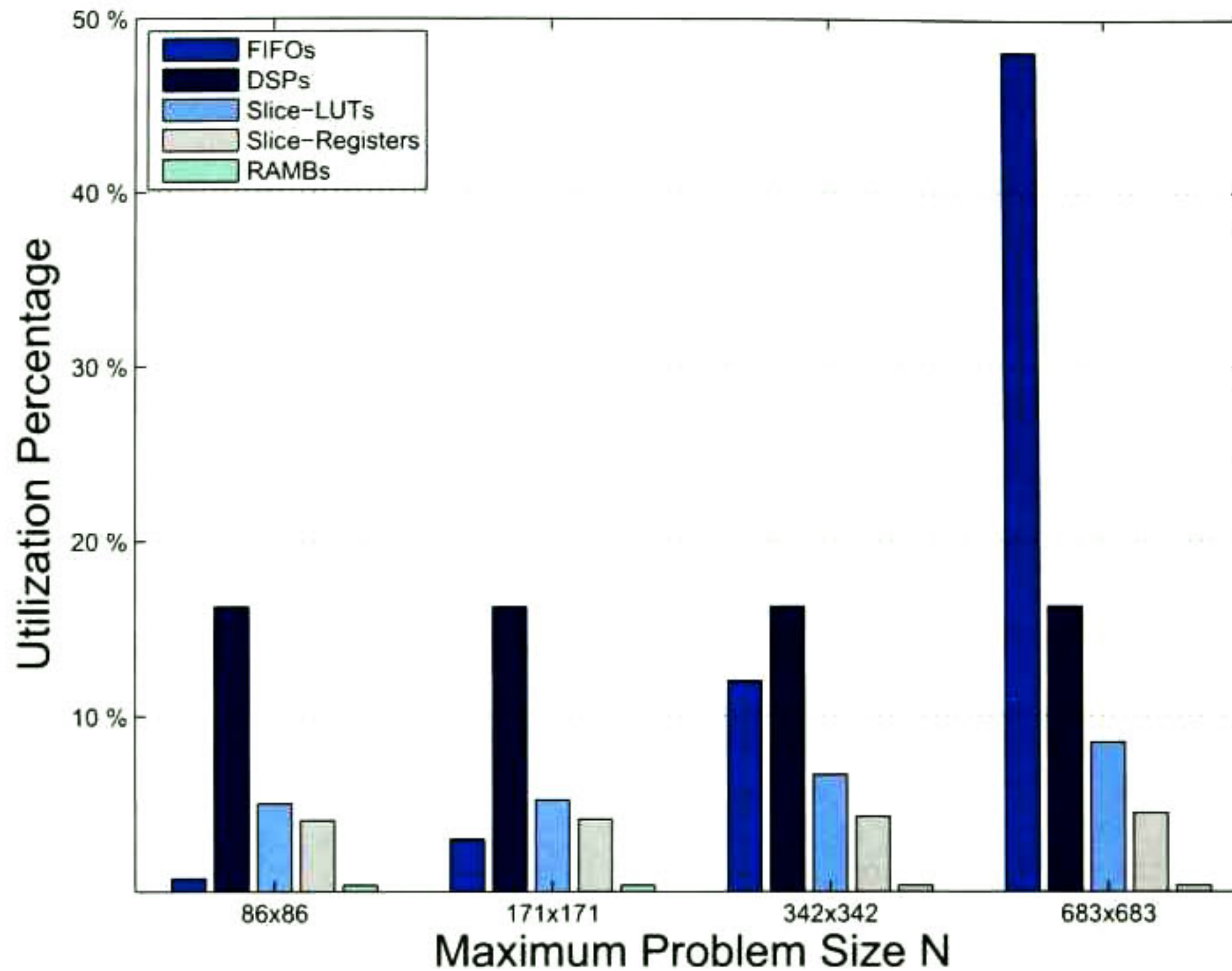


Figure 6.4: FPGA resource percentage utilization varying the maximum problem size (control word) for implementations shown in table 6.20.

Finally, with the purpose of showing that increasing the maximum problem size leads to increase the percentage that an implementation dedicates to memory resources; figure 6.5 shows the percentage of the implementation that is mapped to the FPGA resources considering each Cholesky implementation as a complete design. Recall that for this kind of graphic, a same weight is assigned for each FPGA resource. Also, recall that each bar in figure 6.5 corresponds to a different processor array size, and each set of grouped bars corresponds to the maximum problem size. Similarly to figure 6.3 when the problem size of 683×683 is considered, the percentage that a processor array dedicates to store intermediate data is 89%, 79% and 59% for 2×2 , 4×4 and 8×8 processor arrays, respectively. On the other hand, for the smallest problem size, the DSPs require above of 47% of the processor array design. These percentages indicate that for solving larger problem size, the FPGA resources are more dedicated to store data than for performing computations; whereas for solving smaller problem sizes, the FPGA resources are more dedicate to perform computations

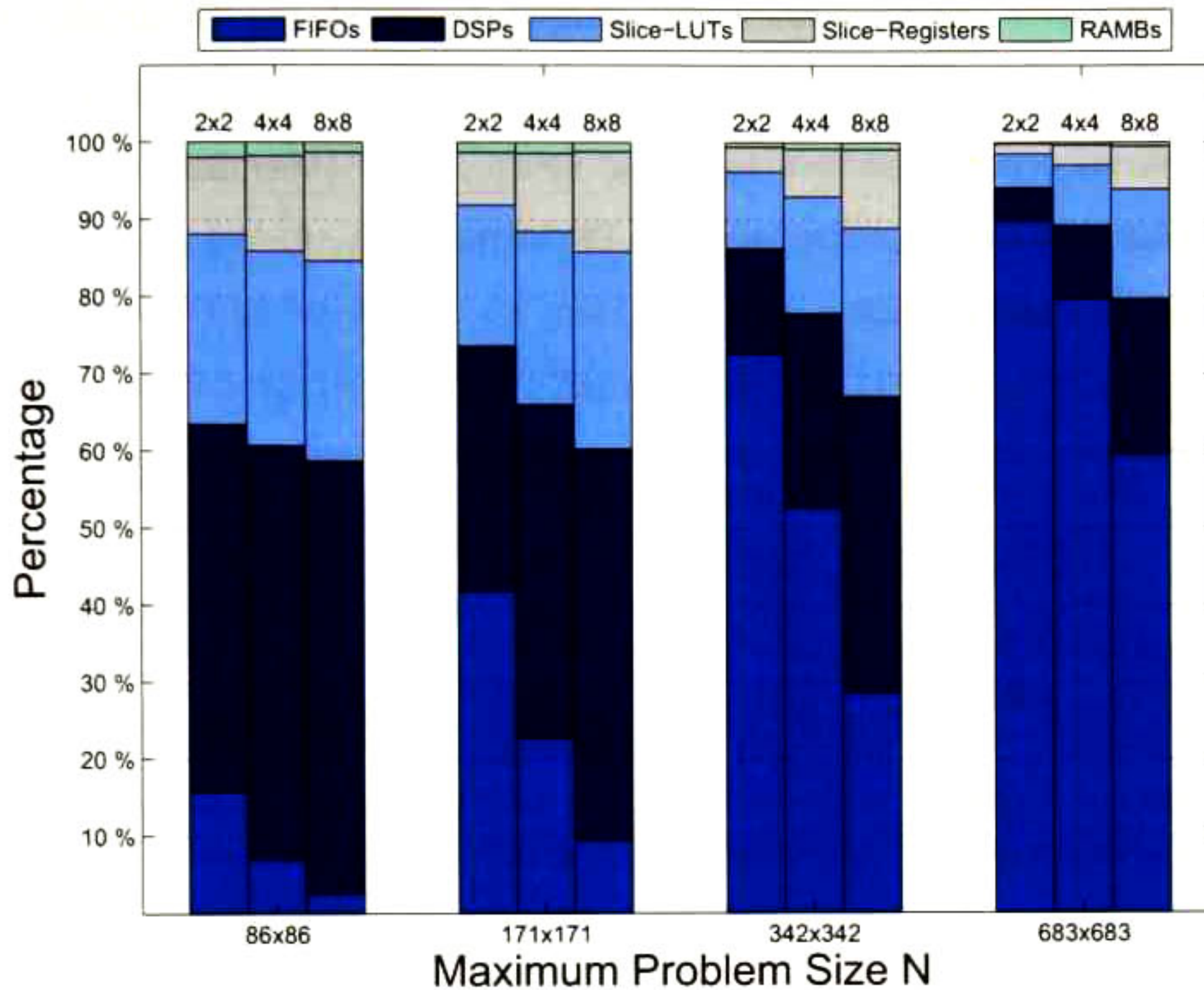


Figure 6.5: Distribution of the FPGA resources for data of tables 6.18, 6.19 and 6.20. Data are grouped according to the maximum problem size, and each member of a group corresponds to different processor array size.

rather than for storing data. Also, note that the percentage of RAMB18E1 dedicated to perform the division operation is low for the smallest array of 2×2 processor arrays, since these BRAMs are not used for storing data but to performing computations.

6.4.3 Embedded Platform

The previous results have been targeted for two Virtex-6 FPGA devices. However, the processor array derived can be targeted for smaller FPGA devices used in embedded platforms, where small problem sizes are required to be solved, and where a balance between power consumption and computational performance is desired. Processor arrays are capable of performing parallel computations and completing a larger volume of work with a low power consumption. In this sense, this section presents the PAR results, speed-up, and power consumption per LUT for the two previous algorithms

implemented as processor arrays targeted for a Spartan-6 family. Mainly, the purpose of this section is showing that the processors arrays, derived by the desing methodology, provide a balance between power and acceleration compared to a sequential soft-processor (MicroBlaze) used in embedded platforms for small problem sizes using the same FPGA technology for the implementation of both sequential and parallel processors.

Implementation Code Name	Implemented Algorithm	Processor Array Size	Max. Prob. Size (N_{max})
<i>MatMul-Off-2x2-250</i>	MatMul	2 × 2 PEs	250 × 250
<i>MatMul-Off-4x4-250</i>	MatMul	4 × 4 PEs	250 × 250
<i>Chol-Off-2x2-342</i>	Cholesky	2 × 2 PEs	180 × 180

Table 6.21: Description of the implementation set targeted for an embedded platform. These implementations assume off-chip memories for storing the I/O matrixes, and a $W_c = 11$ -bit.

The set of implementations, with their respective configurations is shown in table 6.21. This fifth implementation set consists of three different processor arrays, implementing the MatMul and Cholesky decomposition algorithms. The design parameters of the MatMul processor arrays are: $\vec{\lambda}_l = [1, 1, 1]$, $\vec{u} = [1, 0, 0]^t$, and $P = 1$; whereas for Choleksy array the design parameters are: $\vec{\lambda}_l = [1, 1, 1]$, $\vec{u} = [0, 0, 1]^t$, and $P = 21$. Note that the iteration interval for these arrays is equal to the most time expensive operation presented in each algorithm. The data word width is 32-bit and the control $W_c = 11$ for the three implementations. The FIFOs required for the processor arrays were implemented by using the Xilinx's IP-Core as shift registers and as built-in FIFOs for FIFOs L1 and FIFOs L2, respectively. The memory banks required in the memory system are assumed to be off-chip memories. The division and square root operations were implemented by using Xilinx's IP-Cores. Note that these parameters are the same configuration parameters used in subsection 6.4.1 and in subsection 6.4.2.

The fifth implementation set has been targeted for a Spartan-6 XC6SLX45 FPGA device that is included in the Digilent Atlys Development Board [38]. This board includes a 128 MByte DDR2

memory with a 16-bit data bus, which was used for storing the input and output matrixes. Also, for comparison purpose, the MicroBlaze soft-processor has been used for implementing the same two algorithms but in a sequential fashion. The MicroBlaze implementation includes a 64-KB of local memory without cache, and the AXI Bus for peripheral interconnections. The soft-processor was used for computing the loop-kernels and for measuring their execution time including the external memory accesses. The optimization compiling flag was placed in `-O3` in order to obtain the maximum compilation effort.

FPGA Resources		MatMul-Off	MatMul-Off	Chol-Off	MicroBlaze
Name	Available	2x2-250	4x4-250	2x2-180	Processor
Slice Regs	54,576	1,702	5,770	3,207	3,703
Slice LUTs	27,288	2,312	7,607	8,523	3,782
Block RAM	116	116	116	60	42
DSP48A1	58	16	42	26	3
Max. Frequency (MHz)		45.68	44.62	52.45	97.75
Power Consumption (W)		0.398	0.756	0.334	0.973

Table 6.22: Place and Route results for a Microblaze and three processor arrays implementations targeted for a XC6SLX45 FPGA device.

Table 6.22 summarizes the PAR results of the fifth implementation set and the MicroBlaze implementation. Also, table 6.22 shows the operational frequency and the dynamic power consumption estimated by the Xilinx's XPower Analyzer using the maximum operational frequencies obtained after PAR, setting the FPGA supply parameters as $V_{CCINT} = 1.2$, and $V_{CCAUX} = 2.5$ Volts. Note that the operational frequency of the *MatMul-Off-4x4-250* implementation decreases 1% compared against the *MatMul-Off-2x2-250* array implementation, despite that the number of PEs has been quadrupled. Although theoretically with a $W_c = 11$ the processor arrays are able to solve problem sizes no larger than 342×342 , there is again the memory limitation according to the target device characteristics. In the case of the selected XC6SLX45 FPGA device, the number of BRAMs does not allow to solve problem size larger than 250×250 for the MatMul processor array case, and 180×180 for the Cholesky array. In the MatMul case, all the BRAMs are used for storing

intermediate data in FIFO memories. On the other hand, since the IP cores used in the Cholesky array require BRAMs for implementing the division functionality, not all BRAMs are used for data storing. Moreover, the three processor array implementations consume least power compared to the MicroBlaze processor, since their power operational frequencies are almost 2x slower than in the soft-processor. However, such operational frequency disadvantage is overcome with the fact that the processor arrays have at least 4x more processing elements than the MicroBlaze implementation, and that the processor arrays do not stall their computations in order to access to the external memory; thus a speed-up compared to the soft-processor is achieved.

Metric Name	MatMul-Off 2x2-250	MatMul-Off 4x4-250	Chol-Off 2x2-180	MicroBlaze Processor
AVG. Speed-Up	6.05	10.20	5.34	1
AVG. Improvement	24.2	6.5	6.9	1
mW/LUT	0.172	0.099	0.039	0.257

Table 6.23: Average speed-up and energy consumption per LUT of three processor arrays.

Table 6.23 shows the energy consumed by each LUT according to the PAR results shown in table 6.22, the average speed-up and the average improvement compared against the MicroBlaze. The improvement is calculated by multiplying the speed-up, the usage of LUTs and the power consumed of each array compared against the MicroBlaze soft-processor. In order to calculate the speed-up some considerations were made. Recall that the memory system tries to provide as many communication channels as the processor array requires. In the case of the MatMul processor arrays, three and six 32-bit communication channels are required for implementations *MatMul-Off-2x2-250* and *MatMul-Off-4x4-250*, respectively. In contrast, two 32-bit communication channels are required for the *Chol-Off-2x2-180* implementation. Since the MicroBlaze experimental platform has only one 16-bit communication channel, the speed-up results assume the use of the same one-half communication channel. In this sense, although the operational frequencies of the processor arrays are almost 2x slower than the MicroBlaze frequency, an acceleration for the three arrays is achieved. Mainly this is a consequence of that the MicroBlaze processor dedicates more time for performing external memory

accesses than the processor arrays. Recall that the processor arrays include a memory system which is in charge of the external memory accesses while the processor array is working, thus the memory system does not stall the processor array computations.

Besides, note that the processor array implementations consume fewer power per LUT compared against the MicroBlaze processor; therefore the processor arrays perform their operations in a more power-efficient way than the soft-processor. With the power per LUT metric (mW/LUT), a more realistic measurement of the power required for performing computation and comparison against a sequential processor implemented in the same technology can be achieved. In addition, if the speed-up, and usage of LUTs and the total power consumed by each implementation is considered a minimum improvement of 6.5x could be achieved.

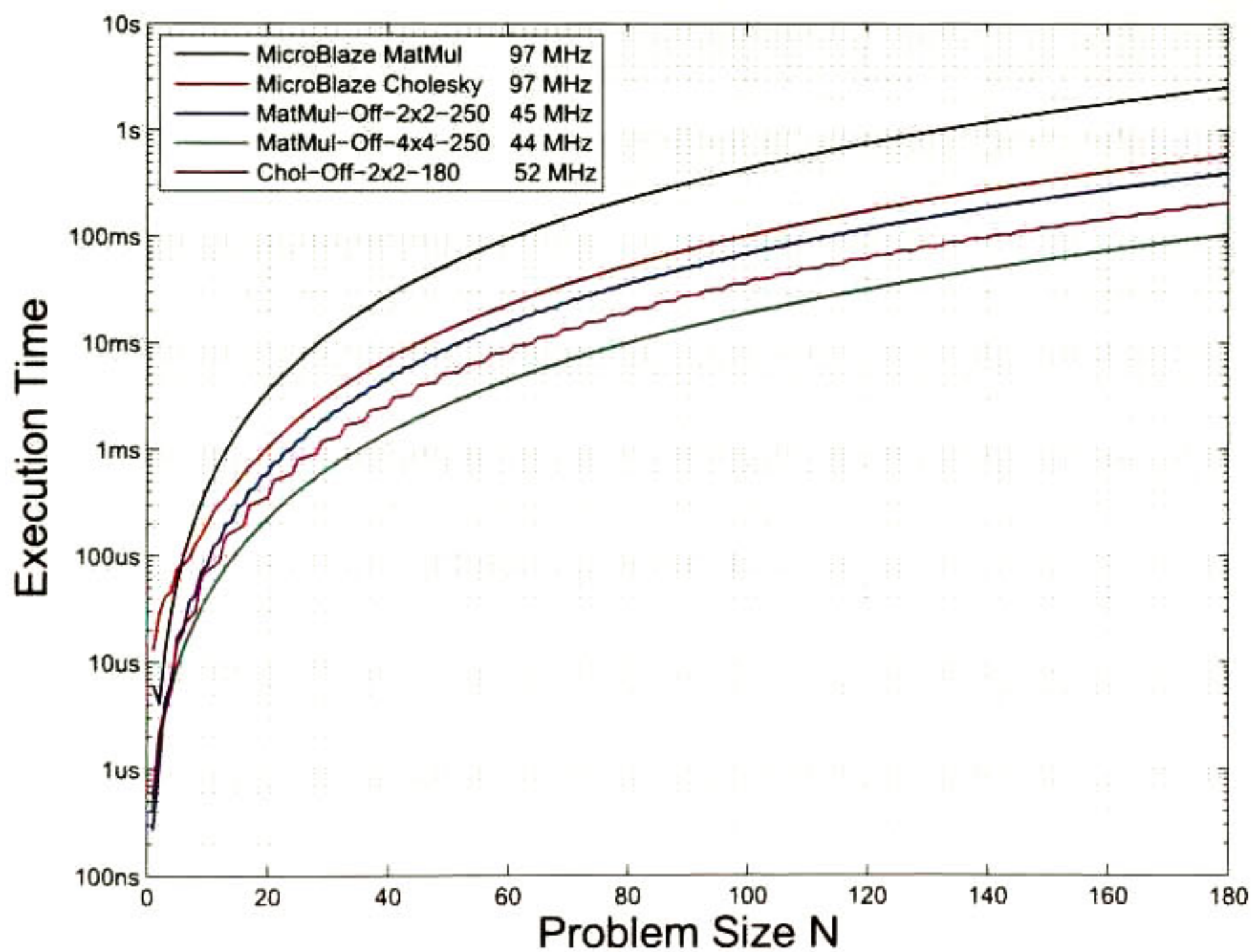


Figure 6.6: MatMul and Cholesky execution times for their processor arrays and their MicroBlaze implementations.

Figure 6.6 shows the time required for solving different problem sizes for the MatMul and Cholesky algorithms implemented in processor arrays compared against their MicroBlaze implementations. The y-axis represents the execution time in logarithmic scale, while the x-axis represents the problem size N from 1×1 to 180×180 . Note that the execution time achieved by each implementation is less than the time required for their corresponding sequential implementation, despite that the processor arrays have slower clock frequencies. In the case of implementation *MatMul-Off-4x4-250* an order of magnitude difference with respect of its sequential implementation is achieved when $N = 80$.

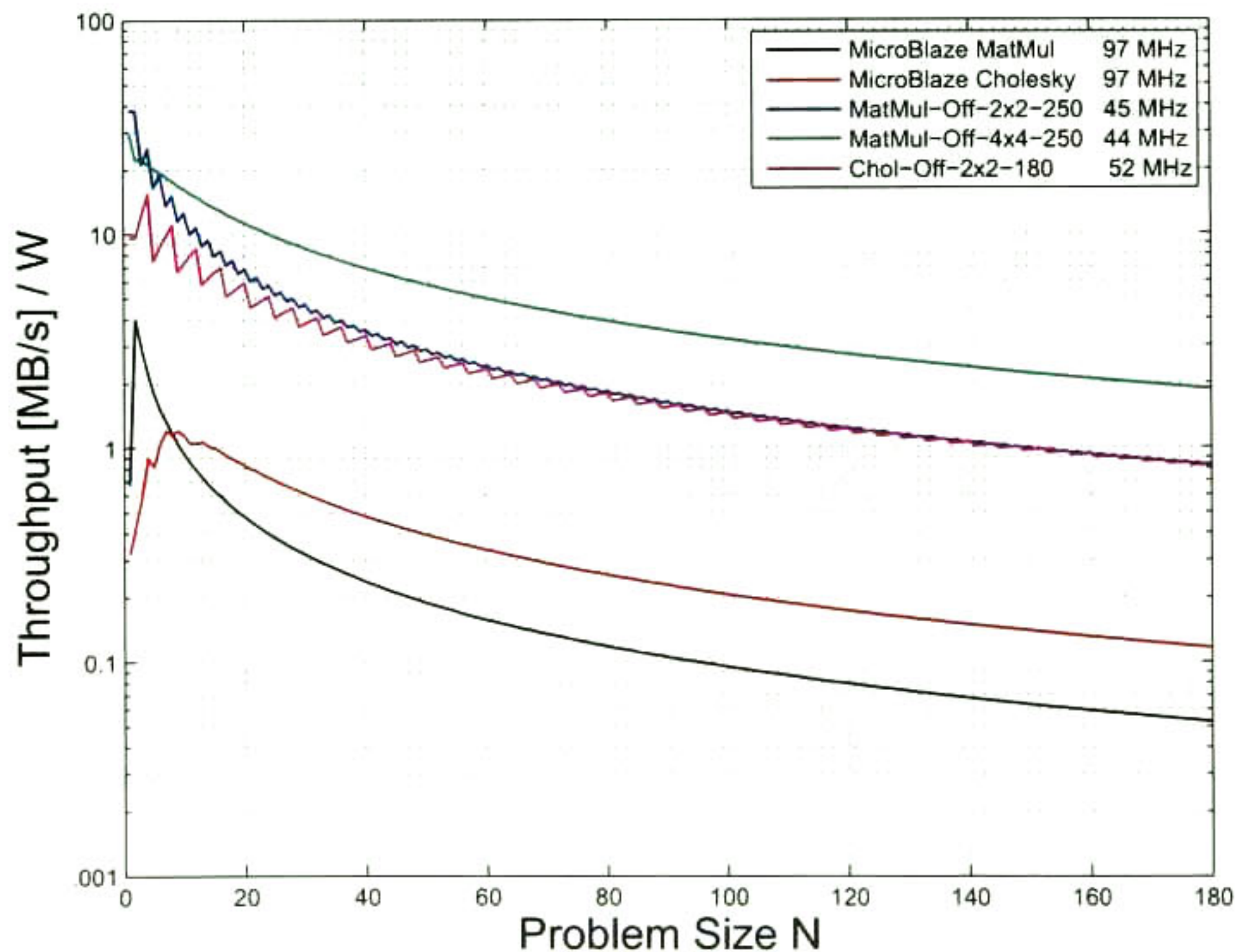


Figure 6.7: MatMul and Cholesky throughput per power unit for their processor arrays and their MicroBlaze implementations.

Finally, figure 6.7 shows the throughput per power unit achieved by the processor arrays implementations and their corresponding MicroBlaze implementations. The y-axis represents the $[\text{MB/s}]/\text{W}$ in logarithmic scale, while the x-axis represents the problem size N_{max} from 1×1 to 180×180 . In this figure, note that the throughput achieved by the three arrays is similar since the

amount of bytes delivered per second is limited by the 16-bit communication channel available in the Atlys Board. Despite this limitation, when $N = 180$ implementation *MatMul-Off-4x4-250* has an improvement of 2.2x on the throughput per power unit compared against the other two processor array implementations. However, it should be noted that implementation *MatMul-Off-4x4-250* has four times more PEs than implementations *MatMul-Off-2x2-250* and *Chol-Off-2x2-180*, and consequently, more communication channels. Nevertheless, the throughputs achieved by the three processor arrays are greater than the throughput of their corresponding sequential implementation.

6.5 Evaluation Metrics

The processor arrays derived by the design methodology followed in this research work could be evaluated using traditional metrics used for parallel systems. Metrics such as the acceleration, efficiency and load imbalance might provide an idea of the expected performance of an algorithm targeted as a processor array before its implementation as an ASIC or into an FPGA. In this sense, this section presents a brief description of these metrics as an introduction to the the subsequent section. It is important to emphasize that the metrics employed for the processor array evaluation are relative, since they are calculated using a single processing element as a baseline instead of using the best sequential implementation known [57]. Despite their apparent limitation, relative metrics are useful for exploring the processor arrays scalability, hence they could be helpful for the exploration of the design space. The following metrics are explained in [57], [75], [97].

6.5.1 Acceleration

Usually, a way used to evaluate a sequential implementation is by measuring its execution time expressed as a function of the problem size, denoted as $T_1(N)$. Similarly, a parallel system with p processors can be evaluated in terms of its parallel execution time expressed as a function of the problem size, denoted as $T_p(N)$. The *relative acceleration* expresses the execution time improvement of a parallel system with respect of a sequential system. Relative acceleration is denoted as:

$$S_p(N) = \frac{T_1(N)}{T_p(N)} \quad (6.4)$$

Additionally, there are two concepts closely related to the acceleration metric: the *degree of parallelism* and *parallelism profile*. The degree of parallelism (DoP) determines the number of active PEs of a parallel system at a certain execution time, whereas the parallelism profile is an abstract measure of the amount of data-parallelism at different time instants during the execution of a parallel system [75]. In other words, the parallelism profile shows all degrees of parallelism of a processor array during the its execution. Figure 6.8 shows an example of a parallelism profile with eight PEs. The average of the area under the parallelism profile line denotes the average parallelism or acceleration during an elapsed time period. In the example shown in figure 6.8, the average parallelism or acceleration is equal to 3.8, and is represented as a dashed line.

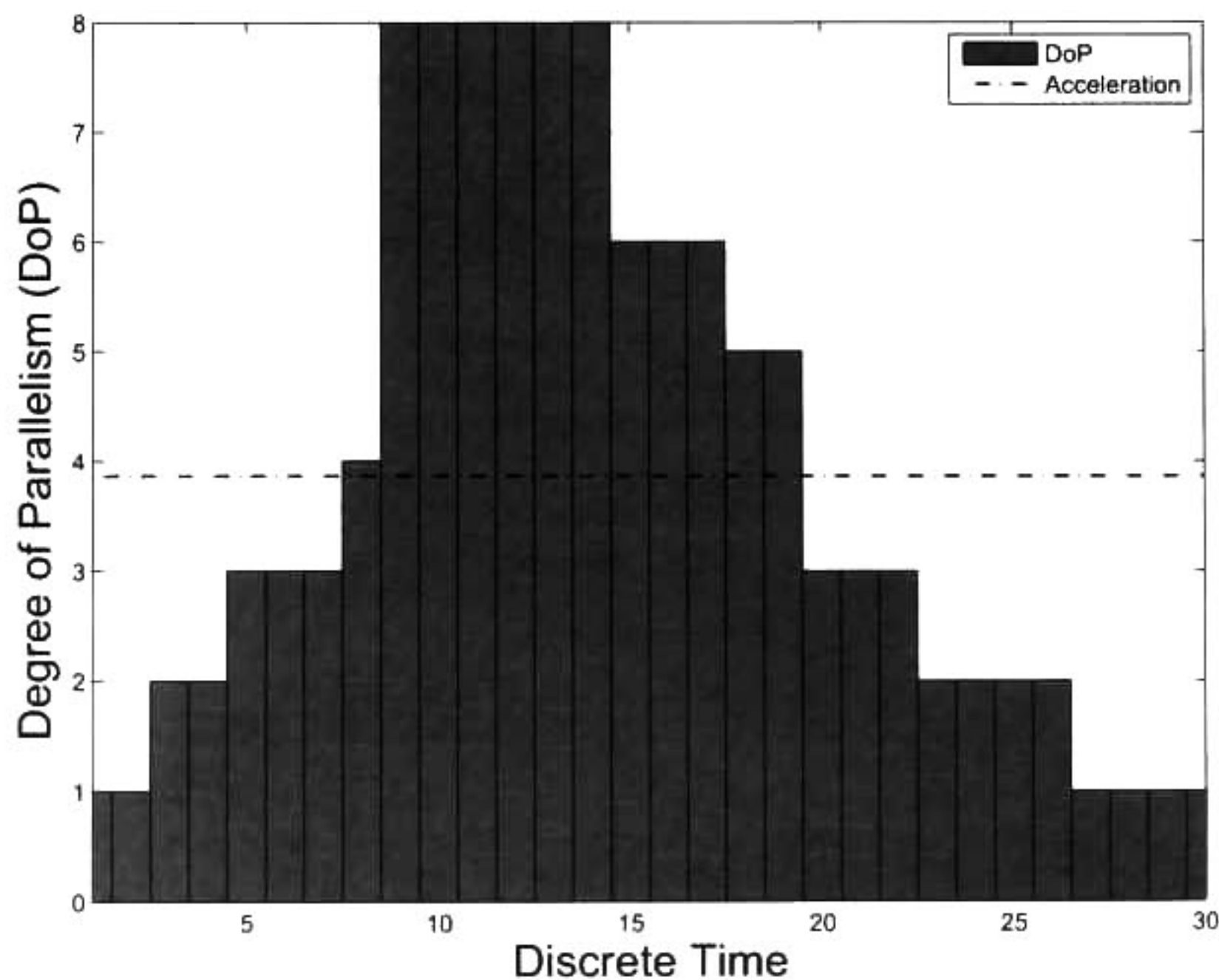


Figure 6.8: Example of a profile of parallelism.

6.5.2 Efficiency

The *relative efficiency*, or *utilization percentage*, is a measure of the fraction of time for which a PE is usefully employed, *i.e.* it measures the cost-effectiveness of the computations. Formally, it is defined as the ratio of relative acceleration to the number of PEs. Only an ideal parallel system of p PEs can deliver an efficiency equal to one. However, in practice the efficiency values are between zero and one. Mathematically, relative efficiency is given by:

$$E_p(N) = \frac{T_1(N)}{pT_p(N)} \quad (6.5)$$

6.5.3 Relative Load Imbalance

Load imbalance refers to the overhead caused by poor distribution of the total computational work as a function of the problem size among the PEs, which is denoted as $W(N)$. In an ideal scenario, an evenly work balance for all the processor is desired, *i.e.* all the p processors doing the same amount of $W(N)/p$ work. If there is an i -th processor for which the difference:

$$L_i(N) = W_i(N) - \frac{W(N)}{p}$$

is a non-zero value, then the workload assigned to the processor has an imbalance equal to $L_i(N)$. The processor with the highest imbalance determines the overall overhead of the parallel system. However, by itself, the value of the load imbalance lacks of a meaning, since it depends on the computer architecture and other parallel overheads. In fact, a same load imbalance value in different parallel systems may result in a different overhead. In order to assess the impact of the load imbalance on performance, the *relative load imbalance* metric is introduced as:

$$Lr_p(N) = 1 - \frac{W(N)}{pW_{\max}(N)} \quad (6.6)$$

where $W_{\max}(N) = \max(W_0(N), W_1(N), \dots, W_{p-1}(N))$. Values of $Lr_p(N)$ close to zero denote a small impact on performance, whereas values close to $1 - 1/p$ denote a highly imbalanced workload

distribution [97]. In the context of processor arrays, relative load imbalance can measure the computational work distribution due to the scheduler, and the tile shape (processor array size). When the processor space fits exactly into the tile shape, the relative load imbalance is very close to $1 - 1/p$. Usually, this occurs when the processor space has a rectangular shape. On the other hand, for processor arrays targeted for algorithms with non-rectangular shapes, the workload distribution is usually imbalanced. Finally, the concept of a unitary amount of workload within the processor array context, can be interpreted as an iteration point mapped to a PE.

6.6 Processor Arrays Evaluation

The metrics shown in previous section could be used for evaluating the processor arrays once they have been implemented, or as a pre-evaluation manner in order to explore the design space and providing an idea of the performance that should be expected given a set of design parameters. In general, these metrics evaluate a processor array as a function of the problem size, resulting in a great amount of data if each problem size is evaluated independently. Calculating an average provides an idea of how well a processor array behaves over a range of values when a specific metric is used. Thus, the processor arrays are evaluated for a range of problem sizes, obtaining an average value for each metric. In this sense, the harmonic mean is used for obtaining the average of the acceleration and efficiency metrics, whereas the average for the imbalance is obtained by the geometric mean. The harmonic mean is appropriate for obtaining the average over a set of rates values, while the geometric mean is appropriate for normalized numbers [81]. If there is a zero value within the evaluated range, the arithmetic mean is used.

Before presenting the evaluation metrics results, it is important to mention two possible assumptions that can be made about the sequential PE used as baseline in the relative acceleration and efficiency metrics. The first assumption consists of conceiving the sequential PE working like a constant-cycle processor, where all the PE's operations require the same number of clock cycles to be executed. The second assumption consists of thinking the sequential PE as a multi-cycle processor

whose operations require different clock cycles to be executed, depending on the operational latencies. In the case of MatMul algorithm, the single PE is thought as a constant-cycle due to only one-cycle operations are required. In contrast, in the case of Cholesky algorithm, both assumptions can be made. If a constant-cycle PE is assumed, then the number of clock cycles to complete an operation will be the same as the iteration interval P . On the other hand, if a multi-cycle PE is assumed, the number of clock cycles to complete an operation will differ according to the different operational latencies of the division, square root and MAC operations.

In the following subsections the relative metrics are applied over the processor arrays implemented in section 6.4. Also, these metrics are used for pre-evaluating other possible processor arrays targeted for algorithm like LU, FIR filter, or Back/Forward substitution with different processor array sizes and allocation functions.

6.6.1 Implemented Processor Arrays Evaluation

Table 6.24 shows the evaluation metric values for each processor array of the second implementation set (see table 6.10). The second column of this table specifies the problem size range used for obtaining the average of the metric. The abbreviation C.C. refers to constant-cycle PE used as baseline for calculating each metric. Although there are some arrays that theoretically support a problem size of 683×683 , their corresponding averages are limited to problem size of 500×500 . In general, it should be noted that metrics shown in table 6.24 tend to improve if the problem size ranges are greater than the processor array size, *i.e.* metric values used for evaluating small processor arrays for solving large problem sizes tend to be more stable than evaluations of larger arrays with the selected problem size ranges. In terms of the efficiency the 2×2 and 4×4 array implementations are above of 60% PE utilization percentage, while in the case of the 8×8 arrays, the best efficiency is 68% achieved by *MatMul-Off-8x8-683* implementation. However, the acceleration achieved by this last implementation is 44x with respect of the sequential PE. The relative load imbalance values show that the processor arrays for MatMul are balanced, since they are close to zero.

Implementation Code Name	Problem Size Range	Acceleration C.C.	Efficiency C.C.	Imbalance C.C.
<i>MatMul-Off-2x2-086</i>	[1, ..., 86]	3.40	0.85	0.045
<i>MatMul-Off-2x2-171</i>	[1, ..., 171]	3.63	0.90	0.027
<i>MatMul-Off-2x2-342</i>	[1, ..., 342]	3.78	0.94	0.015
<i>MatMul-Off-2x2-683</i>	[1, ..., 500]	3.84	0.96	0.011
<i>MatMul-Off-4x4-086</i>	[1, ..., 86]	9.69	0.60	0.112
<i>MatMul-Off-4x4-171</i>	[1, ..., 171]	11.73	0.73	0.068
<i>MatMul-Off-4x4-342</i>	[1, ..., 342]	13.33	0.83	0.040
<i>MatMul-Off-4x4-683</i>	[1, ..., 500]	13.99	0.87	0.029
<i>MatMul-Off-8x8-086</i>	[1, ..., 86]	19.97	0.31	0.205
<i>MatMul-Off-8x8-171</i>	[1, ..., 171]	29.16	0.45	0.130
<i>MatMul-Off-8x8-342</i>	[1, ..., 342]	39.02	0.60	0.078
<i>MatMul-Off-8x8-683</i>	[1, ..., 500]	44.02	0.68	0.059

Table 6.24: Evaluation metrics for the second implementation set corresponding to the MatMul processor array. The average of the imbalance is obtained by the arithmetic mean.

Table 6.25 shows the evaluation metric values for each processor array of the fourth implementation set corresponding to Cholesky decomposition algorithm (see table 6.17). The second column of this table specifies the problem size range used for obtaining the average of the metric. The abbreviations C.C. and M.C. refer to the constant-cycle and multi-cycle PEs used as baseline, respectively. Recall that Cholesky processor arrays assume that each PE requires 21 clock cycles to compute an iteration point (due to the latency of division operation). Also, recall that square root and MAC operations require ten and one clock cycles to be computed. In this sense, note that higher acceleration and efficiency values are achieved when the constant-cycle is used as baseline, *i.e.* when the single PE assumes 21 clock cycles to complete a loop iteration regardless of the operation being executed. On the other hand, when the multi-cycle PE is assumed, the acceleration and efficiency values are degraded. Mainly, this is due to the multi-cycle single PE lacks of idle times compared against the constant-cycle PE, which assumes that all operations require 21 clock cycles to be completed. Also, the difference among the metrics using the constant-cycle and multi-cycle PEs indicates that a high percentage of time dedicated to perform the Cholesky operations is employed to

realize the operations with the lower latency like square root and MAC operations. Both PE baselines are useful for exploring different characteristics. If the constant-cycle PE is used as baseline, the metric values can be viewed as a theoretical bound indicating the maximum performance which could be achieved by using the design methodology followed in this research work. Besides by using the constant-cycle PE, performance comparisons to other algorithms targeted as processor arrays can be realized. On the other hand when the multi-cycle PE is used as baseline, the metric values provide a more realistic perspective.

Implementation Code Name	Problem Size Range	Acceleration		Efficiency		Imbalance	
		C.C.	M.C.	C.C.	M.C.	C.C.	M.C.
<i>Chol-Off-2x2-086</i>	[1, ..., 86]	2.84	0.37	0.71	0.09	0.078	0.078
<i>Chol-Off-2x2-171</i>	[1, ..., 171]	3.22	0.30	0.80	0.07	0.042	0.042
<i>Chol-Off-2x2-342</i>	[1, ..., 342]	3.51	0.26	0.87	0.06	0.022	0.022
<i>Chol-Off-2x2-683</i>	[1, ..., 500]	3.63	0.34	0.90	0.06	0.015	0.015
<i>Chol-Off-4x4-086</i>	[1, ..., 86]	6.27	1.00	0.39	0.06	0.203	0.203
<i>Chol-Off-4x4-171</i>	[1, ..., 171]	8.47	0.95	0.52	0.05	0.117	0.117
<i>Chol-Off-4x4-342</i>	[1, ..., 342]	10.67	0.90	0.66	0.05	0.063	0.063
<i>Chol-Off-4x4-683</i>	[1, ..., 500]	11.74	0.87	0.73	0.05	0.044	0.044
<i>Chol-Off-8x8-086</i>	[1, ..., 86]	10.26	2.37	0.16	0.03	0.379	0.379
<i>Chol-Off-8x8-171</i>	[1, ..., 171]	16.41	2.42	0.25	0.03	0.236	0.236
<i>Chol-Off-8x8-342</i>	[1, ..., 342]	24.80	2.67	0.38	0.04	0.135	0.135
<i>Chol-Off-8x8-683</i>	[1, ..., 500]	30.05	2.77	0.46	0.04	0.097	0.097

Table 6.25: Evaluation metrics for the fourth implementation set corresponding to Cholesky processor array.

Using the constant-cycle PE data shown in table 6.25 as reference, note that the accelerations are less than the accelerations achieved by the MatMul array. This is due to the Cholesky iteration space (\mathcal{I}_{Chol}) has fewer index points than the MatMul iteration space (\mathcal{I}_{MatMul}), and consequently a fewer number of sequential steps. Recall that \mathcal{I}_{Chol} and \mathcal{I}_{MatMul} have a non-rectangular and rectangular shape, respectively. Similarly, the efficiencies achieved by Cholesky implementations are less, since when the Cholesky processor space (\mathcal{P}_{Chol}) is being scanned by the tile indexes, some PEs are not activated due to its non-rectangular shape. These two observations suggest that if the constant-cycle

PE is used as baseline, a higher acceleration and a better efficiency should be expected for algorithms with rectangular space than for algorithms with non-rectangular space. In the case of the multi-cycle PE used as baseline, smaller accelerations are achieved compared to the constant-cycle PE case. In fact accelerations above 2x are accomplished for the implementations of 8×8 PEs. This suggests that larger arrays are required in order to achieve an acceleration in the case of Cholesky algorithm. In general, the Cholesky processor array efficiencies are below 10% due to the sequential PE lacks of idle times compared against the PEs inside the processor arrays, and because there are few times when all the PEs are in parallel performing divisions, square root and MAC operations.

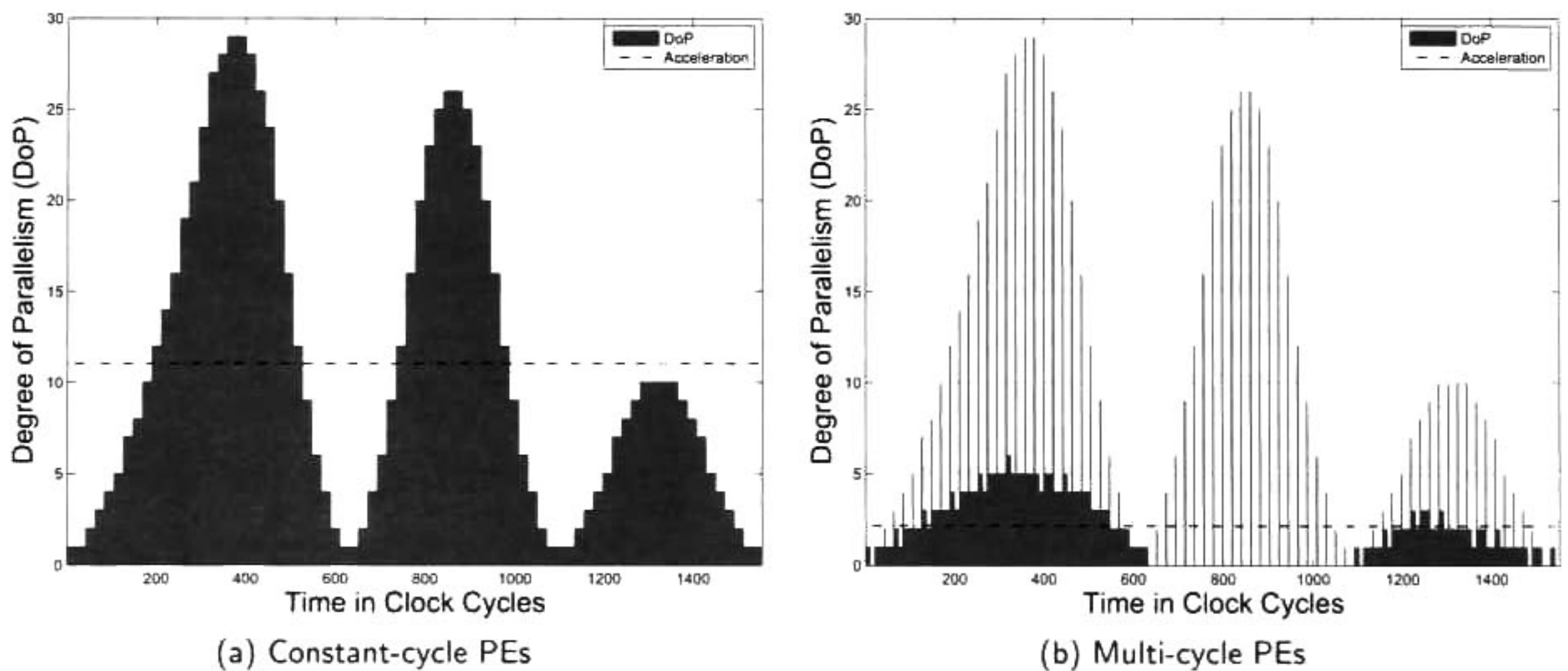


Figure 6.9: Cholesky profiles of parallelism for an 8×8 processor array and using the constant-cycle and multi-cycle PEs as baseline.

For the purpose of visualizing the PEs idle times, figure 6.9 shows the parallelism profiles of two 8×8 processor arrays. Both profiles show the time required for solving a problem size of 16×16 , but each of them assume different operational latencies. In figure 6.9.a, the same 21 clock cycles for all operations are assumed, whereas in figure 6.9.b latencies of 21, 10, and 1 clock cycles for division, square root and MAC operations, respectively, are assumed. For both cases, the acceleration is represented by a dashed-line. Note that both profiles have a similar shape forming three hills. Each hill corresponds to the three different tiles in which is divided \mathcal{P}_{Chol} due to the partitioning. Also,

note that the hills of the right-side profile are sparser than the hills of the left-side since many PEs do not perform an operation at the same time. In fact, in both cases, the maximum DoP is not achieved, *i.e.* there are not time instants where the 64 PEs are used in parallel.

6.6.2 Design Space Exploration

As stated before, the relative metrics (acceleration, efficiency and load imbalance) can be used for pre-evaluating possible processor arrays before their implementation, *i.e.* when the partitioned version of the PRA is obtained. Design parameters like the array sizes, array shapes and different allocations functions could be changed in order to explore the behavior of the processor arrays. This section presents a brief design space exploration for seven algorithms: MatMul, LU and Cholesky decompositions, Back and Forward substitution, FIR filter, matrix-vector multiplication (MatVec). These algorithms can be organized according to the number (dimension) of nested loops. In the case of the two-dimensional algorithms, two different allocation matrixes were used as design parameters, whereas for three-dimensional algorithms six different allocation matrixes were employed. The name of these configurations with their respective allocation functions are shown in tables 6.26 and 6.27. The scheduler function used for space-time mapping is $\vec{\lambda}_l = [1, 1]$ in the case of two-dimensional algorithms; whereas for three-dimensional arrays the scheduler is $\vec{\lambda}_l = [1, 1, 1]$.

Configuration Code Name	Projection Vector	Allocation Matrix
2-Ver-1	$\vec{u} = [1 \ 0]^t$	$\Phi = [0 \ 1]$
2-Ver-2	$\vec{u} = [0 \ 1]^t$	$\Phi = [1 \ 0]$

Table 6.26: Allocation functions used for two-dimensional algorithms.

For each allocation matrix, several possible processor arrays with different number of PEs are evaluated for a range of problem sizes of $N = [0, \dots, 500]$. Recall that the number of PEs in a processor array is determined by the strip size parameters (SSp_0 and SSp_1). In the case of two-

Configuration Code Name	Projection Vector	Allocation Matrix
<i>3-Ver-1</i>	$\vec{u} = [1 \ 0 \ 0]^t$	$\Phi = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
<i>3-Ver-2</i>	$\vec{u} = [1 \ 0 \ 0]^t$	$\Phi = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$
<i>3-Ver-3</i>	$\vec{u} = [0 \ 1 \ 0]^t$	$\Phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
<i>3-Ver-4</i>	$\vec{u} = [0 \ 1 \ 0]^t$	$\Phi = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$
<i>3-Ver-5</i>	$\vec{u} = [0 \ 0 \ 1]^t$	$\Phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$
<i>3-Ver-6</i>	$\vec{u} = [0 \ 0 \ 1]^t$	$\Phi = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

Table 6.27: Allocation functions used for three-dimensional algorithms.

dimensional algorithms, one-dimensional arrays are derived; whereas for three-dimensional algorithms, two-dimensional arrays are obtained. In both cases, the strip size parameters are limited to multiples of power of two, with a maximum of $SSp_0 = 128$ for one-dimensional arrays, and $SSp_0 = SSp_1 = 32$ for two-dimensional arrays. For example, in the LU case, two-dimensional processor arrays with 16 PEs can be derived from arrays with 1×16 , 2×8 , 4×4 , 8×2 and 16×1 PEs; therefore the shape of the processor array is also evaluated. Finally, since Back and Forward substitution have the same loop kernel, iteration space, and operations, their results are identical; thus they are shown in the same set named Back/Forward. Similarly, the loop kernel, iteration space and operations are the same for the MatVec and the FIR filter, and consequently their results are referred as the MatVec/FIR.

6.6.2.1 One-Dimensional Arrays

In the case of one-dimensional arrays, there are 16 possible arrays which can be evaluated with the three different metrics. For the sake of simplicity, this subsection is restricted to discuss only the main aspects involved during the design space exploration of these linear arrays. However, in Appendix A the tables containing the complete evaluation of the two-dimensional algorithms can be found, including metric evaluations using the sequential constant-cycle PE and multi-cycle PE as baseline.

Processors	2-Ver-1	2-Ver-2
001 PE	1.0000	1.0000
002 PE	1.9635	1.9635
004 PE	3.7875	3.7875
008 PE	7.0648	7.0648
016 PE	12.4155	12.4155
032 PE	19.8674	19.8674
064 PE	28.1404	28.1404
128 PE	35.1463	35.1463

(a) MatVec/FIR

Processors	2-Ver-1	2-Ver-2
001 PE	1.0000	1.0000
002 PE	1.9179	1.9354
004 PE	3.5493	3.6325
008 PE	6.1884	6.4528
016 PE	9.8729	10.4927
032 PE	14.0825	15.1413
064 PE	17.9231	19.2190
128 PE	20.7707	21.9027

(b) Back/Foward

Table 6.28: Acceleration for MatVec/FIR and Back/Foward processor arrays using the constant-cycle PE as baseline.

In tables 6.28.a and 6.28.b the relative accelerations for the MatVec/FIR and Back/Foward processor arrays are presented. In both cases the constant-cycle PE is used as baseline. Note that MatVec/FIR processor arrays have a better acceleration than Back/Foward arrays. Mainly, this is due to given the same problem size and same the scheduler function, the four algorithms complete their computations in the same number of parallel steps. However, the number of sequential steps required by MatVec/FIR algorithms is greater than Back/Foward algorithms sequential steps. Consequently, a better acceleration is achieved by MatVec/FIR processor arrays.

Also, note that for MatVec/FIR processor arrays the acceleration achieved is the same regardless of the allocation matrix used; unlike Back/Foward processor arrays where different acceleration

values are obtained from different allocation matrixes. Finally, similar behaviors are obtained when the MatVec/FIR and Back/Foward processor arrays are evaluated using the relative efficiency and relative load imbalance metrics (see Appendix A).

6.6.2.2 Two-Dimensional Arrays

In the case of two-dimensional arrays, there are 216 possible arrays which can be evaluated with the three different metrics for each of three different algorithms (MatMul, Cholesky and LU). Again, for the sake of simplicity, this subsection is also restricted to discuss only the main aspects involved during the design space exploration. In this sense, processor arrays of 32 and 64 PEs are used in order to exemplify the impact of changing the allocation function, and processor array shape. However, in Appendix A the tables containing the complete evaluation of these algorithms can be found.

Table 6.29 shows the efficiency of several MatMul processor arrays derived by the six different allocation matrixes. The efficiency achieved by these arrays is greater than 65%, and in some cases close to 80%. Note that given a processor array of $SSp_0 \times SSp_1$ PEs, the efficiency is the same regardless of the allocation function. Mainly, this is because the MatMul iteration space is rectangular, leading to a uniform distribution of the index points to the PEs. Also, it should be noted that in the case of 64 PEs, the square processor array (8×8 PEs) has a better efficiency than some other non-square arrays. Particularly, the 8×8 array has a better efficiency than 1×32 and 32×1 arrays. This tendency is also observed with other MatMul processor arrays (see Appendix A).

Figure 6.10 shows the efficiency as a function of the problem size N for three different MatMul processor arrays derived by using the configuration 3-Ver-2. The harmonic mean of each line is already shown in table 6.29. Note that these graphs have a sawtooth-like form. Recall that by partitioning the processor space, several tiles containing the processor index points are created, and

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
32 PEs	01 × 32	0.6209	0.6209	0.6209	0.6209	0.6209	0.6209
	02 × 16	0.7343	0.7343	0.7343	0.7343	0.7343	0.7343
	04 × 08	0.7906	0.7906	0.7906	0.7906	0.7906	0.7906
	08 × 04	0.7906	0.7906	0.7906	0.7906	0.7906	0.7906
	16 × 02	0.7343	0.7343	0.7343	0.7343	0.7343	0.7343
	32 × 01	0.6209	0.6209	0.6209	0.6209	0.6209	0.6209
64 PEs	02 × 32	0.5728	0.5728	0.5728	0.5728	0.5728	0.5728
	04 × 16	0.6598	0.6598	0.6598	0.6598	0.6598	0.6598
	08 × 08	0.6878	0.6878	0.6878	0.6878	0.6878	0.6878
	16 × 04	0.6598	0.6598	0.6598	0.6598	0.6598	0.6598
	32 × 02	0.5728	0.5728	0.5728	0.5728	0.5728	0.5728

Table 6.29: Average relative efficiency for several MatMul processor arrays and different allocation matrixes, when $N_{max} = 500$. The processor arrays are organized according to the number of PEs.

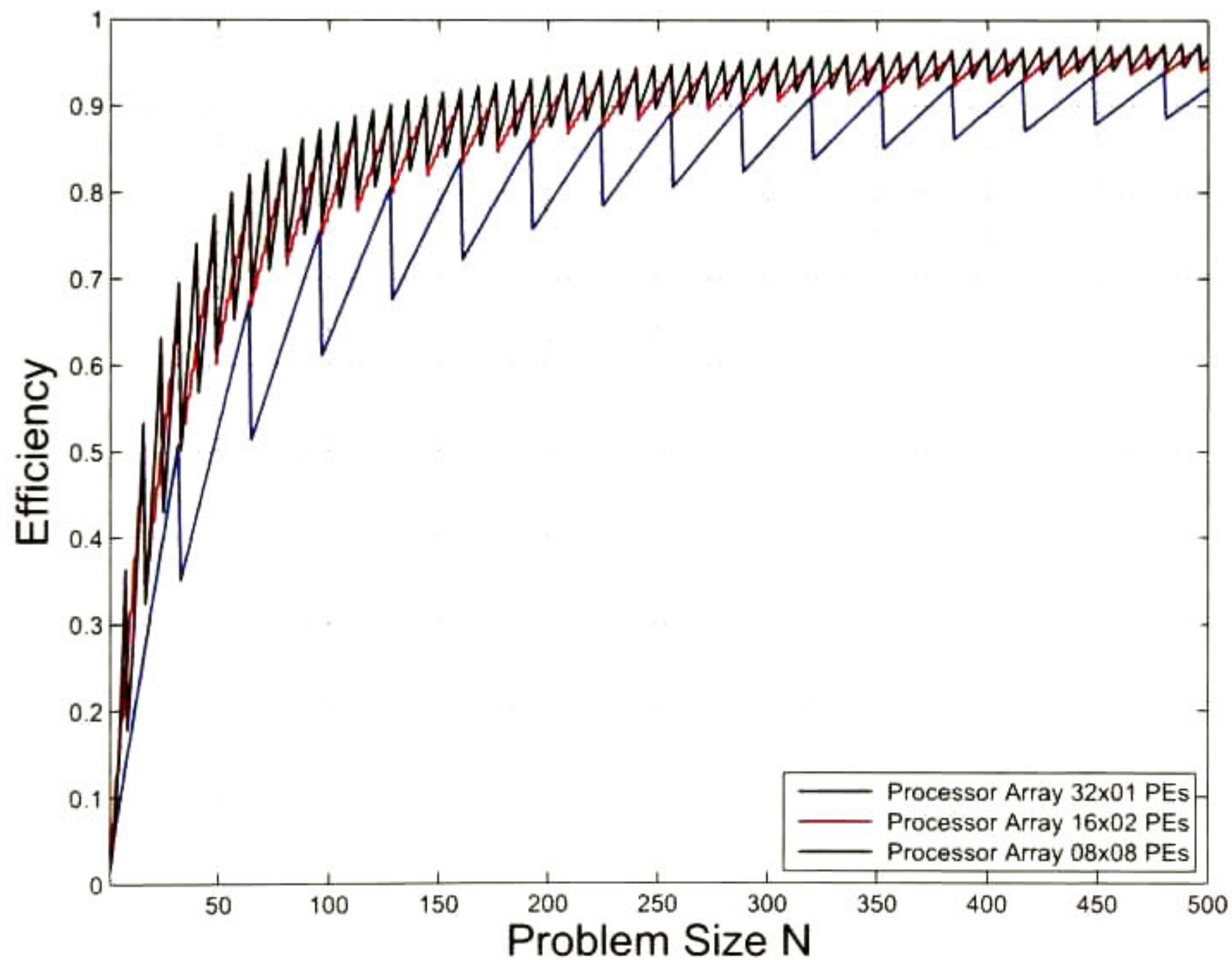


Figure 6.10: Efficiency of three MatMul processor arrays.

some of these tiles do not fit exactly into the physical array. Consequently, some PEs in the physical array remain inactive when a tile is being scanned (see figure 4.13), more precisely when $N \bmod SSp_x \neq 0$. In this sense, the graphic's cliffs represent when given a problem size all the tiles fit exactly into the processor space, *i.e.* when $N \bmod SSp_x = 0$.

From figure 6.10 note that although the 8×8 array has a greater number of PEs than the 16×2 array, both arrays have a similar efficiency. In general, a better efficiency is achieved for processor arrays whose form is close to a square shape. This suggest that square processor arrays have a better work distribution than non-square arrays. Table 6.30 shows the relative load imbalance for the same processor arrays shown in table 6.29. Here, note that lower relative load imbalance are achieved for processor arrays with a closer square shape. Like in table 6.29, changing the allocation function does not change the relative load imbalance given a processor array.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
32 PEs	01 \times 32	0.1045	0.1045	0.1045	0.1045	0.1045	0.1045
	02 \times 16	0.0644	0.0644	0.0644	0.0644	0.0644	0.0644
	04 \times 08	0.0453	0.0453	0.0453	0.0453	0.0453	0.0453
	08 \times 04	0.0453	0.0453	0.0453	0.0453	0.0453	0.0453
	16 \times 02	0.0644	0.0644	0.0644	0.0644	0.0644	0.0644
	32 \times 01	0.1045	0.1045	0.1045	0.1045	0.1045	0.1045
64 PEs	02 \times 32	0.1078	0.1078	0.1078	0.1078	0.1078	0.1078
	04 \times 16	0.0715	0.0715	0.0715	0.0715	0.0715	0.0715
	08 \times 08	0.0590	0.0590	0.0590	0.0590	0.0590	0.0590
	16 \times 04	0.0715	0.0715	0.0715	0.0715	0.0715	0.0715
	32 \times 02	0.1078	0.1078	0.1078	0.1078	0.1078	0.1078

Table 6.30: Average relative load imbalance for several MatMul processor arrays and different allocation matrixes, when $N_{max} = 500$. The processor arrays are organized according to the number of PEs.

In general, when using the relative load imbalance to measure the processor array work distribution, zero values could be obtained, *i.e.* the processor arrays are perfectly balanced for a certain problem sizes. Figure 6.11 depicts the relative work distribution for an 8×8 MatMul

processor array for a range of problem sizes of $[1, \dots, 250]$. Again, note that the graphic has a sawtooth-like form. Also, note that when $N \bmod 8 = 0$ the relative load imbalance is zero, since all the loop iterations are mapped evenly to the PEs in the physical array.

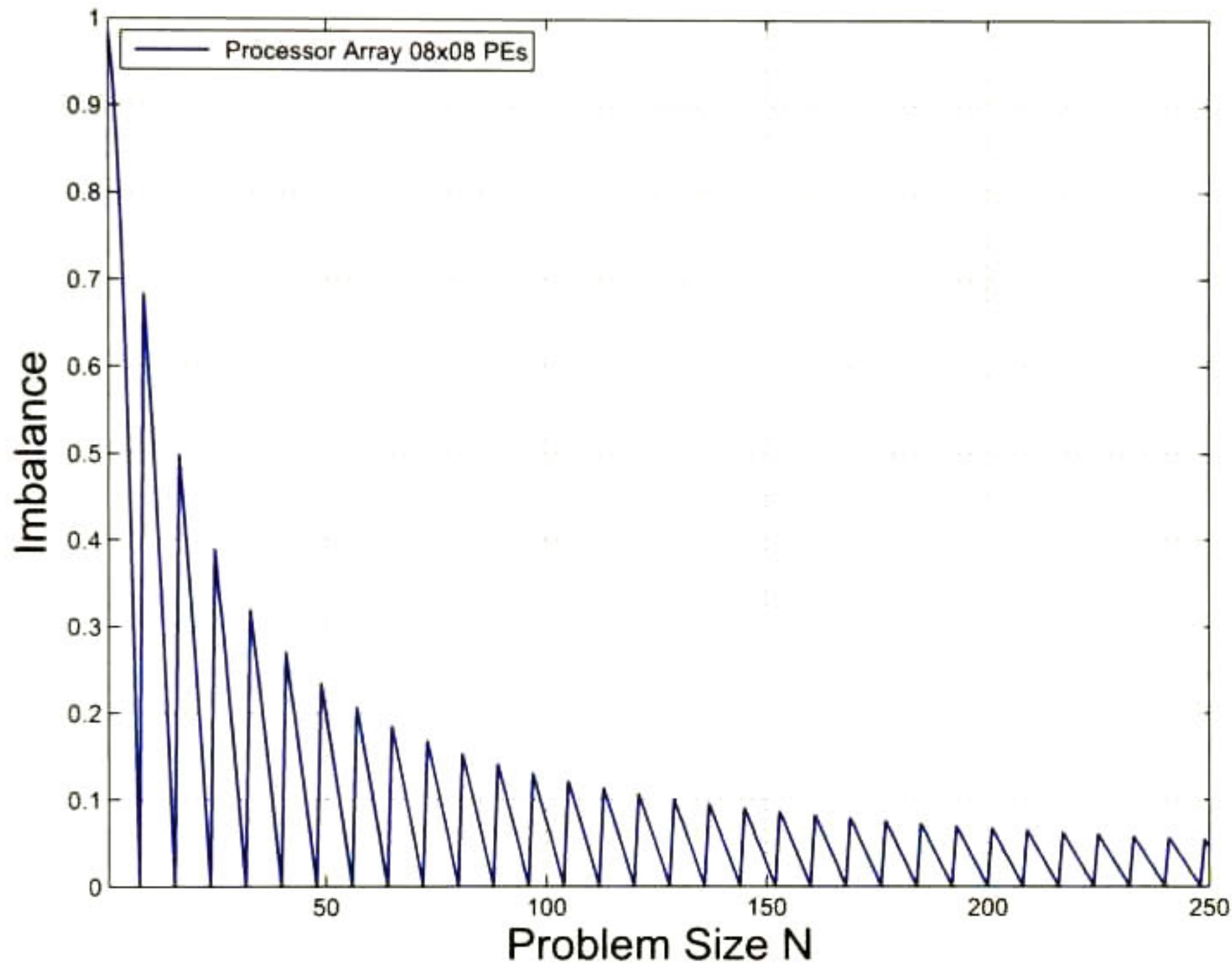


Figure 6.11: Relative load imbalance of an 8×8 MatMul processor array.

Table 6.31 shows the efficiency of several LU processor arrays changing the allocation function. These results use a sequential constant-cycle PE as baseline. Contrary to the MaMul arrays, in the case of LU given an array of size $SSp_0 \times SSp_1$, its efficiency changes if the allocation matrix is changed too. In fact, note that the efficiency results can be paired for the a same processor array. For example, taking the 4×8 array as reference, the configurations *3-Ver-3* and *3-Ver-5* have the same efficiency value; despite that both functions come from different projection vectors. Similarly, the efficiency for configurations *3-Ver-4* and *3-Ver-6* is the same. Mainly, this is a consequence of the LU target polytope iteration space shape after space-time mapping, due to depending on the projection vector \vec{u} , the LU processor space (\mathcal{P}_{LU}) could be rectangular or non-rectangular. In the case of

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
32 PEs	01 × 32	0.4181	0.4181	0.5124	0.4595	0.5124	0.4595
	02 × 16	0.5476	0.5476	0.6284	0.5960	0.6284	0.5960
	04 × 08	0.6269	0.6269	0.6923	0.6826	0.6923	0.6826
	08 × 04	0.6269	0.6269	0.6826	0.6923	0.6826	0.6923
	16 × 02	0.5476	0.5476	0.5960	0.6284	0.5960	0.6284
	32 × 01	0.4181	0.4181	0.4595	0.5124	0.4595	0.5124
64 PEs	02 × 32	0.3769	0.3769	0.4568	0.4125	0.4568	0.4125
	04 × 16	0.4724	0.4724	0.5386	0.5167	0.5386	0.5167
	08 × 08	0.5082	0.5082	0.5679	0.5679	0.5679	0.5679
	16 × 04	0.4724	0.4724	0.5167	0.5386	0.5167	0.5386
	32 × 02	0.3769	0.3769	0.4125	0.4568	0.4125	0.4568

Table 6.31: Average relative efficiency for several LU processor arrays and different allocation matrixes, when $N_{max} = 500$. The processor arrays are organized according to the number of PEs.

$\vec{u} = [1, 0, 0]^t$, the resulting processor space after space-time mapping has a rectangular shape. On the other hand, for projection vectors $\vec{u} = [0, 1, 0]^t$ and $\vec{u} = [0, 0, 1]^t$ the processor spaces obtained are non-rectangular. The unique difference on $\vec{u} = [0, 1, 0]^t$ and $\vec{u} = [0, 0, 1]^t$ is that by selecting the allocation function, the same index of the source polytope is mapped to a different index of \mathcal{P}_{LU} . In this last case, the outer most index of LU algorithm is mapped to the same processor index. Furthermore, note that in the case of the 8×4 array, the efficiency for configurations *3-Ver-4* and *3-Ver-6* is the same as the efficiency for configurations *3-Ver-3* and *3-Ver-5* of the 4×8 processor arrays. Again, this is a consequence of the space-time mapping, since the outer loop index is mapped to the same processor index, which is later partitioned with the same value of SSp_x .

Figures 6.12.a and 6.12.b show the relative efficiency and relative load imbalance, respectively, for three different processor arrays of 4×8 PEs derived by three different configurations. Both graphics show a range of problem sizes of $[1, \dots, 250]$. Although the processor array derived from configuration *3-Ver-1* (red color line) has a lower efficiency than the arrays derived from configurations *3-Ver-3* and *3-Ver-4*, it tends to have lower relative load imbalance, *i.e.* there is a more balanced work distribution for *3-Ver-1* than for configurations *3-Ver-3* and *3-Ver-4*. Furthermore, note that the relative load

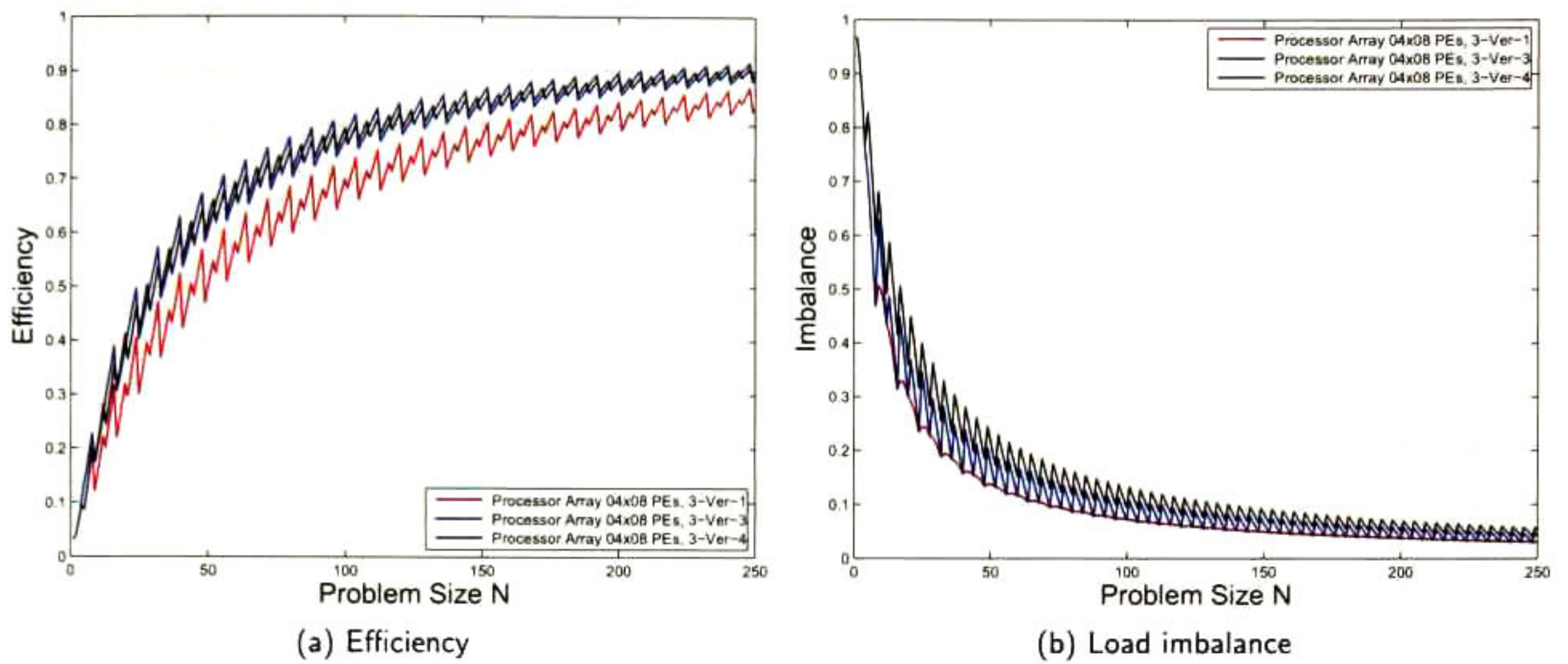


Figure 6.12: Relative efficiency and relative load imbalance of three different processor array derived by different allocation functions.

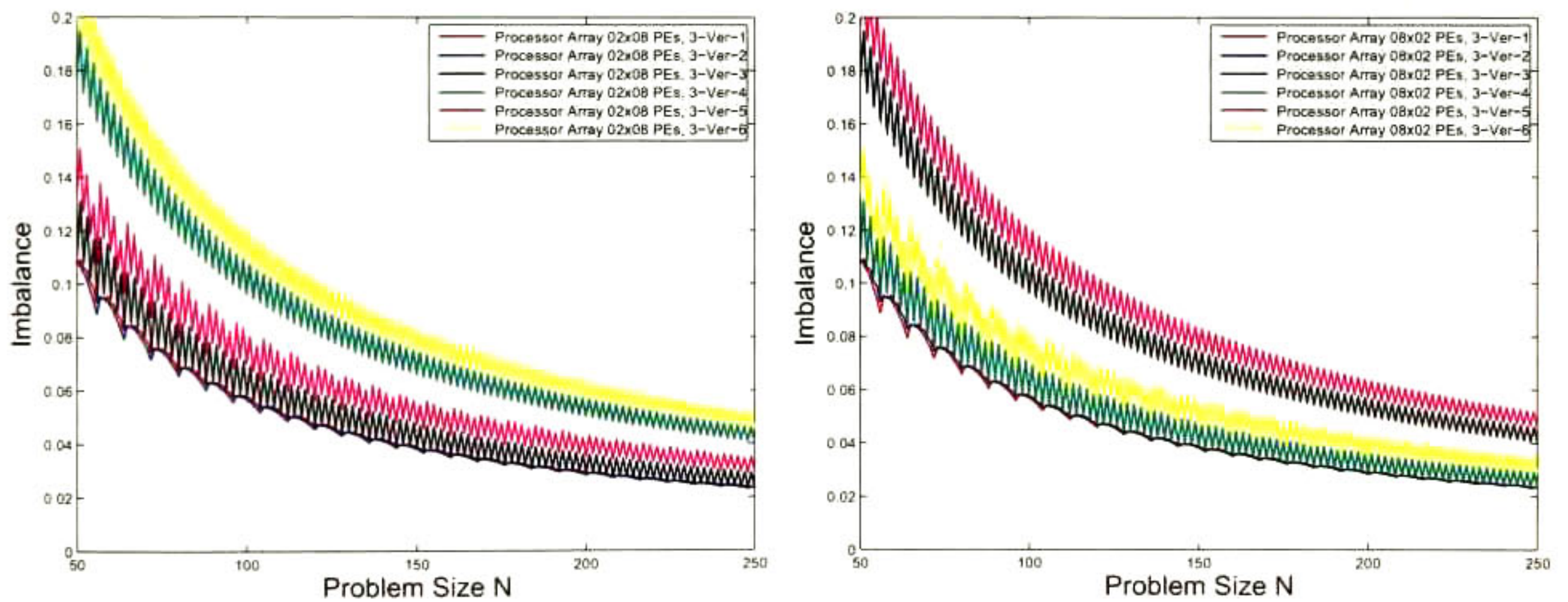


Figure 6.13: Relative load imbalance of different processor array of size 2×8 and 8×2 derived by different allocation functions.

imbalance of the array obtained from configuration *3-Ver-1* does not have abrupt changes when the problem size is increased compared to configurations *3-Ver-3* and *3-Ver-4*. This suggests, that by using projection vector $\vec{u} = [1, 0, 0]^t$ LU processor arrays with a low load imbalance are obtained.

Figure 6.13 shows the relative load imbalance of six LU processor arrays of 2×8 PEs and 8×2 PEs obtained with the six different allocation functions. For purpose of clarity, the problem size range is limited to $[50, \dots, 250]$. Again, the allocation function $\vec{u} = [1, 0, 0]^t$ leads to LU processor arrays with the lower load imbalance. Note that by changing the processor array shape, the load imbalance values are interchanged for the same projection vector. For example the load imbalance of array 2×8 derived by configuration *3-Ver-3* is the same as the load imbalance of the array 8×2 derived by configuration *3-Ver-4*.

Finally, as stated before, using the relative metrics prior to the processor array implementation is helpful for the design space exploration. Additionally, if the number of hardware resources required by each configuration is taken into account, a design decision considering the number of hardware elements and the projected performance (using the acceleration, efficiency and load imbalance) could be taken. Table 6.32 shows this big picture for an 8×8 Cholesky processor array. In this table, the implementation aspects required by each one of the three main components (data-path, the memory and control) of a processor array system are shown in terms of the number of hardware elements required by each main component. Also, the evaluation metrics using the constant-cycle PE are shown. In general, among the six configurations, the number of hardware elements required by the control and memory are very similar. However, the number of dividers required by configurations *3-Ver-5* and *3-Ver-6* is 8x less than the number of dividers required by the other configurations. This indicates that using the allocation matrixes for *3-Ver-5* and *3-Ver-6* configurations result in a relatively low hardware cost. The selection between these two configurations could be done by choosing any of these metrics as decision criterion. In this sense, since acceleration and efficiency are the same, the relative load imbalance could be used as the decision criterion.

		3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
Data-Path	Adders	64	64	64	64	64	64
	Multipliers	64	64	64	64	64	64
	Dividers	64	64	64	64	8	8
	Square Roots	8	8	8	8	8	8
Control	Adders	153	150	147	149	148	157
	Multipliers	2	2	2	2	2	2
	Multiplexers	9	9	7	8	10	9
	Counters	132	132	132	132	132	132
	Registers	263	263	263	263	263	263
	Shifters	7	4	8	7	6	4
Memory	Adders	96	96	96	96	96	96
	Multipliers	32	32	32	32	32	32
	Multiplexers	68	68	72	72	68	68
	Registers	68	68	72	72	68	68
	Input Type	Broadcast	Broadcast	Border	Border	Border	Border
	Output Type	Border	Border	Border	Border	Broadcast	Broadcast
Metrics	Acceleration	28.47	28.47	28.885	28.885	30.056	30.056
	Efficiency	44.49%	44.49%	45.13%	45.13%	46.96%	46.96%
	Load Imbalance	0.119	0.079	0.119	0.119	0.079	0.119

Table 6.32: Information for the implementation of an 8×8 Cholesky processor array gathering the metrics and the number of hardware elements required by the data-path, memory and control.

6.7 Summary

In this chapter, the PAR results concerning the hybrid control scheme, the memory architectural cases and several processor arrays implemented as a complete system were presented. The control scheme was implemented for supporting MatMul and Cholesky decomposition algorithms for different schedulers, allocation and iteration intervals. Mainly these two algorithms were selected with the purpose of showing that the hybrid control scheme is able to support rectangular and non-rectangular iteration spaces. Also, the PAR control results show that maximum operational frequencies do not decrease significantly when the processor array is increased; however changing the control word W_c has a major impact on the frequency degradation compared when the processor array is increased. Although the proposed control scheme requires more FPGA resources than controllers generated

by PARO and MMAAlpha, the proposed solution is able to generate the control signals for several problem sizes, without the need of having several controllers for each possible problem size.

The four memory architectural cases were implemented for supporting the MatMul and Cholesky decomposition algorithms. Their PAR results show that the assumption made for the two clock domains is achievable, since the processor arrays clock frequencies are at least three times slower than the worst case of any of the four architectural memory implementations. Besides, the theoretical peak bandwidth obtained by the four architectural cases exceed the processor array I/O bandwidth requirements, due to the use of the multiple communications channels. Furthermore, comparison against a first attempt for implementing a memory system for MatMul processor array shows that the proposed memory architectural cases deliver a large amount of data per clock cycle.

The PAR results of several processor arrays integrating the processor array data-path, the hybrid controller, and the memory system for MatMul and Cholesky algorithms were presented too. In general, implementation results of both algorithms indicate that one technological limitation for implementing a complete integrated processor arrays system into FPGAs is the number of BRAMs available. However, if the memory banks required for storing the input and output matrixes are placed as off-chip memories, solving larger problem sizes becomes possible with the current FPGA technology, at the price of dedicating more silicon resources to storing data than to computing data. Although in the case of two proposed MatMul processor arrays more FPGA resources are required compared to the processor arrays derived by PARO, the proposed MatMul arrays are able to solve a set of problem sizes without the need of generating several arrays for each problem size. In fact, for multiplying two matrixes of 100×100 , the arrays derived in this research are 5,000 clock cycles faster than the array derived with the PARO methodology. In addition, this research work is one of the first attempts for generating processor arrays for algorithms with non-rectangular iteration spaces like the Cholesky decomposition algorithm, focusing on having processor arrays able to solve several problem sizes.

Besides, for purpose of comparison against a soft-processor implementation, the MatMul and Cholesky processor arrays were implemented in an embedded platform. The results of these implementations show that an acceleration is achieved for both algorithms with a low power consumption. In fact, processor arrays use more efficiently the FPGA LUTs, and deliver a major number of bits per power unit than the soft-processor implementation.

Finally, by using three different metrics, the processor arrays implemented were evaluated in terms of their relative acceleration, relative efficiency and relative load imbalance. In general, a better efficiency (values close to one) are obtained for processor arrays which solve wider range of problem sizes. For algorithms with rectangular iteration spaces, a major acceleration is achieved than in the case of algorithms with non-rectangular iteration spaces. Besides, for processor arrays with a square shape, the work distribution is more balanced than in the case of non-square processor arrays. Also, brief design space exploration was presented.

7

Conclusions and Future Directions

This dissertation has reviewed the process of generating processor arrays, focusing on the generation of a control scheme for algorithms with non-rectangular iteration spaces, and the generation of memory interfaces for inserting/extracting data to/from processor arrays. In Chapter 1 the hypothesis, objectives, research questions and contributions of this dissertation were presented. The origins and an overview of the automatic synthesis tools within the polytope framework were presented in Chapter 2. Chapter 3 presented the mathematical background and concepts required for the generation of processor arrays. An hybrid control scheme able to generate the activation sequence for several problem sizes and able to deal with non-rectangular iteration spaces is presented in Chapter 4. The internal memory for intermediate storage, and external memory interface based on four architectural cases is presented in Chapter 5. The PAR results for the control scheme and memory architectural cases are presented in Chapter 6. Also, Chapter 6 included the PAR results for the integration of two complete processor arrays (including data-path, control and memory) targeted for two different FPGA families. The evaluation of these arrays using traditional metrics employed in parallel computing was presented in Chapter 6 too. Finally, the conclusions and future work derived from this research are presented in this last chapter.

7.1 Conclusions

In this section, the conclusions, the list of contributions, and the list of research questions with their corresponding answers are presented. Mainly, the conclusions derived from this research work can be divided into three different parts: control scheme, external memory system, and processor array systems.

Control Scheme: The control scheme is able to generate the control signals for processor arrays derived from loop-based algorithms with rectangular and non-rectangular iteration spaces by using the MatMul and Cholesky decomposition algorithms as cases of study. Also, the control scheme supports different iteration intervals, schedulers and allocations vectors, thus different space-time mappings can be supported. Besides, it was shown that given a scheduler vector, the control scheme is able to generate the control signals for a set of problem sizes no larger than an N_{max} value, unlike previous works where the controllers are able to generate the control signals for a unique problem size. The use of distributed control cells allow the control signal generation for non-rectangular iteration spaces, and for a set of problem sizes since each cell has the necessary information for activating its neighbors according to valid mapping from the logical array to the physical array. Furthermore, FPGA's PAR results show that the impact of increasing the size of the control array on the operational frequencies is minor when the control word width is increased and that the FPGA resources scale at a similar factor to the rate from which the control array size is changed.

External Memory System: The memory architectural cases enclose the four different types of I/O that can be derived after transforming a piecewise regular algorithm. Each one of the memory cases has a modular architecture allowing to add easily modules in order to provide support larger processor arrays. These four architectural cases were implemented and validated for two different algorithms with different I/O data requirements. Besides, these architectural cases are based on the use of two clock domains, and based on the use dual-port memories. Similarly to the control scheme, the four memory cases support different space-time mappings due to combinational logic is in charge

7

Conclusions and Future Directions

This dissertation has reviewed the process of generating processor arrays, focusing on the generation of a control scheme for algorithms with non-rectangular iteration spaces, and the generation of memory interfaces for inserting/extracting data to/from processor arrays. In Chapter 1 the hypothesis, objectives, research questions and contributions of this dissertation were presented. The origins and an overview of the automatic synthesis tools within the polytope framework were presented in Chapter 2. Chapter 3 presented the mathematical background and concepts required for the generation of processor arrays. An hybrid control scheme able to generate the activation sequence for several problem sizes and able to deal with non-rectangular iteration spaces is presented in Chapter 4. The internal memory for intermediate storage, and external memory interface based on four architectural cases is presented in Chapter 5. The PAR results for the control scheme and memory architectural cases are presented in Chapter 6. Also, Chapter 6 included the PAR results for the integration of two complete processor arrays (including data-path, control and memory) targeted for two different FPGA families. The evaluation of these arrays using traditional metrics employed in parallel computing was presented in Chapter 6 too. Finally, the conclusions and future work derived from this research are presented in this last chapter.

7.1 Conclusions

In this section, the conclusions, the list of contributions, and the list of research questions with their corresponding answers are presented. Mainly, the conclusions derived from this research work can be divided into three different parts: control scheme, external memory system, and processor array systems.

Control Scheme: The control scheme is able to generate the control signals for processor arrays derived from loop-based algorithms with rectangular and non-rectangular iteration spaces by using the MatMul and Cholesky decomposition algorithms as cases of study. Also, the control scheme supports different iteration intervals, schedulers and allocations vectors, thus different space-time mappings can be supported. Besides, it was shown that given a scheduler vector, the control scheme is able to generate the control signals for a set of problem sizes no larger than an N_{max} value, unlike previous works where the controllers are able to generate the control signals for a unique problem size. The use of distributed control cells allow the control signal generation for non-rectangular iteration spaces, and for a set of problem sizes since each cell has the necessary information for activating its neighbors according to valid mapping from the logical array to the physical array. Furthermore, FPGA's PAR results show that the impact of increasing the size of the control array on the operational frequencies is minor when the control word width is increased and that the FPGA resources scale at a similar factor to the rate from which the control array size is changed.

External Memory System: The memory architectural cases enclose the four different types of I/O that can be derived after transforming a piecewise regular algorithm. Each one of the memory cases has a modular architecture allowing to add easily modules in order to provide support larger processor arrays. These four architectural cases were implemented and validated for two different algorithms with different I/O data requirements. Besides, these architectural cases are based on the use of two clock domains, and based on the use dual-port memories. Similarly to the control scheme, the four memory cases support different space-time mappings due to combinational logic is in charge

of calculating the memory banks addresses, and such calculation takes into account the indexes after space-time transformation. Mainly, the FPGA's PAR results shows that the assumption of using two clock domains (one twice faster than the other domain) is technologically achievable. Also, the PAR results show that the number of LUTs required by the broadcast cases are major than the number of LUTs required by the border cases.

Processor Array: MatMul and Cholesky decomposition algorithms were used in order to generate different integrated processor array systems, which include control scheme, external memory system and data-path. Since these processor arrays have the advantages of the control scheme and the external memory system, they are able to provide support to algorithms with non-rectangular iteration spaces, and the support for solving a set of problem sizes depending on the memory available on the implementation platform. Besides, due to mathematical expressions obtained after space-time transformations are mapped to combinational logic, which are included into the control scheme and external memory system, the processor arrays are able to support different schedulers, allocators and iteration intervals. Several processor arrays were evaluated using three different metrics and focusing on changing the processor array size. When evaluating the processor arrays a major acceleration, and a better efficiency (values close to one) should be expected for algorithms whose operations have lower latency (clock cycles to be computed) and for algorithms with a major number of index points (sequential steps). This is the case of the MatMul processor arrays where an acceleration is obtained even for smaller arrays. On the other hand, for algorithms with a high operation latency (as Cholesky) an acceleration should be expected only for larger processor arrays. Besides, when different processor array shapes targeted for a same algorithm are analyzed, better acceleration, efficiency and load imbalance values should be expected for square arrays than for non-square processor arrays. FPGA PAR results indicate that the amount of memory (number of FPGA BRAMs) is a current technological limitation for implementing complete integrated processor array systems into FPGAs. Such limitation could be reduced if the memory banks are assumed to be off-chip memories or if the data word width is reduced. However, solving larger problem sizes comes at the price of dedicating more FPGA silicon to store data than to compute.

7.1.1 Contributions

The list of contributions in this research work and a brief description of them is enumerated:

1. **A novel and general architectural framework able to support space-time transformations in the polytope model, consisting on a control scheme, a memory hierarchy and a processor array data-path.** The architectural framework has been implemented and validated by integrating two processor array systems for two different cases of study, including the processor array data-path, the control scheme and memory system. The implementations of these two cases of study support different iteration spaces, different schedulers and allocation functions (space-time mapping), and different iteration intervals.
2. **A general control scheme able to perform the selection of operations inside the PE and the activation of PEs inside the processor array regardless of the iteration space shape.** A control scheme has been validated for two different algorithms, and for different space-time transformations. Also, this control scheme is able to generate the activation signals for algorithms with non-rectangular iteration spaces and able to deal with several problem sizes.
3. **A novel external memory system able to perform the data feeding to the processor array and the data extraction from the processor array.** Four memory architectural cases for feeding data to the processor array from external memory, and storing the data results produced by the processor array have been implemented and validated.
4. **A set of software tools that helps in the transformation of sequential loop algorithms, used in complex digital signal processing systems, into a processor array representation.** The software tools for obtaining the linear and affine schedulers, and for performing the space-time transformations were implemented in order to obtain a parallel version of the piecewise regular algorithm. From this parallel version, it is possible to derive the loop bounds mapped to the control scheme, to select the architectural cases, and to construct the processor array data-path.

7.1.2 Revisiting the Research Questions

The list of the research questions presented in Chapter 1 is enumerated with their respective answers:

1. **What are the advantages of using the polytope model for the generation of processor arrays?** The polytope model provides an abstraction for modeling the computations of loop-based algorithms. Such abstraction facilitates the data dependence analysis of the sequential algorithms, and it allows the parallelization of the loop computations. By obtaining a parallelized version of the loop-based algorithm, it is possible to derive the control structures required for activating the PEs, and to derive the memory interfaces for inserting/extracting data to/from the processor array. Furthermore, by using the scheduler and allocation functions, which are required prior to parallelize the loop algorithm, the processor array data-path (PE internal data-path and array interconnection topology) can be synthesized. Also, the use of the polytope for modeling the loop-based computations allows the development of advanced synthesis software based on software libraries required for extracting the data dependences, for solving linear programming formulations, and for performing the space-time transformation.
2. **What control schemes are needed in order to support non-rectangular iteration spaces and to provide problem size independency?** Control schemes combining centralized and distributed facilities are required for supporting non-rectangular iteration spaces and for providing problem size independency. In both cases, the most expensive and repetitive calculations are placed into central modules in order to avoid the overhead of placing them into the distributed modules. However, each one of the distributed modules must have some information (like the tile indexes and the problem size) in order to correctly generate the control signals for non-rectangular iteration spaces, and for providing problem size independency. For non-rectangular iteration spaces, tile indexes and problem size provide the information that helps to each distributed module to decide when a valid mapping exists from the processor space (logical array) to the physical array. For providing problem size independency, the centralized and distributed modules require to know the problem size that is being solved.

3. **What memory schemes or hierarchies are able to provide and extract data from the processor array in order to avoid bottleneck problem from memory?** External memory interfaces with multiple communication channels are required for delivering/extracting data to/from the processor array in order to avoid the bottleneck memory problem. Although these communication channels could be implemented physically by the use of a single n -port memory, this multi-port memory could be technologically unfeasible for larger processor arrays. In this sense, distributing the original data into different memory banks (similar to an interleaved scheme) provides the required communication channels regardless of the processor array size. However, in order to reduce the number of memory banks, the use of different clock domains becomes an attractive solution in order to reduce the overhead of dividing and storing data into different memory locations placed in different memory banks. Combining multi-port memory facilities, working twice faster than the processor array, provide the communication channels required by the processor array without stalling the processor array computations. Besides of this external memory interface, memory elements for storing data that has been already read or produced by the array (like FIFO elements) are required in order to avoid unnecessary external memory accesses.

4. **What are the processor array sizes that have a PE utilization percentage above of 50% for the selected loop-kernel algorithms generated by using the polytope model?** Mainly, the utilization percentage (or efficiency) relies on three or two aspects depending on the number of loops in the algorithm. For two-dimensional loops the relation between the processor array size and the problem size to be solved, and the sequential processor used as baseline are two aspects which impact on the efficiency. Additionally on these two aspects, for three-dimensional loops the processor array shape is a third aspect to be considered. A processor array with a high PE efficiency is derived if the problem size for which it was targeted is much larger than the partitioned processor space. Also, the utilization percentage depends on which sequential PE is used as baseline. Using the constant-cycle PE as baseline, higher efficiencies are achieved compared to multi-cycle PE baseline. In general, the loop algorithms

easily achieve an utilization percentages above of 50% when the processor array size is larger than the partitioned processor space and the constant-cycle PE is used as baseline. However, if the multi-cycle PE is used as baseline, then utilization percentages no greater than 6% are achieved. Additionally, in the case of three-dimensional algorithms when processor arrays have a square shape a better efficiency is achieved compared to the case of rectangular processor arrays, regardless of the sequential PE used as baseline.

7.2 Future Work

The research presented in this dissertation could be extended in several directions. One of them consists of using strip mining technique for deriving processor arrays working in a LSGP or in a hierarchical co-partitioning approach. Although there are works that generate processor arrays with these partitioning approaches, none of them supports solving several problem sizes. In this sense, a possible processor array using LSGP should implement an intermediate memory for each PE in the array instead of having an intermediate memory for each processor array row or column. Also, intermediate registers among PEs should be placed in order to respect the scheduler function.

Another issue concerning strip mining is the use non-unimodular transformations. Strip mining could be only used for partitioning processor spaces when the space-time transformation comes from a unimodular matrix. Although using non-unimodular matrixes for space-time mapping derives index spaces which are not longer a polytope but a polyhedra, supporting such non-unimodular matrixes could provide a more general framework for supporting any space-time transformation. More precisely, using non-unimodular transformations would lead to use other kinds of schedulers, like affine schedulers. Generally, these affine schedulers derive faster execution times than the linear scheduler, thus, supporting non-unimodular transformation would lead to derive faster processor arrays with highly pipelined PEs.

In relation to the control scheme, some improvements could be made in order to increase the maximum operational frequency. Since the centralized Max/Min sub-modules map the complex mathematical expressions for calculating the bounds of the n -dimensional target polytope, they generate the longest critical path of the control scheme. One possible improvement is to apply pipelining to these expressions in order to reduce such critical path. Intuitively, all the Max/Min sub-modules should have the same number of pipeline stages, however a deeper analysis would be required in order to prevent if some forwarding units would be required too. Furthermore, another aspect that would require research is the use of memories for storing pre-generated sequences instead of generating them in run-time execution. Using such approach is straightforward for processor arrays dedicated to solve a unique problem size, since only one pre-generated sequence should be stored. However, in order to solve several problem sizes, a mechanism for having a unique pre-generated sequence and for reusing such sequence is required. This would help to avoid the need of having several sequences stored in memories, and consequently reducing the total amount of memory inside the processor array.

With respect to the external memory, some ideas might be applied in order to hide the latency due to the process of inserting input data into the different memory banks, *i.e.* reducing the overhead introduced by dividing and storing data into different memory locations placed in different memory banks. Mainly, these improvements consist of inserting data into the memory banks while the processor array is performing the algorithm computations, without the need of waiting that the matrixes are totally inserted into the memory banks. In other words, storing the complete matrixes into the memory banks is not required for starting the processor array computations, since the array requires only a matrix data subset when a new tile is being scanned. Developing mechanisms for inserting data into the memory banks while the processor array is working would be the next step in order to enhance the external memory system, creating a new level in the memory hierarchy.

Finally, an interesting aspect is the study and comparison of processor arrays derived by following the polytope model and highly-pipelined mono-processors derived by modeling the loop nests as a

polytope in terms of their acceleration, specifically for algorithms with high latency operations like Cholesky and LU decomposition algorithms. Such study would help to determine for which algorithm implemented, either as a processor array or highly-pipelined mono-processor, a higher acceleration is achieved.

A

Appendix

This appendix shows the mean values of relative acceleration, relative efficiency and relative load imbalance metrics applied over a set of seven different nested loop algorithms: matrix-vector multiplication (MatVec), finite impulse response (FIR) filter, back substitution, forward substitution, matrix-matrix multiplication (MatMul), Cholesky decomposition and LU decomposition. These algorithms can be organized according to the number of nested loops. For deriving the processor arrays, the same scheduler function but different allocation matrixes were employed. In the case of two dimensional algorithms, two different allocation matrixes were used as design parameters, whereas for three-dimensional algorithms six different allocation matrixes were employed. The name of these configurations with their respective allocation function are shown in tables A.1 and A.2. The scheduler function used for space-time mapping is $\vec{\lambda} = [1, 1]$ in the case of two-dimensional algorithms; whereas for three-dimensional arrays the scheduler is $\vec{\lambda} = [1, 1, 1]$.

For each allocation matrix, several possible processor arrays with different number of PEs are evaluated for a range of problem sizes of $N = [0, \dots, 500]$. In the case of two-dimensional algorithms, one-dimensional arrays are derived; whereas for three-dimensional algorithms, two-dimensional arrays

Configuration Code Name	Projection Vector	Allocation Matrix
<i>2-Ver-1</i>	$\vec{u} = [1 \ 0]^t$	$\Phi = [0 \ 1]$
<i>2-Ver-2</i>	$\vec{u} = [0 \ 1]^t$	$\Phi = [1 \ 0]$

Table A.1: Allocation functions used for two-dimensional algorithms.

are obtained. In both cases, the processor array sizes are limited to multiples of power of two, with a maximum of $SSp_0 = 128$ for one-dimensional arrays and $SSp_0 = SSp_1 = 32$ for two-dimensional arrays. For example, in the LU case, two-dimensional processor arrays with 16 PEs can be derived with arrays of 1×16 , 2×8 , 4×4 , 8×2 and 16×1 PEs; therefore the shape of the processor array is also evaluated.

Configuration Code Name	Projection Vector	Allocation Matrix
<i>3-Ver-1</i>	$\vec{u} = [1 \ 0 \ 0]^t$	$\Phi = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
<i>3-Ver-2</i>	$\vec{u} = [1 \ 0 \ 0]^t$	$\Phi = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$
<i>3-Ver-3</i>	$\vec{u} = [0 \ 1 \ 0]^t$	$\Phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
<i>3-Ver-4</i>	$\vec{u} = [0 \ 1 \ 0]^t$	$\Phi = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$
<i>3-Ver-5</i>	$\vec{u} = [0 \ 0 \ 1]^t$	$\Phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$
<i>3-Ver-6</i>	$\vec{u} = [0 \ 0 \ 1]^t$	$\Phi = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

Table A.2: Allocation functions used for three-dimensional algorithms.

Since Back and Forward substitution have the same loop kernel, the same iteration space, and the same operations, their results are identical and they are shown in a same set named Back/Forward. Similarly, since MatVec and FIR filter algorithms are similar, their results are referred as the MatVec/FIR. Finally, in the case of Back/Forward, Cholesky and LU processor arrays, the operational latencies used when multi-cycle PE is used as baseline are: 21 clock cycles for division, 10 clock cycles for square root and one clock cycle for MAC.

A.1 Matrix and FIR Filter

Processors	2-Ver-1	2-Ver-2
001 PE	1.0000	1.0000
002 PE	1.9635	1.9635
004 PE	3.7875	3.7875
008 PE	7.0648	7.0648
016 PE	12.4155	12.4155
032 PE	19.8674	19.8674
064 PE	28.1404	28.1404
128 PE	35.1463	35.1463

Table A.3: Average relative acceleration for several MatVec processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	2-Ver-1	2-Ver-2
001 PE	1.0000	1.0000
002 PE	0.9817	0.9817
004 PE	0.9469	0.9469
008 PE	0.8831	0.8831
016 PE	0.7760	0.7760
032 PE	0.6209	0.6209
064 PE	0.4397	0.4397
128 PE	0.2746	0.2746

Table A.4: Average relative efficiency for several MatVec processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	2-Ver-1	2-Ver-2
001 PE	0.0000	0.0000
002 PE	0.0061	0.0061
004 PE	0.0162	0.0162
008 PE	0.0331	0.0331
016 PE	0.0606	0.0606
032 PE	0.1045	0.1045
064 PE	0.1710	0.1710
128 PE	0.2643	0.2643

Table A.5: Average relative load imbalance for several MatVec processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

A.2 Back and Forward Substitution

Processors	2-Ver-1	2-Ver-2
001 PE	1.0000	1.0000
002 PE	1.9179	1.9354
004 PE	3.5493	3.6325
008 PE	6.1884	6.4528
016 PE	9.8729	10.4927
032 PE	14.0825	15.1413
064 PE	17.9231	19.2190
128 PE	20.7707	21.9027

Table A.6: Average relative acceleration for several BackForward processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	2-Ver-1	2-Ver-2
001 PE	1.0000	1.0000
002 PE	0.9589	0.9677
004 PE	0.8873	0.9081
008 PE	0.7735	0.8066
016 PE	0.6171	0.6558
032 PE	0.4401	0.4732
064 PE	0.2800	0.3003
128 PE	0.1623	0.1711

Table A.7: Average relative efficiency for several BackForward processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	2-Ver-1	2-Ver-2
001 PE	0.0000	0.0000
002 PE	0.0112	0.0112
004 PE	0.0296	0.0296
008 PE	0.0598	0.0598
016 PE	0.1079	0.1079
032 PE	0.1820	0.1820
064 PE	0.2890	0.2890
128 PE	0.4303	0.4303

Table A.8: Average relative load imbalance for several BackForward processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	2-Ver-1	2-Ver-2
001 PE	0.0599	0.0599
002 PE	0.1169	0.1176
004 PE	0.2228	0.2266
008 PE	0.4075	0.4220
016 PE	0.6972	0.7405
032 PE	1.0850	1.1831
064 PE	1.5110	1.6695
128 PE	1.8921	2.0653

Table A.9: Average relative aceleration for several BackForward processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential multi-cycle PE as baseline.

Processors	2-Ver-1	2-Ver-2
001 PE	0.0599	0.0599
002 PE	0.0584	0.0588
004 PE	0.0557	0.0567
008 PE	0.0509	0.0528
016 PE	0.0436	0.0463
032 PE	0.0339	0.0370
064 PE	0.0236	0.0261
128 PE	0.0148	0.0161

Table A.10: Average relative efficiency for several BackForward processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential multi-cycle PE as baseline.

A.3 Matrix-Matrix Multiplication

Processors	Array Size	Ver 1	Ver 2	Ver 3	Ver 4	Ver 5	Ver 6
1 PE	01 × 01	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2 PEs	01 × 02	1.9635	1.9635	1.9635	1.9635	1.9635	1.9635
	02 × 01	1.9635	1.9635	1.9635	1.9635	1.9635	1.9635
4 PEs	01 × 04	3.7875	3.7875	3.7875	3.7875	3.7875	3.7875
	02 × 02	3.8447	3.8447	3.8447	3.8447	3.8447	3.8447
	04 × 01	3.7875	3.7875	3.7875	3.7875	3.7875	3.7875
8 PEs	01 × 08	7.0648	7.0648	7.0648	7.0648	7.0648	7.0648
	02 × 04	7.3763	7.3763	7.3763	7.3763	7.3763	7.3763
	04 × 02	7.3763	7.3763	7.3763	7.3763	7.3763	7.3763
	08 × 01	7.0648	7.0648	7.0648	7.0648	7.0648	7.0648
16 PEs	01 × 16	12.4155	12.4155	12.4155	12.4155	12.4155	12.4155
	02 × 08	13.6145	13.6145	13.6145	13.6145	13.6145	13.6145
	04 × 04	13.9906	13.9906	13.9906	13.9906	13.9906	13.9906
	08 × 02	13.6145	13.6145	13.6145	13.6145	13.6145	13.6145
	16 × 01	12.4155	12.4155	12.4155	12.4155	12.4155	12.4155
32 PEs	01 × 32	19.8674	19.8674	19.8674	19.8674	19.8674	19.8674
	02 × 16	23.4989	23.4989	23.4989	23.4989	23.4989	23.4989
	04 × 08	25.2996	25.2996	25.2996	25.2996	25.2996	25.2996
	08 × 04	25.2996	25.2996	25.2996	25.2996	25.2996	25.2996
	16 × 02	23.4989	23.4989	23.4989	23.4989	23.4989	23.4989
	32 × 01	19.8674	19.8674	19.8674	19.8674	19.8674	19.8674
64 PEs	02 × 32	36.6560	36.6560	36.6560	36.6560	36.6560	36.6560
	04 × 16	42.2278	42.2278	42.2278	42.2278	42.2278	42.2278
	08 × 08	44.0224	44.0224	44.0224	44.0224	44.0224	44.0224
	16 × 04	42.2278	42.2278	42.2278	42.2278	42.2278	42.2278
	32 × 02	36.6560	36.6560	36.6560	36.6560	36.6560	36.6560
128 PEs	04 × 32	62.9982	62.9982	62.9982	62.9982	62.9982	62.9982
	08 × 16	69.4969	69.4969	69.4969	69.4969	69.4969	69.4969
	16 × 08	69.4969	69.4969	69.4969	69.4969	69.4969	69.4969
	32 × 04	62.9982	62.9982	62.9982	62.9982	62.9982	62.9982
256 PEs	08 × 32	96.9653	96.9653	96.9653	96.9653	96.9653	96.9653
	16 × 16	101.4313	101.4313	101.4313	101.4313	101.4313	101.4313
	32 × 08	96.9653	96.9653	96.9653	96.9653	96.9653	96.9653
512 PEs	16 × 32	130.8061	130.8061	130.8061	130.8061	130.8061	130.8061
	32 × 16	130.8061	130.8061	130.8061	130.8061	130.8061	130.8061
1024 PEs	32 × 32	156.6683	156.6683	156.6683	156.6683	156.6683	156.6683

Table A.11: Average relative acceleration for several MatMul processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	Ver 1	Ver 2	Ver 3	Ver 4	Ver 5	Ver 6
1 PE	01 × 01	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2 PEs	01 × 02	0.9817	0.9817	0.9817	0.9817	0.9817	0.9817
	02 × 01	0.9817	0.9817	0.9817	0.9817	0.9817	0.9817
4 PEs	01 × 04	0.9469	0.9469	0.9469	0.9469	0.9469	0.9469
	02 × 02	0.9612	0.9612	0.9612	0.9612	0.9612	0.9612
	04 × 01	0.9469	0.9469	0.9469	0.9469	0.9469	0.9469
8 PEs	01 × 08	0.8831	0.8831	0.8831	0.8831	0.8831	0.8831
	02 × 04	0.9220	0.9220	0.9220	0.9220	0.9220	0.9220
	04 × 02	0.9220	0.9220	0.9220	0.9220	0.9220	0.9220
	08 × 01	0.8831	0.8831	0.8831	0.8831	0.8831	0.8831
16 PEs	01 × 16	0.7760	0.7760	0.7760	0.7760	0.7760	0.7760
	02 × 08	0.8509	0.8509	0.8509	0.8509	0.8509	0.8509
	04 × 04	0.8744	0.8744	0.8744	0.8744	0.8744	0.8744
	08 × 02	0.8509	0.8509	0.8509	0.8509	0.8509	0.8509
	16 × 01	0.7760	0.7760	0.7760	0.7760	0.7760	0.7760
32 PEs	01 × 32	0.6209	0.6209	0.6209	0.6209	0.6209	0.6209
	02 × 16	0.7343	0.7343	0.7343	0.7343	0.7343	0.7343
	04 × 08	0.7906	0.7906	0.7906	0.7906	0.7906	0.7906
	08 × 04	0.7906	0.7906	0.7906	0.7906	0.7906	0.7906
	16 × 02	0.7343	0.7343	0.7343	0.7343	0.7343	0.7343
	32 × 01	0.6209	0.6209	0.6209	0.6209	0.6209	0.6209
64 PEs	02 × 32	0.5728	0.5728	0.5728	0.5728	0.5728	0.5728
	04 × 16	0.6598	0.6598	0.6598	0.6598	0.6598	0.6598
	08 × 08	0.6878	0.6878	0.6878	0.6878	0.6878	0.6878
	16 × 04	0.6598	0.6598	0.6598	0.6598	0.6598	0.6598
	32 × 02	0.5728	0.5728	0.5728	0.5728	0.5728	0.5728
128 PEs	04 × 32	0.4922	0.4922	0.4922	0.4922	0.4922	0.4922
	08 × 16	0.5429	0.5429	0.5429	0.5429	0.5429	0.5429
	16 × 08	0.5429	0.5429	0.5429	0.5429	0.5429	0.5429
	32 × 04	0.4922	0.4922	0.4922	0.4922	0.4922	0.4922
256 PEs	08 × 32	0.3788	0.3788	0.3788	0.3788	0.3788	0.3788
	16 × 16	0.3962	0.3962	0.3962	0.3962	0.3962	0.3962
	32 × 08	0.3788	0.3788	0.3788	0.3788	0.3788	0.3788
512 PEs	16 × 32	0.2555	0.2555	0.2555	0.2555	0.2555	0.2555
	32 × 16	0.2555	0.2555	0.2555	0.2555	0.2555	0.2555
1024 PEs	32 × 32	0.1530	0.1530	0.1530	0.1530	0.1530	0.1530

Table A.12: Average relative efficiency for several MatMul processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	Ver 1	Ver 2	Ver 3	Ver 4	Ver 5	Ver 6
1 PE	01 × 01	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2 PEs	01 × 02	0.0061	0.0061	0.0061	0.0061	0.0061	0.0061
	02 × 01	0.0061	0.0061	0.0061	0.0061	0.0061	0.0061
4 PEs	01 × 04	0.0162	0.0162	0.0162	0.0162	0.0162	0.0162
	02 × 02	0.0114	0.0114	0.0114	0.0114	0.0114	0.0114
	04 × 01	0.0162	0.0162	0.0162	0.0162	0.0162	0.0162
8 PEs	01 × 08	0.0331	0.0331	0.0331	0.0331	0.0331	0.0331
	02 × 04	0.0211	0.0211	0.0211	0.0211	0.0211	0.0211
	04 × 02	0.0211	0.0211	0.0211	0.0211	0.0211	0.0211
	08 × 01	0.0331	0.0331	0.0331	0.0331	0.0331	0.0331
16 PEs	01 × 16	0.0606	0.0606	0.0606	0.0606	0.0606	0.0606
	02 × 08	0.0374	0.0374	0.0374	0.0374	0.0374	0.0374
	04 × 04	0.0296	0.0296	0.0296	0.0296	0.0296	0.0296
	08 × 02	0.0374	0.0374	0.0374	0.0374	0.0374	0.0374
	16 × 01	0.0606	0.0606	0.0606	0.0606	0.0606	0.0606
32 PEs	01 × 32	0.1045	0.1045	0.1045	0.1045	0.1045	0.1045
	02 × 16	0.0644	0.0644	0.0644	0.0644	0.0644	0.0644
	04 × 08	0.0453	0.0453	0.0453	0.0453	0.0453	0.0453
	08 × 04	0.0453	0.0453	0.0453	0.0453	0.0453	0.0453
	16 × 02	0.0644	0.0644	0.0644	0.0644	0.0644	0.0644
	32 × 01	0.1045	0.1045	0.1045	0.1045	0.1045	0.1045
64 PEs	02 × 32	0.1078	0.1078	0.1078	0.1078	0.1078	0.1078
	04 × 16	0.0715	0.0715	0.0715	0.0715	0.0715	0.0715
	08 × 08	0.0590	0.0590	0.0590	0.0590	0.0590	0.0590
	16 × 04	0.0715	0.0715	0.0715	0.0715	0.0715	0.0715
	32 × 02	0.1078	0.1078	0.1078	0.1078	0.1078	0.1078
128 PEs	04 × 32	0.1139	0.1139	0.1139	0.1139	0.1139	0.1139
	08 × 16	0.0841	0.0841	0.0841	0.0841	0.0841	0.0841
	16 × 08	0.0841	0.0841	0.0841	0.0841	0.0841	0.0841
	32 × 04	0.1139	0.1139	0.1139	0.1139	0.1139	0.1139
256 PEs	08 × 32	0.1251	0.1251	0.1251	0.1251	0.1251	0.1251
	16 × 16	0.1056	0.1056	0.1056	0.1056	0.1056	0.1056
	32 × 08	0.1251	0.1251	0.1251	0.1251	0.1251	0.1251
512 PEs	16 × 32	0.1446	0.1446	0.1446	0.1446	0.1446	0.1446
	32 × 16	0.1446	0.1446	0.1446	0.1446	0.1446	0.1446
1024 PEs	32 × 32	0.1769	0.1769	0.1769	0.1769	0.1769	0.1769

Table A.13: Average relative load imbalance for several MatMul processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

A.4 Cholesky Decomposition

Processors	Array Size	Ver 1	Ver 2	Ver 3	Ver 4	Ver 5	Ver 6
1 PE	01 × 01	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2 PEs	01 × 02	1.8909	1.8899	1.8909	1.9128	1.8923	1.9128
	02 × 01	1.8899	1.8909	1.9128	1.8909	1.9128	1.8923
4 PEs	01 × 04	3.4106	3.4113	3.4106	3.5140	3.4194	3.5140
	02 × 02	3.5788	3.5788	3.5875	3.5875	3.6314	3.6314
	04 × 01	3.4113	3.4106	3.5140	3.4106	3.5140	3.4194
8 PEs	01 × 08	5.7072	5.7251	5.7072	6.0229	5.7415	6.0229
	02 × 04	6.3841	6.3778	6.4118	6.5126	6.4906	6.5948
	04 × 02	6.3778	6.3841	6.5126	6.4118	6.5948	6.4906
	08 × 01	5.7251	5.7072	6.0229	5.7072	6.0229	5.7415
16 PEs	01 × 16	8.6189	8.6883	8.6189	9.3028	8.7107	9.3028
	02 × 08	10.5053	10.4925	10.5805	10.9418	10.7187	11.0895
	04 × 04	11.3280	11.3280	11.4184	11.4184	11.7485	11.7485
	08 × 02	10.4925	10.5053	10.9418	10.5805	11.0895	10.7187
	16 × 01	8.6883	8.6189	9.3028	8.6189	9.3028	8.7107
32 PEs	01 × 32	11.5895	11.7473	11.5895	12.6465	11.7692	12.6465
	02 × 16	15.5232	15.5098	15.6879	16.4424	15.9167	16.6890
	04 × 08	18.1492	18.0987	18.3825	18.6349	18.8344	19.1637
	08 × 04	18.0987	18.1492	18.6349	18.3825	19.1637	18.8344
	16 × 02	15.5098	15.5232	16.4424	15.6879	16.6890	15.9167
	32 × 01	11.7473	11.5895	12.6465	11.5895	12.6465	11.7692
64 PEs	02 × 32	20.4023	20.3959	20.6877	21.7168	21.0291	22.0816
	04 × 16	25.9469	25.7873	26.4263	26.9662	26.9699	27.7323
	08 × 08	28.4760	28.4760	28.8856	28.8856	30.0563	30.0563
	16 × 04	25.7873	25.9469	26.9662	26.4263	27.7323	26.9699
	32 × 02	20.3959	20.4023	21.7168	20.6877	22.0816	21.0291
128 PEs	04 × 32	33.0013	32.6890	33.7808	34.3159	34.3829	35.3141
	08 × 16	38.9640	38.7657	39.7351	39.9054	40.8485	41.3227
	16 × 08	38.7657	38.9640	39.9054	39.7351	41.3227	40.8485
	32 × 04	32.6890	33.0013	34.3159	33.7808	35.3141	34.3829
256 PEs	08 × 32	47.5968	47.1079	48.7525	48.6917	49.6857	50.2546
	16 × 16	51.6255	51.6255	52.4269	52.4269	54.2695	54.2695
	32 × 08	47.1079	47.5968	48.6917	48.7525	50.2546	49.6857
512 PEs	16 × 32	60.5799	60.2574	61.6864	61.4505	62.9234	63.1454
	32 × 16	60.2574	60.5799	61.4505	61.6864	63.1454	62.9234
1024 PEs	32 × 32	69.4844	69.4844	70.2560	70.2560	71.6935	71.6935

Table A.14: Average relative acceleration for several Cholesky processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2 PEs	01 × 02	0.9454	0.9449	0.9454	0.9564	0.9462	0.9564
	02 × 01	0.9449	0.9454	0.9564	0.9454	0.9564	0.9462
4 PEs	01 × 04	0.8526	0.8528	0.8526	0.8785	0.8549	0.8785
	02 × 02	0.8947	0.8947	0.8969	0.8969	0.9078	0.9078
	04 × 01	0.8528	0.8526	0.8785	0.8526	0.8785	0.8549
8 PEs	01 × 08	0.7134	0.7156	0.7134	0.7529	0.7177	0.7529
	02 × 04	0.7980	0.7972	0.8015	0.8141	0.8113	0.8243
	04 × 02	0.7972	0.7980	0.8141	0.8015	0.8243	0.8113
	08 × 01	0.7156	0.7134	0.7529	0.7134	0.7529	0.7177
16 PEs	01 × 16	0.5387	0.5430	0.5387	0.5814	0.5444	0.5814
	02 × 08	0.6566	0.6558	0.6613	0.6839	0.6699	0.6931
	04 × 04	0.7080	0.7080	0.7137	0.7137	0.7343	0.7343
	08 × 02	0.6558	0.6566	0.6839	0.6613	0.6931	0.6699
	16 × 01	0.5430	0.5387	0.5814	0.5387	0.5814	0.5444
32 PEs	01 × 32	0.3622	0.3671	0.3622	0.3952	0.3678	0.3952
	02 × 16	0.4851	0.4847	0.4902	0.5138	0.4974	0.5215
	04 × 08	0.5672	0.5656	0.5745	0.5823	0.5886	0.5989
	08 × 04	0.5656	0.5672	0.5823	0.5745	0.5989	0.5886
	16 × 02	0.4847	0.4851	0.5138	0.4902	0.5215	0.4974
	32 × 01	0.3671	0.3622	0.3952	0.3622	0.3952	0.3678
64 PEs	02 × 32	0.3188	0.3187	0.3232	0.3393	0.3286	0.3450
	04 × 16	0.4054	0.4029	0.4129	0.4213	0.4214	0.4333
	08 × 08	0.4449	0.4449	0.4513	0.4513	0.4696	0.4696
	16 × 04	0.4029	0.4054	0.4213	0.4129	0.4333	0.4214
	32 × 02	0.3187	0.3188	0.3393	0.3232	0.3450	0.3286
128 PEs	04 × 32	0.2578	0.2554	0.2639	0.2681	0.2686	0.2759
	08 × 16	0.3044	0.3029	0.3104	0.3118	0.3191	0.3228
	16 × 08	0.3029	0.3044	0.3118	0.3104	0.3228	0.3191
	32 × 04	0.2554	0.2578	0.2681	0.2639	0.2759	0.2686
256 PEs	08 × 32	0.1859	0.1840	0.1904	0.1902	0.1941	0.1963
	16 × 16	0.2017	0.2017	0.2048	0.2048	0.2120	0.2120
	32 × 08	0.1840	0.1859	0.1902	0.1904	0.1963	0.1941
512 PEs	16 × 32	0.1183	0.1177	0.1205	0.1200	0.1229	0.1233
	32 × 16	0.1177	0.1183	0.1200	0.1205	0.1233	0.1229
1024 PEs	32 × 32	0.0679	0.0679	0.0686	0.0686	0.0700	0.0700

Table A.15: Average relative efficiency for several Cholesky processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2 PEs	01 × 02	0.0155	0.0019	0.0155	0.0155	0.0019	0.0155
	02 × 01	0.0019	0.0155	0.0155	0.0155	0.0155	0.0019
4 PEs	01 × 04	0.0409	0.0057	0.0409	0.0409	0.0057	0.0409
	02 × 02	0.0291	0.0291	0.0291	0.0291	0.0291	0.0291
	04 × 01	0.0057	0.0409	0.0409	0.0409	0.0409	0.0057
8 PEs	01 × 08	0.0817	0.0133	0.0817	0.0817	0.0133	0.0817
	02 × 04	0.0525	0.0315	0.0525	0.0525	0.0315	0.0525
	04 × 02	0.0315	0.0525	0.0525	0.0525	0.0525	0.0315
	08 × 01	0.0133	0.0817	0.0817	0.0817	0.0817	0.0133
16 PEs	01 × 16	0.1454	0.0288	0.1454	0.1454	0.0288	0.1454
	02 × 08	0.0913	0.0367	0.0913	0.0913	0.0367	0.0913
	04 × 04	0.0733	0.0733	0.0733	0.0733	0.0733	0.0733
	08 × 02	0.0367	0.0913	0.0913	0.0913	0.0913	0.0367
	16 × 01	0.0288	0.1454	0.1454	0.1454	0.1454	0.0288
32 PEs	01 × 32	0.2400	0.0595	0.2400	0.2400	0.0595	0.2400
	02 × 16	0.1530	0.0491	0.1530	0.1530	0.0491	0.1530
	04 × 08	0.1090	0.0769	0.1090	0.1090	0.0769	0.1090
	08 × 04	0.0769	0.1090	0.1090	0.1090	0.1090	0.0769
	16 × 02	0.0491	0.1530	0.1530	0.1530	0.1530	0.0491
	32 × 01	0.0595	0.2400	0.2400	0.2400	0.2400	0.0595
64 PEs	02 × 32	0.2457	0.0762	0.2457	0.2457	0.0762	0.2457
	04 × 16	0.1673	0.0854	0.1673	0.1673	0.0854	0.1673
	08 × 08	0.1404	0.1404	0.1404	0.1404	0.1404	0.1404
	16 × 04	0.0854	0.1673	0.1673	0.1673	0.1673	0.0854
	32 × 02	0.0762	0.2457	0.2457	0.2457	0.2457	0.0762
128 PEs	04 × 32	0.2566	0.1071	0.2566	0.2566	0.1071	0.2566
	08 × 16	0.1935	0.1466	0.1935	0.1935	0.1466	0.1935
	16 × 08	0.1466	0.1935	0.1935	0.1935	0.1935	0.1466
	32 × 04	0.1071	0.2566	0.2566	0.2566	0.2566	0.1071
256 PEs	08 × 32	0.2769	0.1615	0.2769	0.2769	0.1615	0.2769
	16 × 16	0.2392	0.2392	0.2392	0.2392	0.2392	0.2392
	32 × 08	0.1615	0.2769	0.2769	0.2769	0.2769	0.1615
512 PEs	16 × 32	0.3134	0.2501	0.3134	0.3134	0.2501	0.3134
	32 × 16	0.2501	0.3134	0.3134	0.3134	0.3134	0.2501
1024 PEs	32 × 32	0.3748	0.3748	0.3748	0.3748	0.3748	0.3748

Table A.16: Average relative load imbalance for several Cholesky processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	0.0645	0.0645	0.0645	0.0645	0.0645	0.0645
2 PEs	01 × 02	0.1245	0.1245	0.1245	0.1256	0.1246	0.1256
	02 × 01	0.1245	0.1245	0.1256	0.1245	0.1256	0.1246
4 PEs	01 × 04	0.2331	0.2331	0.2331	0.2384	0.2333	0.2384
	02 × 02	0.2419	0.2419	0.2421	0.2421	0.2441	0.2441
	04 × 01	0.2331	0.2331	0.2384	0.2331	0.2384	0.2333
8 PEs	01 × 08	0.4131	0.4136	0.4131	0.4325	0.4142	0.4325
	02 × 04	0.4518	0.4517	0.4523	0.4587	0.4561	0.4625
	04 × 02	0.4517	0.4518	0.4587	0.4523	0.4625	0.4561
	08 × 01	0.4136	0.4131	0.4325	0.4131	0.4325	0.4142
16 PEs	01 × 16	0.6736	0.6768	0.6736	0.7269	0.6780	0.7269
	02 × 08	0.7977	0.7981	0.7996	0.8288	0.8068	0.8357
	04 × 04	0.8510	0.8510	0.8536	0.8536	0.8726	0.8726
	08 × 02	0.7981	0.7977	0.8288	0.7996	0.8357	0.8068
	16 × 01	0.6768	0.6736	0.7269	0.6736	0.7269	0.6780
32 PEs	01 × 32	0.9849	0.9967	0.9849	1.0937	0.9984	1.0937
	02 × 16	1.2936	1.2970	1.2986	1.3839	1.3133	1.3956
	04 × 08	1.4927	1.4925	1.5009	1.5323	1.5322	1.5656
	08 × 04	1.4925	1.4927	1.5323	1.5009	1.5656	1.5322
	16 × 02	1.2970	1.2936	1.3839	1.2986	1.3956	1.3133
	32 × 01	0.9967	0.9849	1.0937	0.9849	1.0937	0.9984
64 PEs	02 × 32	1.8791	1.8932	1.8895	2.0640	1.9208	2.0821
	04 × 16	2.3976	2.3984	2.4188	2.5313	2.4674	2.5851
	08 × 08	2.6440	2.6440	2.6676	2.6676	2.7773	2.7773
	16 × 04	2.3984	2.3976	2.5313	2.4188	2.5851	2.4674
	32 × 02	1.8932	1.8791	2.0640	1.8895	2.0821	1.9208
128 PEs	04 × 32	3.4432	3.4515	3.4871	3.7220	3.5624	3.8005
	08 × 16	4.1802	4.1766	4.2393	4.3359	4.3862	4.5019
	16 × 08	4.1766	4.1802	4.3359	4.2393	4.5019	4.3862
	32 × 04	3.4515	3.4432	3.7220	3.4871	3.8005	3.5624
256 PEs	08 × 32	5.8939	5.8837	6.0121	6.2439	6.1901	6.4654
	16 × 16	6.6394	6.6394	6.7559	6.7559	7.1305	7.1305
	32 × 08	5.8837	5.8939	6.2439	6.0121	6.4654	6.1901
512 PEs	16 × 32	9.1233	9.0989	9.3448	9.4699	9.7209	9.9192
	32 × 16	9.0989	9.1233	9.4699	9.3448	9.9192	9.7209
1024 PEs	32 × 32	12.4939	12.4939	12.7792	12.7792	13.4351	13.4351

Table A.17: Average relative acceleration for several Cholesky processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential multi-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	0.0645	0.0645	0.0645	0.0645	0.0645	0.0645
2 PEs	01 × 02	0.0623	0.0623	0.0623	0.0628	0.0623	0.0628
	02 × 01	0.0623	0.0623	0.0628	0.0623	0.0628	0.0623
4 PEs	01 × 04	0.0583	0.0583	0.0583	0.0596	0.0583	0.0596
	02 × 02	0.0605	0.0605	0.0605	0.0605	0.0610	0.0610
	04 × 01	0.0583	0.0583	0.0596	0.0583	0.0596	0.0583
8 PEs	01 × 08	0.0516	0.0517	0.0516	0.0541	0.0518	0.0541
	02 × 04	0.0565	0.0565	0.0565	0.0573	0.0570	0.0578
	04 × 02	0.0565	0.0565	0.0573	0.0565	0.0578	0.0570
	08 × 01	0.0517	0.0516	0.0541	0.0516	0.0541	0.0518
16 PEs	01 × 16	0.0421	0.0423	0.0421	0.0454	0.0424	0.0454
	02 × 08	0.0499	0.0499	0.0500	0.0518	0.0504	0.0522
	04 × 04	0.0532	0.0532	0.0534	0.0534	0.0545	0.0545
	08 × 02	0.0499	0.0499	0.0518	0.0500	0.0522	0.0504
	16 × 01	0.0423	0.0421	0.0454	0.0421	0.0454	0.0424
32 PEs	01 × 32	0.0308	0.0311	0.0308	0.0342	0.0312	0.0342
	02 × 16	0.0404	0.0405	0.0406	0.0432	0.0410	0.0436
	04 × 08	0.0466	0.0466	0.0469	0.0479	0.0479	0.0489
	08 × 04	0.0466	0.0466	0.0479	0.0469	0.0489	0.0479
	16 × 02	0.0405	0.0404	0.0432	0.0406	0.0436	0.0410
	32 × 01	0.0311	0.0308	0.0342	0.0308	0.0342	0.0312
64 PEs	02 × 32	0.0294	0.0296	0.0295	0.0323	0.0300	0.0325
	04 × 16	0.0375	0.0375	0.0378	0.0396	0.0386	0.0404
	08 × 08	0.0413	0.0413	0.0417	0.0417	0.0434	0.0434
	16 × 04	0.0375	0.0375	0.0396	0.0378	0.0404	0.0386
	32 × 02	0.0296	0.0294	0.0323	0.0295	0.0325	0.0300
128 PEs	04 × 32	0.0269	0.0270	0.0272	0.0291	0.0278	0.0297
	08 × 16	0.0327	0.0326	0.0331	0.0339	0.0343	0.0352
	16 × 08	0.0326	0.0327	0.0339	0.0331	0.0352	0.0343
	32 × 04	0.0270	0.0269	0.0291	0.0272	0.0297	0.0278
256 PEs	08 × 32	0.0230	0.0230	0.0235	0.0244	0.0242	0.0253
	16 × 16	0.0259	0.0259	0.0264	0.0264	0.0279	0.0279
	32 × 08	0.0230	0.0230	0.0244	0.0235	0.0253	0.0242
512 PEs	16 × 32	0.0178	0.0178	0.0183	0.0185	0.0190	0.0194
	32 × 16	0.0178	0.0178	0.0185	0.0183	0.0194	0.0190
1024 PEs	32 × 32	0.0122	0.0122	0.0125	0.0125	0.0131	0.0131

Table A.18: Average relative efficiency for several Cholesky processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential multi-cycle PE as baseline.

A.5 LU Decomposition

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2 PEs	01 × 02	1.9104	1.9104	1.9384	1.9349	1.9384	1.9349
	02 × 01	1.9104	1.9104	1.9349	1.9384	1.9349	1.9384
4 PEs	01 × 04	3.5119	3.5119	3.6547	3.6269	3.6547	3.6269
	02 × 02	3.6347	3.6347	3.7518	3.7518	3.7518	3.7518
	04 × 01	3.5119	3.5119	3.6269	3.6547	3.6269	3.6547
8 PEs	01 × 08	6.0592	6.0592	6.5662	6.4177	6.5662	6.4177
	02 × 04	6.6388	6.6388	7.0066	6.9649	7.0066	6.9649
	04 × 02	6.6388	6.6388	6.9649	7.0066	6.9649	7.0066
	08 × 01	6.0592	6.0592	6.4177	6.5662	6.4177	6.5662
16 PEs	01 × 16	9.5303	9.5303	10.9312	10.3436	10.9312	10.3436
	02 × 08	11.3279	11.3279	12.3842	12.1209	12.3842	12.1209
	04 × 04	11.9706	11.9706	12.9136	12.9136	12.9136	12.9136
	08 × 02	11.3279	11.3279	12.1209	12.3842	12.1209	12.3842
	16 × 01	9.5303	9.5303	10.3436	10.9312	10.3436	10.9312
32 PEs	01 × 32	13.3808	13.3808	16.3954	14.7048	16.3954	14.7048
	02 × 16	17.5239	17.5239	20.1104	19.0736	20.1104	19.0736
	04 × 08	20.0614	20.0614	22.1533	21.8434	22.1533	21.8434
	08 × 04	20.0614	20.0614	21.8434	22.1533	21.8434	22.1533
	16 × 02	17.5239	17.5239	19.0736	20.1104	19.0736	20.1104
	32 × 01	13.3808	13.3808	14.7048	16.3954	14.7048	16.3954
64 PEs	02 × 32	24.1221	24.1221	29.2355	26.4007	29.2355	26.4007
	04 × 16	30.2348	30.2348	34.4672	33.0678	34.4672	33.0678
	08 × 08	32.5245	32.5245	36.3481	36.3481	36.3481	36.3481
	16 × 04	30.2348	30.2348	33.0678	34.4672	33.0678	34.4672
	32 × 02	24.1221	24.1221	26.4007	29.2355	26.4007	29.2355
128 PEs	04 × 32	40.3957	40.3957	47.6727	43.9532	47.6727	43.9532
	08 × 16	47.2157	47.2157	53.0988	51.9756	53.0988	51.9756
	16 × 08	47.2157	47.2157	51.9756	53.0988	51.9756	53.0988
	32 × 04	40.3957	40.3957	43.9532	47.6727	43.9532	47.6727
256 PEs	08 × 32	60.6073	60.6073	68.7852	65.4399	68.7852	65.4399
	16 × 16	64.9747	64.9747	72.0163	72.0163	72.0163	72.0163
	32 × 08	60.6073	60.6073	65.4399	68.7852	65.4399	68.7852
512 PEs	16 × 32	79.8593	79.8593	87.2455	85.5267	87.2455	85.5267
	32 × 16	79.8593	79.8593	85.5267	87.2455	85.5267	87.2455
1024 PEs	32 × 32	93.8045	93.8045	99.8135	99.8135	99.8135	99.8135

Table A.19: Average relative acceleration for several LU processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2 PEs	01 × 02	0.9552	0.9552	0.9692	0.9674	0.9692	0.9674
	02 × 01	0.9552	0.9552	0.9674	0.9692	0.9674	0.9692
4 PEs	01 × 04	0.8780	0.8780	0.9137	0.9067	0.9137	0.9067
	02 × 02	0.9087	0.9087	0.9380	0.9380	0.9380	0.9380
	04 × 01	0.8780	0.8780	0.9067	0.9137	0.9067	0.9137
8 PEs	01 × 08	0.7574	0.7574	0.8208	0.8022	0.8208	0.8022
	02 × 04	0.8299	0.8299	0.8758	0.8706	0.8758	0.8706
	04 × 02	0.8299	0.8299	0.8706	0.8758	0.8706	0.8758
	08 × 01	0.7574	0.7574	0.8022	0.8208	0.8022	0.8208
16 PEs	01 × 16	0.5956	0.5956	0.6832	0.6465	0.6832	0.6465
	02 × 08	0.7080	0.7080	0.7740	0.7576	0.7740	0.7576
	04 × 04	0.7482	0.7482	0.8071	0.8071	0.8071	0.8071
	08 × 02	0.7080	0.7080	0.7576	0.7740	0.7576	0.7740
	16 × 01	0.5956	0.5956	0.6465	0.6832	0.6465	0.6832
32 PEs	01 × 32	0.4181	0.4181	0.5124	0.4595	0.5124	0.4595
	02 × 16	0.5476	0.5476	0.6284	0.5960	0.6284	0.5960
	04 × 08	0.6269	0.6269	0.6923	0.6826	0.6923	0.6826
	08 × 04	0.6269	0.6269	0.6826	0.6923	0.6826	0.6923
	16 × 02	0.5476	0.5476	0.5960	0.6284	0.5960	0.6284
	32 × 01	0.4181	0.4181	0.4595	0.5124	0.4595	0.5124
64 PEs	02 × 32	0.3769	0.3769	0.4568	0.4125	0.4568	0.4125
	04 × 16	0.4724	0.4724	0.5386	0.5167	0.5386	0.5167
	08 × 08	0.5082	0.5082	0.5679	0.5679	0.5679	0.5679
	16 × 04	0.4724	0.4724	0.5167	0.5386	0.5167	0.5386
	32 × 02	0.3769	0.3769	0.4125	0.4568	0.4125	0.4568
128 PEs	04 × 32	0.3156	0.3156	0.3724	0.3434	0.3724	0.3434
	08 × 16	0.3689	0.3689	0.4148	0.4061	0.4148	0.4061
	16 × 08	0.3689	0.3689	0.4061	0.4148	0.4061	0.4148
	32 × 04	0.3156	0.3156	0.3434	0.3724	0.3434	0.3724
256 PEs	08 × 32	0.2367	0.2367	0.2687	0.2556	0.2687	0.2556
	16 × 16	0.2538	0.2538	0.2813	0.2813	0.2813	0.2813
	32 × 08	0.2367	0.2367	0.2556	0.2687	0.2556	0.2687
512 PEs	16 × 32	0.1560	0.1560	0.1704	0.1670	0.1704	0.1670
	32 × 16	0.1560	0.1560	0.1670	0.1704	0.1670	0.1704
1024 PEs	32 × 32	0.0916	0.0916	0.0975	0.0975	0.0975	0.0975

Table A.20: Average relative efficiency for several LU processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2 PEs	01 × 02	0.0087	0.0087	0.0087	0.0159	0.0087	0.0159
	02 × 01	0.0087	0.0087	0.0159	0.0087	0.0159	0.0087
4 PEs	01 × 04	0.0232	0.0232	0.0232	0.0415	0.0232	0.0415
	02 × 02	0.0171	0.0171	0.0230	0.0230	0.0230	0.0230
	04 × 01	0.0232	0.0232	0.0415	0.0232	0.0415	0.0232
8 PEs	01 × 08	0.0469	0.0469	0.0469	0.0824	0.0469	0.0824
	02 × 04	0.0307	0.0307	0.0359	0.0475	0.0359	0.0475
	04 × 02	0.0307	0.0307	0.0475	0.0359	0.0475	0.0359
	08 × 01	0.0469	0.0469	0.0824	0.0469	0.0824	0.0469
16 PEs	01 × 16	0.0853	0.0853	0.0853	0.1462	0.0853	0.1462
	02 × 08	0.0536	0.0536	0.0581	0.0873	0.0581	0.0873
	04 × 04	0.0440	0.0440	0.0582	0.0582	0.0582	0.0582
	08 × 02	0.0536	0.0536	0.0873	0.0581	0.0873	0.0581
	16 × 01	0.0853	0.0853	0.1462	0.0853	0.1462	0.0853
32 PEs	01 × 32	0.1452	0.1452	0.1452	0.2408	0.1452	0.2408
	02 × 16	0.0910	0.0910	0.0947	0.1500	0.0947	0.1500
	04 × 08	0.0657	0.0657	0.0781	0.0964	0.0781	0.0964
	08 × 04	0.0657	0.0657	0.0964	0.0781	0.0964	0.0781
	16 × 02	0.0910	0.0910	0.1500	0.0947	0.1500	0.0947
	32 × 01	0.1452	0.1452	0.2408	0.1452	0.2408	0.1452
64 PEs	02 × 32	0.1500	0.1500	0.1527	0.2437	0.1527	0.2437
	04 × 16	0.1017	0.1017	0.1120	0.1573	0.1120	0.1573
	08 × 08	0.0871	0.0871	0.1128	0.1128	0.1128	0.1128
	16 × 04	0.1017	0.1017	0.1573	0.1120	0.1573	0.1120
	32 × 02	0.1500	0.1500	0.2437	0.1527	0.2437	0.1527
128 PEs	04 × 32	0.1591	0.1591	0.1670	0.2492	0.1670	0.2492
	08 × 16	0.1210	0.1210	0.1429	0.1708	0.1429	0.1708
	16 × 08	0.1210	0.1210	0.1708	0.1429	0.1708	0.1429
	32 × 04	0.1591	0.1591	0.2492	0.1670	0.2492	0.1670
256 PEs	08 × 32	0.1758	0.1758	0.1931	0.2596	0.1931	0.2596
	16 × 16	0.1546	0.1546	0.1949	0.1949	0.1949	0.1949
	32 × 08	0.1758	0.1758	0.2596	0.1931	0.2596	0.1931
512 PEs	16 × 32	0.2056	0.2056	0.2385	0.2786	0.2385	0.2786
	32 × 16	0.2056	0.2056	0.2786	0.2385	0.2786	0.2385
1024 PEs	32 × 32	0.2560	0.2560	0.3116	0.3116	0.3116	0.3116

Table A.21: Average relative load imbalance for several LU processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential constant-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	0.0574	0.0574	0.0574	0.0574	0.0574	0.0574
2 PEs	01 × 02	0.1112	0.1112	0.1123	0.1122	0.1123	0.1122
	02 × 01	0.1112	0.1112	0.1122	0.1123	0.1122	0.1123
4 PEs	01 × 04	0.2092	0.2092	0.2151	0.2145	0.2151	0.2145
	02 × 02	0.2147	0.2147	0.2197	0.2197	0.2197	0.2197
	04 × 01	0.2092	0.2092	0.2145	0.2151	0.2145	0.2151
8 PEs	01 × 08	0.3744	0.3744	0.3971	0.3937	0.3971	0.3937
	02 × 04	0.4021	0.4021	0.4183	0.4175	0.4183	0.4175
	04 × 02	0.4021	0.4021	0.4175	0.4183	0.4175	0.4183
	08 × 01	0.3744	0.3744	0.3937	0.3971	0.3937	0.3971
16 PEs	01 × 16	0.6200	0.6200	0.6896	0.6730	0.6896	0.6730
	02 × 08	0.7139	0.7139	0.7640	0.7580	0.7640	0.7580
	04 × 04	0.7462	0.7462	0.7925	0.7925	0.7925	0.7925
	08 × 02	0.7139	0.7139	0.7580	0.7640	0.7580	0.7640
	16 × 01	0.6200	0.6200	0.6730	0.6896	0.6730	0.6896
32 PEs	01 × 32	0.9256	0.9256	1.0952	1.0323	1.0952	1.0323
	02 × 16	1.1681	1.1681	1.3039	1.2738	1.3039	1.2738
	04 × 08	1.3067	1.3067	1.4186	1.4109	1.4186	1.4109
	08 × 04	1.3067	1.3067	1.4109	1.4186	1.4109	1.4186
	16 × 02	1.1681	1.1681	1.2738	1.3039	1.2738	1.3039
	32 × 01	0.9256	0.9256	1.0323	1.0952	1.0323	1.0952
64 PEs	02 × 32	1.7169	1.7169	2.0224	1.9115	2.0224	1.9115
	04 × 16	2.0942	2.0942	2.3477	2.3018	2.3477	2.3018
	08 × 08	2.2319	2.2319	2.4768	2.4768	2.4768	2.4768
	16 × 04	2.0942	2.0942	2.3018	2.3477	2.3018	2.3477
	32 × 02	1.7169	1.7169	1.9115	2.0224	1.9115	2.0224
128 PEs	04 × 32	3.0009	3.0009	3.4985	3.3315	3.4985	3.3315
	08 × 16	3.4660	3.4660	3.9020	3.8556	3.9020	3.8556
	16 × 08	3.4660	3.4660	3.8556	3.9020	3.8556	3.9020
	32 × 04	3.0009	3.0009	3.3315	3.4985	3.3315	3.4985
256 PEs	08 × 32	4.7872	4.7872	5.4845	5.2940	5.4845	5.2940
	16 × 16	5.1384	5.1384	5.8048	5.8048	5.8048	5.8048
	32 × 08	4.7872	4.7872	5.2940	5.4845	5.2940	5.4845
512 PEs	16 × 32	6.7924	6.7924	7.6100	7.4789	7.6100	7.4789
	32 × 16	6.7924	6.7924	7.4789	7.6100	7.4789	7.6100
1024 PEs	32 × 32	8.5435	8.5435	9.3733	9.3733	9.3733	9.3733

Table A.22: Average relative acceleration for several LU processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential multi-cycle PE as baseline.

Processors	Array Size	3-Ver-1	3-Ver-2	3-Ver-3	3-Ver-4	3-Ver-5	3-Ver-6
1 PE	01 × 01	0.0574	0.0574	0.0574	0.0574	0.0574	0.0574
2 PEs	01 × 02	0.0556	0.0556	0.0561	0.0561	0.0561	0.0561
	02 × 01	0.0556	0.0556	0.0561	0.0561	0.0561	0.0561
4 PEs	01 × 04	0.0523	0.0523	0.0538	0.0536	0.0538	0.0536
	02 × 02	0.0537	0.0537	0.0549	0.0549	0.0549	0.0549
	04 × 01	0.0523	0.0523	0.0536	0.0538	0.0536	0.0538
8 PEs	01 × 08	0.0468	0.0468	0.0496	0.0492	0.0496	0.0492
	02 × 04	0.0503	0.0503	0.0523	0.0522	0.0523	0.0522
	04 × 02	0.0503	0.0503	0.0522	0.0523	0.0522	0.0523
	08 × 01	0.0468	0.0468	0.0492	0.0496	0.0492	0.0496
16 PEs	01 × 16	0.0387	0.0387	0.0431	0.0421	0.0431	0.0421
	02 × 08	0.0446	0.0446	0.0477	0.0474	0.0477	0.0474
	04 × 04	0.0466	0.0466	0.0495	0.0495	0.0495	0.0495
	08 × 02	0.0446	0.0446	0.0474	0.0477	0.0474	0.0477
	16 × 01	0.0387	0.0387	0.0421	0.0431	0.0421	0.0431
32 PEs	01 × 32	0.0289	0.0289	0.0342	0.0323	0.0342	0.0323
	02 × 16	0.0365	0.0365	0.0407	0.0398	0.0407	0.0398
	04 × 08	0.0408	0.0408	0.0443	0.0441	0.0443	0.0441
	08 × 04	0.0408	0.0408	0.0441	0.0443	0.0441	0.0443
	16 × 02	0.0365	0.0365	0.0398	0.0407	0.0398	0.0407
	32 × 01	0.0289	0.0289	0.0323	0.0342	0.0323	0.0342
64 PEs	02 × 32	0.0268	0.0268	0.0316	0.0299	0.0316	0.0299
	04 × 16	0.0327	0.0327	0.0367	0.0360	0.0367	0.0360
	08 × 08	0.0349	0.0349	0.0387	0.0387	0.0387	0.0387
	16 × 04	0.0327	0.0327	0.0360	0.0367	0.0360	0.0367
	32 × 02	0.0268	0.0268	0.0299	0.0316	0.0299	0.0316
128 PEs	04 × 32	0.0234	0.0234	0.0273	0.0260	0.0273	0.0260
	08 × 16	0.0271	0.0271	0.0305	0.0301	0.0305	0.0301
	16 × 08	0.0271	0.0271	0.0301	0.0305	0.0301	0.0305
	32 × 04	0.0234	0.0234	0.0260	0.0273	0.0260	0.0273
256 PEs	08 × 32	0.0187	0.0187	0.0214	0.0207	0.0214	0.0207
	16 × 16	0.0201	0.0201	0.0227	0.0227	0.0227	0.0227
	32 × 08	0.0187	0.0187	0.0207	0.0214	0.0207	0.0214
512 PEs	16 × 32	0.0133	0.0133	0.0149	0.0146	0.0149	0.0146
	32 × 16	0.0133	0.0133	0.0146	0.0149	0.0146	0.0149
1024 PEs	32 × 32	0.0083	0.0083	0.0092	0.0092	0.0092	0.0092

Table A.23: Average relative efficiency for several LU processor arrays and different allocation matrixes, when $N_{max} = 500$, and using the sequential multi-cycle PE as baseline.

Bibliography

- [1] GNU Linear Programming Kit. <http://www.gnu.org/software/glpk/> Last seen in November, 2013.
- [2] IBM ILOG CPLEX Optimizer. ILOG CPLEX 10.0, Users Manual, 2006. <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/> Last seen in November, 2013.
- [3] Pico technology. <http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SymphonyC-Compiler.aspx> Last seen in November, 2013.
- [4] Polyhedral Loop Parallelization: LooPo. <http://www.infosun.fmi.uni-passau.de/cl/loopo/> Last seen in November, 2013.
- [5] Shail Aditya and Vinod Kathail. *Algorithmic Synthesis Using PICO: An Integrated Framework for Application Engine Synthesis and Verification from High Level C Algorithms*. Chapter 4, pages 53–74. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, 2008.
- [6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison–Wesley Longman, Second edition, 2007.
- [7] Christophe Alias, Alain Darté, and Alexandru Plesco. *Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA*. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 285–286, New Orleans, LA, USA, February, 2012.
- [8] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. *FPGA-Specific Synthesis of Loop-Nests with Pipelined Computational Cores*. *Microprocessors and Microsystems*, 36(8), pages 606–619, November, 2012.

- [9] Randy Allen and Ken Kennedy. *Automatic Translation of Fortran Programs to Vector Form*. ACM Transactions on Programming Languages and Systems (TOPLAS), 9(4):491–542, October, 1987
- [10] Altera Company. *Aria V Device Overview*. Data Sheet.
- [11] Altera Company. *Cyclone V Device Overview*. Data Sheet.
- [12] Altera Company. *Nios Processor Technical Documentation*. Data Sheet.
- [13] Uptal Banerjee, Shyh-Ching Chen, David J. Kuck, and Ross A. Towle. *Time and Parallel Processor Bounds for Fortran-like Loops*. IEEE Transactions on Computers, 28(9):660–670, September, 1979.
- [14] Cédric Bastoul. *Efficient Code Generation for Automatic Parallelization and Optimization*. In Proceedings of the 2nd IEEE International Symposium on Parallel and Distributed Computing (ISPDC), pages 23–30, Ljubljana, Slovenia, October, 2003
- [15] Cédric Bastoul. *Improving Data Locality in Static Control Programs*. PhD Thesis, University Paris 6, Pierre et Marie Curie, France, December, 2004.
- [16] Samuel Bayliss and George A. Constantinides. *Optimizing SDRAM Bandwidth for Custom FPGA Loop Accelerators*. In Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pages 195–204, Monterey, CA, USA, February, 2012.
- [17] Marcus Bednara, Frank Hannig, and Jürgen Teich. *Boundary Control: A New Distributed Control Architecture for Space-Time Transformed (VLSI) Processor Arrays*. In Proceedings of the 35th IEEE Asilomar Conference on Signals, Systems, and Computers, vol. 1 pages 468–474, Pacific Grove, CA, USA, November, 2001.
- [18] A. J. Bernstein. *Analysis of Programs for Parallel Processing*. IEEE Transactions on Electronic Computers, EC-15(5):757–763, October, 1966.

- [19] Aart J. C. Bik and Harry A. G. Wijshoff. *Implementation of Fourier-Motzkin Elimination*. Technical Report 94-42, Dept. of Computer Science, Leiden University, 1994.
- [20] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. *A Survey of Multicore Processors*. IEEE Signal Processing Magazine, 26(6), pages 27–37, November, 2009.
- [21] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer Publishing Company, 2007.
- [22] Uday Bondhugula, Albert Hartono, J. Ramanujam, and Ponnuswamy Sadayappan. *A Practical and Fully Automatic Polyhedral Program Optimization System*. In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 101–113, Tucson, AZ, USA, June, 2008.
- [23] Uday Bondhugula, J. Ramanujam, and Ponnuswamy Sadayappan. *Automatic Mapping of Nested Loops to FPGAs*. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 101–111, San Jose, CA, USA, March, 2007.
- [24] Uday Kumar Reddy Bondhugula. *Effective Automatic Parallelization and Locality Optimization using the Polyhedral Model*. PhD Thesis, Ohio State University, 2008.
- [25] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. *Spatial Computation*. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 14–26, Boston, MA, USA, October, 2004.
- [26] Betul Buyukkurt, Zhi Guo, and Walid A. Najjar. *Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs*. In Proceedings of the 2nd International Workshop on Applied Reconfigurable Computing (ARC), volume 3985 of Lecture Notes in Computer Science (LNCS), pages 401–412, Delft, The Netherlands, March, 2006.

- [27] Alain Darte, Steven Derrien, and Tanguy Risset. *Hardware-software Interface for Multi-Dimensional Processor Arrays*. In Proceedings of 16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP), pages 28–35, Samos, Greece, July, 2005.
- [28] Alain Darte, Leonid Khachiyan, and Yves Robert. *Linear Scheduling is Nearly Optimal*. *Parallel Processing Letters*, 1(2), pages 73-81, December, 1991.
- [29] Alain Darte, Tanguy Risset, and Yves Robert. *Synthesizing Systolic Arrays: Some Recent Developments*. In Proceedings of the International Conference on Application Specific Array Processors, pages 372–386, Barcelona, Spain, September, 1991.
- [30] Alain Darte and Frédéric Vivien. *Automatic Parallelization Based on Multi-Dimensional Scheduling*. Technical Report 94–24, Laboratoire de L'Informatique du Parallélisme, École Normale Supérieure de Lyon de Lyon, France, September, 1994.
- [31] Alain Darte and Frédéric Vivien. *Revisiting the Decomposition of Karp, Miller and Winograd*. In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP), pages 13–25, Strasbourg, France, July, 1995.
- [32] Alain Darte and Frédéric Vivien. *Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs*. In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT), pages 281–291, Boston, MA, USA, October, 1996.
- [33] Florent de Dinechin and Bogdan Pasca. *Designing Custom Arithmetic Data Paths with FloPoCo*. *IEEE Design and Test of Computers*, 28(4), pages 18–27, July, 2011.
- [34] Steven Derrien. *Platforms, Methodologies and Tools for Designing Reconfigurable Hardware Architectures*. PhD Thesis, L'Université de Rennes 1, December, 2011.
- [35] Steven Derrien and Patrice Quinton. *Hardware Acceleration of HMMER on FPGAs*. *Journal of Signal Processing Systems*, 58(1), pages 53–67, January 2010.

- [36] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset. *High-Level Synthesis of Loops Using the Polyhedral Model The MMAAlpha Software*, Chapter 12, pages 215–230. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, 2008.
- [37] Steven Derrien, Alexandru Turjan, Claudiu Zissulescu, Bart Kienhuis, and Ed F. Deprettere. *Deriving Efficient Control in Process Networks with Compaan/Laura*. *International Journal of Embedded Systems*, 3(3), pages 170–180, September, 2008.
- [38] Digilent Company. *Atlys Board Reference Manual*. Data Sheet.
- [39] Harald Devos. *Loop Transformations for the Optimized Generation of Reconfigurable Hardware*. PhD Thesis, Ghent University, February, 2008.
- [40] Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, and Heidi Ziegler. *Automatic Mapping of C to FPGAs with the DEFACTO Compilation and Synthesis System*. *Microprocessors and Microsystems*, 29(2–3), pages 51–62, April, 2005.
- [41] Clémentin Tayou Djamegni, Patrice Quinton, Sanjay Rajopadhye, Tanguy Risset, and Maurice Tchuenté. *A Reindexing Based Approach Towards Mapping of DAG with Affine Schedules onto Parallel Embedded Systems*. *Journal of Parallel and Distributed Computing*, 69(1), pages 1–11, January, 2009.
- [42] Scott C. Douglas. *Simple Adaptive Algorithms for Cholesky, LDL/sup T/, QR, and Eigenvalue Decompositions of Autocorrelation Matrices for Sensor Array Data*. In *Proceedings of the 35th IEEE Asilomar Conference on Signals, Systems, and Computers*, vol. 2 pages 1134–1138, Pacific Grove, CA, USA, November, 2001.
- [43] Hritam Dutta. *Mapping of Hierarchically Partitioned Regular Algorithms onto Processor Arrays*. Master Thesis, University of Erlangen-Nuremberg, October, 2004.
- [44] Hritam Dutta. *Synthesis and Exploration of Loop Accelerators for Systems-on-a-Chip*. PhD Thesis, University of Erlangen-Nuremberg, Germany, 2011.

- [45] Hritam Dutta, Frank Hannig, and Jürgen Teich. *Controller synthesis for Mapping Partitioned Programs on Array Architectures*. In Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS), volume 3894 of Lecture Notes in Computer Science (LNCS), pages 176–190, Frankfurt am Main, Germany, March, 2006.
- [46] Hritam Dutta, Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. *Efficient Control Generation for Mapping Nested Loop Programs onto Processor Arrays*. *Journal of Systems Architecture*, 53(5-6), pages 300–309, May, 2007.
- [47] Hritam Dutta, Frank Hannig, and Jürgen Teich. *A Formal Methodology for Hierarchical Partitioning of Piecewise Linear Algorithms*. Technical Report 03–2005, Department of Computer Science 12, University of Erlangen-Nuremberg, November, 2005.
- [48] Hritam Dutta, Frank Hannig, and Jürgen Teich. *Controller Synthesis for Mapping Partitioned Programs on Array Architectures*. Technical Report 04–2006, Department of Computer Science 12, University of Erlangen-Nuremberg, April, 2005.
- [49] Paul Feautrier. *Parametric Integer Programming*. *RAIRO Recherche Op'erationnelle*, 22(3), pages 243–268, 1988.
- [50] Paul Feautrier. *Dataflow Analysis of Array and Scalar References*. *International Journal of Parallel Programming*, 20(1), pages 23–53, February, 1991.
- [51] Paul Feautrier. *Some Efficient Solutions to the Affine Scheduling Problem – Part I One-Dimensional Time*. *International Journal of Parallel Programming*, 21(5), pages 313–348, October, 1992.
- [52] Paul Feautrier. *Some Efficient Solutions to the Affine Scheduling Problem – Part II Multi-Dimensional Time*. *International Journal of Parallel Programming*, 21(6), pages 389–420, December, 1992.
- [53] Paul Feautrier. *The Polytope Model, Past, Present, Future*. Keynote presentation to the 2009 Language and Compilers for Parallel Computing (LCPC) Workshop, October, 2009.

- [54] Dirk Fimmel and Renate Merker. *Determination of Processor Allocation in the Design of Processor Arrays*. *Microprocessors and Microsystems*, 22(3-4), pages 149–155, August, 1998.
- [55] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. *Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies*. *International Journal of Parallel Programming*, 34(3), pages 261–371, June, 2006.
- [56] Sylvain Girbal, Sami Yehia, Hugues Berry, and Olivier Temam. *Stream and Memory Hierarchy Design for Multi-Purpose Accelerators*. In *Proceedings of the 1st Workshop on SoC Architecture, Accelerators and Workloads (SAW)*, Bangalore, India, October, 2010.
- [57] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing, Second edition, 2002.
- [58] Martin Griehl and Christian Lengauer. *The Loop Parallelizer LooPo-Announcement*. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 1239 of *Lecture Notes in Computer Science (LNCS)*, pages 603–604, San Jose, CA, USA, August, 1997.
- [59] Armin Größlinger. *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation*. PhD Thesis, University of Passau, Germany, July, 2009.
- [60] Rohan Grover, Dimitris A. Pados, and Michael J. Medley. *Adaptive Optimization of Binary/Quaternary CDMA Signatures in Multipath Fading Environments*. In *Proceedings of the IEEE Military Communications Conference (MILCOM)*, vol. 2, pages 930–935, Atlantic City, NJ, USA, October, 2005.
- [61] Anne-Claire Guillou, Patrice Quinton, and Tanguy Risset. *Hardware Synthesis for Multi-Dimensional Time*. In *Proceedings of the IEEE 14th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 40–50, The Hague, The Netherlands, June, 2003.

- [62] Rajesh Gupta and Forrest Brewer. *High-Level Synthesis: A Retrospective*, Chapter 2, pages 13–28. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, 2008.
- [63] Sumit Gupta, Rajesh K. Gupta, Nikil D. Dutt, and Alexandru Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.
- [64] Frank Hannig. *Scheduling Techniques for High-Throughput Loop Accelerators*. PhD Thesis, University of Erlangen-Nuremberg, Germany, August, 2009.
- [65] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. *PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications*. In *Proceedings of the 4th International Workshop on Applied Reconfigurable Computing (ARC)*, volume 4943 of *Lecture Notes in Computer Science (LNCS)*, pages 287–293, London, UK, March, 2008.
- [66] Scott Hauck and Andre DeHon. *Reconfigurable Computing The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers, 2007.
- [67] Yunhui He. *Real-time Nonlinear Facial Feature Extraction using Cholesky Decomposition and QR Decomposition for Face Recognition*. In *Proceedings of the International Conference on Electronic Computer Technology (ICECT)*, pages 306–310, Macau, China, February, 2009.
- [68] Francois Irigoin and Remi Triolet. *Supernode Partitioning*. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 319–329, San Diego, California, USA, January, 1988.
- [69] Marta Jiménez. *Multilevel Tiling for Non-Rectangular Iteration Spaces*. PhD Thesis, Universitat Politècnica de Catalunya, May, 1999.
- [70] Zille Huma Kamal, Ajay Gupta, Leszek Lilien, and Ashfaq Khokhar. *An Efficient Map Classifier for Sensornets*. In *Proceedings of the 13th IEEE International Conference on High Performance Computing (HiPC)*, volume 4297 of *Lecture Notes in Computer Science (LNCS)*, pages 287–293, Bangalore, India, December, 2008.

- [71] Richard M. Karp, Raymon E. Miller, and Shmuel Winograd. *The Organization of Computations for Uniform Recurrence Equations*. *Journal of the ACM*, 14(3), pages 563–590, July, 1967.
- [72] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, Mukund Sivaraman *Pico-NPA: High-level synthesis of Nonprogrammable Hardware Accelerators*. *Journal of VLSI Signal Processing*, 31(2), pages 127–142, June, 2002.
- [73] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. *Compaan: Deriving Process Networks from MATLAB for Embedded Signal Processing Architectures*. In *Proceedings of the 8th International Workshop on Hardware/Software CoDesign (CODES)*, pages 13–17, San Diego, CA, USA, May, 2000.
- [74] Dmitriy Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. *Hardware Cost Analysis for Weakly Programmable Processor Arrays*. In *Proceedings of the International Symposium on System-on-Chip (SoC)*, pages 1–4, Tampere, Finland, November, 2006.
- [75] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Casçaval. *How Much Parallelism is There in Irregular Applications?* In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 3–14, Raleigh, NC, USA, 2009.
- [76] Sun Yang Kung. *VLSI Array Processors*. Prentice-Hall, 1987.
- [77] Leslie Lamport. *The Parallel Execution of DO Loops*. *Communications of the ACM*, 17(2), pages 83–93, February, 1974.
- [78] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhye. *Advanced Systolic Design*, Chapter 5, pages 657–692. *Digital Signal Processing for Multimedia Systems*. Signal Processing and Communications, 1999.
- [79] Peizong Lee and Zvi Meir Kedem. *Synthesizing Linear Array Algorithms from Nested FOR Loop Algorithms*. *IEEE Transactions on Computers*, 37(12):1578–1598, December, 1988.

- [80] Christian Lengauer. *Loop Parallelization in the Polytope Model*. In Proceedings of the 4th International Conference on Concurrency Theory (CONCUR), volume 715 of Lecture Notes in Computer Science (LNCS), pages 398–416, Hildesheim, Germany, August, 2008.
- [81] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005.
- [82] Kuang-Hao Lin, Robert C. Chang, Chien-Lin Huang, Feng-Chi Chen, and Shih-Chun Lin. *Implementation of QR Decomposition for MIMO OFDM Detection Systems*. In Proceedings of the 15th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pages 57–60, Malta, September, 2008.
- [83] Peng Liu, Pengheng Zhu Yan Du, and Wei Zhang. *A New Efficient MIMO Detection Algorithm Based on Cholesky Decomposition*. In Proceedings of the 6th International Conference on Advanced Communication Technology (ICACT), pages 264–268, Phoenix Park, Korea, February, 2004.
- [84] Qiang Liu, Dong Tong, and Xu Cheng. *A New Systolic Architecture without Global Broadcast*. In Proceedings of the 7th International Conference on Signal Processing (ICSP), vol. 1, pages 527–531, Beijing, China, August, 2004.
- [85] Mojtaba Mehrara, Thomas Jablin, Dan Upton, David August, Kim Hazelwood, and Scott Mahlke. *Multicore Compilation Strategies and Challenges*. IEEE Signal Processing Magazine, 26(6), pages 55–63, November, 2009.
- [86] Wolfgang Meisl. *Practical Methods for Scheduling and Allocation in the Polytope Model*. Master Thesis, University of Passau, Germany, September, 1996.
- [87] Étienne Mémi and Tanguy Risset. *Hardware Driven Considerations for Energy Based Applications*. Technical Report 1220, Université de Rennes 1, IRISA, April, 1999.
- [88] Dan I. Moldovan and Jose A. B Fortes. *Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays*. IEEE Transactions on Computers, C-35(1):1–12, January, 1986.

- [89] Chrysostomos L. Nikias, Andreas P. Chrysafis, and Anastasios N. Venetsanopoulos. *The LU Decomposition Theorem and its Implications to the Realization of Two-Dimensional Digital Filters*. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, 33(3):694–711, June, 1985.
- [90] Alex Orailoglu and Daniel D. Gajski. *Flow Graph Representation*. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference (DAC)*, pages 503–509, Las Vegas, NV, USA, June, 1986.
- [91] Alexandru Plesco. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*. PhD Thesis, École Normale Supérieure de Lyon, September, 2010.
- [92] John G. Proakis and Dimitris K. Manolakis. *Digital Signal Processing : Principles, Algorithms and Applications*. Prentice-Hall, Fourth edition, 2006.
- [93] William Pugh. *Uniform Techniques for Loop Optimization*. In *Proceedings of the 5th International Conference on Supercomputing (ICS)*, pages 341–352, Cologne, West Germany, June, 1991.
- [94] Patrice Quinton, Tanguy Risset, Katell Morin-Allory, and David Cachera. *Designing Parallel Programs and Integrated Circuits*. In *Proceedings of the 8th International Mathematica Symposium (IMS)*, pages 1–13, Avignon, France, June, 2006.
- [95] Sailesh K. Rao. *Regular Interactive Algorithms and their Implementations on Processor Arrays*. PhD Thesis, Stanford University, 1986.
- [96] Holger Ruckdeschel, Hritam Dutta, Frank Hannig, and Jürgen Teich. *Automatic FIR Filter Generation for FPGAs*. In *Proceedings of the 5th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 51–61, Samos, Greece, July, 2005.

- [97] Rizos Sakellariou. *On the Quest for Perfect Balance in Loop-Based Parallel Computations*. PhD Thesis, University of Manchester, March, 1996.
- [98] Tsutomu Sasao. *Memory-Based Logic Synthesis*. Springer Publishing Company, March, 2011.
- [99] Dinesh C. Suresh, Satya R. Mohanty, Walid A. Najjar, Laxmi N. Bhuyan, and Frank Vahid. *Loop Level Analysis of Security and Network Applications*. In Proceedings of the 6th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), pages 44–50, Anaheim, CA, USA, February, 2003.
- [100] The Polylib Team. *Polylib User's Manual*. IRISA, April, 2002.
- [101] Paul Teehan, Mark Greenstreet, and Guy Lemieux. *A Survey and Taxonomy of GALS Design Styles*. IEEE Design and Test of Computers, 24(5), pages 418 –428, September, 2007.
- [102] Jürgen Teich. *A Compiler for Application-Specific Processor Arrays*. PhD Thesis, 1993.
- [103] Jürgen Teich, Jorg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schroder-Preikschat, and Gregor Snelting. *Invasive Computing: An Overview*. Chapter 11, pages 241–268. Multiprocessor System-on-Chip – Hardware Design and Tool Integration Springer Publishing Company, 2011.
- [104] Jürgen Teich, Alexandru Tanase, and Frank Hannig. *Symbolic Parallelization of Loop Programs for Massively Parallel Processor Arrays*. In Proceedings of the IEEE 24th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP), pages 1–9, Washington, DC, USA, June, 2013.
- [105] Jürgen Teich and Lothar Thiele. *Exact Partitioning of Affine Dependence Algorithms*. In Proceedings of the Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS), volume 2268 of Lecture Notes in Computer Science (LNCS), pages 135–153, March, 2002.

- [106] Lothar Thiele. *On the Hierarchical Design of VLSI Processor Arrays*. In Proceedings of the International Symposium on Circuits and Systems (ISCAS), vol. 3, pages 2517–2520, Espoo, Finland, June, 1988.
- [107] Lothar Thiele. *Resource Constrained Scheduling of Uniform Algorithms*. Journal of VLSI Signal Processing, 10(3), pages 295–310, August, 1995.
- [108] Lothar Thiele and Vwani P. Roychowdhury. *Systematic Design of Local Processor Arrays for Numerical Algorithms*. In Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures, vol. A, pages 329–229, Amsterdam, The Netherlands, January, 1991.
- [109] Konrad Trifunovic, Albert Cohen, David Edelsohn, Li Feng, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. *Graphite: Two Years After First Lessons Learned From Real-World Polyhedral Compilation*. In Proceedings of the 2nd International Workshop GCC Research Opportunities (GROW), pages 4–19, Pisa, Italy, January, 2010.
- [110] Hervé Le Verge, Christophe Maurus, and Patrice Quinton. *The Alpha Language and its Use for the Design of Systolic Arrays*. Journal of VLSI Signal Processing Systems, 3(3), pages 173–182, September, 1991.
- [111] Sabine Wetzel. *Automatic Code Generation in the Polytope Model*. Master Thesis, University of Passau, Germany, September, 1996.
- [112] Michael Wolf and Monica Lam. *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*. IEEE Transactions on Parallel and Distributed Systems, 2(4):452–471, October, 1991.
- [113] Hsiao-Chun Wu, Shih Yu Chang, and Tho Le-Ngoc. *Efficient Rank-Adaptive Least-Square Estimation and Multiple-Parameter Linear Regression Using Novel Dyadically Recursive Hermitian Matrix Inversion*. In Proceedings of the International Wireless Communications and Mobile Computing (IWCMC), pages 1064–1069, Crete Island, Greece, August, 2008.

- [114] Dan Xia, Huan He, Youyun Xu, and Yueming Cai. *A Novel Construction Scheme with Linear Encoding Complexity for LDPC Codes*. In Proceedings of 4th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM), pages 1–4, Dalian, China, October, 2008.
- [115] Xilinx Company. *LogiCore IP CORDIC*. Data Sheet.
- [116] Xilinx Company. *LogiCore IP Divider Generator*. Data Sheet.
- [117] Xilinx Company. *MicroBlaze Micro Controller System*. Data Sheet.
- [118] Xilinx Company. *Spartan-6 Family Overview*. Data Sheet.
- [119] Xilinx Company. *Virtex-4 Family Overview*. Data Sheet.
- [120] Xilinx Company. *Virtex-6 Family Overview*. Data Sheet.
- [121] Jingling Xue and Christian Lengauer. *The Synthesis of Control Signals for One-Dimensional Systolic Arrays*. Integration, the VLSI Journal, 14(1), pages 1–32, November, 1992.
- [122] Depeng Yang, Gregory D. Peterson, Husheng Li, and Junqing Sun. *An FPGA Implementation for Solving Least Square Problem*. In Proceedings of 17th IEEE Symposium on Field Programmable Custom Computing Machines, pages 303–306, Napa, CA, USA, April, 2009.
- [123] Claudiu Zissulescu, Bart Kienhuis, and Ed Depretere. *Expression Synthesis in Process Networks Generated by Laura*. In Proceedings of 16th IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP), pages 15–21, Samos, Greece, July, 2005.



CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL IPN

UNIDAD TAMAULIPAS

Cd. Victoria, Tamaulipas, a 7 de marzo de 2014.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará el C. José Roberto Pérez Andrade, declaramos que hemos revisado la tesis titulada:

"Síntesis de arreglo de procesadores para ciclos anidados con espacios de iteración no-rectangulares usando el modelo del politopo"


Y consideramos que cumple con los requisitos para obtener el grado de Doctor en Ciencias en Computación.

Atentamente,

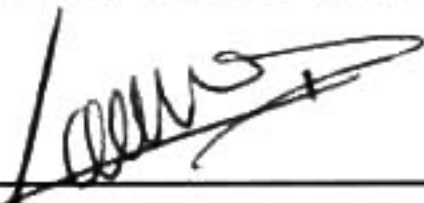
Dr. José Juan García Hernández



Dr. Arturo Díaz Pérez



Dr. José Javier Díaz Carmona



Dr. René Armando Cumplido Parra



Dr. José Luis Tecpanecatí Xihuitl



Dr. César Torres Huitzil





CINVESTAV - IPN
Biblioteca Central



SSIT0012342