MAPPING REGULAR RECURSIVE ALGORITHMS TO FINE-GRAINED PROCESSOR ARRAYS

BY

KUMAR NANJUNDA GANAPATHY

B.Tech., Indian Institute of Technology at Madras, 1987
M.S., University of Massachusetts at Amherst, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ABSTRACT

With the continuing growth of VLSI technology, special-purpose parallel processors have become a promising approach in the quest for high performance. Fine-grained processor arrays have become popular as they are suitable for solving problems with a high degree of parallelism, and can be inexpensively built using custom designs or commercially available field programmable gate arrays (FPGA). Such specialized designs are often required in portable computing and communication systems with real-time constraints, as software-controlled processors often fail to provide the necessary throughput. This thesis addresses many issues in designing such application-specific systems built with fine-grained processor arrays for regular recursive uniform dependence algorithms. A uniform dependence algorithm consists of a set of indexed computations and a set of uniform dependence vectors which are independent of the indices of computations. Many important applications in signal/image processing, communications, and scientific computing can be formulated as uniform dependence algorithms.

The first part of this thesis addresses the problem of designing algorithm-specific processor arrays. A systematic parameter-based method, called the General Parameter Method (GP-M), to design *optimal*, lower-dimensional processor arrays for uniform dependence algorithms has been developed. The GPM can be used to derive optimal arrays for any user-specified objective expressed in terms of the parameters. The proposed approach employs an efficient search technique to explore the design space and arrive at the optimal designs. Equivalence between the parameter and dependence-based methods [1, 2, 3] can be used to find optimal designs in the dependence-based approaches. The GPM has also been extended to derive optimal two-level pipelined algorithm-specific processor arrays. Such two-level pipelined arrays can be clocked at higher rates than can nonpipelined designs for real-time applications.

The second part of this thesis presents a parallel VLSI architecture for a general-purpose coprocessor for uniform dependence algorithms. The architecture consists of a linear array of processors and a linear chain of buffer memories organized as FIFO queues to store the buffered data. Such an architecture is advantageous from the point of view of scalability and wafer-level integration. A distinguishing feature is the assumption of a limited-bandwidth interface to external memory modules for accessing the data. Such an assumption allows the coprocessor to be integrated easily into existing systems. Efficient techniques to partition the dependence graph into blocks, sequence the blocks through the buffer memory to reduce the number of data accesses to main memory, and map the blocks using GPM have been developed. An important result obtained is the square-root relationship between clock-rate reduction and area of the coprocessor under fixed main-memory bandwidth. From the square-root relationship, it can found that the system yield improves with the area of the coprocessor when chip yield decreases as the inverse square of the clock frequency. Results on matrix-product and transitive-closure applications indicate that the coprocessor can be used to deliver higher speedup or lower clock rate than a reference one-processor design. Thus, the coprocessor can be used as a general-purpose back-end accelerator for loop-based matrix algorithms.

DEDICATION

To my parents, P. Bagyam and R. Ganapathy

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support of many people, and it is with pleasure and gratitude that I acknowledge their efforts. First, I would like to thank my advisor, Professor Benjamin Wah, for his advice and guidance. If this thesis amounts to something, it is probably so because of his persistence and attention to detail. I would like to express my sincere gratitude to Professor Michael Loui for his advice and encouragement over the past few years. Our weekly meetings were very enjoyable and a learning experience for me. I thank him for his thorough reading of my papers and this thesis, which greatly improved the presentation and quality of this work. I would like to thank the other members of my committee, Professors Prith Banerjee and Pen Yew, for their comments and suggestions on this work. I would like to thank Professors Jose Fortes and Weijia Shang for reading and providing feedback on parts of this work. I would also like to acknowledge the financial support I received from the following sponsors: IBM (graduate fellowship), Joint Services Electronics Program Contract N00014-90-J1270, National Aeronautics and Space Administration Grant NCC-2-481, and National Science Foundation Grant MIP 88-10584.

Many thanks to my current and former colleagues – Steve Schwartz, Vijay Karamcheti, Arthur Ieumwananonthachai, Tanay Karnik, Chin-Chi Teng, Lon-Chan Chu, Akiko Aizawa, Yi Shang, Yo-Jen Chang – for many helpful discussions about ideas presented in this thesis. I certainly cannot ever thank my friend Bapi enough for the support, encouragement and hope he has given me over the past many, many years. To my dear friends who have helped me cope with the pressures and frustrations of graduate school – Shanthi, Vijay, Nagi, Venkat, Vivek, Rashmi, Ajit, Pallab – thanks for your encouragement, support, and your friendship! I would especially like to thank Pallab for being there always, and for the late-night "chai" discussions and perspectives on life. I wish to thank Donna Guzy for her support, help and delicious cakes and pies.

I cannot express in words how much my wife has meant to me, and how her love and understanding sustained me during the final phases of graduate life. My other family members – sister Latha and Uncle Sankaran – have been a wonderful source of love, encouragement and motivation throughout my life. I would always treasure the part of my life in Amherst with my friend Rajiv Jindani, who succumbed due to leukemia. His ideas and thoughts have changed my life greatly.

Finally, I wish to thank my parents, who have sacrificed so much to get me to this stage in my career, and for their never-ending love, encouragement, faith, and care.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1
   1.1 Notations 4
   1.2 Algorithm Model 5
      1.2.1 Relation to nested-loop programs 6
   1.3 Overview of the Thesis 9
   1.4 Contributions of This Thesis 15

2. GENERAL PARAMETER METHOD 17
   2.1 Previous Work 18
   2.2 General Parameter Method: Parameters 21
   2.3 General Parameter Method: Constraint Equations 24
      2.3.1 Constraints for correct systolic processing 25
      2.3.2 Constraints for linearly dependent dependence vectors 27
      2.3.3 Constraints to govern valid space-time mappings 31
      2.3.4 Constraints in preloaded data 33
   2.4 Design Method 33
      2.4.1 Formulation of the search problem 33
      2.4.2 Pruning strategy 35
      2.4.3 Search procedure 38
   2.5 Summary 39

3. APPLICATIONS OF GENERAL PARAMETER METHOD 42
   3.1 Algebraic Path-finding Problems: Transitive Closure 42
      3.1.1 Performance attributes and constraints 44
      3.1.2 Time-optimal and processor-optimal linear array designs 49
      3.1.3 Processor-time trade-offs 56
   3.2 Three-dimensional Cube Graph Algorithms: Matrix Product 59
      3.2.1 Performance attributes 60

3.2.2 Time-optimal linear array designs 63
3.2.3 Load and drain times for stationary data in linear arrays 65
3.2.4 Processor-optimal designs 76
3.2.5 Processor-time trade-offs 79
3.3 Summary 81

4. SYNTHESIS OF PROCESSOR ARRAYS WITH PIPELINED ARITHMETIC UNITS 82
4.1 Review of General Parameter Method 83
4.2 New Constraints for Pipelined Functional Units 85
4.2.1 Minimum initiation interval greater than one 86
4.3 Pipelined Designs for n-D Meshes 87
4.4 Applications 89
4.4.1 Evaluation metric 89
4.4.2 Matrix product: results 90
4.4.3 Transitive closure: results 94
4.5 Comparisons with Existing Work 96
4.6 Summary 98

5. COPROCESSOR DESIGN 99
5.1 Coprocessor Architecture and Rationale 101
5.1.1 Coprocessor architecture 101
5.1.2 Proposed partitioning of the coprocessor architecture 105
5.1.3 Design approach 105
5.1.4 Comparison to existing architectures 106
5.2 Target Algorithms 108
5.3 Mapping Process 111
5.3.1 Partitioning 112
5.3.2 Sequencing blocks through the access-unit 118
5.3.3 Executing a block in the processor array 122
5.3.4 Overview of mapping process 124
5.4 Applications: Matrix Product and Transitive Closure 124
5.4.1 Evaluation metrics 125
5.4.2 Relationship between clock-rate reduction and area index 130
5.4.3 Area allocation between AU and PA 131
5.4.4 Cost-performance trade-offs: matrix product 132
5.4.5 Cost-performance trade-offs: transitive closure 138
5.5 Discussion 143
5.5.1 Yield 143
5.5.2 Speedup 144
5.5.3 Power 145
5.6 Summary 146

6. CONCLUSIONS AND FUTURE WORK 147
6.1 Conclusions 147
6.2 Future Work 149

REFERENCES 152

VITA 158

LIST OF TABLES

Table		Page
1.1:	A sample of applications that can be represented as uniform dependence algorithms, or can be converted to uniform dependence algorithms	10
2.1:	Comparison between dependency-based and parameter-based methods.	41
3.1:	T_{comp}-optimal linear processor arrays for finding transitive closures of $N \times N$ matrices.	50
3.2:	T_c-optimal linear processor arrays for finding transitive closures of $N \times N$ matrices.	50
3.3:	$\#PE$-optimal linear processor arrays for finding transitive closures of $N \times N$ matrices. (Parameters for GPM are shown in Theorem 3.2.)	53
3.4:	T_{comp} optimal linear arrays for product of two $N \times N$ matrices.	64
3.5:	T_c-optimal linear PAs for product of two $N \times N$ matrices.	64
3.6:	$\#PE$ optimal linear arrays for product of two $N \times N$ matrices.	78
3.7:	Feasible designs found by DM ($T_c = (T_{load}, T_{comp}, T_{drain})$).	79

LIST OF FIGURES

Figure Page

1.1: Overview and organization of this thesis: Objectives and resource constraints considered in the design process. 11

2.1: Application of GPM to find optimal designs in DM. 21

2.2: Proof of Theorem 1: Data movement between variables 1 and 2. For $p > 1$, token $Z_2(\vec{I} - p\,\vec{d_2})$ becomes $Z_2(\vec{I} - \vec{d_2})$ as it travels towards PE A. . 26

2.3: The dependency loop: $\sum_{j=1}^{r} \alpha_{i,j}\,\vec{d_j} = 0$ or $\sum_{j=1}^{j_1}(-\alpha_{i,j})\,\vec{d_j} = \sum_{j_1+1}^{r} \alpha_{i,j}\,\vec{d_j}$. . 28

3.1: PE allocation with $\vec{k}_1$, $\vec{k}_2 \geq 0$ and $\vec{k}_3'' \leq 0$. 45

3.2: Linear processor array for finding the transitive closure of a 3×3 matrix using parameters $(t_1, t_2, t_3) = (1, 1, 2)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. The array is optimal for minimum T_{comp}, minimum $\#PE$, and minimum $\#PE^x \times T_{comp}^y$, $x, y \geq 1$. The PE used is the same as in Lee and Kedem's design [3]. 54

3.3: Linear processor array for finding the transitive closure of a 3×3 matrix using parameters $(t_1, t_2, t_3) = (1, 2, 1)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. The array is optimal for minimum T_c, minimum $\#PE$, and minimum $\#PE^x \times T_c{}^y$, $x, y \geq 1$. The PE used is the same as in Lee and Kedem's design [3]. 55

3.4: Performance trade-offs: Variation in minimum $\#PE$ with time bound T_{comp}^{ub} and variation in minimum T_{comp} with processor bound $\#PE^{ub}$. The plots are given for three problem sizes $N = 100$, 200 and 300. . . . 58

3.5: Performance trade-offs: Variation in $\#PE$ with T_c or T_{comp}. The plots are given for two problem sizes $N = 100$ and 200. 59

3.6: PE allocation with $\vec{k}_1$, $\vec{k}_2 \geq 0$ and $\vec{k}_3 \leq 0$. 61

3.7: Draining $X = \sum_{i=1}^{l} x_i$ tokens distributed in a linear array. Each PE has k_1 left output ports and k_2 right output ports, and all data have to be drained out through the boundary PEs: PE_1 and PE_l. 66

3.8: Eliminating zig-zag paths in the optimal schedule. The dashed line represents the new schedule of token d from PE_i to PE_l without direction changes. 71

3.9: Draining set L left from PE_1 and set R right from PE_l. The dashed line shows the modified schedule. 72
3.10: Greedy draining of tokens from PEs. The dashed line shows the modified optimal schedule where token d is moved in a greedy way. 74
3.11: Variation of $T_d^L(i)$ and $T_d^R(i)$, $i = 1, \ldots, X$. The total drain time is minimized when $L = \{d_1, \ldots, d_b\}$ and $R = \{d_{b+1}, \ldots, d_X\}$ such that $T_d^L(b) = T_d^R(b)$. 75
3.12: Linear processor array for finding the product $\mathbf{C}$ of two 4×4 matrices $\mathbf{A}$ and $\mathbf{B}$ using parameters $(t_1, t_2, t_3) = (1, 2, 2)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. The array is optimal for minimum T_{comp} and minimum T_c. 77
3.13: Performance trade-offs: Variation in $\#PE$ with T_c or T_{comp}. The plots are given for two problem sizes $N = 50$ and 100. 80
4.1: Linear PA for 4×4 matrix product with nonpipelined functional units. 85
4.2: Pipelining is equivalent to adding delays to dependent data. 86
4.3: Matrix Product: Ratio of clock periods $R = \frac{t_c^S}{t_c^1}$ in order for the the S-pipelined and nonpipelined designs to have the same performance, i.e., $t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1$. 91
4.4: A PA for matrix multiplication with 3-stage PFUs. Note that this is obtained by a cyclic interchange of the inputs as compared to those shown in Figure 4.1. 92
4.5: Processor-time trade-offs for matrix product: Variation in $\#PE$ with T_{comp}. 93
4.6: Transitive Closure: Ratio of clock periods $R = \frac{t_c^S}{t_c^1}$ in order for the the S-pipelined and nonpipelined designs to have the same performance, i.e., $t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1$. 94
4.7: Linear PA for transitive closure with 3-stage PFUs with $(t_1, t_2, t_3) = (1, 1, 3)$, $(k_1, k_2, k_3) = (0, -1, 1)$. 95
4.8: Processor-time trade-offs for transitive closure: Variation in $\#PE$ with T_{comp}. 96
4.9: Relationship between proposed and existing methods [4, 5] of synthesizing PAs with PFUs. 97
5.1: Coprocessor architecture proposed to solve a class of algorithms modeled by uniform recurrences. 102
5.2: Proposed partitioning of the coprocessor architecture into four chip types. 105
5.3: Proposed approach for designing the coprocessor architecture. 106
5.4: Comparing the coprocessor to other architectures with limited memory bandwidth. 108
5.5: Cubical-mesh dependence graph for computing the product of two $N \times N$ matrices. 110
5.6: Cubical-mesh dependence graph for computing the transitive closure of an $N \times N$ matrix [6, 7]. The figure is drawn for $N = 4$. 111
5.7: Algorithms targeted for the coprocessor. The words URE, ARE, and NURE stand for Uniform Recurrence Equations, Affine Recurrence Equations, and Nonuniform Recurrence Equations, respectively. 112

5.8: Hyperplane partitioning of dependence graphs. The index set is the 2-D plane. The vectors $\vec{p}_1$, $\vec{p}_2$, and $\vec{p}_3$ correspond to a family of regularly spaced hyperplanes normal to them. The vectors determine only the orientation of the hyperplanes, i.e., block shape, and not the spacing between them. 114
5.9: A pair of cyclically dependent blocks. Block 2 depends on block 1 through dependence $\vec{d}_1$, while block 1 depends on block 2 through $\vec{d}_2$. Hence, partitioning vector $\vec{p}_i$ is invalid. 115
5.10: Partitioned dependence graph for the matrix-product problem. The partitioning matrix $\mathbf{P}$ is equal to the dependence matrix $\mathbf{D}$. 116
5.11: Partitioned multimesh dependence graph of the transitive-closure problem. The partitioning matrix is equal to the identity matrix $\mathbf{I}_3$. The figure is drawn for $N = 4, m = 2$. 117
5.12: Storage-window movement in the block-level DG for the matrix-product problem. The darkened areas show the storage window. For $N = 6m$, $p = 9$, there are 24 tiles. 121
5.13: Storage-window movement in the block-level DG for the transitive-closure problem. The darkened areas show the storage window. For $N = 4m$, $p = 4$, there are 36 tiles. 122
5.14: Data-input patterns for a linear array that computes the product of two 4×4 matrices. 123
5.15: Mapping the MMG of an algorithm to the coprocessor architecture. . . . 124
5.16: Area allocation between AU and PA of the coprocessor for masking MM latency fully. The area of the reference design is equal to 230 words for $A_{pe} = 100$ and 1130 words for $A_{pe} = 1000$ (Eq. (5.13)). 131
5.17: Cost-performance trade-offs of the coprocessor on the matrix-product problem. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = CRR^3_{max} = CRR^4_{max} = 512.5$, $CRR^5_{max} = CRR^6_{max} = 51.7$. The values of critical $AreaIndex$ in megawords beyond which clock-rate reduction saturates are: $A^1_{crit} = 105.9, A^2_{crit} = 111.4, A^3_{crit} = 140, A^4_{crit} = 155, A^5_{crit} = 1.44, A^6_{crit} = 2.97$. 133
5.18: Effect of local memory per PE (l) on cost-performance trade-offs for the matrix-product problem. Local memory in the PE is used to simulate a set of virtual PEs reducing the PE-count. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = 512.5$, $CRR^3_{max} = CRR^4_{max} = 445.6$, $CRR^5_{max} = CRR^6_{max} = 44.9$. The values of critical $AreaIndex$ in megawords are: $A^1_{crit} = 107.7, A^2_{crit} = 112.8, A^3_{crit} = 428.05, A^4_{crit} = 433.1, A^5_{crit} = 4.93, A^6_{crit} = 5.53$. 135

5.19: Effect of bandwidth (B_{MM}) on cost-performance trade-offs for the matrix-product problem. The values of maximum clock-rate reduction for plots $2, 3, 4$ are: $CRR^2_{max} = 1280.5$, $CRR^3_{max} = 512.5$, $CRR^4_{max} = 256.5$. The values of critical *AreaIndex* in megawords are: $A^2_{crit} = 211.6, A^3_{crit} = 142.8, A^4_{crit} = 123$. Plot 1 does not saturate as the bandwidth to MM is higher than the data rate required by PA. 136
5.20: Sensitivity of clock-rate reduction of coprocessor to changes in AU size for the matrix-product problem. If ALPHA < 1, the PEs are idle due to insufficient data accessed from MM. If ALPHA > 1, chip area is wasted by the extra memory in AU. The maximum clock-rate reduction for all curves is 512.5. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = A^2_{crit} = 142.8, A^3_{crit} = 175.1, A^4_{crit} = 770.7, A^5_{crit} = 2463.8$. . . 137
5.21: Cost-performance trade-offs for the transitive-closure problem. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = CRR^3_{max} = 342$, $CRR^4_{max} = CRR^5_{max} = CRR^6_{max} = 34.8$. The values of critical *AreaIndex* in megawords beyond which clock-rate reduction saturates are: $A^1_{crit} = 140, A^2_{crit} = 144, A^3_{crit} = 170.6, A^4_{crit} = 1.44, A^5_{crit} = 1.73, A^6_{crit} = 4.6$. 139
5.22: Effect of local memory per PE (l) on cost-performance trade-offs for the transitive-closure problem. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = CRR^3_{max} = 310.9$, $CRR^4_{max} = CRR^5_{max} = CRR^6_{max} = 31.6$. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = 426.1, A^2_{crit} = 427, A^3_{crit} = 436.3, A^4_{crit} = 4.23, A^5_{crit} = 4.33, A^6_{crit} = 5.26$. 140
5.23: Effect of bandwidth (B_{MM}) on cost-performance trade-offs for the transitive-closure problem. The values of maximum clock-rate reduction for plots $2, 3, 4$ are: $CRR^2_{max} = 854$, $CRR^3_{max} = 342$, $CRR^4_{max} = 171.3$. The values of critical *AreaIndex* in megawords are: $A^2_{crit} = 215.4, A^3_{crit} = 144, A^4_{crit} = 123$. Plot 1 does not saturate as the bandwidth to MM is higher than the data rate required by PA. 141
5.24: Sensitivity of cost-performance trade-offs to changes in AU size for the transitive-closure problem. The maximum clock-rate reduction for all curves is 342. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = A^2_{crit} = 144, A^3_{crit} = 177.8, A^4_{crit} = 780, A^5_{crit} = 2498.3$. 142
5.25: Chip yield y versus clock frequency f_{clk}; $y \geq \frac{c}{f_{clk}}$ (Region 1) to reduce the expected number of chips tested (Eq. (5.16)). 144

1. INTRODUCTION

In the last decade, the rapid development of VLSI computing techniques has had a significant impact on the development of novel computer architectures. One class of architectures, the so-called *systolic arrays*, first introduced by Kung [8, 9], has gained popularity because of its ability to exploit massive parallelism and pipelining to achieve high performance. Informally, a systolic system can be envisaged as an array of synchronized processors (or processing-elements, abbreviated as PEs) which process data in parallel by passing them from PE to PE in a regular, rhythmic fashion. Systolic arrays have balanced, uniform, grid-like architectures of special PEs that process data like an n-dimensional pipeline. In physiology, the term *systolic* describes the contraction (systole) of the heart, which regularly sends blood to all of the cells in the body. Analogously, a systolic computer performs operations in a repetitive, rhythmic manner.

The fundamental concept behind a systolic architecture is that the von Neumann bottleneck is greatly alleviated by repeated use of a fetched data item in a physically distributed array of processing elements [9]. The regularity of these arrays leads to inexpensive and dense VLSI implementations, which imply high performance and low cost. Application-specific processor arrays fit naturally into the concept of a hardware library [10], where functional units are in relation to the host computer as subroutines from a software library are to production code.

Systolic arrays have been designed for a wide variety of computationally intensive problems in signal processing, numerical problems, pattern recognition, database and dictionary machines, graph algorithms, and automata. To meet the performance requirements of these applications, it is often necessary to dedicate hardware with parallel processing capabilities to these specialized operations. Processor arrays or systolic arrays, due to their structural regularity and consequent suitability for VLSI implementation, are frequently used for this purpose.

Three factors have contributed to the evolution of systolic arrays into a leading approach for computationally intensive applications [11]:

- Technology Advances: The growth of VLSI/WSI (Wafer Scale Integration) technology. Smaller and faster gates allow a higher rate of on-chip communication as data have to travel a shorter distance. Higher gate densities permit more complex PEs with higher performance and granularity. Economical design and fabrication processes produce less expensive systolic chips, even in small quantities. Progress in design tools and simulation techniques ensure that a systolic PE can be fully simulated before fabrication, reducing the chances that it will fail to work as designed. The regular modular arrays also decrease time to design and test, as fully tested unique cells can be copied quickly and arranged into a systolic array. In addition, as VLSI/WSI designs become more complex, a regular systolic array provides an efficient way to ensure fault tolerance: any fault-tolerance mechanism built into one PE is extensible to all PEs. Relatively new Field Programmable Gate Arrays (FPGA) permit a reconfigurable architecture in which the architecture of the PE can be programmed to match the computation required by the algorithm.

- Parallel Processing: Efforts to add concurrency to conventional von Neumann computers have yielded a variety of techniques such as coprocessors, multiple functional units, data pipelining (and parallelism), and multiple homogeneous processors. Systolic arrays combine features from all of these architectures in a massively parallel architecture that can be integrated into existing platforms without a complete redesign. A systolic array can act as a coprocessor with multiple functional units and/or processors in an n-dimensional pipeline. The I/O requirements are mitigated using data pipelining by allowing adjacent PEs to reuse the input data, but a systolic array also has incremental instruction processing or computational pipelining. Each PE computes an incremental result, and both the input data and partial results flow through the array.

- Demanding Scientific Applications: The technology growth in the past three decades has produced computers that make it feasible to attack scientific applications on a larger scale. New applications requiring increased computational performance have been developed that were not possible earlier. Examples of these applications include interactive language (or speech) recognition, text recognition, virtual reality, database operations, and real-time image and signal processing [12, 13]. These applications require massive, repetitive parallel processing, and hence, systolic computing.

A number of implementation issues determine a systolic array's performance and efficiency. Designers must be able to understand the performance trade-offs early in the design cycle to quickly produce economical designs. Some of the issues are

- Algorithm mapping
- Integration into existing systems
- Extensibility
- PE functionality
- Clock synchronization
- Reliability.

The most important aspect in the design of a systolic computer is the mapping of the algorithm to the processor array. In the systolic paradigm, every algorithm requires a specialized systolic design in which communication data streams, PE definitions, and input-output patterns are customized. Consequently, the terms *systolic algorithm* and *systolic array* are often synonymous, and designs are referred to as algorithmically specialized. The early arrays were designed using a "seat-of-the-pants" method, an ad hoc process in which a designer sits down with a pen and tries to draft data flows until one that works is found. This early approach was criticized as "black art," and it was clear that if systolic design was to develop and influence parallel computing, new and more systematic methods of design were required. Hence, for the past 10 years, there has been a concerted effort to develop a mathematical framework based on dependence manipulation and the mapping of loop programs and recurrences into lattice space. These important developments utilized concepts from source-to-source program transformation, signal-flow graphs from control engineering, and formulation of loop computations as uniform or linear recurrence equations [12]. They have been shown to be useful for a wide class of algorithms in many applications. The theoretical framework has also allowed considerable progress in extending the range of problems considered.

This thesis addresses a number of the issues in designing systolic processor arrays. A major contribution of this thesis is a systematic parameter-based approach to design optimal processor array for algorithms with uniform dependencies. Section 1.2 provides a precise characterization of the class of algorithms for which the results obtained in this thesis are valid. Section 1.1 describes the notation used in this thesis.

1.1 Notations

The general notation used in this thesis is as follows. Vectors are in lower case with arrows on top, and matrices are in upper-case bold font. The transpose of vector $\vec{v}$ and matrix $\mathbf{M}$ are denoted by $\vec{v}^t$ and $\mathbf{M}^t$, respectively. The absolute value of vector $\vec{v}$ defined as the Euclidean norm is denoted by $|\vec{v}|$. Notation $\vec{v} \geq \vec{u}$ means that every component of $\vec{v}$ is greater than or equal to the corresponding component of $\vec{u}$. Vector $\vec{0}$ denotes a row or column vector whose entries are all zeroes. The dimensions of vector $\vec{0}$, and whether it denotes a row or column vector, are implied by the context in which it is used. The scalar product of two vectors $\vec{v}_1$ and $\vec{v}_2$, and the product of a vector $\vec{v}$ and matrix $\mathbf{M}$ are written (without transposes) as $\vec{v}_1 \cdot \vec{v}_2$ and $\vec{v} \cdot \mathbf{M}$ (or $\mathbf{M} \cdot \vec{v}$), respectively. The product of two matrices $\mathbf{M}_1$, $\mathbf{M}_2$, and a scalar s and a vector $\vec{v}$ are simply written as $\mathbf{M}_1\mathbf{M}_2$ and $s\,\vec{v}$ without any dot symbol. The set of integers is denoted by $\mathcal{Z}$, and the cardinality of a set C by $|C|$.

1.2 Algorithm Model

Uniform dependence algorithms (or *Uniform Recurrence Equations* – UREs) are formally defined as follows. A Uniform dependence algorithm is an algorithm that can be described by an equation of the form

$$Z_i(\vec{J}) = \phi\left[Z_1(\vec{J}-\vec{d}_1), Z_2(\vec{J}-\vec{d}_2), \ldots, Z_r(\vec{J}-\vec{d}_r)\right], \qquad 1 \le i \le r\,, \tag{1.1}$$

where the column vector $\vec{J} = [j_1, j_2, \ldots, j_n]^t \in J \subset \mathcal{Z}^n$ is an index point; J is the index set of the algorithm n is the number of components of $\vec{J}$, i.e., the algorithm dimension; ϕ is a single-valued function computed "at index point $\vec{J}$" in a single unit of time; $\mathcal{Z}_i(\vec{J})$, $1 \le i \le r$ is the result of computing the right-hand side of Eq. (1.1); output variables correspond to values at particular index points $\vec{J}$; if $\vec{J}$ is not in J, then $Z_i(\vec{J})$ is an input variable; $\vec{d}_j$, $j = 1, \ldots, r$ are constant vectors independent of $\vec{J}$ called dependences; and matrix $\mathbf{D} = [\vec{d}_1 \vec{d}_2 \ldots \vec{d}_r]$ is called the dependence matrix.

Example 1.1 A well-known simple example of an URE is

$$\mathcal{C}(i,j,k) \;=\; \mathcal{C}(i,j,k-1) + \mathbf{A}(i,k)\,\mathbf{B}(k,j), \qquad 1 \le i,j,k \le N, \tag{1.2}$$

which describes the computation of an $N \times N$ matrix $\mathbf{C}$, as the product of two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$. The index set consists of all of the integer points within a 3-D cube with sides of length N. ■

Uniform dependence algorithms can be found in many scientific computations, digital signal processing applications and other fields. However, a much larger class of algorithms called affine dependence algorithms (or Affine Recurrence Equations – AREs) can also benefit from the techniques proposed for UREs. For AREs, Eq. (1.1) is replaced by

$$Z_i(y(\vec{J})) = \phi\left[Z_1(x_1(\vec{J})), \ldots, Z_r(x_r(\vec{J}))\right], \qquad 1 \le i \le r\,, \tag{1.3}$$

where indices y and x_i, $i = 1, \ldots, r$ are affine functions of $\vec{J}$, i.e., functions of the form $\mathbf{A}\,\vec{J} + \vec{b}$ where $\mathbf{A}$ is a matrix with n columns, and $\vec{b}$ is a constant vector with as many elements as the number of rows of $\mathbf{A}$.

There exist many techniques to transform AREs into UREs, or to *uniformize* AREs. The basic idea is to select a few integral basis vectors such that all affine dependence vectors of the ARE can be expressed as nonnegative integer linear combinations of the basis vectors. These vectors correspond to uniform dependences after the uniformization.

Example 1.2 For example, in the URE shown in Eq. (1.2) for matrix multiplication, $\mathbf{A}(i,k)$ (respectively, $\mathbf{B}(k,j)$) are inputs used in several computations to generate $\mathcal{C}(i,j,k)$ for all values of j (respectively, i). This can be shown explicitly as follows:

$$\begin{aligned} \mathcal{A}(i,j,k) &= \mathcal{A}(i,0,k) \\ \mathcal{B}(i,j,k) &= \mathcal{B}(0,j,k) \\ \mathcal{C}(i,j,k) &= \mathcal{C}(i,j,k-1) + \mathcal{A}(i,j,k)\mathcal{B}(i,j,k),\ \ 1 \le i,j,k \le N, \end{aligned} \tag{1.4}$$

where $\mathcal{A}(i,0,k) = \mathbf{A}(i,k)$ and $\mathcal{B}(0,j,k) = \mathbf{B}(k,j)$. The dependencies for the first two statements are affine, i.e., they are $[0,j,0]^t$ and $[i,0,0]^t$, respectively. Reusing, or "pipelining" of these data among different computations can be done as follows (yielding a decomposition of the affine dependencies in terms of uniform basic vectors $(0,1,0)^t$ and $(1,0,0)^t$):

$$\begin{aligned} \mathcal{A}(i,j,k) &= \mathcal{A}(i,j-1,k), \\ \mathcal{B}(i,j,k) &= \mathcal{B}(i-1,j,k), \\ \mathcal{C}(i,j,k) &= \mathcal{C}(i,j,k-1) + \mathcal{A}(i,j,k)\mathcal{B}(i,j,k),\ \ 1 \le i,j,k \le N. \end{aligned} \tag{1.5}$$

■

This simple example illustrates another advantage of uniformization in which broadcasts of data to many processing elements can be eliminated. In the above uniformized algorithm, distinct variables (which may have identical values) are used to compute distinct $\mathcal{C}(i,j,k)$. Procedures for uniformization and broadcast removal share many similarities and are discussed in the references [14, 15, 16, 17, 18, 19].

1.2.1 Relation to nested-loop programs

Affine dependence algorithms are common in image processing, digital signal processing, and other scientific applications in which regular compute-intensive operations are required

[12, 20, 21]. In practice, many of the algorithms to be executed by processor arrays are described in a procedural high-level language such as FORTRAN. Nested loops are often the most time consuming kernels of these programs and are, therefore, targets of hardware accelerators based on processor arrays. It turns out that a large number of nested loops can be modeled as affine recurrences. For instance, it is relatively easy to relate the following generic nested loop to a corresponding system of affine recurrences:

```
DO ($j_1 = l_1, u_1$ ; $j_2 = l_2, u_2$ ; ... ; $j_n = l_n, u_n$)
    $H_1(\vec{J})$ ;
    $H_2(\vec{J})$ ;
    ⋮
    $H_t(\vec{J})$ ;
END
```

where $\vec{J} = [j_1, j_2, \ldots, j_n]^t$ is the index vector and $H_i(\vec{J})$, $i = 1, \ldots, t$, are assignment statements of the form of Eq. (1.3), i.e., $Z_i(y(\vec{J})) = \phi\left[Z_1(x_1(\vec{J})), \ldots, Z_r(x_r(\vec{J}))\right], \qquad 1 \leq i \leq r.$ Each appearance of a variable on the right-hand side may cause a dependence [22]. If all loop bounds l_i and u_i, $i = 1, \ldots, n$, are linear functions of index variables $j_1, \ldots, j_{i-1}$, then the set of all iteration vectors $\vec{J}$ of the loop can be described by a convex polyhedron.

Affine recurrence equations with a convex polyhedral domain can be used to model the above program if *(i)* all loop bounds l_i and u_i are affine functions of loop variables $j_1, \ldots, j_{i-1}$; *(ii)* indexing functions $y()$ and $x_k()$, $k = 1, \ldots, r$, are affine functions of the form $\mathbf{A} \cdot \vec{J} + \vec{d}$; and *(iii)* branch statements are defined in terms of the loop variables $j_1, \ldots, j_{i-1}$, and do not go outside the loop containing the branch statement. Given a nested loop program, reference [23] describes how to obtain the set of uniform dependencies using the techniques of uniformization.

Example 1.3 It is easy to see that the following nested-loop program corresponds to the pipelined version of the matrix multiplication algorithm described in Eq. (1.5).

```
DO ($i = 1, N$ ; $j = 1, N$ ; $k = 1, N$)
```

```
    $A(i,j,k) = A(i,j-1,k)$
    $B(i,j,k) = B(i-1,j,k)$
    $C(i,j,k) = C(i,j,k-1) + A(i,j,k) \times B(i,j,k)$
END
```

Intuitively, data $A_{i,k}$ are pipelined along the j axis, and $B(k,j)$ are pipelined along i axis. ■

The following example shows the application of pipelining to obtain uniform dependencies to another problem. The set of dependence vectors obtained after uniformization are equivalent to those of the Warshall-Floyd algorithm of computing the transitive closure of a $N \times N$ matrix in Section 3.1.

Example 1.4 Consider the following 3-dimensional (3-D) recurrence with $n = 3$, $r = 5$:

$$\mathcal{Z}(k,i,j) = \mathbf{X}(k,i)\,\mathbf{Y}(j,k) + \mathcal{Z}(k-1,i+1,j+1) \tag{1.6}$$

$$+\mathcal{Z}(k-1,i+1,j) + \mathcal{Z}(k-1,i,j+1). \tag{1.7}$$

After pipelining and uniformization, Eq. (1.7) becomes

$$\begin{aligned} \mathcal{Z}(k,i,j) &= \mathcal{X}(k,i,j-1)\,\mathcal{Y}(k,i-1,j) + \mathcal{Z}(k-1,i+1,j+1) + \mathcal{Z}(k-1,i+1,j) \\ &+ \mathcal{Z}(k-1,i,j+1). \end{aligned} \tag{1.8}$$

The dependence vectors collected into a matrix are

$$\mathbf{D} = \begin{array}{c} \left[\begin{array}{ccccc} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 0 \\ 1 & 0 & -1 & 0 & -1 \end{array}\right] \\ \begin{array}{ccccc} \mathcal{X} & \mathcal{Y} & \mathcal{Z} & \mathcal{Z} & \mathcal{Z} \end{array} \end{array} \tag{1.9}$$

■

In short, the focus in this work is on algorithms that can be modeled as uniform recurrences and affine recurrences that can be uniformized. Hence, only the structural information of the algorithm, i.e., index set J and dependence matrix D, is needed. Note that all computations are identical in the processor array when uniform recurrences are mapped, and that inputs/outputs occur at the peripheral boundary of the processor array.

If iteration $\vec{J}$ depends on iteration $\vec{J}'$, then this dependence can be described by a dependence vector $\vec{d} = \vec{J} - \vec{J}'$, which is the vector difference of the index vectors of these two iterations. The dependencies in the algorithm can be shown by a dependence graph (DG) over an n-dimensional (n-D) domain (integer lattice), where nodes are labeled by index vectors corresponding to the operations in the innermost loop body, and arcs correspond to the loop-carried dependencies between two instances of the loop body. Hence, the loop body for scheduling is the set of statements in loop nests enclosing all of the branch statements. Thus, the techniques developed in this thesis are applicable to algorithms represented as dependence graphs with uniform dependencies, and whose domain is defined in terms of a set of problem parameters.

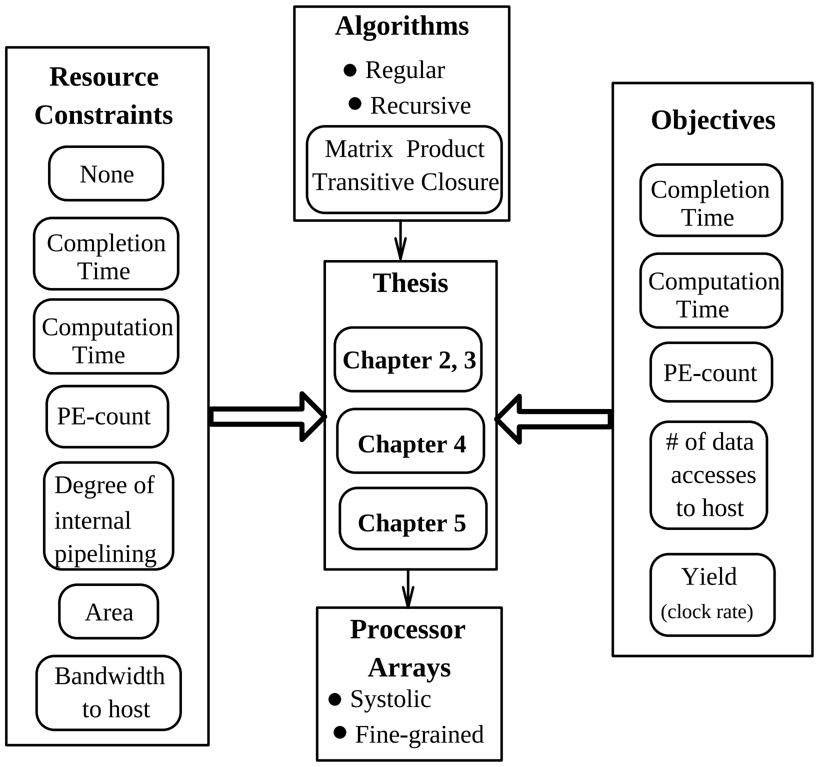
To illustrate the importance of uniform dependence algorithms, a list of some of the applications is given in Table 1.1.

1.3 Overview of the Thesis

This section gives a brief overview and organization of the different chapters in this thesis. The thesis is conceptually subdivided into two parts. In the first part, consisting of Chapters 2, 3, and 4, a systematic method of synthesizing *optimal* algorithm-specific systolic processor arrays from uniform dependence algorithms (UDA) is described. In the second part, consisting of Chapter 5, the design of a scalable VLSI coprocessor suitable for the class of recursive loop algorithms in Section 1.2 is presented. Such a class-specific coprocessor can be used as a back-end accelerator for loop algorithms. Figure 1.1 shows the overview of this thesis. As shown in the figure, this thesis deals with the problem of mapping recursive, uniform dependence algorithms described by nested loops to systolic processor arrays with globally synchronized data flow. The individual chapters describe how the designer can include different resource constraints and specify different objectives in the design process.

Table 1.1: A sample of applications that can be represented as uniform dependence algorithms, or can be converted to uniform dependence algorithms

Signal and Image Processing	Finite impluse response (FIR) filtering Convolution, Deconvolution Discrete transforms (DFT, DCT, DHT) Vector quantization SAR beamforming Template matching, Feature vector computation
Matrix Operations Linear Algebra	Matrix-matrix, Matrix-vector products Triangular linear system LU, QR, SVD decomposition Matrix Inversion, Gauss-Jordan Elimination
Algebraic Computations	Polynomial multiplication and division Interpolation Binary long multiplication
Pattern Matching	String matching, 2-D tuple comparison Correlation Levenstein algorithm Longest common subsequence
Graph Applications	Shortest paths Transitive closure (Floyd-Warshall algorithm) 2-D routing
Others	Sorting Database operations: Join, Cartesian product



Objectives	Resource Constraints	
Completion time	None	Chapter 3
Computation time	PE-count	Chapter 3
	Degree of internal pipelining	Chapter 4
	Degree of internal pipelining, PE-count	Chapter 4
PE-count	None	Chapter 3
	Completion time	Chapter 3
	Computation time	Chapter 3
	Degree of internal pipelining, Computation time	Chapter 4
# of data accesses to host	Bandwidth to host	Chapter 5
Yield (Clock rate)	Area	

Figure 1.1: Overview and organization of this thesis: Objectives and resource constraints considered in the design process.

Figure 1.1 lists a few of the common performance measures that are optimized and resource constraints considered in this thesis. The resource restrictions considered are

- maximum completion time (including load and drain times),
- maximum computation time (without load and drain times),
- maximum PE-count (number of processors),
- degree of internal pipelining which is the number of stages in the functional units of the PEs,
- maximum total area which includes area of PEs and area of memory,
- maximum bandwidth to transfer data from the host.

Similarly, the objectives considered are minimizing completion or computation time, minimizing PE-count, minimizing total area, minimizing number of data accesses to host to execute the given algorithm on the VLSI array processor, and minimizing power consumption by minimizing clock rate required for a given performance level. Figure 1.1 also shows the combination of design objectives and resource constraints considered in the different chapters of this thesis.

Chapters 2 and 3 describe the development and application of a systematic array synthesis technique, called the *General Parameter Method* (GPM), for specialized systolic arrays. Chapter 2 considers how to synthesize a systolic processor array with the input data flows to solve the given uniform dependence algorithm. The chapter also describes the parameter definitions, constraint equations for valid choice of parameters, and an efficient polynomial-time search procedure to find the optimal parameter values. The unique feature of GPM is the ability to synthesize *optimal* arrays of any lower dimension for a general nonlinear and nonmonotonic user-specified objective expressed in terms of the parameters in GPM. This is in contrast to other existing synthesis procedures such as the dependency method (described in Section 2.1), in which the design problem has been cast as an integer programming problem with an exponential worst-case complexity to find the optimal designs. In addition, in Chapter 2, equivalence between the GPM and the dependency method (DM) has been presented. This equivalence permits designers familiar with DM to utilize the efficiency of GPM to find optimal designs.

Chapter 3 discusses the application of the proposed GPM to two important application problems: finding the transitive closure of a matrix, and computing the product of two matrices. This chapter presents the following optimal array designs for the two application problems.

- Minimizing completion time, computation time and PE-count without any resource constraints,
- Minimizing PE-count with upper bounds on completion time or computation time,
- Minimizing completion time or computation time with upper bound on PE-count.

The processor-time trade-offs allow a designer to choose the appropriate architecture when faced with resource restrictions on processor count or time or both. These processor-time trade-offs require the exploration of a large portion of the design space, and is possible because of the efficiency of GPM and its search procedure.

In Chapter 4, an extension of GPM to design of two-level pipelined systolic arrays has been described. Two-level pipelining is a technique of pipelining operations across PEs of the array, and also pipelining the functional units in the PEs, to achieve higher throughput. However, its success depends greatly on the ability to design data flows that can fully utilize the two levels of pipelines. This chapter considers how to synthesize a two-level pipelined processor array and input data flows to optimize a given objective to solve the given uniform dependence algorithm.

The enhanced GPM, presented in Chapter 4, is a systematic approach that evaluates alternative designs and determines data flows that utilize the two-level pipelined structures. Chapter 4 describes the additional constraints incorporated into GPM for handling pipelined functional units and presents closed-form results for a class of algorithms represented by n-dimensional mesh graphs. Also, Chapter 4 presents the application of the enhanced GPM to matrix product problem by considering the degree of internal pipelining as a user-specified parameter. Array designs that optimize computation time for a given degree of internal pipelining are presented for the matrix-product and transitive-closure applications. In addition, processor-time tradeoff figures that can be used by a designer to derive arrays that minimize computation time given a maximum PE-count, or minimize PE-count given a maximum computation time, for any given degree of internal pipelining, are presented.

The first part of this thesis, Chapters 2, 3, and 4, describes a systematic synthesis method for a special-purpose or fixed-function systolic architecture designed as a hardware implementation of a given algorithm. In mass quantities, the production of such fixed-function arrays is manageable and economical, and thus suited for common applications. Every time a systolic array is to be used for a new application, the manufacturer has to take the long and costly process of designing and fabricating the application-specific integrated chip (ASIC). Although the cost and risks of ASIC development have decreased in recent years, budget constraints have motivated a trend away from unique hardware development except in cases in which the performance required justifies the cost of developing such specialized hardware. Consequently, general-purpose or versatile systolic architectures have been a logical alternative.

The second part of this thesis, Chapter 5, describes the design of a scalable VLSI coprocessor to execute the class of algorithms defined in Section 1.2. This chapter describes the architecture design, partitioning and mapping techniques for a coprocessor to execute the entire class of uniform dependence algorithms. Such a common retargetable architecture for the coprocessor allows the processing power to be reused for all of the applications instead of an algorithm-specific design. This chapter deals with the design issues (page 3) of (i) integration into existing systems, by considering a fixed bandwidth connection to main memory, and (ii) extensibility, by choosing a modularly expandable parallel architecture. The architecture consists of a linear array of PEs and buffer memories (called access-units), and good performance is achieved by developing efficient software techniques.

Chapter 5 discusses the architectural issues involved in designing the coprocessor, the mapping and partitioning techniques, and results on matrix-product and transitive-closure applications. Constraints on completion time (throughput) and total silicon area are considered, and trade-offs between the performance (clock-rate reduction) and cost (silicon area index) of the resulting architecture are given for matrix-product and transitive-closure problems. An important resource limitation is the assumption of a fixed bandwidth to access data from the external main memory. For large problem sizes, managing large quantities of data efficiently through the low-bandwidth connection is critical to achieve good performance from the coprocessor. Thus, the objective of minimizing the number of accesses over the limited bandwidth connection is considered in the mapping process. This is in contrast to other approaches of building general-purpose systolic computers [24, 25, 26, 27, 7]. Thus, Chapter 5 discusses design methods under constraints of fixed bandwidth and area, and objectives of yield (clock frequency) or speedup, and number of accesses.

The mapping process incorporates the General Parameter Method (Chapters 2 and 4) to map partitioned dependence graphs of the given loop algorithm. The objective of maximizing the utilization of the processor array is used in GPM. An important result obtained is the square-root relationship between the clock-rate reduction and the area of the coprocessor under fixed main-memory bandwidth. From the square-root relationship, it can found that the system yield improves with the area of the coprocessor when chip yield decreases as the inverse square of the clock frequency.

Finally, in Chapter 6, we summarize the thesis, and discuss some future avenues of research.

1.4 Contributions of This Thesis

The following are the main contributions of this thesis:

- General Parameter Method (Chapter 2) [28, 29, 30, 31] A systematic array synthesis technique for uniform dependence algorithms that can optimize a given user-specified objective subject to some resource constraints. Chapter 3 presents the application of GPM to the problems to transitive-closure and matrix-product problems.

- Extension of General Parameter Method [32] (Chapter 4) to design two-level pipelined processor arrays for uniform dependence algorithms. Results are given for designing optimal two-level pipelined processor arrays for matrix-product and transitive-closure applications.

- Architectural design of a parallel VLSI coprocessor [33] (Chapter 5) for the class of uniform dependence algorithms. Efficient techniques are given to partition the dependence graph into blocks and sequence the blocks through the fixed size buffer memory to reduce data accesses to main memory. Results on the matrix-product and transitive-closure applications indicate that the coprocessor can be used for higher throughput or lower clock rate compared to that for a reference 1-PE design. A square-root relationship between the clock-rate reduction and area of the coprocessor under fixed main-memory bandwidth has been derived. From the square-root relationship, it can found that the system yield improves with the area of the coprocessor when chip yield decreases as the inverse square of the clock frequency.

2. GENERAL PARAMETER METHOD

This chapter presents a systematic parameter-based approach to synthesize algorithm-specific systolic arrays for uniform dependence algorithms. The techniques discussed here apply to algorithms described as recurrences, either by mathematical expressions or by high-level-language programs. Section 1.2 provides a precise characterization of the class of algorithms for which our results are valid. The techniques are illustrated by examples involving linear arrays of processors (1-dimensional processor arrays) (Chapter 3); however, unless otherwise stated, the results can be extended to processor arrays of arbitrary dimensions. We choose to study linear arrays because they are easier to build and program than arrays of higher dimension. Hence, several linear arrays have been implemented for specific applications as well as for "general-purpose" computing [34, 35, 24, 25].

The organization of this chapter is as follows. Section 1.2 describes the model of algorithms targeted in this thesis, followed by a discussion of previous and related work in Section 2.1. Section 2.2 presents the definitions of parameters, followed by the constraint equations for valid systolic processing in Section 2.3. The design method, pruning strategy, and search procedure for optimal array parameters are given in Sections 2.4, and the chapter concludes with a summary and comparison to the dependence-based methods in Section 2.5.

2.1 Previous Work

Initial designs of processor arrays were ad hoc and relied heavily on the designers' skill and intuition. Since every algorithm requires a specialized design customized to its communication patterns, a systematic technique for generating processor arrays from the algorithm description is necessary. Therefore, a great deal of effort has been devoted by numerous researchers to mapping uniform dependence algorithms to processor arrays systematically. Most of these methods are based on or derived from the *dependency method* (*DM*) [36, 37, 38]. An overview of the different methods can be found in the references [39, 12].

In the dependency method (denoted as DM), an algorithm (A) is represented as a 5-tuple (J^n, C, D, X, Y), where J^n is a finite n-dimensional index set of A; C is the set of triples that represents the set of computations performed; D is the set of dependencies; X is the set of input variables; and Y is the set of output variables. A feasible design (i.e., mapping an algorithm to processor array) is obtained by a linear transformation, represented as an $n \times n$ matrix $\mathbf{T} \in \mathcal{Z}^{n \times n}$, where $\mathcal{Z}$ is the set of integers. Thus,

$$\mathbf{T} = \begin{bmatrix} \vec{\pi} \\ \mathbf{S} \end{bmatrix}$$

where $\vec{\pi}$ is $1 \times n$ schedule vector and $\mathbf{S}$ is the processor allocation matrix. For any index point $\vec{J}$, $\mathbf{S} \cdot \vec{J}$ denotes the processor at which the computation corresponding to the index point executes, and $\vec{\pi} \cdot \vec{J}$ is the time of execution at that processor. Constraints are imposed on matrix $\mathbf{T}$ to ensure valid execution of the algorithm. The design of a systolic array is then equivalent to determining the n^2 parameters of the transformation matrix, $\mathbf{T}$.

This general representation of a feasible design as a particular mapping matrix allows DM to be applied to uniform as well as nonuniform recurrences. However, in DM, the generality in representation leads to large search spaces for optimal designs, as the problem of finding optimal designs is posed as an integer programming problem [2, 40]. In contrast, the method presented in this chapter, the *General Parameter Method* (*GPM*), is restricted to uniform recurrences, but can be used to generate optimal designs for user-specified objectives (including nonmonotonic and nonlinear ones) using efficient search techniques of polynomial complexity.

There have been several earlier attempts to map algorithms onto lower-dimensional arrays [3, 40, 41]. Important steps towards a formal solution were first made by Lee and Kedem [3]. They presented the concept of data-link collisions (two data tokens contending for the same link simultaneously) and conditions to avoid collisions. They also presented a method that analyzes all computations in the domain of the recurrence in order to detect computational conflicts (two computations scheduled to execute simultaneously in the same processor). To identify feasible designs, they provided necessary and sufficient conditions for designs that avoid computational and data-link conflicts. However, they did not present any systematic procedure for finding optimal designs. Subsequently, Shang and Fortes [2] have developed closed-form conditions for a mapping to be free of computational conflicts. These closed-form conditions also eliminate data-link conflicts for active data[1] participating in the computations.

In general, in DM, feasible designs are found heuristically by first specifying a "good" allocation matrix $\mathbf{S}$ and then subsequently determining the schedule vector $\vec{\Pi}$ that minimizes the computation time. Note that the number of choices for matrix $\mathbf{S}$ could be very large or even infinite, making it difficult (or impossible) to enumerate over them.

Initial work on parameter-based methods was done by Li and Wah [42] (denoted as OPM or Original Parameter Method) for a restricted set of uniform recurrences. They considered specifically 3-D and 2-D recurrences and mapped them to 2-D and 1-D processor arrays, respectively. The structure of the recurrence was such that the dependence vectors were unit vectors and the dependency matrix, an identity matrix. This research generalizes the above initial work into a powerful and efficient array-synthesis technique called the General Parameter Method (GPM) by making three important and nontrivial extensions.

(a) We consider the recurrence model as a general n-D recurrence with arbitrary constant dependence vectors instead of a specific 3-D one. The target processor arrays are also allowed

[1]The lifetime of a data token in the processor array can be viewed as consisting of an active phase, in which the token is involved in its chain of computations, and a passive phase, in which the token is moving from the input peripheral processor to become active or is moving to an output peripheral processor after its active phase.

to be of any lower dimension m, where $1 \leq m \leq n$. We provide new necessary conditions to guarantee the correctness of systolic processing in mapping high-dimensional recurrences to lower-dimensional processor arrays. These conditions define a search space, whose size is polynomial with respect to the size of the recurrence to be mapped. In contrast, previous methods for finding optimal designs are based on integer linear programming with a search space of exponential complexity.

(b) We extend our search method to handle general nonlinear objectives that may vary nonmonotonically with the parameters, and introduce new pruning strategies to prune suboptimal designs in the search space so that optimal designs can be found efficiently. We show *(i)* optimal designs that include load and drain times in the objective (which introduce nonlinearity in the objective function and constraints), and *(ii)* optimal designs with constraints on number of allowable processing elements and/or completion time. Such designs cannot be found by previous methods.

(c) We show the equivalence between DM and GPM by providing necessary equations to transform parameters used in DM to those used in GPM, and vice versa. The DM can be considered as a mapping problem in the Cartesian coordinate system with unit vectors as basis vectors, whereas GPM can be considered as mapping in a possibly nonorthogonal coordinate system with dependence vectors as basis vectors. The equivalence allows the designers familiar with DM to utilize the efficiency of GPM to find optimal designs.

The potential simplicity of GPM over DM described in (c) is explained by observing that in mapping an n-D algorithm to an m-D processor array, the number of variables to be determined in DM is $(m+1) \times n$, whereas the number of parameters in GPM is $(m+1) \times g$, where $g = rank(\mathbf{D})$. Since $g \leq n$ (as $\mathbf{D}$ is an $n \times r$ matrix), the number of variables in GPM is often less than that in DM, and is at worst equal to the number of variables in DM. Hence, there is potential reduction in complexity by performing the transformation, especially if there are only a few dependence vectors in a high-dimensional space.

Our transformation between GPM and DM extends the work of O'Keefe, Fortes and Wah [43], who showed the equivalence between DM and GPM for 2-D and 3-D uniform recurrences. Our transformation also allows efficient search strategies developed in GPM to be used to find optimal designs in DM. Consequently, designers familiar with DM can

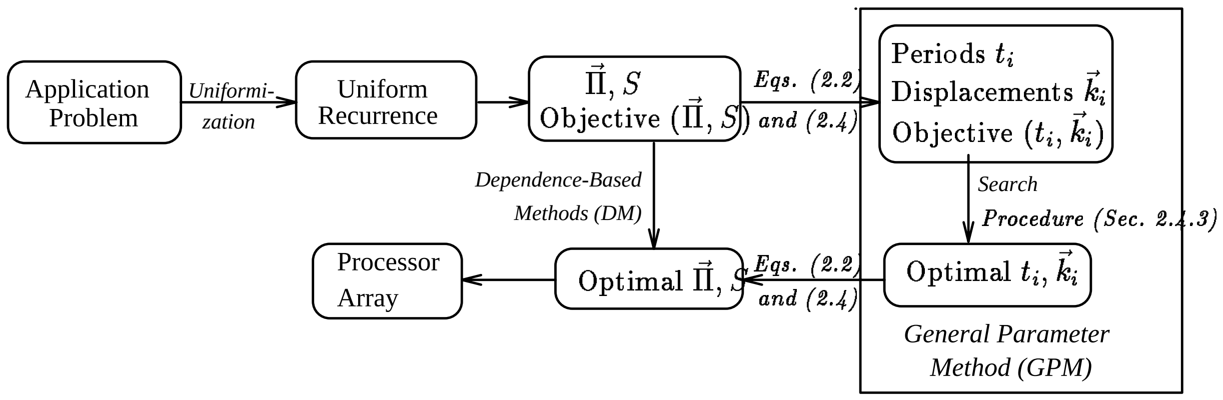

Application Problem
Uniformi-zation
Uniform Recurrence
$\vec{\Pi}, S$ Objective $(\vec{\Pi}, S)$
Eqs. (2.2) and (2.4)
Periods t_i Displacements $\vec{k}_i$ Objective $(t_i, \vec{k}_i)$
Dependence-Based Methods (DM)
Search Procedure (Sec. 2.4.3)
Processor Array
Optimal $\vec{\Pi}, S$
Eqs. (2.2) and (2.4)
Optimal $t_i, \vec{k}_i$
General Parameter Method (GPM)

Figure 2.1: Application of GPM to find optimal designs in DM.

obtain better (or optimal) array designs using GPM. Referring to Figure 2.1, after defining the objective (possibly nonlinear and nonmonotonic) in terms of the representation chosen (i.e., $\vec{\Pi}$ and **S**), the designer converts the objective in terms of the parameters of GPM using the equivalence given in Eqs. (2.2) and (2.4) (p. 22 and p. 23 in the next section). Once the objective and variables have been converted, GPM is used to generate optimal arrays efficiently. The solutions obtained by GPM are then converted to $\vec{\Pi}$ and **S** in DM again using Eqs. (2.2) and (2.4). This step involves solving two sets of simultaneous equations for $\vec{\Pi}$ and **S** from the periods and displacements in GPM, and has a worst-case complexity of $O(n^3)$.

The next three sections describe the parameters used in GPM, the constraints that must be satisfied for correct operation, the specification of the objective function, and the search strategy. We assume that the processing elements are equally spaced in m dimensions with unit distance between directly connected processing elements, and that buffers between directly connected processing elements, if any, are assumed to be equally spaced along the link.

2.2 General Parameter Method: Parameters

The intuition behind GPM is as follows. It is known that the semantics of processor arrays can be formally described by uniform recurrence equations, i.e., processor arrays are *isomorphic* to uniform recurrences. This implies that as long as the computations defined by the UREs are well-formed, there is a direct mapping from the recurrence to the processor array. In fact, this mapping is equivalent to a *linear* transformation of the index set. Hence, for a linear mapping, the time (respectively, the distance) is constant between execution of any two points $\vec{I}_1$ and $\vec{I}_2$ in the index set separated by a dependence vector $\vec{d}$, where $\vec{I}_1 = \vec{I}_2 + \vec{d}$. This constant is equal to $\vec{\Pi} \cdot \vec{d}$ (respectively, $\mathbf{S} \cdot \vec{d}$) independent of index points $\vec{I}_1$ and $\vec{I}_2$. For recurrences with uniform indexing functions (i.e., UREs and uniformized AREs), the dependences are constant vectors and homogeneous (i.e., the set of dependence vectors at any point in the index set is the same as any other in the index set). Thus, the computation of the recurrence on the processor array is periodic in time and space along dependence vectors in the index space. This periodicity is succinctly captured and exploited in GPM, which considers the mapping problems in a possibly nonorthogonal coordinate system with dependence vectors as basis vectors. In other words, in GPM, a representation that captures the above periodicity is used, which allows the optimal target array to be found efficiently.

In GPM, the characterization of the behavior, correctness, and performance of a processor array is defined in terms of a set of scalar and vector parameters. When a uniform recurrence is executed on a processor array, the computations are periodic and equally spaced in the processor array. The GPM captures this periodicity through a minimal set of parameters defined as follows.

Parameter 1: Periods. These capture the time between execution of the computations corresponding to the source and sink index points of a dependence vector. Suppose that the time at which an index point $\vec{I}$ (defined for the uniform recurrence equation) is executed is given by function $\tau_c(\vec{I})$, then the period of computation t_j along dependence vector $\vec{d_j}$ is defined as

$$t_j \;=\; \tau_c(\vec{I} + \vec{d_j}) - \tau_c(\vec{I}), \quad j = 1, 2, \cdots, r. \tag{2.1}$$

The number of periods defined is equal to r, the number of dependencies in the algorithm. In terms of DM, period t_j is related to $\vec{\Pi}$, the schedule vector in DM, as follows [31]:

$$t_j = \vec{\Pi} \cdot \vec{d_j}\ . \tag{2.2}$$

Parameter 2: Velocity. $\vec{V}_j$, velocity of a datum along a dependence vector $\vec{d}_j$, for $j = 1, 2, \cdots, r$, is defined as the directional distance passed during a clock cycle. Since PEs are at unit distance from their neighbors, and buffers (if present) must be equally spaced between PEs, the magnitude of the velocity must be a rational number of the form x/y, where x and y are integers and $x \leq y$ (to prevent broadcasting). This implies that in y clock cycles, a datum propagates through x PEs and $y - x$ buffers. All tokens of the same variable have the same velocity (both in speed and direction) which is constant during the execution in the processor array. The total number of velocity parameters is r (one for each dependence vector) with each velocity an m-element vector, where m is the dimension of the processor array. Hence, velocity $\vec{V}_j$ is given by

$$\vec{V}_j = \frac{\vec{k}_j}{t_j}, \quad j = 1, 2, \cdots, r, \tag{2.3}$$

where $\vec{k}_j$ is the (vector) distance between the execution locations of the source and sink index points of $\vec{d}_j$. In the notation of DM, $\mathbf{S}$, the allocation matrix, is related to $\vec{k}_j$ and $\vec{d}_j$ as follows:

$$\vec{k}_j = \mathbf{S} \cdot \vec{d}_j \ . \tag{2.4}$$

Parameter 3: Spacing or Data distribution. Consider variable Ω_i pipelined along dependence vector $\vec{d}_i$, $1 \leq i \leq r$. Consider an index point $\vec{I}$ where Ω_i is used in the computation. Data token $\Omega_i(\vec{I} - \vec{d}_i)$ is also used at index points $\vec{I} + t\,\vec{d}_i$, $t = \cdots, -1, 0, 1, \cdots$, in computing the recurrence. In other words, this token moves through the processors that use datum Ω_i at index points $(\vec{I} + t\,\vec{d}_i)$. Consider another token $\Omega_i(\vec{I} - \vec{d}_j)$ of the same variable Ω_i that is used at index points $(\vec{I} - \vec{d}_j + t\,\vec{d}_i)$, $j \neq i$. The directional distance in the processor space from token $\Omega_i(\vec{I} - \vec{d}_j)$ to $\Omega_i(\vec{I} - \vec{d}_i)$ is defined as spacing parameter $\vec{S}_{i,j}$. Since there are r variables Ω_i, $1 \leq i \leq r$, each associated with dependence vector $\vec{d}_i$, there are $r - 1$ nontrivial spacing parameters for each variable and one trivial spacing parameter, $\vec{S}_{i,i} = \vec{0}$. These denote the r distances for variable i: $\Omega_i(\vec{I} - \vec{d}_j) \longrightarrow \Omega_i(\vec{I} - \vec{d}_i)$, $j = 1, 2, \cdots, r$. Each spacing parameter $\vec{S}_{i,j}$ is an m-D vector, where m is the dimension of the processor array.

The notation $\vec{S}_{i,j}$ denotes that it is the j-th spacing parameter of the i-th variable. A total of $r(r-1)$ nontrivial spacing parameters are defined. In the notation of DM, we have

$$\begin{aligned}
\vec{S}_{i,j} &= \vec{V}_j\, t_j - \vec{V}_i\, t_j && \text{(from Theorem 2.1, p. 25, Section 2.3.1)}\\
&= \vec{k}_j - \vec{V}_i\, t_j && \text{(from Eq. (2.3))}\\
&= \mathbf{S}\cdot\vec{d}_j - \tfrac{\vec{k}_i}{t_i}\, t_j && \text{(from Eqs. (2.3) and (2.4))}\\
&= \mathbf{S}\cdot\vec{d}_i - \tfrac{\vec{\Pi}\cdot\vec{d}_j}{\vec{\Pi}\cdot\vec{d}_i}\mathbf{S}\cdot\vec{d}_i && \text{(from Eqs. (2.2) and (2.4)).}
\end{aligned} \tag{2.5}$$

The total number of parameters defined is $r \times (r+2)$ of which r parameters are periods (scalars); the remaining $r^2 + r$ parameters are m-D vectors, of which r parameters are velocities and r^2 parameters are spacings (r of these spacings are trivially zero).

Example 2.1 For the recurrence in Eq. (1.5), there are 3 periods t_1, t_2, t_3 and 3 velocities $\vec{V}_1$, $\vec{V}_2$, $\vec{V}_3$. There are 9 spacing parameters $\vec{S}_{i,j}, i, j = 1,2,3$, of which $\vec{S}_{i,i} = \vec{0}$. For instance, for variable $\mathcal{C}$, $\vec{S}_{1,2}$, $\vec{S}_{1,3}$ define distances between $(\mathbf{C}(i,j) \rightarrow \mathbf{C}(i,j+1))$, $(\mathbf{C}(i,j) \rightarrow \mathbf{C}(i+1,j))$, respectively. ■

Example 2.2 For the recurrence in Eq. (1.8) the parameters defined are as follows. There are 5 periods t_1, t_2, t_3, t_4, t_5, and 5 velocities $\vec{V}_1$, $\vec{V}_2$, $\vec{V}_3$, $\vec{V}_4$, $\vec{V}_5$. There are 25 spacing parameters $\vec{S}_{i,j}, i, j = 1,2,3,4,5$, where $\vec{S}_{i,i} = \vec{0}$. For instance, for variable $\mathcal{X}$, $\vec{S}_{1,2}, \vec{S}_{1,3}, \vec{S}_{1,4}, \vec{S}_{1,5}$ define distances $(\mathbf{X}(k,i) \rightarrow \mathbf{X}(k,i-1)), (\mathbf{X}(k,i) \rightarrow \mathbf{X}(k-1,i+1)), (\mathbf{X}(k,i) \rightarrow \mathbf{X}(k-1,i+1))$, and $(\mathbf{X}(k,i) \rightarrow \mathbf{X}(k-1,i))$, respectively. ■

2.3 General Parameter Method: Constraint Equations

In Section 2.2, a set of $r^2 + r$ parameters has been introduced to define a mapping on the target processor array. The assignment of values to the parameters defines a specific processor array with a particular number of processors, buffers, and data-input pattern. It is also easy to see that all processor arrays that solve a given algorithm (or uniform recurrence) correspond to some assignment of values to the parameters. Hence, choosing different values for these parameters leads to different array configurations with different performance, and the problem of array design has been reduced to that of choosing appropriate parameter values.

In this section, constraint equations relating the parameters are given such that the set of values for the parameters is meaningful and defines a valid processor array. Theorems 2.1 and 2.2 provide the fundamental space-time relationship that must be satisfied by the parameters for correct systolic processing. Computational and data-link conflicts are avoided by enforcing the condition in Theorem 2.3.

The following notation is introduced to simplify the presentation of the theorems. Let $\vec{T} = [t_1, t_2, \cdots, t_r]^t$ be a vector composed of periods, and let $\mathbf{K} = \left[\vec{k}_1, \vec{k}_2, \cdots, \vec{k}_r\right]$ be a matrix (of size $m \times r$, where m is the dimension of the processor array) composed of displacements $\vec{k}_i = \vec{V}_i t_i$. Note that $\vec{T}$ is an $r \times 1$ column vector, and that $\vec{k}_i$ is an $m \times 1$ column vector. Given the periods t_i, choosing $\vec{k}_i$ immediately determines the velocity $\vec{V}_i$ and vice versa. In searching for parameter values, we choose to consider $\vec{k}_i$ and not $\vec{V}_i$.

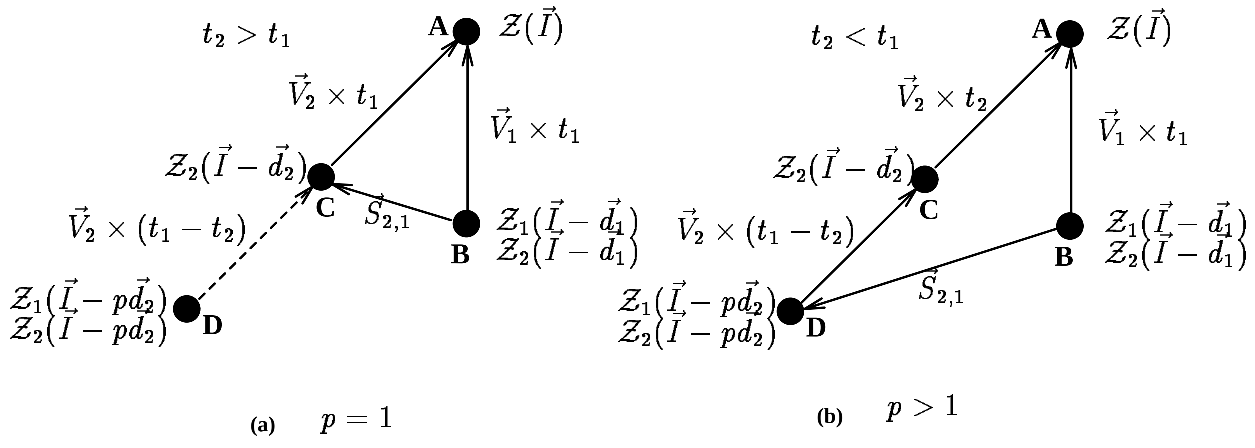
2.3.1 Constraints for correct systolic processing

The following theorem relates the parameters defined in GPM for correct systolic processing of the given uniform dependence algorithm.

Theorem 2.1 *The parameters velocities, spacings, and periods must satisfy the following constraint equations for correct systolic processing of the given uniform recurrence equations (Eq. (1.1)):*

$$\vec{V}_i t_i = \vec{V}_j t_i + \vec{S}_{j,i}, \quad i, j = 1, 2, \cdots, r \ . \tag{2.6}$$

Proof. Consider the execution of index point $\vec{I}$ at PA A as shown in Figure 2.2. Eq. (2.6) can be proved by considering the movement of data tokens of variables i and j to PE A. Without loss of generality, let $i = 1$ and $j = 2$. Consider the movement of data token $Z_1(\vec{I} - \vec{d}_1)$ of the first variable to PE A. Let B be the PE where it was generated. In time



$t_2 > t_1$
A
$\mathcal{Z}(\vec{I})$
$\vec{V}_2 \times t_1$
$\vec{V}_1 \times t_1$
$\mathcal{Z}_2(\vec{I} - \vec{d}_2)$
C
$\vec{S}_{2,1}$
$\vec{V}_2 \times (t_1 - t_2)$
B
$\mathcal{Z}_1(\vec{I} - \vec{d}_1)$
$\mathcal{Z}_2(\vec{I} - \vec{d}_1)$
$\mathcal{Z}_1(\vec{I} - p\vec{d}_2)$
$\mathcal{Z}_2(\vec{I} - p\vec{d}_2)$
D
(a) $p = 1$
$t_2 < t_1$
A
$\mathcal{Z}(\vec{I})$
$\vec{V}_2 \times t_2$
$\vec{V}_1 \times t_1$
$\mathcal{Z}_2(\vec{I} - \vec{d}_2)$
C
$\vec{V}_2 \times (t_1 - t_2)$
B
$\mathcal{Z}_1(\vec{I} - \vec{d}_1)$
$\mathcal{Z}_2(\vec{I} - \vec{d}_1)$
$\vec{S}_{2,1}$
$\mathcal{Z}_1(\vec{I} - p\vec{d}_2)$
$\mathcal{Z}_2(\vec{I} - p\vec{d}_2)$
D
(b) $p > 1$

Figure 2.2: Proof of Theorem 1: Data movement between variables 1 and 2. For $p > 1$, token $Z_2(\vec{I} - p\,\vec{d}_2)$ becomes $Z_2(\vec{I} - \vec{d}_2)$ as it travels towards PE A.

t_1, when $Z_1(\vec{I} - \vec{d}_1)$ moves from PE B to PE A, the other $r - 1$ data tokens must move from their respective locations to PE A.

When $Z_1(\vec{I} - \vec{d}_1)$ was generated at PE B, $Z_2(\vec{I} - \vec{d}_1)$ also resides at PE B. However, when $Z_1(\vec{I} - \vec{d}_1)$ was generated, $Z_2(\vec{I} - \vec{d}_2)$ might not exist in the array. Let $Z_2(\vec{I} - p\vec{d}_2)$, $p \geq 1$, be the value available in the dependence chain along direction $\vec{d}_2$ passing through index point $\vec{I}$, when $Z_1(\vec{I} - \vec{d}_1)$ is generated at PE B. Therefore, if t_2, the period along dependence $\vec{d}_2$, is greater than t_1, then $p = 1$, else $p > 1$.

Case 1: $p = 1$ (refer to Figure 2.2(a)). By definition, $\vec{S}_{2,1}$ denotes the distance between $Z_2(\vec{I} - \vec{d}_1)$ and $Z_2(\vec{I} - \vec{d}_2)$. By vector composition, we have $\vec{BA} = \vec{BC} + \vec{CA}$ which leads to Eq. (2.6).

Case 2: $p > 1$ (refer to Figure 2.2(b)). The distance between $Z_2(\vec{I} - \vec{d}_1)$ and $Z_2(\vec{I} - p\vec{d}_2)$ (or $\vec{BD}$) is needed to prove the theorem. The key observation is that token $Z_2(\vec{I} - p\vec{d}_2)$ refers to the *same* element of variable 2 for all p. This is true because variable 2 is pipelined along $\vec{d}_2$ in the index space and propagates through the array between the execution of indices differing by $\vec{d}_2$. Hence, irrespective of the value of p, $\vec{BD} = \vec{S}_{2,1}$. Again, by vector composition, the theorem is proved. ■

The constraints in Eq. (2.6) ensure that in computing an index point $\vec{I}$ at any processor in the array, all of the participating data tokens are present at the processor at the same time, moving from their respective processors where they were used earlier. A total of r^2 vector constraints are obtained from Theorem 2.1.

2.3.2 Constraints for linearly dependent dependence vectors

Let $\mathbf{S} = \left[\vec{S}_{i,j}\right]$, $i, j = 1, 2, \cdots, r$, be an $r \times r$ "matrix" (actually, a matrix of vectors) of spacings such that the (i,j)-th element of the matrix is $\vec{S}_{i,j}$. Note by definition that $\vec{S}_{i,i} = 0$. Let $\mathbf{S}_i$ be the i-th "row" of $\mathbf{S}$, i.e., $\mathbf{S}_i = \left[\vec{S}_{i,1}\vec{S}_{i,2}\cdots\vec{S}_{i,r}\right]$ (where $\mathbf{S}_i$ is an $m \times r$ matrix). Since $\vec{S}_{i,j} = \vec{V}_j t_j - \vec{V}_i t_j = \vec{k}_j - \vec{V}_i t_j$ from Theorem 2.1, $\mathbf{S}_i$ can be written in matrix form as

$$\mathbf{S}_i = \mathbf{K} - \vec{V}_i \otimes \vec{T}, \tag{2.7}$$

where $\vec{T}$ is a vector composed of periods and $\otimes$ is the outer product or tensor product, i.e., $\vec{a} \otimes \vec{b} = \vec{a}\,\vec{b}^t = [a_i b_j]$.

The next theorem characterizes the constraints on the periods and displacements if the dependence vectors in the recurrence are not linearly independent.

Let g be the rank of dependency matrix $\mathbf{D}$. Therefore, $\mathbf{N}$, the *null space* of $\mathbf{D}$, has $r-g$ basis vectors (as $\mathbf{D}$ has r columns). Let $\mathbf{N} = [\vec{\alpha}_1\ \vec{\alpha}_2\ \cdots\ \vec{\alpha}_{r-g}]$ be an $r \times (r-g)$ matrix, where $\vec{\alpha}_i$, $i = 1, 2, \cdots (r-g)$, are the basis vectors of the null space of $\mathbf{D}$. Hence,

$$\mathbf{D} \cdot \vec{\alpha}_i \;=\; 0, \quad 1 \le i \le (r-g)\,. \tag{2.8}$$

Theorem 2.2 *The periods t_i and the displacements $\vec{k}_i$ are related as follows:*

$$\vec{T} \cdot \mathbf{N} \;=\; \vec{0} \tag{2.9}$$

$$\mathbf{K}\,\mathbf{N} \;=\; \vec{0} \tag{2.10}$$

where $\mathbf{N}$ is the matrix consisting of the basis vectors of the null space of $\mathbf{D}$.

Proof. Consider some column $\vec{\alpha}_i$ of matrix $\mathbf{N}$. To prove the theorem, we show that $\vec{T} \cdot \vec{\alpha}_i = 0$ and $\mathbf{K} \cdot \vec{\alpha}_i = 0$. If the recurrence is computable, i.e., the DG is acyclic, then vector $\vec{\alpha}_i$ should

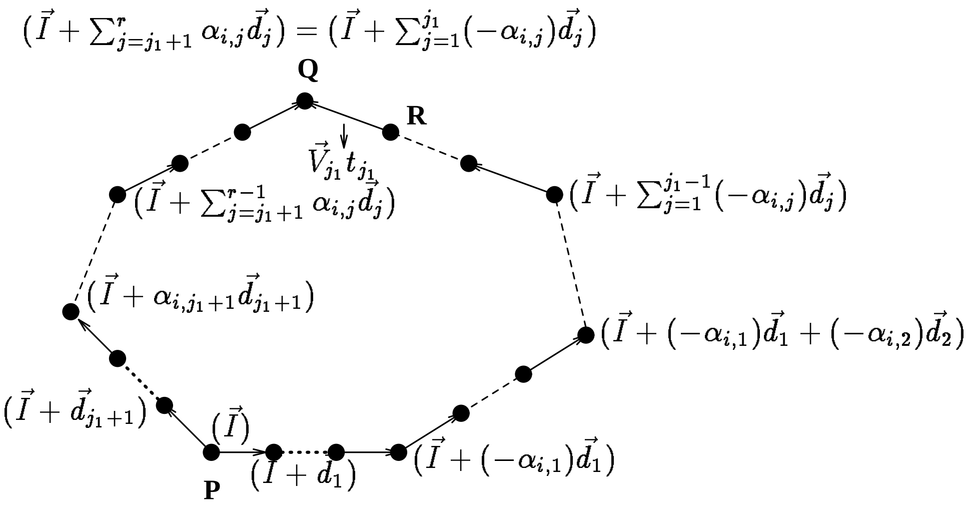

$(\vec{I}+\sum_{j=j_1+1}^{r}\alpha_{i,j}\vec{d}_j) = (\vec{I}+\sum_{j=1}^{j_1}(-\alpha_{i,j})\vec{d}_j)$
Q
R
$\vec{V}_{j_1} t_{j_1}$
$(\vec{I}+\sum_{j=j_1+1}^{r-1}\alpha_{i,j}\vec{d}_j)$
$(\vec{I}+\sum_{j=1}^{j_1-1}(-\alpha_{i,j})\vec{d}_j)$
$(\vec{I}+\alpha_{i,j_1+1}\vec{d}_{j_1+1})$
$(\vec{I}+(-\alpha_{i,1})\vec{d}_1+(-\alpha_{i,2})\vec{d}_2)$
$(\vec{I}+\vec{d}_{j_1+1})$
$(\vec{I})$
$(\vec{I}+\vec{d}_1)$
$(\vec{I}+(-\alpha_{i,1})\vec{d}_1)$
P

Figure 2.3: The dependency loop: $\sum_{j=1}^{r}\alpha_{i,j}\,\vec{d}_j = 0$ or $\sum_{j=1}^{j_1}(-\alpha_{i,j})\,\vec{d}_j = \sum_{j_1+1}^{r}\alpha_{i,j}\,\vec{d}_j$.

have at least one negative component, i.e., $\alpha_{i,j} < 0$ for some j, $1 \le j \le r$. Let j_1 be the number of negative components of basis vector $\vec{\alpha}_i$. Without loss of generality, assume that the first j_1 components of $\vec{\alpha}_i$ are negative, i.e., $\alpha_{i,j} < 0$, $j = 1, \ldots, j_1$. Since $\vec{\alpha}_i$ is a vector in the null space of $\mathbf{D}$, $\sum_{j=1}^{r}\alpha_{i,j}\,\vec{d}_j = 0$, which leads to $\sum_{j=1}^{j_1}(-\alpha_{i,j})\,\vec{d}_j = \sum_{j=j_1+1}^{r}\alpha_{i,j}\,\vec{d}_j$.

Consider the execution locations of indices $\vec{I}$, $\vec{I}+d_1$, $\cdots$, $\vec{I}+\sum_{j=1}^{r}\alpha_{i,j}\,\vec{d}_j = 0$ as shown in Figure 2.3. Let PE P execute index $\vec{I}$, and PE Q, index $\vec{I}+\sum_{j=1}^{j_1}(-\alpha_{i,j})\,\vec{d}_j$. Since $\sum_{j=1}^{j_1}(-\alpha_{i,j})\,\vec{d}_j = \sum_{j=j_1+1}^{r}\alpha_{i,j}\,\vec{d}_j$, there are two distinct paths from PE P to PE Q: path 1 composed of $\vec{d}_1 \ldots \vec{d}_{j_1}$; and path 2 composed of $\vec{d}_{j_1+1} \ldots \vec{d}_r$.

The time elapsed between the execution of the index point $\vec{I}$ at PE P and the corresponding index point at PE Q must be the same along paths 1 and 2. Therefore,

$$\tau_c\left(\vec{I}+\sum_{j=1}^{j_1}(-\alpha_{i,j})\,\vec{d}_j\right) - \tau_c(\vec{I}) \;=\; \underset{\text{(path 1)}}{\sum_{j=1}^{j_1}(-\alpha_{i,j})\,t_j} \;=\; \underset{\text{(path 2)}}{\sum_{j=j_1}^{r}\alpha_{i,j}\,t_j}$$

$$\Longrightarrow\; \sum_{j=1}^{r}\alpha_{i,j}\,t_j = 0 \;\Longrightarrow\; \vec{T}\cdot\vec{\alpha}_i = 0,$$

Similarly, by considering the displacement between P and Q along paths 1 and 2, we have $\mathbf{K} \cdot \vec{\alpha}_i = 0$. ■

Theorem 2.2, therefore, provides a total of $2(r-g)$ constraints: $(r-g)$ scalar constraints and $(r-g)$ vector constraints.

The following corollary shows the constraints on spacings that follow from Theorem 2.2. In fact, these constraints can be shown to be equivalent to those in Theorem 2.2. The implication of this corollary is that, of the r spacing parameters for each variable, only $g-1$ of them are independent, one of them is zero, and the rest can be expressed as linear combinations of the $g-1$ independent ones.

Corollary 2.1 *The spacing parameters* $\mathbf{S}_i = \left[\vec{S}_{i,1} \cdots \vec{S}_{i,r}\right]$ *are constrained by the equations* $\mathbf{S}_i\mathbf{N} = 0$, $i = 1, 2, \cdots, r$, *where* $\mathbf{N}$ *is the matrix consisting of the basis vectors of the null space of* $\mathbf{D}$.

Proof. From Eq. (2.7), we know that $\mathbf{S}_i = \mathbf{K} - \vec{V}_i \otimes \vec{T}$. Using the property of outer products that $(\vec{a} \otimes \vec{b}) \cdot \vec{c} = (\vec{b} \cdot \vec{c})\, \vec{a}$, we have

$$\mathbf{S}_i \cdot \vec{\alpha}_i = \mathbf{K} \cdot \vec{\alpha}_i - (\vec{T} \cdot \vec{\alpha}_i)\, \vec{V}_i = 0$$

for any column $\vec{\alpha}_i$ of matrix $\mathbf{N}$. The corollary is proved by applying Theorem 2.2. ■

Example 2.3 For the recurrence in Eq. (1.5), the following constraint equations can be obtained by applying Theorem 2.1:

$$\vec{V}_1 t_1 = \vec{V}_2 t_1 + \vec{S}_{2,1} = \vec{V}_3 t_1 + \vec{S}_{3,1}$$
$$\vec{V}_2 t_2 = \vec{V}_1 t_2 + \vec{S}_{1,2} = \vec{V}_3 t_2 + \vec{S}_{3,2}$$
$$\vec{V}_3 t_3 = \vec{V}_2 t_2 + \vec{S}_{2,3} = \vec{V}_3 t_2 + \vec{S}_{1,3}.$$

Since $\mathbf{D}$ is an identity matrix with full rank, matrix $\mathbf{N}$ has only one all-zero vector $\vec{0}$, and Theorem 2.2 is trivially satisfied. ■

Example 2.4 From Theorem 2.1, the constraint equations for the recurrence in Eq. (1.7) (excluding the trivial constraint $\vec{V}_1 t_1 = \vec{V}_1 t_1 + \vec{S}_{1,1}$) are

$$\vec{V}_1 t_1 = \vec{V}_2 t_1 + \vec{S}_{2,1} = \vec{V}_3 t_1 + \vec{S}_{3,1} = \vec{V}_4 t_1 + \vec{S}_{4,1} = \vec{V}_5 t_1 + \vec{S}_{5,1}.$$

Similarly, there are 16 additional equations related to $\vec{V}_2 t_2$, $\vec{V}_3 t_3$, $\vec{V}_4 t_4$, and $\vec{V}_5 t_5$.

The rank of $\mathbf{D}$ defined in Eq. (1.9) is 3. Hence, $\mathbf{N}$ is comprised of two basis vectors.

$$\mathbf{N} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}.$$

From Theorem 2.2, the additional constraints are

$$t_4 = t_1 + t_3 \qquad t_5 = t_2 + t_3 \tag{2.11}$$

$$\vec{k}_4 = \vec{k}_1 + \vec{k}_3 \qquad \vec{k}_5 = \vec{k}_2 + \vec{k}_3. \tag{2.12}$$

In this example, there are a total of 27 vector constraints and 2 scalar constraints. ■

To summarize, a total of $r^2 + r$ vector parameters and r scalar parameters have been defined whose values have to be determined. Theorems 2.1 and 2.2 give a total of $r^2 + (r - g)$ vector constraints and $(r - g)$ scalar constraints. Hence, g of the scalar parameters (periods) and g of the vector parameters have to be chosen such that the other $(r - g)$ scalar parameters and $r^2 + (r - g)$ vector parameter values can be determined from the chosen scalar and vector parameters. Since the performance of the design can naturally be expressed in terms of the periods and displacements, our strategy is to choose the g periods and g displacements to optimize a given performance criterion. The remaining $(r - g)$ periods, $(r - g)$ displacements, and all of the spacings can be determined from Theorems 2.1 and 2.2. All of the vector equations are solved in m-D space in order to obtain m-D vector parameters.

2.3.3 Constraints to govern valid space-time mappings

The validity of a space-time mapping is governed by the following fundamental conditions:

1. **Precedence Constraints.** An index point should be executed only after all of the index points on which it depends on have been executed. In DM, $\vec{\Pi} \cdot \mathbf{D} > 0$.

2. **Avoidance of Computational Conflicts.** No two index points should be executed on the same processor at the same time. In DM, $\vec{\Pi} \cdot \vec{I_1} = \vec{\Pi} \cdot \vec{I_2}$, implying that $\mathbf{S} \cdot \vec{I_1} \neq \mathbf{S} \cdot \vec{I_2}$.

3. **Avoidance of Data-Link Conflicts.** No two data tokens should contend for a given link at the same time.

Having established the parameters and the basic relationships among them in Theorems 2.1 and 2.2, we show how the fundamental conditions for valid space-time mappings are satisfied in GPM.

By definition, periods denote the time difference between the source and sink of dependencies. Hence, the precedence constraint is satisfied by simply enforcing $t_i \geq 1, i = 1, \cdots, r$. In the array model, all tokens of the same variable move with the same velocity. Hence, data-link conflicts can exist if and only if two tokens of a variable are input at the same time into the same processor and travel together contending for links. This condition is called a *data-input conflict* in GPM, as two data tokens are in the same physical location and conflict with each other as they move through the processors together.

It is important to note that in GPM, computational conflicts can exist if and only if data-input conflicts occur. This can be seen by the following simple argument. If two index points are evaluated in the same processor at the same time, then for each variable, at least two distinct tokens exist together in the same processor. Hence, if there is at least one nonstationary variable, then there are data-input conflicts for the tokens of that variable. Otherwise, all variables are stationary, and the entire computation is executed in one processor, i.e., there is no processor array. Hence, by enforcing that no data-input conflicts exist, both computational and data-link conflicts are avoided. Theorem 2.3 presents conditions under which data-input conflicts can be eliminated.

Consider the spacings of variable i. Let $\mathbf{S}_i'$ be an $m \times (g-1)$ matrix:

$$\mathbf{S}_i' = \left[\vec{S}_{i,1}\vec{S}_{i,2}\cdots\vec{S}_{i,g-1}\right] \tag{2.13}$$

where $\vec{S}_{i,1}\vec{S}_{i,2},\ldots,\vec{S}_{i,g-1}$ are the $g-1$ independent spacings. Let $\vec{\alpha},\vec{\beta},\vec{\gamma}$ be vectors with $g-1$ integral elements. Let $L_j, U_j, j = 1,2,\cdots,g-1$, be defined such that the distance of any token of the input matrix from a reference element (usually the first element) is equal to $\sum_{j=1}^{g-1}\vec{S}_{i,j}\beta_j$, where $L_j \le \beta_j \le U_j$, and L_j and U_j are functions of the size of the input matrix.

Theorem 2.3 *Data-input conflicts occur in the input matrix of nonstationary input i if and only if $\mathbf{S}_i' \cdot \vec{\alpha} = \vec{0}$, where $\vec{\alpha} = [\alpha_1,\alpha_2,\ldots,\alpha_{g-1}]^t \neq \vec{0}$, and $\alpha_i \in [(L_i - U_i),\ldots,(L_i + U_i)]$ for all i such that $1 \le i \le g-1$.*

Proof. The position of any element of input i can be described as $\mathbf{S}_i' \cdot \vec{\beta}$, where $\vec{\beta} = [\beta_1,\ldots,\beta_{g-1}]$ and $L_i \le \beta_i \le U_i$. Therefore,

$$\begin{aligned}
\textit{Data-input conflicts} &\iff \mathbf{S}_i' \cdot \vec{\beta} = \mathbf{S}_i' \cdot \vec{\gamma}, \quad \text{where } \vec{\beta} \neq \vec{\gamma} \text{ and } L_i \le \vec{\gamma}_i,\ \beta_i \le U_i \\
&\iff \mathbf{S}_i' \cdot (\vec{\beta} - \vec{\gamma}) = \vec{0} \\
&\iff \mathbf{S}_i' \cdot \vec{\alpha} = \vec{0}, \ \text{where} \vec{\alpha} = \vec{\beta} - \vec{\gamma} \neq \vec{0}, \alpha_i \in [(L_i - U_i),\ldots,(L_i + U_i)].
\end{aligned}$$

■

Note that in Theorem 2.3, we have defined conservative bounds on α_i. Better estimates can be obtained [44] and will result in less overhead when the conditions in Theorem 2.3 are checked in the design process.

Example 2.5 For the recurrence in Eq. (1.7), if the array sought is 1-D, then the spacing parameters are all scalars. Let $\vec{S}_{1,2}, \vec{S}_{1,5}$ be the two independent spacings for input $\mathbf{X}$, and we choose the values of $L_1 = L_2 = 1$ and $U_1 = U_2 = N$. According to Theorem 2.3, data-input conflicts occur in input $\mathbf{X}$ if and only if

$$\left[\vec{S}_{1,2}\ \vec{S}_{1,5}\right] \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = 0 \tag{2.14}$$

where $-(N-1) \leq \alpha_1, \alpha_2 \leq (N-1)$ and $\alpha_1, \alpha_2 \neq 0$. For instance, if $N = 5$, $\vec{S}_{1,2} = 6$, and $\vec{S}_{1,5} = 4$, then $\alpha_1 = 2$ and $\alpha_2 = -3$ satisfys Eq. (2.14). (In one dimension, the vector spacings are positive or negative numbers.) Hence, there are data-input conflicts in input **X**. ■

2.3.4 Constraints in preloaded data

If the velocity of a variable is zero, then the data corresponding to the variable have to be preloaded in the processors before computation begins. This problem involves designing a schedule that can overlap as much as possible the preloading of data with the systolic computations without delaying these computations. A general approach is to decide when a particular stationary datum is to be used in its first computation and to develop a preloading schedule so that the bandwidth constraint of the processor array is satisfied and that the first computation can begin with the minimum delay. We like to point out (a) that data do not have to be preloaded in any order governed by a dependence relation (as in systolic processing) as long as they do not conflict in using the interprocessor links, and the bandwidth of the input ports is not exceeded; (b) that the optimal preloading schedule may depend on the velocities and data distributions of the moving data; and (c) that preloading data may result in problem-size-dependent memory in each processor (a design alternative often disallowed in systolic arrays).

Since, the preloading schedule is problem-dependent, Section 3.1 will discuss how the effect of preloading is taken into account to find optimal designs for matrix product and transitive closure problems.

2.4 Design Method

2.4.1 Formulation of the search problem

The design of a feasible processor array is equivalent to choosing an appropriate set of parameters that satisfy the constraints imposed by dependency and application requirements for a *specific* uniform recurrence equation and a *specific* problem size N. The search for the "best" design can be represented by the following optimization problem.

$$\textbf{Minimize} \quad b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) \tag{2.15}$$

$$\textbf{Subject To:} \begin{cases} 1 \le t_i, \quad i = 1, \ldots, r, \\ 0 \le \left|\vec{k}_i\right| \le t_i, \quad i = 1, \ldots, r \\ \text{constraints defined in Theorems 2.1, 2.2 and 2.3} \\ \#PE \le \#PE^{UB} \text{ and } T_c \le T_c^{UB}. \end{cases} \tag{2.16}$$

The objective function b defined in Eq. (2.15) is expressed in terms of attributes such as T_{comp}, computation time of the algorithm, T_{load}, load time for the initial inputs, T_{drain}, drain time for the final results, and $\#PE$, number of processing elements in the design. Note that the completion time of evaluating a recurrence is

$$T_c = T_{comp} + T_{load} + T_{drain}. \tag{2.17}$$

All of the attributes are then expressed in terms of the parameters defined in GPM.

The first two constraints in Eq. (2.16) follow directly from the definition of the parameters in GPM. Since the target array is systolic, displacement $\left|\vec{k}_i\right|$ should not exceed period t_i in order to prevent data broadcasting (velocities should not exceed one). In addition, the constraints $t_i \ge 1$, $i = 1, 2, \ldots, r$, mean that precedence constraints are satisfied.

The third constraint indicates that the recurrence is evaluated correctly by the processor array, satisfying dependency requirements (Theorems 2.1 and 2.2) and be free of data-link and computational conflicts (Theorem 2.3).

The fourth constraint indicates bounds on T_c and $\#PE$ imposed on the design to be obtained. For instance, the following are two possible formulations of the optimization problem:

- Minimize T_c for a design with a maximum bound on $\#PE$, $\#PE^{UB}$;
- Minimize $\#PE$ for a design with a maximum bound on T_c, T_c^{UB}.

Both of these formulations represent trade-offs between T and $\#PE$.

The optimal design for the formulation given by Eqs. (2.15) and (2.16) is found by a search algorithm. Since, in general, the objective function is nonlinear, involving functions such as ceiling, floor, and maximum/minimum of a set of terms, it is difficult to describe a comprehensive algorithm that covers all possible cases. In the rest of this section, we first describe a pruning strategy used in our search algorithm, followed by a discussion on searches with objectives that are functions of T_c, T_{comp}, T_{drain}, and $\#PE$. We then present the search algorithm and show its application for special cases of optimizing T_c and $\#PE$.

2.4.2 Pruning strategy

The search space defined by the constraints in Eq. (2.16) results in a worst-case size of

$$O\left(\sum_{i=1}^{g}(t_i^{max})^2\right) = O\left((T_{comp}^{seq})^{2g}\right), \tag{2.18}$$

where T_{comp}^{seq} is the time needed to process the recurrence sequentially, and t_i^{max} is the maximum value of period t_i such that the computation time $T_{comp} \leq T_{comp}^{seq}$. Eq. (2.18) is true because we iterate in the worst case all combinations of t_i and $\left|\vec{k}_i\right| \leq t_i$, $i = 1, \ldots, r$. Note that the size of this search space is polynomial in terms of the parameters in GPM and the size of the URE to be evaluated.

To reduce this search space, we have to develop effective pruning strategies so that suboptimal designs do not have to be evaluated. In this section, we present one such strategy that prunes based on incumbent designs obtained in the search. Our pruning strategy takes the objective function b (assuming to be minimized) and decomposes it as follows:

$$b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) = f\left(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r, e(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r)\right), \tag{2.19}$$

where N is not represented explicitly since it is a constant in the optimization. The decomposition is done in such a way that $e()$[2] is a monotonic function of its variables, which may be a subset of $t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r$. The intuition behind this decomposition is as follows.

[2]For notational ease, we denote functions without their arguments

If the objective function $b(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r)$ is a monotone function of its variables, then the optimal value of the parameters can be found by enumerating combinations of values of variables from their smallest permissible values (given by Eq. (2.16)) until a feasible design that satisfies Theorems 2.1, 2.2 and 2.3 is found. Since $b()$ is monotonic, the first feasible design obtained is also the optimal design.

The above idea of enumerating values of a monotone function can be extended to the general case of nonmonotonic objective functions. This is done by first identifying $e()$, a monotonic component of the objective that can be enumerated efficiently. The search proceeds by enumerating designs so that values of $e()$ grow monotonically. (The combination of parameter values used in $e()$ are substituted into Eq. (2.16), and the constraint equations are solved to see if there exists a feasible design.) Whenever a feasible design is obtained, an upper bound on $e()$ is computed by setting variables in $b()$ that are not included in $e()$ to their extremum values. (This upper bound means that no optimal design will have an objective value whose monotonic component $e()$ is larger than the upper bound.) The search is then repeated, refining the upper bound each time a feasible design is found. It stops when the upper bound on $e()$ is smaller than or equal to $e()$ of the best feasible design.

From the above description, it is clear that the search strategy is an adaptation of a general branch-and-bound search. The monotonic component $e()$ corresponds to the variables searched in the branch-and-bound process, and bounding is performed by evaluating the lower bound every time a feasible solution is found.

For complex objective functions, rewriting the objective in terms of composite variables (expressed in terms of the primary variables $t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r$) can simplify the choice of the extremum values for variables other than those in $e()$. This is illustrated as follows.

Consider an objective expressed as a function of composite variables T_{comp}, T_{load}, T_{drain}, and $\#PE$ as follows:

$$B = b\left(T_{comp}, T_{load}, T_{drain}, \#PE\right). \tag{2.20}$$

It is easy to see that $T_{comp} = T_{comp}(t_1, \ldots, t_r)$ is monotonic with respect to the g periods $t_1, \ldots, t_r$. (An exact characterization is shown in Lemma 3.1 (p. 44) in Section 3.1.1 for the transitive-closure problem.) Hence, we choose T_{comp} as the monotonic component of objective function $b()$ and enumerate the periods $t_1, \ldots, t_r$ in an increasing order from their smallest permissible values, i.e., $t_1, \ldots, t_r = 1$.

The upper bound T_{comp}^{UB} can be refined if $b()$ is monotonically increasing with T_{comp}, T_{load}, T_{drain} and $\#PE$. In this case, T_{comp}^{UB} can be obtained by setting $T_{load} = T_{drain} = 0$, $T_{comp} = T_{comp}^{min}$, and $\#PE = \#PE^{min}$ and solving

$$\begin{aligned} B^{inc} &= b\left(T_{comp}^{UB}, T_{load}^{min}, T_{drain}^{min}, \#PE^{min}\right) && (2.21) \\ &= b\left(T_{comp}^{UB}, 0, 0, \#PE^{min}\right), && (2.22) \end{aligned}$$

where B^{inc} is the objective value of the current incumbent design (best feasible design found so far). Hence,

$$T_{comp}^{UB} = b^{-1}\left(B^{inc}, T_{load}^{min}, T_{drain}^{min}, \#PE^{min}\right), \tag{2.23}$$

where $b^{-1}()$ is the inverse function of $b()$ that rearranges Eq. (2.22) to compute T_{comp}^{UB} in terms of known constants.

The upper bound T_{comp}^{UB} can further be refined if $\#PE$ can be expressed as a function of $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_r\right|$. In this case, $\#PE$ is minimum when exactly one of the $\left|\vec{k}_i\right|$s is 1, and the rest of the $\left|\vec{k}_j\right|$, $j \neq i$, are 0. (An exact characterization is shown in Lemma 3.2 (p. 45) in Section 3.1.1 for the transitive-closure problem.)

For instance, suppose the objective function is

$$B = (T_{comp} + T_{load} + T_{drain})^2 \times \#PE\ . \tag{2.24}$$

According to Eq. (2.22), we have

$$\begin{aligned} B^{inc} &= (T_{comp} + 0 + 0)^2 \times \#PE \\ \Longrightarrow\ T_{comp}^{UB} &= \sqrt{B^{inc}/\#PE^{min}}. && (2.25) \end{aligned}$$

Similarly, if the objective function to minimize completion time T_c,

$$\begin{aligned} B &= T_c = T_{comp} + T_{load} + T_{drain} \\ \Longrightarrow\ T_{comp}^{UB} &= B^{inc} - (T_{load}^{min} + T_{drain}^{min}) = B^{inc} - (0+0) = B^{inc} = T_c^{inc}. && (2.26) \end{aligned}$$

The upper bound T_{comp}^{UB} is refined continuously as new incumbent designs are found in the search. The search stops when there is no combination of t_i, $i = 1, \ldots, r$, that satisfies $T_{comp} \leq T_{comp}^{UB}$.

A special case of the optimization is to find a design with the minimum computation time T_{comp} (not including load and drain times). Here, $T_{comp}^{UB} = B^{inc} = T_{comp}^{inc}$, and the first feasible design is the optimal design that minimizes T_{comp}.

2.4.3 Search procedure

In this section, we present our search procedure for minimizing $b(\#PE, T_c) = b(T_{comp}, T_{load}, T_{drain}, \#PE)$ (Eq. (2.20)), where T_{comp} is a function of $t_1, \ldots, t_r$; T_{load} and T_{drain} are functions of $t_1, \ldots, t_r$, $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_r\right|$; and $\#PE$ is a function of $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_r\right|$.

1. Choose g periods and g displacements to be unconstrained parameters. Without loss of generality, let these periods and displacements be t_i and $\vec{k}_i$, $1 \leq i \leq g$, respectively.

2. Initialize T_{comp}^{UB} to be T_{comp}^{seq}, the computation time required to evaluate the recurrence sequentially.

3. Set the values of all the g unconstrained periods t_i, $i = 1, \ldots, g$, to be 1.

4. Choose the magnitude of the g unconstrained displacements $\left|\vec{k}_i\right|$, $i = 1, \ldots, g$, to be zero.

5. Compute the values of the other dependent $r-g$ periods and $r-g$ displacements using the conditions of Theorem 5.2.

6. Compute T_{comp}^{cur} using the periods and displacements found, where T_{comp}^{cur} is the computation time (without load and drain times) required for processing the recurrence. The value of T_{comp}^{cur} is found by substituting the current values of t_i, $i = 1, \ldots, r$, in Eq. (2.15). (Note that the design may not be feasible at this time.) If $T_{comp}^{cur} > T_{comp}^{UB}$, then exit with the incumbent design.

7. Solve for the spacing parameters from Eq. (2.6) defined in Theorem 2.1.

8. Check for data-input conflicts using Theorem 2.1 on the spacing parameters; also, check whether the constraints on T_c and $\#PE$ are violated (Constraint 4 in Eq. (2.16)).

9. If the solution is not feasible, then increment one of the $\left|\vec{k}_i\right|$s and repeat Steps 5, 6, 7 and 8 until all $\left|\vec{k}_i\right|$ equal t_i, $i = 1, \ldots, r$. If all $\left|\vec{k}_i\right|$ equal t_i and no feasible design is found, then go to Step 10. If a feasible design is found, then go to Step 11.

10. Increment one of the periods such that T_{comp}^{cur} increases by the lowest possible value. Go to Step 4.

11. Compute B^{cur}, the objective value achieved by the current design found. If $B^{cur} < B^{inc}$, then set $B^{inc} = B^{cur}$, and compute T_{comp}^{UB} for the current design using Eq. (2.23). Increment one of the $\left|\vec{k}_i\right|$s and go to Step 5.

For a design that minimizes $\#PE$, the search procedure described above has to be changed. In this case, $e()$ should be defined as a function of $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_r\right|$, and the search should start iterating with the smallest combinations of $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_g\right|$.

2.5 Summary

Algorithm-specific parallel processing with processor arrays can be systematically accomplished with the help of the general parameter-based approach (GPM) discussed in this chapter. The techniques discussed in this chapter are ideally suited to nested loops described as uniform recurrences or as affine recurrences that can be uniformized.

In GPM, the behavior of the target array is captured by a set of parameters, and the design problem is formulated as an optimization problem with an objective and a set of constraints specified in terms of the parameters. The parameters in GPM can be expressed in terms of the processor-allocation matrix $\mathbf{S}$ and the time schedule vector $\vec{\Pi}$ in dependency-based methods (DMs), thereby allowing GPM to be used in DMs to find optimal designs. We present an efficient search procedure for finding T_c-optimal or T_{comp}-optimal (respectively, $\#PE$-optimal) designs for specified bounds on $\#PE$ (respectively, T_c or T_{comp}), as well as optimal designs with general objective functions. The distinct features of GPM lie in its ability to systematically search for optimal designs with specific design requirements on T_c (or T_{comp}) and $\#PE$, and in its ability to include constraints on data-link and computational conflicts in the optimization procedure. In conclusion, Table 2.1 summarizes the unique features of GPM and DM. The next chapter presents the results of applying GPM to two important applications: computing transitive closures and matrix products.

Table 2.1: Comparison between dependency-based and parameter-based methods.

Feature	Dependency-Based Method	Generalized Parameter Method
Applicable Recurrences	General and applicable to uniform as well as nonuniform recurrences.	Homogeneous uniform recurrences or uniformized affine recurrences.
Representation	Schedule Vector and Allocation Matrix: they are represented in the Cartesian coordinate system with unit vectors as basis vectors; for the dimension-reduction technique [2], the mappings are rank-deficient; (i.e., $\vec{\Pi}$ and $\mathbf{S}$ yield $\mathbf{T}$ where $rank(\mathbf{T}) \leq n$).	Periods and Displacements: they are represented in a possibly nonorthogonal coordinate system with dependence vectors as basis vectors; hence, for uniform recurrences, the representations in DM and GPM are equivalent and are derivable from each other by a coordinate (linear) transformation.
Characteristics of controls in processor array	Nonuniform in the general case by specifying a general processor allocation matrix; processor arrays derived may have in the general case arbitrary speed/direction changes for data tokens and have aperiodic computations.	Uniform controls throughout the processor array, resulting in constant velocities and periodic computations.
Design objective and constraints	Computation-time optimal designs or processor-optimal designs with linear objective function and linear constraints.	General nonlinear objective function and constraints with certain monotonicity properties on the objective function; new constraints have been developed that avoid data-link conflicts.
Search methods for finding processor array designs	Choose processor-allocation matrix heuristically, and find schedule vector satisfying processor-allocation constraints; methods for finding designs are based on linear/integer programming or intelligent searches.	Search method is systematic enumeration and pruning on a search space whose size is polynomial with respect to problem size.
Designs obtained	Designs found are optimal in computation time with respect to a given choice of processor-allocation matrix; possible allocation matrices chosen are those that minimize the number of processing elements.	Trade-offs between number of processors and computation time, or between number of processors and completion time (including load and drain times) for a specific problem instance can be obtained (Chapter 3).
Summary	The two methods are equivalent representations for synthesizing uniform recurrences. The formulation of the design optimization problem and the search techniques developed are equally applicable in both representations.	

3. APPLICATIONS OF GENERAL PARAMETER METHOD

This chapter presents the application of the GPM developed in Chapter 2 to two important application problems: computing the transitive closure of a matrix (Section 3.1) and computing the product of two matrices (Section 3.2). For each application, the performance attributes are derived first (Sections 3.1.1, 3.2.1), followed by time-optimal and processor-optimal linear array designs for different objectives (Section 3.1.2, 3.2.2, 3.2.4). Sections 3.1.3 and 3.2.5 show the continuous processor-time trade-offs obtained using GPM, and the chapter concludes with a summary in Section 3.3.

3.1 Algebraic Path-finding Problems: Transitive Closure

Algebraic path-finding problems (APPs) belong to an important class of optimization problems. Typical examples include computing the transitive closure and the shortest paths of a graph. Other problems in the class of APPs include the generation of regular expressions from finite automata [45], matrix inversion, and Gauss-Jordan elimination. For an overview of applications of APPs see references [46, 47, 48]. A common algebraic framework for graph algorithms and numerical algorithms in terms of APPs was first achieved by Lehmann [49]. The application of APPs in global flow analysis of programs useful for code optimization is discussed in reference [50]. Two-dimensional processor arrays for finding transitive closures have been presented before [51, 52]. In this section we synthesize a one-pass linear processor array for the transitive-closure problem using the Floyd-Warshall path-finding algorithm.

The transitive-closure problem is defined as follows. Given an N-node directed graph with an $N \times N$ Boolean adjacency matrix $\mathbf{C}[i,j]$, $1 \leq i,j \leq N$, the transitive closure $\mathbf{C}^{+}[i,j] = 1$ if there exists a path from node i to node j, where $\mathbf{C}[i,j] = 1$ if there is an edge from node i to node j or $i = j$, and $\mathbf{C}[i,j] = 0$ otherwise. That is,

$$\begin{aligned} &\textbf{for} \quad (k=1,N; i=1,N; j=1,N) \\ &\qquad \mathbf{C}(i,j) = \mathbf{C}(i,j) + \mathbf{C}(i,k) \cdot \mathbf{C}(k,j). \end{aligned} \tag{3.1}$$

The dependence structure of a general dynamic programming formulation of the transitive-closure problem is irregular and difficult to map on a regularly connected planar processor array. To cope with this mapping problem, S.Y. Kung et. al., have converted the transitive-closure algorithm into an reindexed form and have mapped it onto 2-D spiral and orthogonal arrays [51]. Based on their algorithm we obtain the following five dependence vectors:

$$\begin{aligned} \vec{d_1} &= (0,0,1)^t \text{ for } (k,i,j)^t \leftarrow (k,i,j-1)^t, \ \ 2 \leq j \leq N, \\ \vec{d_2} &= (0,1,0)^t \text{ for } (k,i,j)^t \leftarrow (k,i-1,j)^t, \ \ 2 \leq i \leq N, \\ \vec{d_3} &= (1,-1,-1)^t \text{ for } (k,i,j)^t \leftarrow (k-1,i+1,j+1)^t, 2 \leq k \leq N, 1 \leq i,j \leq N-1, \\ \vec{d_4} &= (1,-1,0)^t \text{ for } (k,i,N)^t \leftarrow (k-1,i+1,N)^t, \ \ 2 \leq k \leq N, \ \ 1 \leq i \leq N-1, \\ \vec{d_5} &= (1,0,-1)^t \text{ for } (k,N,j)^t \leftarrow (k-1,N,j+1)^t, \ \ 2 \leq k \leq N, \ \ 1 \leq j \leq N-1, \end{aligned} \tag{3.2}$$

where $\vec{I_1} \leftarrow \vec{I_2}$ means that the data at index point $\vec{I_2}$ are used at index point $\vec{I_1}$. For nodes on the boundary of the dependence graph, where $i = N$ (respectively, $j = N$), and dependence $\vec{d_4}$ (respectively, $\vec{d_5}$) is present instead of dependence $\vec{d_3}$. For other interior points, only 3 dependencies, $\vec{d_1}, \vec{d_2}, \vec{d_3}$, exist. The five dependence vectors listed above are identical to the dependencies in Example 1.4 (Eq. (1.9)).

The key observation is as follows. Matrix $\mathbf{C}$ (whose transitive closure is to be found) is input along dependence vector $\vec{d_3}$. Inputs along other dependence vectors, $\vec{d_1}$, $\vec{d_2}$, $\vec{d_4}$, $\vec{d_5}$, are nonexistent; i.e., they are never sent into the array from the external host. Hence, there are no data-input conflicts along these dependence directions. As a result, we have to consider data-input conflicts only along direction $\vec{d}_3$. Since dependencies $\vec{d}_3$, $\vec{d}_4$ and $\vec{d}_5$ never co-exist, there are only two spacings for data along direction $\vec{d}_3$, namely, $\vec{S}_{3,1}$ and $\vec{S}_{3,2}$.

A total of 8 relevant parameters are defined for the transitive-closure problem: 3 periods t_1, t_2, t_3, 3 displacements $\vec{k}_1, \vec{k}_2, \vec{k}_3$, and 2 spacings $\vec{S}_{3,1}$, $\vec{S}_{3,2}$. For a linear processor array, all of the parameters are scalars. As derived in Example 2.5, the periods and velocities along directions $\vec{d}_4$ and $\vec{d}_5$ are given as $t_4 = t_1 + t_3$, $t_5 = t_2 + t_3$ (Eq. (2.11)), $\vec{k}_4 = \vec{k}_1 + \vec{k}_3$, and $\vec{k}_5 = \vec{k}_2 + \vec{k}_3$ (Eq. (2.12)), respectively. From Theorem 2.1 and Eq. (2.3), we have

$$\vec{S}_{3,1} = \frac{t_3\vec{k}_1 - t_1\vec{k}_3}{t_3}, \quad \vec{S}_{3,2} = \frac{t_3\vec{k}_2 - t_2\vec{k}_3}{t_3}. \tag{3.3}$$

In the rest of this section, linear array designs are presented for the following formulations of the optimization (Section 2.4): i) T_{comp}-optimal designs without bound on $\#PE$, ii) T_c-optimal designs without bound on $\#PE$, iii) $\#PE$-optimal designs without bound on T_c or T_{comp}, and iv) optimal designs with specific bounds on T_{comp} or $\#PE$, and v) optimal designs with specific bounds on T_c or $\#PE$.

3.1.1 Performance attributes and constraints

Before optimal designs can be found, we have to express performance attributes in the objective function in terms of the parameters in GPM. The attributes of interest are T_{comp}, T_{load}, T_{drain}, $\#PE$, and T_c, where $T_c = T_{load} + T_{comp} + T_{drain}$. In this section, we show three lemmas that express these performance attributes in terms of the parameters defined. We also show two constraints that refine the constraints defined in Theorem 2.3.

Lemma 3.1 *The computation time T_{comp} without load and drain times for finding an $N \times N$ transitive closure is given by*

$$T_{comp} \quad = \quad (N-1)(2t_1 + 2t_2 + t_3) + 1. \tag{3.4}$$

Proof. The critical path in the execution is as follows:

$$(1,1,1) \overset{(N-1)t_1}{\longrightarrow} (1,1,N) \overset{(N-1)t_2}{\longrightarrow} (1,N,N) \overset{(N-1)t_3}{\longrightarrow} (N,1,1) \overset{(N-1)t_1}{\longrightarrow} (N,1,N) \overset{(N-1)t_2}{\longrightarrow} (N,N,N).$$

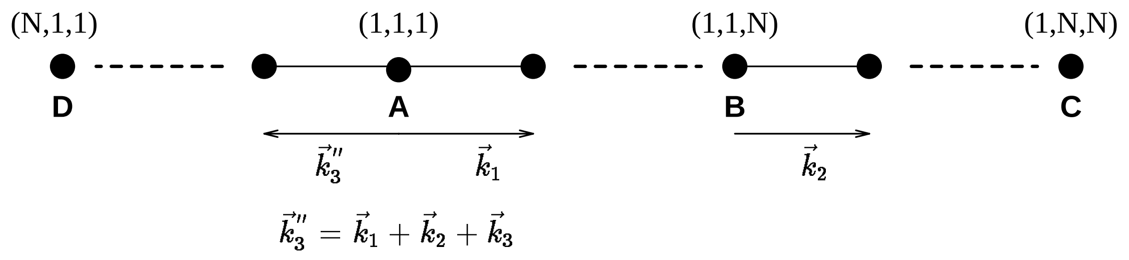


Figure 3.1: PE allocation with $\vec{k}_1$, $\vec{k}_2 \geq 0$ and $\vec{k}_3'' \leq 0$.

Thus, T_{comp} is $(N-1)(2t_1 + 2t_2 + t_3) + 1$. ■

Lemma 3.2 *#PE, the number of processors for computing an $N \times N$ transitive closure on a linear processor array satisfying the dependencies in Eq. (3.2), is given by*

$$\#PE = (N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_1 + \vec{k}_2 + \vec{k}_3\right|) + 1. \tag{3.5}$$

Proof. Let $\vec{k}_3''$ be the displacement from the execution location of index point (k, i, j) to index point $(k+1, i, j)$. From the definition of displacements $\vec{k}_1, \vec{k}_2, \vec{k}_3$ (Eq. (2.4)) for the dependencies $\vec{d}_1, \vec{d}_2, \vec{d}_3$ (Eq. (3.2)), the displacement $\vec{k}_3''$ is equal to $\vec{k}_1 + \vec{k}_2 + \vec{k}_3$ as depicted below.

$$(k, i, j) \xrightarrow{\vec{k}_1} (k, i, j+1) \xrightarrow{\vec{k}_2} (k, i+1, j+1) \xrightarrow{\vec{k}_3} (k+1, i, j) \; .$$

Consider the displacements $\vec{k}_1$, $\vec{k}_2$, and $\vec{k}_3''$. Two of these 3 displacements should be in the same direction, since the array is 1-D. Assume that $\vec{k}_1$ and $\vec{k}_2$ are positive displacements, i.e., they correspond to velocities flowing to the right (refer to Figure 3.1). Let A be the PE where the computation indexed by $(1, 1, 1)$ occurs. Therefore, computation corresponding to index point $(1, 1, N)$ is executed at PE B at a distance $(N-1)\left|\vec{k}_1\right|$ from PE A. Similarly, computation corresponding to the index point $(1, N, N)$ is executed at PE C that is $(N-1)\left|\vec{k}_2\right|$ PEs to the right of B. On the other hand, computation corresponding to index point $(N, 1, 1)$ is executed at PE D, that is, at a distance of $(N-1)\left|\vec{k}_3''\right|$ to the left of PE A (since $\vec{k}_3$ corresponds to the left moving variable). All other computations in the domain are executed by PEs between C and D. Therefore, the total number of PEs is $(N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_3''\right|) + 1 = (N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_1 + \vec{k}_2 + \vec{k}_3\right|) + 1$. ■

Lemma 3.3 *Assuming that the input matrix is nonstationary, T_{load}, the load time, and T_{drain}, the drain time, for computing an $N \times N$ transitive closure on a linear processor array satisfying the dependencies defined in Eq. (3.2) are given by*

$$\begin{aligned} T_{load} = T_{drain} = 1 \quad &+ \quad (N-1)\left\lceil \frac{t_3\left\{\mathcal{G}(\vec{k}_1, \vec{k}_3) + \mathcal{G}(\vec{k}_2, \vec{k}_3) + \mathcal{G}((\vec{k}_1 + \vec{k}_2 + \vec{k}_3), \vec{k}_3)\right\}}{\left|\vec{k}_3\right|}\right\rceil \\ &+ \quad (N-1)\frac{\left[\mathcal{G}(\vec{S}_{3,1}, -\vec{k}_3) + \mathcal{G}(\vec{S}_{3,2}, -\vec{k}_3)\right]}{|\vec{v}_3|} \end{aligned} \tag{3.6}$$

where

$$\mathcal{G}(\vec{x}, \vec{y}) = \begin{cases} |\vec{x}| & \text{if } \vec{x} \text{ and } \vec{y} \text{ are in opposite directions} \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

Proof. Since the first index point executed is $(1, 1, 1)$, the load time is the time for $\mathbf{C}_{1,1}$ to reach the PE executing index $(1, 1, 1)$. Let A be the PE that executes index $(1, 1, 1)$ (refer to Figure 3.1). Let C be the boundary PE for the inputs (assuming **C** flows to the left). The load time, T_{load}, is the sum of the time for $\mathbf{C}_{1,1}$ to reach PE C, and the time for $\mathbf{C}_{1,1}$ to move from PE C to PE A.

Since the layout of the processor array is governed by the displacements $\vec{k}_1$, $\vec{k}_2$ and $\vec{k}_3''$ $(= \vec{k}_1 + \vec{k}_2 + \vec{k}_3$; see proof for Lemma 3.2), the distance (l_1) from PE C to PE A depends on the relative signs of $\vec{k}_1$, $\vec{k}_2$, and $\vec{k}_3''$ with respect to $\vec{k}_3$. If $\vec{k}_1$ and $\vec{k}_2$ are in directions opposite to $\vec{k}_3$, then $l_1 = (N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right|)$. In general, $l_1 = (N-1)\left[\mathcal{G}(\vec{k}_1, \vec{k}_3) + \mathcal{G}(\vec{k}_2, \vec{k}_3) + \mathcal{G}(\vec{k}_3'', \vec{k}_3)\right]$, and the time to move from PE C to PE A is given by $\left\lceil \frac{l_1}{V_3} \right\rceil$.

The distance from $\mathbf{C}_{1,1}$ to PE C (denoted as l_2) is equal to the number of elements between $\mathbf{C}_{1,1}$ and PE C before any input element is sent into the array. Since the data-input pattern is dictated by $\vec{S}_{3,1}$ and $\vec{S}_{3,2}$, the distance l_2 from $\mathbf{C}_{1,1}$ to PE C again depends on the relative signs of $\vec{S}_{3,1}$ and $\vec{S}_{3,2}$ with respect to $\vec{k}_3$. If $\vec{S}_{3,1}$ and $\vec{S}_{3,2}$ are in the same direction as $\vec{k}_3$, then $\mathbf{C}_{1,1}$ is the first element of the input, and $l_2 = 1$. Similarly, if $\vec{S}_{3,1}$ and $\vec{S}_{3,2}$ are in the opposite direction to $\vec{k}_3$, then $\mathbf{C}_{1,1}$ is the last element of the input, and $l_2 = (N-1)(\left|\vec{S}_{3,1}\right| + \left|\vec{S}_{3,2}\right|)$. In general, $l_2 = (N-1)\left[\mathcal{G}\left((\vec{S}_{3,1}), (-\vec{k}_3)\right) + \mathcal{G}\left((\vec{S}_{3,2}), (-\vec{k}_3)\right)\right]$, and the time to reach PE C is equal to $\frac{l_2}{|\vec{v}_3|}$. Hence, T_{load} is given by Eq. (3.6).

By symmetry, we can verify easily that T_{drain}, the time to drain the outputs from the array, is equal to T_{load}. ■

Lemma 3.3 does not cover the case in which the input matrix is stationary. As pointed out in Section 2.3.4, stationary inputs have to be preloaded in the processor array before computation begins. Since there is only one input matrix $\mathbf{C}$, we assume that preloading takes a lower-bound time computed as the floor of the number of elements to be preloaded divided by the maximum number of input ports. A similar assumption is made when the final stationary results have to be drained. Even with this optimistic assumption, we did not find any design with stationary inputs/outputs that outperforms designs with moving inputs. Although this observation is not true in general, we like to point out that a schedule to preload data in the processor array may not be governed by the data dependence relations, and that a general preloading schedule may depend on specific design parameters (such as values of the GPM parameters) and architecture constraints (such as bandwidth and memory).

For linear-array synthesis, the condition for data-input conflicts given in Theorem 2.3 can be refined further as follows. Since all of the spacings are scalars, let $s_{3,1}$ be $\left|\vec{S}_{3,1}\right|$ and $s_{3,2}$ be $\left|\vec{S}_{3,2}\right|$.

Theorem 3.1 *Data-input conflicts occur in the N-by-N input matrix* $\mathbf{C}$ *if and only if*

$$\frac{s_{3,1}}{\xi} < N, \quad \textit{and} \quad \frac{s_{3,2}}{\xi} < N \tag{3.8}$$

where $\xi = \gcd(s_{3,1}, s_{3,2})$ *is the greatest common divisor of* $s_{3,1}$ *and and* $s_{3,2}$*.*

Proof.

If Part. Since $\xi = \gcd(s_{3,1}, s_{3,2})$, we have $s_{3,1} = \xi.\alpha_2$ and $s_{3,2} = \xi.\alpha_1$, where α_1, α_2 are integers such that $|\alpha_1|, |\alpha_2| < N$. Hence,

$$\begin{aligned}
& \tfrac{s_{3,1}}{\alpha_2} = \tfrac{s_{3,2}}{\alpha_1}, \ \text{where } 1 \le |\alpha_1|, |\alpha_2| \le N-1, \\
\Longrightarrow \ & \begin{bmatrix} s_{3,1} \ s_{3,2} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2' \end{bmatrix} = 0, \ \text{where } \alpha_2' = -\alpha_2, \ 1 \le |\alpha_1|, |\alpha_2| \le N-1 \\
\Longrightarrow \ & \text{Data-input conflicts in input (according to Theorem 2.3).}
\end{aligned}$$

Only-if Part. From Theorem 2.3,

$$\begin{aligned}
\text{Data-input conflicts in input } \mathbf{C} & \Longrightarrow s_{3,1}\alpha_2 = s_{3,2}\alpha_1 \\
& \Longrightarrow \tfrac{s_{3,1}}{\alpha_1} = \tfrac{s_{3,2}}{\alpha_2}
\end{aligned}$$

where $\alpha_1, \ \alpha_2 \in \{-(N-1), \ldots, (N-1)\}$ (since $L_3 = 1, U_3 = 1, L_3 - U_3 = -(N-1), L_3 + U_3 = (N-1)$), and $\gcd(\alpha_1, \alpha_2) = 1$ (if not, scale α_1 and α_2 by their greatest common divisor). Since α_1 and α_2 are relatively prime,

$$\begin{aligned}
\tfrac{s_{3,1}}{\alpha_1} = \tfrac{s_{3,2}}{\alpha_2} = \xi & \Longrightarrow \alpha_1 = \tfrac{s_{3,1}}{\xi} \text{ and } \alpha_2 = \tfrac{s_{3,2}}{\xi} \\
& \Longrightarrow \tfrac{s_{3,1}}{\xi} < N \text{ and } \tfrac{s_{3,2}}{\xi} < N.
\end{aligned}$$

■

Corollary 3.1 *For any feasible design,* $t_3(s_{3,1} + s_{3,2}) \ge N + 1$.

Proof. Let $s'_{3,1} = t_3 s_{3,1}$ and $s'_{3,2} = t_3 s_{3,2}$. From Eq. (3.3), $s'_{3,1}$ and $s'_{3,2}$ are integers. Assume for contradiction that there exists a solution such that $s'_{3,1} + s'_{3,2} = x$, $x < N + 1$. Then $1 \leq s'_{3,1}, s'_{3,2} \leq (x-1)$. If $\xi = \gcd(s_{3,1}, s_{3,2})$ and $\xi' = \gcd(s'_{3,1}, s'_{3,2})$, then $\xi' = t_3 \xi$. Therefore,

$$1 \leq \frac{s'_{3,1}}{\xi'}, \frac{s'_{3,2}}{\xi'} \leq (x-1) \leq N$$

$$1 \leq \frac{s_{3,1}}{\xi}, \frac{s_{3,2}}{\xi} \leq (x-1) \leq N \text{ (since } \frac{s'_{3,1}}{\xi'} = \frac{s_{3,1}}{\xi}).$$

Hence, according to Theorem 3.1, data-input conflicts are present, and the solution is not feasible. Hence, for all feasible solutions, $s'_{3,1} + s'_{3,2} \geq N + 1$. ■

3.1.2 Time-optimal and processor-optimal linear array designs

Table 3.1 shows the optimal linear-array designs found by the search procedure of GP-M (see Section 2.4.3) in which the objective is to minimize T_{comp} (computation time, not including load and drain times) without bounds on $\#PE$. In finding these designs, t_3 is incremented before t_1 or t_2 in Step 10 of the search procedure. This is done as it increases T_{comp} by the smallest amount. Among all of the designs that have the minimum T_{comp}, designs with minimum $\#PE$ are chosen first, followed by designs that require the minimum T_{load} and T_{drain}. We list T_{load}, T_{comp}, T_{drain}, $\#PE$s needed, and the CPU time used by the search procedure running on a Sun Sparcstation 10/30. We also list the equivalent values of schedule vector $\vec{\Pi}$ and allocation matrix $\mathbf{S}$ of DM by solving Eqs. (2.2) and (2.4).

In a similar way, we find designs that optimize T_c (completion time, including load and drain times) without bounds on $\#PE$. (See Table 3.2.) Note that these designs have smaller total completion times and larger $\#PE$s than the corresponding designs in Table 3.1. For instance, for $N = 300$, the completion time for the design optimizing T_c requires 7% less completion time and 35% more PEs than the one optimizing T_{comp}. We also list the equivalent $\vec{\Pi}$ and $\mathbf{S}$ in DM for minimizing T_c.

Table 3.1: T_{comp}-optimal linear processor arrays for finding transitive closures of $N \times N$ matrices.

N	GPM: T_{comp}-Optimal Linear-Array Designs						
	Periods	Distances	Schedule	Allocation	Min T_{comp} Designs		SS10/30
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$\vec{\Pi}$	**S**	$(T_{load}, T_{comp}, T_{drain})$	#PE	(sec)
3	(1,1,2)	(0,−1,1)	(4,1,1)	(0,−1,0)	(5,13,5)	3	-
4	(1,1,3)	(0,−1,1)	(5,1,1)	(0,−1,0)	(10,22,10)	4	-
8	(1,1,5)	(0,−1,3)	(7,1,1)	(2,−1,0)	(13,64,13)	22	-
16	(1,2,5)	(0,−2,3)	(8,2,1)	(1,−2,0)	(51,166,51)	46	-
32	(1,3,6)	(0,−3,5)	(10,3,1)	(2,−3,0)	(113,435,113)	156	-
64	(1,5,7)	(0,−5,6)	(13,5,1)	(1,−5,0)	(369,1198,369)	379	1
100	(1,5,11)	(0,−5,9)	(17,5,1)	(4,−5,0)	(606,2278,606)	892	5
200	(1,8,13)	(1,−8,12)	(22,8,1)	(5,−8,1)	(1743,6170,1743)	2787	27
300	(1,9,18)	(0,−9,17)	(28,9,1)	(8,−9,0)	(2851,11363,2851)	5084	88

Table 3.2: T_c-optimal linear processor arrays for finding transitive closures of $N \times N$ matrices.

N	GPM: T_c-Optimal Linear-Array Designs						
	Periods	Distances	Schedule	Allocation	Min T_c Designs		SS10/30
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$\vec{\Pi}$	**S**	$(T_{load}, T_{comp}, T_{drain})$	#PE	(sec)
3	(1,2,1)	(0,−1,1)	(4,2,1)	(0,−1,0)	(3,15,3)	3	-
4	(1,1,5)	(0,1,4)	(7,1,1)	(5,1,0)	(2,28,2)	19	-
8	(1,1,5)	(0,1,−4)	(7,1,1)	(−3,1,0)	(10,64,10)	29	-
16	(1,2,6)	(0,−1,5)	(9,2,1)	(4,−1,0)	(19,181,19)	76	-
32	(1,3,9)	(0,−1,8)	(13,2,1)	(7,−1,0)	(36,528,36)	249	2
64	(1,4,14)	(0,−1,13)	(19,4,1)	(12,−1,0)	(69,1513,69)	820	12
100	(1,4,15)	(0,−3,14)	(20,4,1)	(11,−3,0)	(320,2476,320)	1387	33
200	(1,9,21)	(0,−1,20)	(31,9,1)	(19,−1,0)	(210,8160,210)	3981	206
300	(1,11,26)	(0,−1,25)	(38,11,1)	(24,−1,0)	(312,14951,312)	7476	609

The results in Tables 3.1 and 3.2 demonstrate that GPM, based on the equivalence between GPM and DM as shown in Eqs. (2.2) and (2.4), can serve as a powerful tool to find optimal designs in DM.

It is important to point out that the objective function used (whether to minimize T_{comp} or T_c) depends on the application. If the linear processor array is used to evaluate the transitive closure of one matrix, then minimizing T_c will be important. On the other hand, if the processor array is used for pipelined evaluation of transitive closures of multiple matrices, then minimizing T_{comp} may be important.

If the objective is to minimize $\#PE$ in the linear processor array, then Theorem 3.2 characterizes the $\#PE$-optimal design.

Theorem 3.2 *The combinations of parameters* $(t_1, t_2, t_3) = (1, 1, N-1)$ *and* $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, \pm 1, \mp 1)$ *or* $(\pm 1, 0, \mp 1)$ *result in linear processor arrays with a primary objective of minimizing the number of PEs, and a secondary objective of minimizing the computation time.*

Proof. We show that the parameter values defined in Theorem 3.2 minimize the completion time of a $\#PE$-optimal linear processor array. From Lemma 3.2, $\#PE = (N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_1 + \vec{k}_2 + \vec{k}_3\right|) + 1$. Hence,

$$\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_1 + \vec{k}_2 + \vec{k}_3\right| \geq 1, \text{ else } \#PE = 1 \text{ and the computation is serial,} \quad (3.9)$$

$$\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_1 + \vec{k}_2 + \vec{k}_3\right| = 1, \text{ for minimum PE-count with } \#PE = N. \quad (3.10)$$

The table below lists all of the possible values of displacements and spacings for a $\#PE$-optimal linear processor array ($\#PE = N$) with periods $\vec{T} = (t_1, t_2, t_3)^t$ and $\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_1 + \vec{k}_2 + \vec{k}_3\right| = 1$.

Case	$\vec{k}_1$	$\vec{k}_2$	$\vec{k}_3$	$s_{3,1}$	$s_{3,2}$
1	0	0	± 1	$\frac{t_1}{t_3}$	$\frac{t_2}{t_3}$
2	0	± 1	∓ 1	$\frac{t_1}{t_3}$	$\frac{t_2+t_3}{t_3}$
3	± 1	0	∓ 1	$\frac{t_1+t_3}{t_3}$	$\frac{t_2}{t_3}$

Case 1. From Corollary 3.1, $t_1+t_2 \geq N+1$. For minimum T_{comp}, $t_3 = 1$ and $t_1+t_2 = N+1$. Hence, $\vec{T} = (t_1, t_2, 1)^t$ and $\mathbf{K} = (0, 0, \pm 1)^t$, and T_{comp} is equal to $(N-1)(2N+3)+1$.
Case 2. From Corollary 3.1, we have $t_1+t_2+t_3 \geq N+1$, and

$$T_{comp} \;=\; (N-1)(2t_1+2t_2+t_3)+1. = (N-1)(2(t_1+t_2+t_3)-t_3)+1$$

Therefore, for minimum T_{comp}, $t_1+t_2+t_3$ should be minimized and t_3 should be maximized. The maximum value of $t_3 = N-1$ as $t_1 \geq 1$ and $t_2 \geq 1$. Hence, $T_{comp}^{min} = (N-1)(N+3)+1$
Case 3. Similar to Case 2, the best computation time $T_{comp}^{min} = (N-1)(N+3)+1$.

Therefore, the minimum computation time for the minimum-processor designs occur for Cases 2 and 3 above. ■

Table 3.3 shows the $\#PE$-optimal designs obtained by GPM as well as those obtained by Lee and Kedem (LK) [40] and Shang and Fortes (SF) [2]. In this table, we show the load and drain times, computation times, and $\#PE$s for designs derived by these three methods. Vector $\vec{\Pi}$, matrix $\mathbf{S}$, and the corresponding parameters in GPM are summarized as follows.

Method	$\vec{\Pi}$	$\mathbf{S}$	$(t_1,\ t_2,\ t_3)$	$(\vec{k}_1,\ \vec{k}_2,\ \vec{k}_3)$
LK	$[2N-1,\ 2,\ 1]^t$	$[0,\ 1,\ 1]^t$	$(1,\ 2,\ 2N-4)$	$(1,\ 1,\ -2)$
SF	$[N,\ 1,\ 1]^t$	$[0,\ 0,\ -1]^t$	$(1,\ 1,\ N-2)$	$(-1,\ 0,\ 1)$
GPM	$[N+1,\ 1,\ 1]^t$	$[0,\ 0,\ -1]^t$	$(1,\ 1,\ N-1)$	$(-1,\ 0,\ 1)$

Table 3.3 shows that both the SF and GPM designs require the minimum number of PEs. The SF designs, however, were developed based on different assumptions. According to Lemma 3.1 and the table above, the SF designs have a computation time $T_{comp} = (N-1)(N+2)+1$. This computation time is lower than that of the GPM designs characterized by Theorem 3.2. This difference is attributed to the fact that Shang and Fortes assumed that contention must be avoided only after the first use of a variable and before its last use or generation. This is a valid assumption for systems with fast I/O (or where each PE has its own I/O), or in cases in which inputs are preloaded and outputs need not be drained or are postdrained. In GPM, we consider both contentions in computations as well as in data

Table 3.3: $\#PE$-optimal linear processor arrays for finding transitive closures of $N \times N$ matrices. (Parameters for GPM are shown in Theorem 3.2.)

	Designs by LK [40]		Designs by SF [2]		Designs by GPM	
N	$(T_{load}, T_{comp}, T_{drain})$	#PE	$(T_{load}, T_{comp}, T_{drain})$	#PE	$(T_{load}, T_{comp}, T_{drain})$	#PE
3	(5, 17, 5)	5	(3, 11, 3)	3	(5, 13, 5)	3
4	(13, 31, 13)	7	(7, 19, 7)	4	(10, 22, 10)	4
8	(85, 127, 85)	15	(43, 71, 43)	8	(50, 78, 50)	8
16	(421, 511, 421)	31	(211, 271, 211)	16	(226, 286, 226)	16
32	(1861, 2047, 1861)	63	(931, 1055, 931)	32	(962, 1086, 962)	32
64	(7813, 8191, 7813)	127	(3907, 4159, 3907)	64	(3970, 4222, 3970)	64
100	(19405, 19999, 19405)	199	(9703, 10099, 9703)	100	(9802, 10198, 9802)	100
200	(78805, 79999, 78805)	399	(39403, 40199, 39403)	200	(39602, 40398, 39602)	200
300	(178205, 179999, 178205)	599	(89103, 90299, 89103)	300	(89402, 90598, 89402)	300

links. Excluding designs that have computational and data-link conflicts results in designs that require slightly longer load, drain, and computation times.

To illustrate the point above, we compute using Eq. (3.3) the spacings used in the SF design [2]: $s_{3,1} = -(N-1)/(N-2)$ and $s_{3,2} = -1/(N-2)$). These values of spacings result in data-input conflicts between tokens ($\mathbf{C}_{1,j}$ and $\mathbf{C}_{N,j-1}$), $j = 2, 3, \ldots, N$, of input matrix $\mathbf{C}$ (Theorem 3.1).

The space-time diagrams of two linear processor arrays, one optimizing T_{comp} and the other optimizing T_c, for $N = 3$, are shown in Figures 3.2 and 3.3, respectively.

The design in Figure 3.2 optimizes T_{comp} and has parameters: $(t_1, t_2, t_3) = (1, 1, 2)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, 1, -1)$. This design minimizes both T_{comp} and $\#PE$, and therefore, minimizes any objective of the form $\#PE^x \times T_{comp}^y$ for x, $y \geq 1$. The space-time diagram shows the execution times and locations of all of the index points in the domain of the algorithm. The entire diagram can be derived recursively if the distance and time between index points separated by basis vectors $(0,0,1)^t = \vec{d}_1, (0,1,0)^t = \vec{d}_2, (1,0,0)^t = \vec{d}_1 + \vec{d}_2 + \vec{d}_3$ are known. For example, consider the execution of two index points $(1,1,1)$ and $(2,1,1)$ in Figure 3.2 separated by the vector $(1,0,0) = \vec{d}_1 + \vec{d}_2 + \vec{d}_3$. From the definition of the periods, the time difference between the execution of these two index points is $t_1 + t_2 + t_3 = 1 + 1 + 2 = 4$. Similarly, the displacement between the PEs executing the two index points is given by $\vec{k}_1 + \vec{k}_2 + \vec{k}_3 = 0 + 1 + (-1) = 0$, Hence, in Figure 3.2, they are executed by the same

Input Matrix on Link 3

C11 C12 C13 C21 C22 C23 C31 C32 C33

$\overleftarrow{\quad\quad}$

$\vec{V}_3 = -1/2$

Time	PE $_1$	PE $_2$	PE $_3$
1	1 C11 1 C11 1 C11		
2	1 C12 1 C11 2 C12	1 C21 2 C21 1 C11	
3	1 C13 1 C11 3 C13	1 C22 2 C21 2 C22	1 C31 3 C31 1 C11
4		1 C23 2 C21 3 C13	1 C32 3 C31 2 C12
5	2 C22 1 C22 1 C22		1 C33 3 C31 3 C13
6	2 C23 1 C22 2 C23	2 C32 2 C32 1 C22	
7	2 C21 1 C22 3 C21	2 C33 2 C32 2 C23	2 C12 3 C12 1 C22
8		2 C31 2 C32 3 C21	2 C13 3 C12 2 C23
9	3 C33 1 C33 1 C33		2 C11 3 C12 3 C21
10	3 C31 1 C33 2 C31	3 C13 2 C13 1 C33	
11	3 C32 1 C33 3 C32	3 C11 2 C13 2 C31	3 C23 3 C23 1 C33
12		3 C12 2 C13 3 C32	1 C21 3 C23 2 C31
13			3 C22 3 C23 3 C32

Index (2,3,2) executes with inputs C13, C12, C23

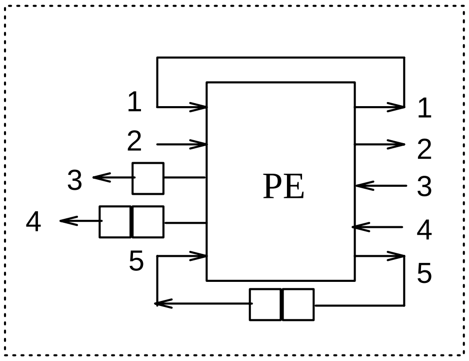


Figure 3.2: Linear processor array for finding the transitive closure of a 3×3 matrix using parameters $(t_1, t_2, t_3) = (1, 1, 2)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. The array is optimal for minimum T_{comp}, minimum $\#PE$, and minimum $\#PE^x \times T^y_{comp}$, $x, y \geq 1$. The PE used is the same as in Lee and Kedem's design [3].

Time	PE_1	PE_2	PE_3
1	1 C11 1 C11 1 C11		
2	1 C12 1 C11 2 C12		
3	1 C13 1 C11 3 C13	1 C21 2 C21 1 C11	
4		1 C22 2 C21 2 C22	
5	2 C22 1 C22 1 C22	1 C23 2 C21 3 C13	1 C31 3 C31 1 C11
6	2 C23 1 C22 2 C23		1 C32 3 C31 2 C12
7	2 C21 1 C22 3 C21	2 C32 2 C32 1 C22	1 C33 3 C31 3 C13
8		2 C33 2 C32 2 C23	
9	3 C33 1 C33 1 C33	2 C31 2 C32 3 C21	2 C12 3 C12 1 C22
10	3 C31 1 C33 2 C31		2 C13 3 C12 2 C23
11	3 C32 1 C33 3 C32	3 C13 2 C13 1 C33	2 C11 3 C12 3 C21
12		3 C11 2 C13 2 C31	
13		3 C12 2 C13 3 C32	3 C23 3 C23 1 C33
14			1 C21 3 C23 2 C31
15			3 C22 3 C23 3 C32

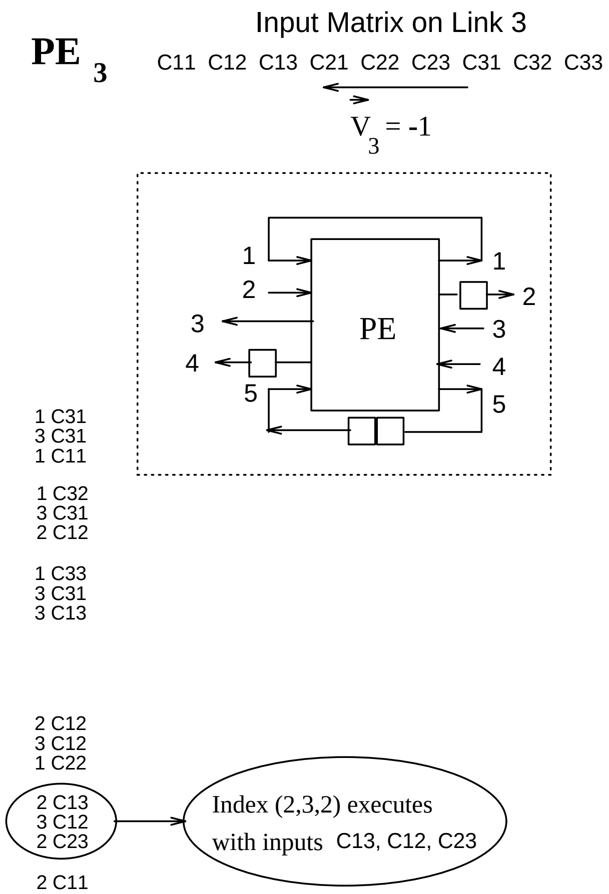

Input Matrix on Link 3
C11 C12 C13 C21 C22 C23 C31 C32 C33
V_3 = -1
PE
1
2
3
4
5
1
2
3
4
5
Index (2,3,2) executes with inputs C13, C12, C23

Figure 3.3: Linear processor array for finding the transitive closure of a 3×3 matrix using parameters $(t_1, t_2, t_3) = (1, 2, 1)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. The array is optimal for minimum T_c, minimum $\#PE$, and minimum $\#PE^x \times T_c{}^y$, $x, y \geq 1$. The PE used is the same as in Lee and Kedem's design [3].

processor $\mathbf{PE}_1$ at times 1 and 5, respectively. In a similar fashion, the entire space-time diagram can be derived mechanically from a knowledge of the periods and displacements.

The design in Figure 3.3 has parameters $(t_1, t_2, t_3) = (1, 2, 1)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. It uses less load and drain times (3 units each), but its computation time T_{comp} is higher than that in Figure 3.2. It minimizes both T_c and $\#PE$, and therefore, minimizes any objective of the form $\#PE^x \cdot T_c^y$ for $x, y \geq 1$. Note that the load and drain times are not shown in these diagrams. Further, for correct execution of the Floyd-Warshall algorithm, control signals are needed to govern the index-dependent assignments performed by the PEs in the array. These assignments are given in Tables I and II in reference [3].

3.1.3 Processor-time trade-offs

Comparing the results in Tables 3.2 and 3.3, we found, for instance, that for a problem of size of 200, the T_c-optimal design is 13.94 times faster than the $\#PE$-optimal design in terms of completion time, and uses 19.91 times more PEs than the $\#PE$-optimal design. (The T_c-optimal design for $N = 200$ requires 8580 time units and 3981 PEs, whereas the $\#PE$-optimal design requires 119602 time units and 200 PEs.) A designer might be unwilling to settle for either the large number of PEs required in the minimum-time design or the long completion time of the minimum-processor design. In realistic design situations, there may be bounds on the number of processors or the completion time or both. Hence, a possible objective could be to have as few processors as possible, so long as the time is within a preset upper limit, T_c^{ub} (or T_{comp}^{ub}), or to minimize T_c (or T_{comp}) with $\#PE$ less than a given upper bound $\#PE^{ub}$.

In the following discussion, let T_{comp}^{min} and $\#PE^{max}$ be, respectively, the completion time and $\#PE$ of the minimum-T_{comp} design. Designs with $\#PE \geq \#PE^{max}$ would not be useful as their completion times have to be at least T_{comp}^{min}. On the other hand, let T_{comp}^{max} and $\#PE^{min}$ be, respectively, the computation time and $\#PE$ of the minimum-processor design (from Theorem 3.2 and Lemma 3.2, $\#PE^{min} = N$). Again, there is no benefit in obtaining designs with $T_{comp} \geq T_{comp}^{max}$, as the number of PEs cannot be reduced below $\#PE^{min}$. In this case, we are interested in finding designs with completion time greater than T_{comp}^{min} and $\#PE$ less than $\#PE^{max}$.

Figure 3.4 shows how $\#PE$ varies with T_{comp} for 3 different problem sizes: $N = 100$, 200, and 300. The y-axis $\#PE$ is normalized by $\#PE^{max}$, and the x-axis T_{comp} is normalized by T_{comp}^{max}. This let us compare the different problem sizes uniformly on the same scale. The stepped curves are obtained by bounding T_{comp} and finding the $\#PE$-optimal designs for specific recurrence sizes. These curves are stepped because there are only a small and finite number of processor-array configurations that can satisfy the given time constraints. If the goal is to find the $\#PE$-optimal designs, then we will have a small number of array configurations; for each configuration, we will select the one with the minimum computation time.

Given the bound T_{comp}^{ub} (respectively, $\#PE^{ub}$) the designer can use Figure 3.4 to find the minimum $\#PE$ (respectively, T_{comp}) required, and decide (possibly from a cost perspective) if it is acceptable. Again, the designer can exploit the initial steep decline in the plots to choose an alternative design that trades performance for cost. For instance, the minimum $\#PE$ for $N = 200$ drops by 43% for only a 19% increase in computation time.

If both T_{comp} and $\#PE$ are bounded from above, then the design with the minimum $\#PE$ for a given time bound is determined using Figure 3.4. First, a horizontal line is drawn across the graph for the desired bound on $\#PE$. The intersection between this line and the stepped curve represents the minimum T_{comp} needed for any feasible design. If this minimum T_{comp} is less than the desired T_{comp}, then a feasible design can be obtained by the procedure discussed in Section 2.4.3. This now represents the best design under both time and processor constraints.

Another observation from Figure 3.4 is that the plots for larger N decrease more rapidly than those for smaller N. Hence, for larger N, there is a substantial reduction in $\#PE$ (respectively, T_{comp}) for a relatively small increase of the computation time (respectively, $\#PE$) from the optimum. Hence, for large N, there are more attractive alternatives than the time- or $\#PE$-optimal designs.

Figure 3.5 shows a similar plot as in Figure 3.4 except that we depict the difference between trade-offs obtained on T_c and $\#PE$ versus trade-offs obtained on T_{comp} and $\#PE$. Two sets of curves are shown, one for designs that minimize T_{comp}, and the other for designs that minimize T_c, for N equal to 100 and 200, respectively. The y-axis of these curves is

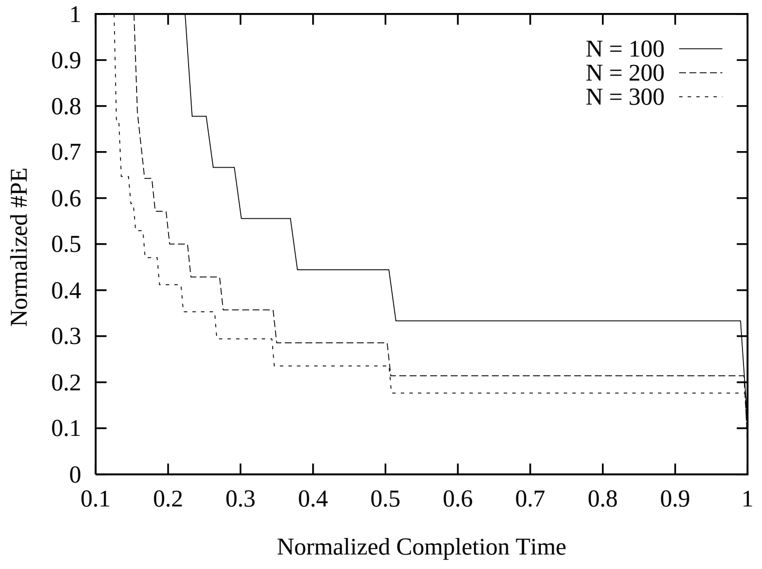


Figure 3.4: Performance trade-offs: Variation in minimum $\#PE$ with time bound T_{comp}^{ub} and variation in minimum T_{comp} with processor bound $\#PE^{ub}$. The plots are given for three problem sizes $N = 100,\ 200$ and 300.

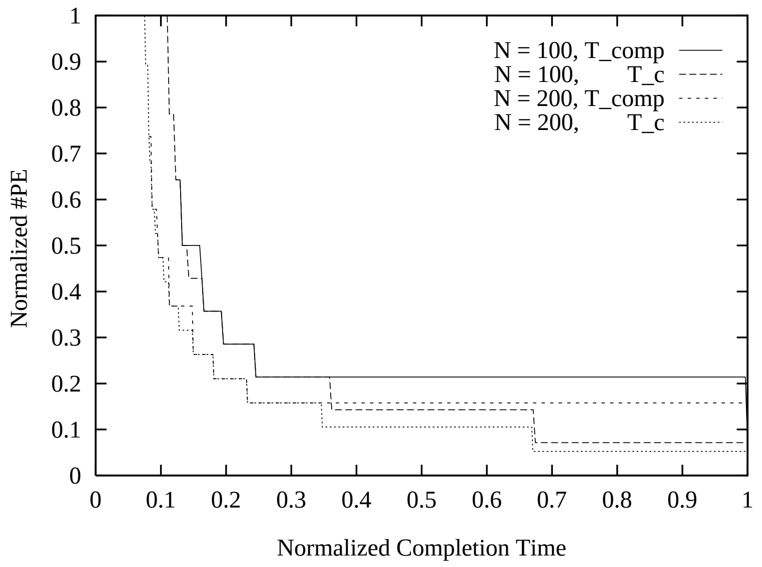


Figure 3.5: Performance trade-offs: Variation in $\#PE$ with T_c or T_{comp}. The plots are given for two problem sizes $N = 100$ and 200.

normalized with respect to $\#PE$ when T_c is minimum (since these designs require more PEs and less T_c), and the x-axis is normalized with respect to T_c when $T_{comp} = T_{comp}^{max}$. These graphs show the difference between designs obtained by different objectives. Given a bound T_c^{ub}, we can see that the number of processors obtained by minimizing T_c is less than or equal to the number of processors obtained by minimizing T_{comp}.

3.2 Three-dimensional Cube Graph Algorithms: Matrix Product

Three-dimensional cube graph algorithms form an important fundamental class of problems in signal and image processing. The dependence graph for 3-D cube graph algorithms is an $N \times N \times N$ cubical mesh. Beyond matrix product, Ibarra and Palis [53] point out that the cubical mesh is the dependence graph for a variety of recurrences in three variables (e.g., finding the longest common subsequence over three strings). Other computations include L-U factorization [6], a three-pass transitive closure [54], matrix triangularization, matrix inversion [52], and two-dimensional tuple comparison [42]. A special case of the matrix-product is the matrix-vector product, which models FIR-filtering, convolution, and polynomial multiplication [42].

In this section we synthesize a linear processor array for computing the product of two $N \times N$ matrices. The basic operation in this algorithm is the multiply-and-accumulate computation, and most commercially available DSP microprocessors are evaluated on the number of multiply-and-accumulate operations they can perform per second. The recurrences describing the matrix product (Example 1.3), parameter definitions (Example 2.1) and constraints on parameters (Example 2.3) are indicated in Chapter 2. A total of 8 relevant parameters are defined: 3 periods t_1 ($\mathbf{C}$), t_2 ($\mathbf{A}$), t_3 ($\mathbf{B}$), 3 displacements $\vec{k}_1$ ($\mathbf{C}$), $\vec{k}_2$ ($\mathbf{A}$), $\vec{k}_3$ ($\mathbf{B}$), and 6 spacings. For a linear array, all of the parameters are scalars. The six spacings are given by

$$\mathbf{C} : s_{1,2} = \frac{t_1 k_2 - t_2 k_1}{t_1}, \quad s_{1,3} = \frac{t_1 k_3 - t_3 k_1}{t_1} \tag{3.11}$$

$$\mathbf{A} : s_{2,1} = \frac{t_2 k_1 - t_1 k_2}{t_2}, \quad s_{2,3} = \frac{t_2 k_3 - t_3 k_2}{t_2} \tag{3.12}$$

$$\mathbf{B} : s_{3,1} = \frac{t_3 k_1 - t_1 k_3}{t_3}, \quad s_{3,2} = \frac{t_3 k_2 - t_2 k_3}{t_3}. \tag{3.13}$$

We illustrate in this section five formulations of the optimization of PAs: a) T_{comp}-optimal designs without bound on $\#PE$, b) T_c-optimal designs without bound on $\#PE$, c) $\#PE$-optimal designs without bound on T_c or T_{comp}, d) optimal designs with specific bounds on T_{comp} or $\#PE$, and e) optimal designs with specific bounds on T_c or $\#PE$.

3.2.1 Performance attributes

The attributes of interest are T_{comp} (computation time), T_{load} (load time), T_{drain} (drain time), $\#PE$ (processor count), and T_c (completion time), where $T_c = T_{load} + T_{comp} + T_{drain}$. For the case of computing the matrix product in a linear PA, T_{comp}, $\#PE$, T_{load} and T_{drain} are given below.

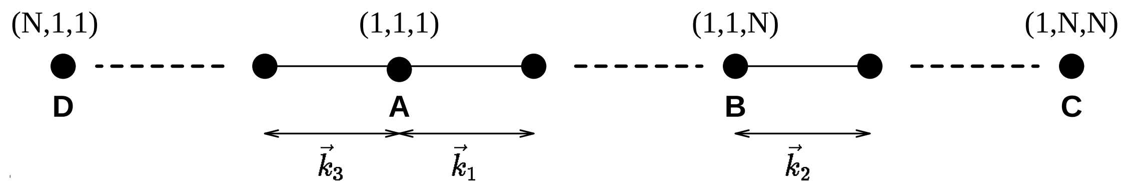


Figure 3.6: PE allocation with $\vec{k}_1$, $\vec{k}_2 \geq 0$ and $\vec{k}_3 \leq 0$.

Lemma 3.4 *The computation time T_{comp} without load and drain times for finding the product of two $N \times N$ matrices is*

$$T_{comp} \quad = \quad (N-1)(t_1+t_2+t_3)+1. \tag{3.14}$$

Proof. The critical path in the execution is as follows:

$$(1,1,1) \overset{(N-1)t_1}{\longrightarrow} (1,1,N) \overset{(N-1)t_2}{\longrightarrow} (1,N,N) \overset{(N-1)t_3}{\longrightarrow} (N,N,N).$$

Thus, T_{comp} is $(N-1)(t_1+t_2+t_3)+1$. ■

Lemma 3.5 *$\#PE$, the number of processor for computing the product of two $N \times N$ matrices on a linear processor array is given by*

$$\#PE \quad = \quad (N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_3\right|)+1. \tag{3.15}$$

Proof. Consider the displacements $\vec{k}_1$, $\vec{k}_2$, and $\vec{k}_3$. Two of these 3 displacements should be in the same direction, since the array is 1-D. Assume that $\vec{k}_1$ and $\vec{k}_2$ are positive displacements, i.e., they correspond to velocities flowing to the right (refer to Figure 3.6). Let A be the PE where the computation indexed by $(1,1,1)$ occurs. Therefore, computation $(1,1,N)$ is executed at PE B that is at a distance $(N-1)\left|\vec{k}_1\right|$ from PE A. Similarly, computation $(1,N,N)$ is executed at PE C, that is, $(N-1)\left|\vec{k}_2\right|$ PEs to the right of B. On the other hand, computation $(N, 1, 1)$ is executed at PE D that is at a distance of $(N-1)\left|\vec{k}_3\right|$ to the left of PE A (since $\vec{k}_3$ corresponds to the left moving variable). All other computations in the domain are executed by PEs between C and D. Therefore, the total number of PEs is $(N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_3\right|) + 1$. ■

The load and drain times are given as follows. Note that the **C** matrix does not have to be loaded, and the **A** and **B** matrices do not have to be postdrained.

$$\begin{aligned} T_{load} &= T_{load}(\mathbf{A}) + T_{load}(\mathbf{B}), && (3.16) \\ T_{drain} &= T_{drain}(\mathbf{C}). && (3.17) \end{aligned}$$

The load time $T_{load}(\mathbf{A})$ for nonstationary matrix **A** is given below in Lemma 3.6. The load times $T_{load}(\mathbf{B})$ and $T_{drain}(\mathbf{C})$ for nonstationary inputs **B** and **C** are given similar to Eq. (3.19). If some $\vec{k}_i = 0$, for $i = 1, 2, 3$, then that input (or output) is stationary and preloaded into (or postdrained from) the processor array.

Lemma 3.6 *The loading time, $T_{load}(A)$, for nonstationary input* **A** *(i.e., $\vec{k}_2 \neq 0$) for computing the product of two $N \times N$ matricex* **A** *and* **B** *is given by*

$$\begin{aligned} T_{load}(\mathcal{A}) &= 1 + (N-1)\frac{\left(\mathcal{G}(\vec{S}_{2,1}, -\vec{v}_2) + \mathcal{G}(\vec{S}_{2,3}, -\vec{v}_2)\right)}{|\vec{v}_2|} && (3.18) \\ &\quad +(N-1)\left\lceil \frac{t_3\left\{\mathcal{G}(\vec{k}_1, \vec{k}_2) + \mathcal{G}(\vec{k}_2, \vec{k}_2) + \mathcal{G}(\vec{k}_3, \vec{k}_2)\right\}}{\left|\vec{k}_2\right|} \right\rceil \end{aligned}$$

where

$$\mathcal{G}(\vec{x}, \vec{y}) = \begin{cases} |\vec{x}| & \text{if } \vec{x}, \vec{y} \text{ are in opposite directions} \\ 0 & \text{otherwise.} \end{cases} \qquad (3.19)$$

Proof. Similar to the proof of Lemma 3.3 where **A** is used instead of **C**, and $\vec{k}_3$ is substituted for $\vec{k}_1 + \vec{k}_2 + \vec{k}_3$. ■

For a linear PA all of the parameters are scalars. The condition for data-input conflict for each of the three variables **C**, **A**, **B** can be refined similar to Theorem 3.1. The refined conditions are given in Theorem 3.3.

Theorem 3.3 *Data-input conflicts occur in the N-by-N input matrix* **X** *(***X** *can be* **C**, **A**, *or* **B***) if and only if*

$$\frac{s_1}{\xi} < N, \quad \textit{and} \quad \frac{s_2}{\xi} < N \tag{3.20}$$

where $\xi = \gcd(s_1, s_2)$ *is the greatest common divisor of* s_1 *and and* s_2*, and* s_1, s_2 *are the two spacings associated with the appropriate input matrix* **X** *(*$s_1 = s_{1,2}, s_2 = s_{1,3}$ *for* **C***,* $s_1 = s_{2,1}, s_2 = s_{2,3}$ *for* **A***,* $s_1 = s_{3,1}, s_2 = s_{3,2}$ *for* **B***).*

Proof. Similar to the proof of Theorem 3.1. ■

Corollary 3.2 captures the equivalent of Corollary 3.1 for the transitive-closure example.

Corollary 3.2 *In any feasible design, for an input* **X** *(***X** *can be* **C**, **A**, *or* **B***) with period* t_i*, displacement* $\vec{k}_i$*,* $(i = 1, 2, or\ 3)$*, and spacings* s_1, s_2 *(*$s_1 = s_{1,2}, s_2 = s_{1,3}$ *for* **C***,* $s_1 = s_{2,1}, s_2 = s_{2,3}$ *for* **A***,* $s_1 = s_{3,1}, s_2 = s_{3,2}$ *for* **B***),*

$$t_i(s_1 + s_2) \geq N + 1, \ \textit{if}\ \vec{k}_i \neq 0. \tag{3.21}$$

Proof. Similar to the proof of Corollary 3.1 using Theorem 3.3 ■

3.2.2 Time-optimal linear array designs

Tables 3.4 and 3.5 show the optimal linear designs found by the search procedure of GPM. The objectives used are to minimize T_{comp} and T_c, respectively. In finding these designs, t_3 is incremented before t_1 or t_2 in Step 10 of the search procedure presented in Section 2.4 (refer to (3.14)). This is done as it increases T_{comp} by the smallest amount. Note that the designs were developed without bounds on $\#PE$s. In these tables, we list T_{load}, T_{comp}, T_{drain}, $\#PE$s

Table 3.4: T_{comp} optimal linear arrays for product of two $N \times N$ matrices.

N	GPM: T_{comp}-Optimal Linear PA Designs						
	Periods	Distances	Schedule	Allocation	Min T_{comp} Designs		SS10/30
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$\vec{\Pi}$	$\mathbf{S}$	$(T_{load}, T_{comp}, T_{drain})$	#PE	(sec)
3	(1,1,2)	(0,−1,1)	(2,1,1)	(1,−1,0)	(5,9,4)	5	-
4	(1,2,2)	(0,−1,1)	(2,2,1)	(1,−1,0)	(7,16,6)	7	-
8	(1,3,3)	(0,−1,2)	(3,3,1)	(2,−1,0)	(43,50,23)	22	-
16	(1,3,4)	(0,−2,3)	(4,3,1)	(3,−2,0)	(69,121,90)	76	-
32	(1,5,5)	(0,−3,4)	(5,5,1)	(4,−3,0)	(208,342,351)	218	-
64	(1,6,7)	(0,−5,6)	(7,6,1)	(6,−5,0)	(455,883,1391)	694	1
100	(1,7,9)	(0,−6,7)	(9,7,1)	(7,−6,0)	(810,1684,3370)	1288	6
200	(1,11,11)	(0,−9,10)	(11,11,1)	(10,−9,0)	(2434,4578,13415)	3782	99
300	(1,12,14)	(0,−11,13)	(14,12,1)	(13,−11,0)	(4242,8074,30132)	7177	101

Table 3.5: T_c-optimal linear PAs for product of two $N \times N$ matrices.

N	Periods	Distances	Schedule	Allocation	Min T_c Designs		SS10/30
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$\vec{\Pi}$	$\mathbf{S}$	$(T_{load}, T_{comp}, T_{drain})$	#PE	(sec)
4	(1,2,2)	(0,−1,1)	(2,2,1)	(1,−1,0)	(7,16,6)	7	-
8	(1,1,5)	(0,−1,3)	(5,1,1)	(3,−1,0)	(22,50,23)	29	1
16	(1,3,4)	(0,−2,3)	(4,3,1)	(3,−2,0)	(69,121,90)	76	18
32	(1,4,7)	(0,−3,4)	(7,4,1)	(4,−3,0)	(167,373,351)	218	1161
35	(1,4,7)	(0,-3,4)	(7,4,1)	(4,-3,0)	(183.0,409.0,418.0)	239.0	1781
36	(5,4,6)	(4,3,-5)	(6,4,5)	(-5,3,4)	(295.0,526.0,220.0)	421.0	1893
64	(6,5,8)	(5,4,−7)	(8,5,6)	(−7,4,5)	(649,1198,531)	1009	176
100	(7,7,9)	(6,5,−8)	(9,7,7)	(−8,5,6)	(1249,2278,925)	1882	483
200	(9,8,14)	(8,7,−13)	(14,8,9)	(−13,7,8)	(3216,6170,2912)	5573	3400
300	(11,11,16)	(10,9,−15)	(16,11,11)	(−15,9,10)	(6061,11363,4935)	10167	10903

needed, the equivalent $\vec{\Pi}$ and $\mathbf{S}$ in DM, and the CPU time used by the search procedure running on a Sun Sparcstation 10/30. Thus, by establishing the equivalence between DM and GPM, GPM serves as a powerful tool to find optimal designs in DM (Figure 2.1).

Note that the designs in Table 3.4 require $\mathbf{C}$ to be stationary (as $\vec{k}_1 = 0$), and $\mathbf{A}$ and $\mathbf{B}$ moving. These designs are found by optimizing T_{comp} and are not affected by the values of T_{drain}. We further note that the designs in Table 3.5 are based on optimizing T_c. As a result, they have less total completion time and more $\#PE$s than the corresponding designs in Table 3.4.

Table 3.5 shows that for N up to 35, the optimal designs for T_c have stationary output matrix $\mathbf{C}$. For $N > 35$, the optimal designs to minimize T_c have moving input and output matrices. To arrive at this conclusion, we have to compute the lower and upper bounds of

T_{drain} (T^{lb}_{drain} and T^{ub}_{drain}) when **C** is stationary. Given N^2 elements in the PA to be drained, we assume that each PE in the PA has 3 input and 3 output ports (one each for **A**, **B**, **C**). Hence, T^{lb}_{drain} can be estimated as $\lceil \frac{N^2}{3} \rceil$. To compute T^{ub}_{drain} we develop an optimal drain schedule (given in Section 3.2.3) for elements of **C** when computation in the linear array is completed. (It is possible to start draining elements of **C** even before computations in the PA are completed; however, the resulting lower and upper bounds will be much more complex and will be case dependent). For $N \leq 35$, we found that designs with stationary **C** have smaller T_c even when T^{ub}_{drain} is used as compared to the best designs when **C** is moving. On the other hand, for $N > 35$, designs with moving **C** have smaller T_c as compared to designs with stationary **C** even when T^{lb}_{drain} is used. These lead us to conclude that **C** should be stationary for $N \leq 35$ and moving for $N > 35$.

It is important to point out that the objective used (whether to minimize T_{comp} or T_c) depends on the application. If the linear PA is used to evaluate a single matrix product, then minimizing T_c will be important. On the other hand, if the PA is used for pipelined evaluation of matrix products, then minimizing T_{comp} may be important.

3.2.3 Load and drain times for stationary data in linear arrays

For stationary variables, the assumption is to preload (respectively, postdrain) the tokens of the stationary input (respectively, output) so that the computation phase can proceed without any further delays. In this subsection, an optimal postdraining (or preloading) schedule for stationary data on linear arrays is presented. This schedule is used to compute the upper bounds of load and drain times of stationary inputs in Tables 3.4 and 3.5. The precise statement of the problem and the proposed algorithm for computing the optimal schedule are described below. The discussion below is with respect to the draining of a stationary matrix from a linear array. The results would be immediately applicable to the case of preloading data into a linear array.

Problem Statement: Given

- a linear array of l processing elements, $PE_1, PE_2, \ldots, PE_l$, with two boundary PEs, PE_1 and PE_l; k input ports and k output ports in each PE; of the k output ports in

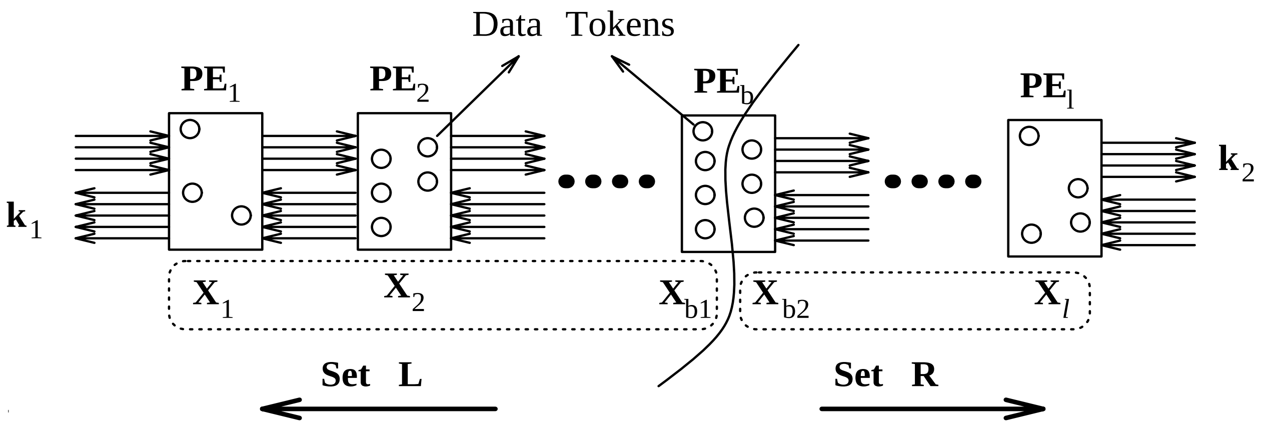


Figure 3.7: Draining $X = \sum_{i=1}^{l} x_i$ tokens distributed in a linear array. Each PE has k_1 left output ports and k_2 right output ports, and all data have to be drained out through the boundary PEs: PE_1 and PE_l.

$PE_i, i = 1, \ldots, l$, p_L ports connect PE_i to PE_{i-1}, and p_R ports connect PE_i to PE_{i+1}, where $p_L + p_R = k$ (Figure 3.7) .

- a data distribution of $x_1, x_2, \ldots, x_l$ data tokens in $PE_1, PE_2, \ldots, PE_l$, respectively.

The goal is to find a drain schedule to drain all of the $X = \sum_{i=1}^{l} x_i$ data tokens out of the linear array in minimum time. Another assumption made in the proposed algorithm to compute the optimal schedule is that each PE has unbounded storage. A drain schedule for a token d is the trajectory of d from the PE it resides in, to an external port in one of the two boundary PEs.

Optimal Schedule

The intuition is to find a line dividing the data tokens in the linear array such that all of the tokens to the left of the line are drained left from PE_1, and the tokens to the right of the line are drained right from PE_l. The optimal schedule to drain the X data tokens in the linear array is given as follows.

Schedule Opt:

- Partition the data tokens into two sets L and R, such that set L contains all the tokens in $PE_1, PE_2, \ldots, PE_{b-1}$ and x_{b1} tokens from PE_b, and set R contains all the tokens in $PE_{b+1}, PE_{b+2}, \ldots, PE_l$ and x_{b2} tokens from PE_b (Figure 3.7). This division is done such that $x_{b1} + x_{b2} = x_b$ and the total drain time $T_{drain} = \max(T_{drain}(L), T_{drain}(R))$ is minimized, where $T_{drain}(L)$, $T_{drain}(R)$ are the time to drain set L and time to drain set R, respectively. Procedure 3.1 describes an algorithm that given a data distribution of $x_1, x_2, \ldots, x_l$, finds the optimal partitioning of the $X = \sum_{i=1}^{l} x_i$ data tokens into the sets L and R such that $T_{drain}(L) = T_{drain}(R) + \epsilon, \epsilon = 0, \pm 1$ (since it minimizes $\max(T_{drain}(L), T_{drain}(R))$).

- Drain all the tokens of set L from PE_1 in a greedy fashion without direction changes, and drain all tokens of set R from PE_l in a greedy fashion without direction changes. In the greedy schedule, PE_i, $1 \leq i \leq b$, whose tokens belong to set L, transfers as many tokens as possible through its p_L ports in each cycle, i.e., $\max(x_i, p_L)$ tokens are transferred left in each cycle. Similarly, PE_j, $b \leq j \leq l$, transfers $\max(x_j, p_R)$ tokens through its p_R ports in each cycle.

Procedure 3.1

Input: Data distribution $x_1, x_2, \ldots, x_l$, where x_i is the number of tokens in PE_i with p_L left output ports and p_R right output ports (Figure 3.7).
Output: Values b, x_{b1} and x_{b2} such that

- set L contains all of the tokens in $PE_1, PE_2, \ldots, PE_{b-1}$and x_{b1} tokens from PE_b,

- set R contains all of the tokens in $PE_{b+1}, PE_{b+2}, \ldots, PE_l$ and x_{b2} tokens from PE_b,

- $T_{drain}(L) = T_{drain}(R) + \epsilon$, $\epsilon = 0, \pm 1$.

Algorithm: To find the value of b, the following four arrays are defined.

- $TimeLeft(i)$ is the time at which the last token of PE_i is drained out of PE_1, when all of the tokens in $PE_1, PE_2, \ldots, PE_i$ are drained left from the p_L ports of PE_1.

- $TimeRight(i)$ is the time at which the last token of PE_i is drained out of PE_l, when all of the tokens in $PE_i, PE_{i+1}, \ldots, PE_l$ are drained right from the p_R ports of PE_l.

- $RemLeft(i)$ is the difference between p_L and the number of tokens drained out of PE_1 at time step $TimeLeft(i)$, when all of the tokens in $PE_1, \ldots, PE_i$ are drained left from PE_1. Hence, $RemLeft(i)$ is the maximum number of tokens that can be added to PE_i without increasing $TimeLeft(i)$.

- $RemRight(i)$ is the difference between p_R and the number of tokens drained out of PE_l at time step $TimeRight(i)$, when all of the tokens in $PE_i, \ldots, PE_l$ are drained right from PE_l. Hence, $RemRight(i)$ is the maximum number of tokens that can be added to PE_i without increasing $TimeRight(i)$.

The two arrays, $TimeLeft(i)$ and $RemLeft(i)$, $i = 1, 2, \ldots, l$, are computed as shown in the pseudo-code below.

```
$TimeLeft(0) = RemLeft(0) = 0$
$Empty := 1$
for $i = 1$ to $l$ step 1
    if $(x_i = 0)$ then
        $Empty := Empty + 1$
        $TimeLeft(i) := TimeLeft(i-1)$
        $RemLeft(i) := RemLeft(i-1)$
    else
        if $(TimeLeft(i-1) \leq Empty)$ then
            $TimeLeft(i) := TimeLeft(i-1) + \left\lceil \frac{x_i}{p_L} \right\rceil$
            $RemLeft(i) := x_i \bmod p_L$
        else
            $TimeLeft(i) := TimeLeft(i-1) + \left\lceil \frac{x_i - RemLeft(i-1)}{p_L} \right\rceil$
            $RemLeft(i) := (x_i - RemLeft(i-1)) \bmod p_L$
        end-if
        $Empty := 1$
    end-if
end-for
```

The variable $Empty$ for any given value of i corresponds to the distance between PE_i and the nearest nonempty PE to the left of PE_i, i.e., $Empty$ is the smallest k such that $k > 0$ and $x_{i-k} > 0, x_{i-k+1} = \ldots == x_{i-1} = 0$. $TimeLeft(i-1)$ is the time at which the last token of $PE_{i-Empty}$ went out of PE_1, and $(p_L - RemLeft(i-1))$ is the number of tokens drained from the p_L output ports of PE_1 at time $TimeLeft(i-1)$. Hence, if there are fewer than $TimeLeft(i-1)$ empty PEs before PE_i, then $RemLeft(i-1)$ tokens in PE_i can be drained along with the set of tokens drained at time $TimeLeft(i-1)$.

Similarly, the pseudo-code to compute the two arrays, $TimeRight(i)$, $RemRight(i)$ can be obtained by replacing $TimeLeft$ by $TimeRight$, $RemLeft$ by $RemRight$, i by $(l+1-i)$, p_L by p_R in the above pseudo-code with the initialization $TimeLeft(l+1) = RemLeft(l+1) = 0$. The variable $Empty$ now corresponds to the distance between PE_i and the nearest nonempty PE to the right of PE_i, i.e., $Empty$ is the smallest k such that $k > 0$ and $x_{i+k} > 0, x_{i+1} = \ldots = x_{i+k-1} = 0$.

Note that $TimeLeft(i)$ is a nondecreasing function of i, and $TimeRight(i)$ is a nonincreasing function of i. The value of b is equal the value of i such that

$$TimeLeft(i-1) \leq TimeRight(i) \text{ and } TimeLeft(i) > TimeRight(i+1). \tag{3.22}$$

Since $TimeLeft(i)$ and $TimeRight(i)$ are monotone functions of i, b will be uniquely defined. Once b is known, x_{b1} is found by equating the time to drain set L $(T_{drain}(L))$ and time to drain set R $(T_{drain}(R))$ (Eq. (3.23)). Since x_{b1} is an integer, Eq. (3.23) is solved for a real value which is then relaxed to the next integer. Let e_1 be the distance between PE_b and the nearest nonempty PE to the left of PE_b $(x_{b-e_1} > 0, x_b > 0, x_{b-e_1+1} = \ldots = x_{b-1} = 0)$, and e_2 be the distance between PE_b and the nearest nonempty PE to the right of PE_b $(x_{b+e_2} > 0, x_b > 0, x_{b+1} = \ldots = x_{b+e_2-1} = 0)$. The following equation is solved for x'_{b1} in the set of real numbers:

$$TimeLeft(b-1) + \frac{y_1}{p_L} = TimeRight(b+1) + \frac{y_2}{p_R} \tag{3.23}$$

$$y_1 = \begin{cases} x'_{b1} \text{ if } TimeLeft(b-1) \leq e_1 \\ x'_{b1} - RemLeft(b-1), \text{ else} \end{cases} \tag{3.24}$$

$$y_2 = \begin{cases} x_b - x'_{b1}, \text{ if } TimeRight(b+1) \leq e_2 \\ x_b - x'_{b1} - RemRight(b+1), \text{ else} \end{cases} \tag{3.25}$$

The values of x_{b1} and x_{b2} are given by

$$x_{b1} = \left\lceil x'_{b1} \right\rceil \tag{3.26}$$

$$x_{b2} = x_b - x_{b1}. \tag{3.27}$$

It is easily verified that the above integral value of x_{b1} leads to $|T_{drain}(L) - T_{drain}(R)| \leq 1$, where $T_{drain}(L) = TimeLeft(b-1) + \frac{y_1}{p_L}$ and $T_{drain}(R) = TimeRight(b+1) + \frac{y_2}{p_R}$.

Set L is defined as all of the tokens in $PE_1, \ldots, PE_{b-1}$, and x_{b1} tokens in PE_b. Set R is defined as all of the tokens in $PE_{b+1}, \ldots, PE_l$, and x_{b2} tokens in PE_b. Since, $x_{b1} = \lceil x'_{b1} \rceil$, $T_{drain}(L) = TimeLeft(b-1) + \lceil \frac{y_1}{p_L} \rceil$, $T_{drain}(R) = TimeRight(b+1) + \lceil \frac{y_2}{p_R} \rceil$, and $|T_{drain}(L) - T_{drain}(R)| \leq 1$ (Eq. (3.23)). ■

The running time of the above algorithm is $O(l)$, as it takes $O(l)$ time to compute each of the four arrays, $O(l)$ to find b using arrays $TimeLeft(i)$ and $TimeRight(i)$ (Eq. (3.22)), and $O(1)$ time to find x_{b1} and x_{b2} (Eq. (3.23)). Hence, the algorithm is optimal as it takes $\Omega(l)$ time to read the l numbers, $x_1, \ldots, x_l$.

The following lemmas are used to prove that the proposed schedule **Opt** minimizes the total drain time.

Lemma 3.7 *There exists an optimal drain schedule in which the trajectory of each token to the boundary PEs is not a zig-zag path, i.e., there are no direction changes.*

Proof. Consider an optimal schedule in which some data token d in PE_i moves in a zig-zag path from PE_i to a boundary PE PE_l. Let PE_j, $1 \leq j \leq l$, be a PE which is visited twice by token d traveling in opposite directions as it moves to PE_j from PE_l (PE_j must exist else the path is not zig-zag). Therefore, the path of d is $PE_i \rightarrow PE_j \rightarrow PE_k \rightarrow PE_j \rightarrow PE_l$ as depicted in Figure 3.8. Let t_i be the time at which d starts from PE_i, t_j be the time at which d enters PE_j first, t'_j be the time at which d reenters PE_j, and t_l be the time at which it leaves the array from PE_l, such that $t_i \leq t_j < t'_j \leq t_l$. The total transit time for d is then $t_l - t_i + 1$ steps.

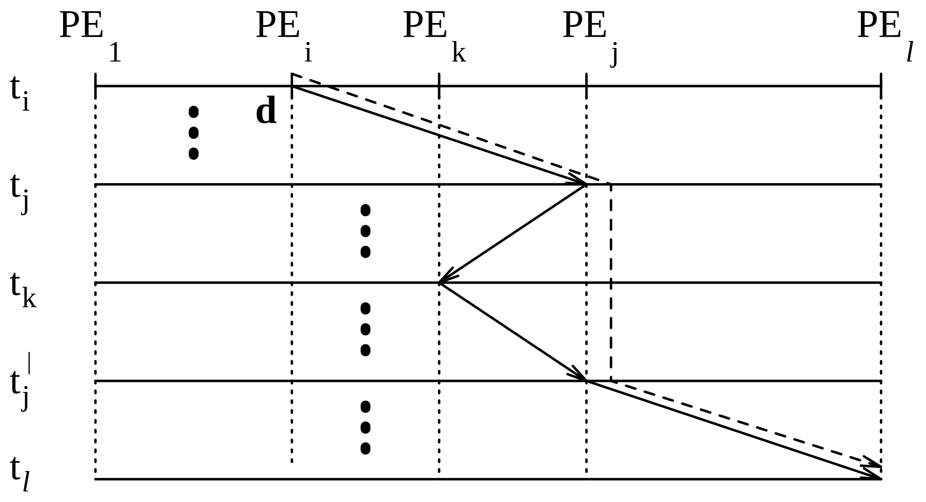


Figure 3.8: Eliminating zig-zag paths in the optimal schedule. The dashed line represents the new schedule of token d from PE_i to PE_l without direction changes.

Our approach is to construct an equivalent schedule without zig-zag paths that moves d from PE_i to PE_l in the time interval t_i to t_l without interfering with the draining of other tokens in the array. The new schedule (indicated by a dashed line in Figure 3.8) is

- move d from PE_i to PE_j in $t_j - t_i + 1$ steps starting from time t_i as per the original schedule,
- retain token d in PE_j from time step t_j to $t_j^{'}$, and
- move token d from PE_j to PE_l starting from time $t_j^{'} + 1$ to t_l following the original schedule.

Thus, the new schedule for token d is also optimal as token d leaves PE_l at time t_l. Hence, there exists an optimal schedule in which the trajectory of each token to the boundary PE is not a zig-zag path. ■

Let set L be the tokens drained out of the PE_1 and set R be the tokens drained out of PE_l. The following lemma establishes that there exists an optimal schedule where the trajectories of the tokens in set L and those in set R do not cross. Hence, the optimal

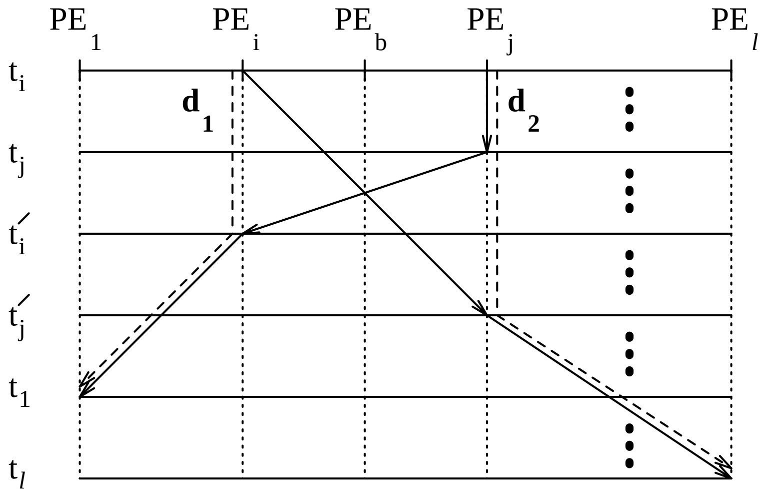


Figure 3.9: Draining set L left from PE_1 and set R right from PE_l. The dashed line shows the modified schedule.

partitioning of the tokens in the linear array can be done by finding a dividing line (as in Figure 3.7) such that all of the tokens to the left of the line are drained left from PE_1, and the tokens to the right of the line are drained right from PE_l.

Lemma 3.8 *Consider a partitioning of the tokens into sets L and R, where set L consists of the tokens drained out of PE_1, and set R consists of the tokens drained out of PE_l. There exists an optimal drain schedule in which the trajectories of the tokens in set L do not intersect with the trajectories of the tokens in set R.*

Proof. Consider an optimal schedule in which every token moves in only one direction as guaranteed by Lemma 3.7. Assume for contradiction that there exist tokens d_1 and d_2 in PE_i and PE_j such that $d_1 \in R$, $d_2 \in L$ and $i < j$ (Figure 3.9).

Let t_i and t_j be the times when d_1 and d_2 are moved from PE_i and PE_j respectively. Let t_i' and t_j' be the times when d_1 and d_2 are moved out of PE_j and PE_i as they move to PE_l and PE_1, respectively. Let t_l and t_1 be the times at which d_1 and d_2 exit the array from PE_l and PE_1, respectively. Consider the following modified drain schedules for tokens d_1 and d_2:

- d_1 remains in PE_i from time t_i to t_i', and moves from PE_i to PE_1 in time t_i' to t_1 following the trajectory of d_2 from time t_i' to t_1, and

- d_2 remains in PE_j from time t_j to $t_j^{'}$, and moves from PE_j to PE_l in time $t_j^{'}$ to t_l following the trajectory of d_1 from time $t_j^{'}$ to t_l.

The schedule of d_2 is used to drain d_1 out of PE_1, and the schedule of d_1 is used to drain d_2 out of PE_l. The token d_1 exits the array at time t_1, and d_2 exits the array at time t_l, without interfering with the movement of other tokens. Therefore, the above modified schedule for d_1 and d_2 does not increase the optimal drain time. Hence, there exists an optimal schedule in which the trajectories of the tokens of set L and the tokens in set R do not intersect. ■

Lemma 3.9 *There exists an optimal drain schedule in which the PEs move data tokens through their output links in a greedy fashion.*

Proof. Consider PE_i, $1 < i \leq b$, whose tokens belong to set L. Note that PE_i can move at most p_L tokens through its p_L ports to PE_{i-1}, $1 < i \leq b$, in each cycle. Assume that in the optimal schedule PE_i transfers $s < \max(x_i(t_i), p_L)$ tokens at some time t_i, where $x_i(t_i)$ is the number of tokens residing in PE_i at time t_i. Note that the full link bandwidth between PE_i and PE_{i-1} is not used at time t_i, as fewer than k_i tokens are transferred. Let d be some token in PE_i that was not transferred at time t_i from PE_i to PE_{i-1} in the optimal schedule. Without loss of generality, d does not have a zig-zag path from the result of Lemma 3.7. Hence, d remains in PE_i until time t_j, $t_j > t_i$, and is then transferred to PE_{i-1} as shown in Figure 3.10 by solid lines. Let $t_j^{'}$ be the time at which d is moved out of PE_{i-1}. Consider a modified drain schedule in which

- d is moved from PE_i to PE_{i-1} at time t_i with the unused link bandwidth between PE_i and PE_{i-1}, and

- d is moved from PE_{i-1} to PE_1 in a greedy fashion, whenever link bandwidth is available.

Since in the original optimal schedule d moves out of PE_{i-1} at time $t_j^{'}$, in the modified schedule d will have to wait at worst until time $t_j^{'}$ in PE_{i-1} before it is moved to PE_{i-2}. Thus, d will reach PE_1 at or before time t_1. (Note that if link bandwidth is available in

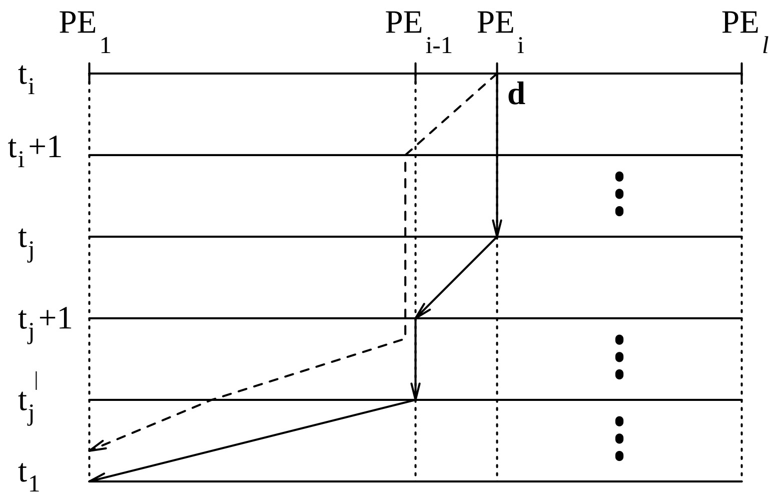


Figure 3.10: Greedy draining of tokens from PEs. The dashed line shows the modified optimal schedule where token d is moved in a greedy way.

PE_{i-1} before $t_j^{'}$, d will be moved out earlier than t_j and may exit the array before time t_1 from PE_1.) Therefore, the new greedy schedule for token d does not increase the optimal drain time. Hence, there exists an optimal schedule where the PEs move tokens greedily through their links. ■

Theorem 3.4 *The schedule* **Opt** *in Section 3.2.3 minimizes the total time to drain all the tokens in the linear array.*

Proof. The proposed schedule **Opt** in Section 3.2.3 consists of the following steps:

1. Divide the tokens into two sets L and R, such that all of the tokens in L are drained left from PE_1, all of the tokens in R are drained right from PE_l, and the time to drain L differs from the time to drain R by at most 1, i.e., $|T_{drain}(L) - T_{drain}(R)| \leq 1$ (Procedure 3.1)

2. Drain set L left greedily without direction changes

3. Drain set R right greedily without direction changes

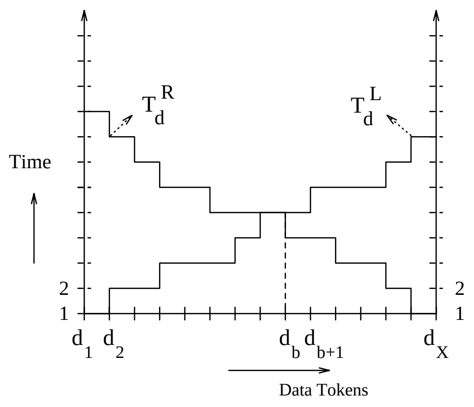


Figure 3.11: Variation of $T_d^L(i)$ and $T_d^R(i)$, $i = 1, \ldots, X$. The total drain time is minimized when $L = \{d_1, \ldots, d_b\}$ and $R = \{d_{b+1}, \ldots, d_X\}$ such that $T_d^L(b) = T_d^R(b)$.

Lemmas 3.7, 3.9, and 3.8 have established that there exists an optimal schedule that follows steps 2,3 in the above schedule. To prove that the schedule **Opt** is optimal, it has to be shown that the division should be done such that $|T_{drain}(L) - T_{drain}(R)| \leq 1$.

Let $d_{x_{i-1}+1}, d_{x_{i-1}+2}, \ldots, d_{x_i}$ denote the x_i tokens in PE_i, $i = 1, 2, \ldots, l$. Thus, $d_1, \ldots, d_{x_1}$ are the tokens in PE_1, $d_{x_1+1}, \ldots, d_{x_2}$ are the tokens in PE_2 and so on. Let $T_d^L(i)$ be the time taken to drain set L, when L consists of the tokens $d_1, \ldots, d_i$, $i = 1, \ldots, X$, $(X = \sum_{i=1}^{l} x_i)$. Let $T_d^R(i)$ be the time taken to drain set R, when R consists of tokens $d_i, \ldots, d_X$, $i = 1, \ldots, X$. Figure 3.11 shows the variation of $T_d^L(i)$ and $T_d^R(i)$ as functions of i. The total drain time T_{drain} is the maximum of $T_d^L(i)$ and $T_d^R(i+1)$, i.e., $T_{drain} = \max(T_d^L(i), T_d^R(i+1))$, where set $L = \{d_1, \ldots, d_i\}$ and set $R = \{d_{i+1}, \ldots, d_X\}$. Since, $T_d^L(i)$ is a nondecreasing function of i and $T_d^R(i)$ is a nonincreasing function of i, the total drain time T_{drain} is minimized when $T_d^L(i) - T_d^R(i+1)$ is minimized. Let b be an integer between 1 and X such that $T_d^L(b) = T_d^R(b)$ (at least one such b has to exist as $T_d^L(1) = T_d^R(X) = 1$, $T_d^L(X), T_d^R(1) \geq 1$, and $T_d^L(i) \leq T_d^L(i+1)$, $T_d^R(i) \geq T_d^R(i+1)$). Let $L = \{d_1, \ldots, d_b\}$, and $R = \{d_{b+1}, \ldots, d_X\}$. Hence, $T_d^L(b) = T_d^R(b) = T_d^R(b+1) + \epsilon$, where $\epsilon = 0, -1$ (since $T_{drain}^R(i)$ is a nonincreasing function of i that decreases by at most 1 when i is increased by 1). Any other choice of L and R will result in a higher drain time T_{drain}. The partitioning of the tokens into sets $L = \{d_1, \ldots, d_b\}$ and $R = \{d_{b+1}, \ldots, d_X\}$ such that $|T_{drain}(L) - T_{drain}(R)| \leq 1$ minimizes the total drain time T_{drain}. Hence, the schedule *Opt* minimizes the total time to drain all of the tokens in the linear array. ■

Example 3.1 Figure 3.12 shows the PE structure and the data input patterns for a linear processor array that optimizes computation time T_{comp} and completion time T_c, for $N = 4$. Matrices **A** and **B** are moving and encounter a delay of one cycle between PEs. Matrix **C** is stationary and distributed in the PEs as shown in Figure 3.12. The number of left output ports $p_L = 1$ and number of right output ports $p_R = 2$. The optimal drain schedule works as follows. The 16 data tokens are partitioned into two sets, L and R, having 6 and 10 tokens, respectively. Set L contains data tokens in PEs 1,2,3, and set R contains data tokens in PEs 4,5,6,7. The time at which each data token exits the array using the above described schedule is indicated in Figure 3.12. The drain time obtained is 6 time steps, which is optimal, since the minimum time to drain 16 tokens using 3 output ports is 6 time steps ($\lceil \frac{16}{3} \rceil = 6$). ■

3.2.4 Processor-optimal designs

If the objective is to minimize $\#PE$, then Theorem 3.5 characterizes the $\#PE$-optimal design.

Theorem 3.5 *The parameters* $(t_1, t_2, t_3) = (N, 1, 1)$ *and* $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, 0, 1)$ *result in a linear PA with a primary objective of minimizing the number of PEs, and a secondary objective of minimizing the completion time.*

Proof. From Lemma 3.2, $\#PE = (N-1)(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_3\right|) + 1$. All three displacements, $\vec{k}_1$, $\vec{k}_2$, and $\vec{k}_3$, cannot be zero simultaneously, as all of the computations will be executed serially on one PE. Hence,

$$\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_3\right| \geq 1. \tag{3.28}$$

For minimum PE count, $\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_3\right| = 1$. Therefore, $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, 0, 1)$ gives the minimum PE count of N.

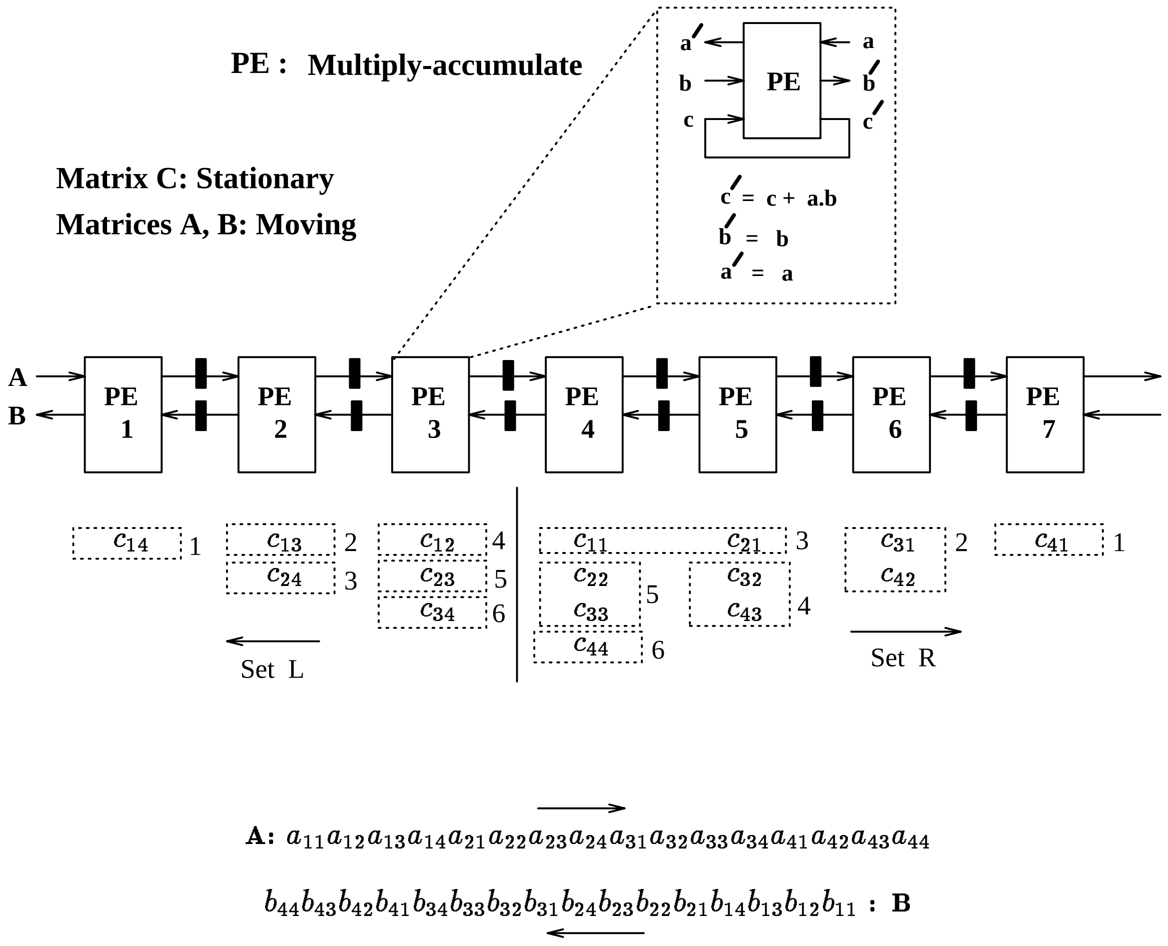


Figure 3.12: Linear processor array for finding the product **C** of two 4×4 matrices **A** and **B** using parameters $(t_1, t_2, t_3) = (1, 2, 2)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. The array is optimal for minimum T_{comp} and minimum T_c.

Table 3.6: $\#PE$ optimal linear arrays for product of two $N \times N$ matrices.

N	GPM: $\#PE$-optimal Designs (Theorem 3.5) $(t_1,t_2,t_3) = (N,1,1)$, $(\vec{k}_1,\vec{k}_2,\vec{k}_3) = (0,0,1)$ $(T_{load},T_{comp},T_{drain})$	#PE
3	(4,11,3)	3
4	(7,19,6)	4
8	(23,71,22)	8
16	(87,271,86)	16
32	(343,1055,342)	32
64	(1367,4159,1366)	64
100	(3335,10099,3334)	100
200	(13335,40199,13334)	200
300	(30001,90299,30000)	300

To prove that $(t_1, t_2, t_3) = (N, 1, 1)$ minimizes the completion time given the displacements $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, 0, 1)$, consider the spacings of the nonstationary input **B**. From Eq. (3.12), $s_{3,1} = \frac{t_1}{t_3}$, and $s_{3,2} = \frac{t_2}{t_3}$. From Corollary 3.2, $t_1 + t_2 \geq N + 1$, for any feasible design. Since matrices **C** and **A** are stationary (as $\vec{k}_1 = \vec{k}_2 = 0$), minimizing the completion time $T_c = T_{load}(\mathcal{A}) + T_{load}(\mathcal{B}) + T_{comp} + T_{drain}(\mathcal{C})$ is equivalent to minimizing T_{comp} ($T_{load}(\mathcal{A}) = T_{drain}(\mathcal{C}) = \lceil \frac{N^2}{3} \rceil$, and $T_{load}(\mathcal{B}) = 1$ from Lemma 3.6 as $\vec{S}_{3,1}, \vec{S}_{3,2}$ are both in direction opposite to $\vec{k}_3$). Thus, we have to choose values of t_1, t_2, t_3, such that $t_1 + t_2 \geq N + 1$, and $t_1 + t_2 + t_3$ is minimized, which leads to $t_3 = 1$ and $t_1 + t_2 = N + 1$ for minimum completion time. Therefore, $(t_1, t_2, t_3) = (N, 1, 1)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, 0, 1)$ result in a linear PA with a primary objective of minimizing the number of PEs and a secondary objective of minimizing the completion time. ■

Note that in the optimal-$\#PE$ designs, input **A** and output **C** are stationary (preloaded) with each PE containing one row of **A** and **C**. The number of PEs and the load, drain and computation times of these minimum-$\#PE$ designs are shown in Table 3.6. From Eqs. (2.2) and (2.4), the corresponding DM parameter for the minimum-$\#PE$ designs are $\vec{\Pi} = [1, 1, N]^t$ and $\mathbf{S} = [1, 0, 0]^t$.

As a comparison, Table 3.7 shows the values of $T_{load}, T_{comp}, T_{drain}$, and $\#PE$ for designs obtained by Lee and Kedem (LK) [40] and Shang and Fortes (SF) [2]. These designs are

Table 3.7: Feasible designs found by DM ($T_c = (T_{load}, T_{comp}, T_{drain})$).

N	Designs by LK [40]		Designs by SF [2]	
	T_c	#PE	T_c	#PE
3	(7,11,9)	7	(7,9,5)	7
7	(19,55,73)	19	(67,49,13)	19
15	(43,239,393)	43	(379,225,29)	43
31	(91,991,1801)	91	(1771,961,61)	91
63	(187,4031,7689)	187	(7627,3969,125)	187
101	(301,10301,20001)	301	(19901,10201,201)	301
201	(601,40601,80001)	601	(79801,40401,401)	601
301	(901,90901,180001)	901	(179701,90601,601)	901

feasible ones that do not optimize T_c, T_{comp}, or $\#PE$. The following table presents the parameters of these designs as well as the corresponding GPM parameters.

Method	$\vec{\Pi}$	$\mathbf{S}$	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$
LK	$[1, 2, N-1]^t$	$[1, 1, -1]^t$	$(N-1, 2, 1)$	$(-1, 1, 1)$
SF	$[1, N-1, 1]^t$	$[1, 1, -1]^t$	$(1, N-1, 1)$	$(-1, 1, 1)$

3.2.5 Processor-time trade-offs

Comparing the results in Tables 3.4, 3.5, and 3.6, we found, for instance, that for a problem of size 200, the T_c-optimal design is 5.44 times faster than the $\#PE$-optimal design in terms of completion time, and uses 27.87 times more PEs than the $\#PE$-optimal design. (The T_c-optimal design for $N = 200$ requires 12,298 time units and 5573 PEs, whereas the $\#PE$-optimal design requires 66,868 time units and 200 PEs.) A designer might be unwilling to settle for either the large number of PEs required in the minimum-time design or the long completion time of the minimum-processor design. In practice, there may be bounds on the number of processors or the completion time or both. In the following, we present optimal designs with bounds on T_c and $\#PE$.

Figure 3.13 shows how $\#PE$ varies with T_c for 2 different problem sizes: $N = 50$, and 100. The y-axis $\#PE$ is normalized by $\#PE^{max}$ (processor count of the minimum-T_c design), and the x-axis T_c is normalized by T_c^{max} (completion time of the minimum-processor design). This

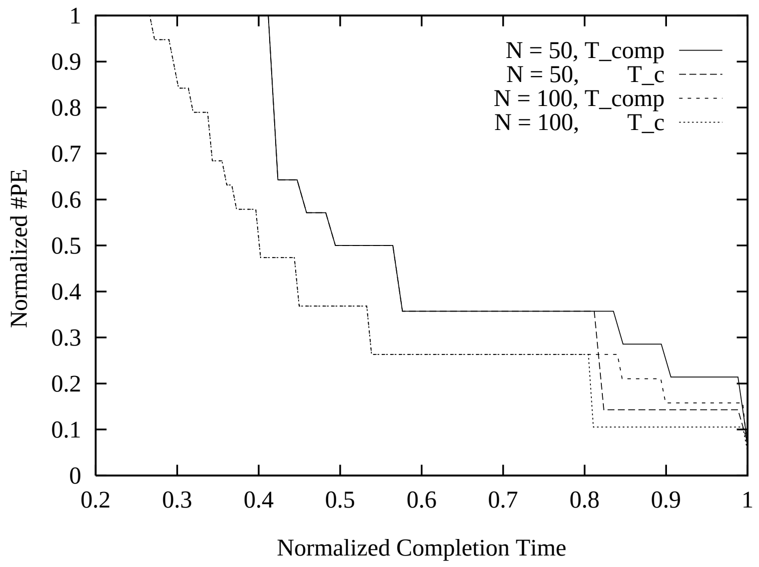


Figure 3.13: Performance trade-offs: Variation in $\#PE$ with T_c or T_{comp}. The plots are given for two problem sizes $N = 50$ and 100.

allows a comparison of different problem sizes uniformly on the same scale. It is also clear from Figure 3.13 that there are better alternatives than the time-optimal or $\#PE$-optimal designs.

Given the bound T_c^{ub} (respectively, $\#PE^{ub}$) the designer can use Figure 3.13 to find the minimum $\#PE$ (respectively, T_c) required and decide (possibly from a cost perspective) if it is acceptable. The designer can exploit the initial steep decline in the plots to choose an alternative design that trades performance for cost. For instance, the minimum $\#PE$ required for $N = 50$ drops by 43% for only a 15% increase in completion time.

If both T_c and $\#PE$ are bounded from above, then we draw a horizontal line across the graph for the desired bound on $\#PE$. The intersection between this line and the stepped curve represents the minimum T_c needed for any feasible design. If this minimum T_c is less than the desired T_c, then a feasible design can be obtained that represents the best design under both time and processor constraints.

Figure 3.13 also shows the trade-offs when T_{comp} is bounded instead of T_c. These plots (denoted by T_{comp} in Figure 3.13) are obtained by computing the completion times T_c of designs that optimize T_{comp}. Given a bound T_c^{ub}, we can see that the number of processors obtained by minimizing T_c is less than or equal to the number of processors obtained by minimizing T_{comp}.

3.3 Summary

This chapter presents the results of applying the GPM to derive linear processor arrays for two important problems: finding the transitive closure of an $N \times N$ matrix and computing the product of two $N \times N$ matrices. Both of the chosen problems are benchmark problems whose structure is found in many different applications. The results of the transitive closure problem is applicable to other members of a larger class of problems called the algebraic path problems (APP), of which transitive closure is a special case. Similarly, the results for the matrix-product problem can be easily extended to other member in the class of 3-D cube graph algorithms.

In this chapter, results have been presented for the following five formulations of the problem of finding optimal linear processor array designs:

- T_{comp}-optimal designs without bound on $\#PE$ (Tables 3.1, 3.4),
- T_c-optimal designs without bound on $\#PE$ (Tables 3.2, 3.5),
- $\#PE$-optimal designs without bound on T_c or T_{comp} (Theorems 3.2, 3.5),
- Optimal designs with specific bounds on T_{comp} or $\#PE$ (Figures 3.4, 3.13), and
- Optimal designs with specific bounds on T_c or $\#PE$ (Figures 3.5, 3.13).

The space-time diagrams and data-input patterns are shown for finding the transitive closure of a 4×4 matrix in Figures 3.2 and 3.3, and for computing the product of two 4×4 matrices in Figure 3.12. Tables 3.1, 3.4, 3.2, 3.5 also indicate the optimal values of $\vec{\Pi}$, and $\mathbf{S}$ of dependence method, and illustrate that the GPM can be used to obtain optimal designs in the representation of dependence-based methods (Figure 2.1). In the next chapter, the extension of the GPM to two-level pipelined processor arrays with each PE having pipelined functional units is presented.

4. SYNTHESIS OF PROCESSOR ARRAYS WITH PIPELINED ARITHMETIC UNITS

This chapter describes an extension of the General Parameter Method to synthesize processor arrays (PAs) with pipelines of PEs, where each individual PE has pipelined functional units (PFU). This technique of pipelining operations across PEs, and pipelining operations in the functional units of the PEs, is known as *two-level pipelining.*

The motivation behind two-level pipelined PAs is as follows. Given the design of a PA, its performance can be improved only by increasing the rate at which the PA is clocked. But, it has become clear that it is increasingly difficult to reduce the cycle time of non-pipelined arithmetic-logic units (ALUs) in the PEs beyond a certain limit. In fact there is a steep increase (possibly exponential) in design cost as well as design time when the the array is designed for higher clock rates. For instance, the design and simulation tools existing today can be used to design an array at a 10-20 MHz clock rate rather than at a 150-200 MHz clock rate. A possible solution, then, is to design the ALUs in the PE of the array as pipelined functional units and subdivide computations performed by each ALU into multiple stages. This permits the array with PFU to be clocked at a much higher rate resulting in a net increase in performance.

Using two-level pipelining, it is much easier to design a PA with the same clock rate, using PFUs that perform fewer operations per cycle than a PA with nonpipelined functional units. This leads to designs that require less power and are easier to manufacture. On the other hand, the pipelined design can be clocked at a much higher rate using superpipelining (up to S times faster for an S-stage pipeline than a nonpipelined design since there is less circuitry in each pipeline stage). This leads to a higher throughput than a nonpipelined design.

A major problem in designing two-level pipelines is the lack of a systematic method that evaluates alternative designs and finds optimal designs with respect to user-specified objective function and design constraints. The search for the optimal design is different than that of PAs without PFUs because internal pipelining introduces a new dimension in the design problem (effectively constraining the search space). Further, internal pipelines in a PE can have a large number of stages (over 20 stages) when several functional units in a PE are chained together to perform the operations in the body of a complex inner loop.

This chapter describes an enhancement to the General Parameter method to map nested-loop algorithms into PAs with PFUs. The organization of the chapter is as follows. Section 4.1 presents a brief review of GPM, followed by presentation of the new constraints incorporated into GPM to automatically synthesize PAs with PFUs in Section 4.2. In Section 4.3, a closed-form condition for deriving pipelined designs from nonpipelined ones for n-dimensional (written as n-D) mesh algorithms is described. In Section 4.4, the results of applying the enhanced GPM to the matrix-product and transitive-closure applications are shown, followed in Section 4.5 by a comparison to previous work. Section 4.6 concludes this chapter with a summary.

4.1 Review of General Parameter Method

The target algorithms in GPM are those that can be modeled directly as uniform recurrences as well as *affine recurrences* that can be "uniformized" to yield equivalent sets of *uniform recurrences* (Section 1.2).

The GPM synthesizes a set of n-D uniform recurrence equations into a set of vector and scalar parameters that characterize the behavior, correctness, and performance of an m-D PA, where $m < n$. A total of $r \times (r + 2)$ parameters are defined, of which r parameters are periods (scalars); the remaining $r^2 + r$ are m-D vectors, of which r parameters are displacements (or velocities), and r^2 are spacings (r of these spacings are trivially zero) (Section 2.2).

The search for the "best" design can be represented by the following optimization problem:

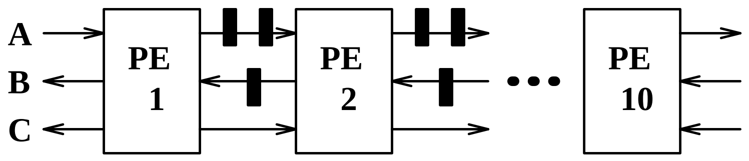
$$\textbf{Minimize} \quad b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) \tag{4.1}$$

$$\textbf{Subject To:} \begin{cases} 1 \leq t_i, \quad i = 1, \ldots, r, \\ 0 \leq \left|\vec{k}_i\right| \leq t_i, \quad i = 1, \ldots, r \\ \text{constraints defined in Theorems 2.1, 2.2 and 2.3} \\ \#PE \leq \#PE^{UB} \text{ and } T_c \leq T_c^{UB}. \end{cases} \tag{4.2}$$

The design objective at the logical level is a function of the number of PEs and the completion time of processing an application problem. The optimal design is *problem-size dependent.* If the objective is a nonmonotonic function of the parameters, a monotonic component of the objective is identified and enumerated efficiently. The difference, however, is that a feasible solution defines an upper bound on the value of the monotonic component being enumerated. The search is then continued to find better designs until the upper bound is reached. A special case happens when the objective function is monotonic. In this case, the upper bound defined by the feasible solution is equal to the feasible objective value itself, making the feasible solution optimal. Additional design constraints such as (i) maximum number of PEs allowed in a design for a given application, (ii) maximum allowed completion time, and (iii) additional delays incurred inside a PE (such as those due to internal pipelines in a PE) can be easily incorporated into the optimization. The next section details the constraints for such PAs with two-level pipelining. The following example shows a design of a processor array with nonpipelined functional units using techniques presented in Chapter 2.

Example 4.1 Figure 4.1 shows a linear PA that minimizes computation time and #PE for multiplying two 4-by-4 matrices using nonpipelined functional units. Note that *computation time* measures the time between the first and last computations in the PA, and does not include load time of inputs and drain time of outputs. For a linear PA, the parameter values are $(t_1, t_2, t_3) = (1, 2, 3)$ and $(k_1, k_2, k_3) = (1, 1, -1)$. Each PE is an inner-product processor, with computation time and $\#PE$ given as

$$T_{comp} \;=\; 1 + (N-1)(t_1 + t_2 + t_3) = 19, \tag{4.3}$$



$a_{44} - - - a_{43}a_{34} - - a_{42}a_{33}a_{24} - a_{41}a_{32}a_{23}a_{14} - a_{31}a_{22}a_{13} - - a_{21}a_{12} - - - a_{11}$

$b_{14}b_{24}b_{34}b_{44} - b_{13}b_{23}b_{33}b_{43} - b_{12}b_{22}b_{32}b_{42} - b_{11}b_{21}b_{31}b_{41} - - - - - - - - - -$

$c_{44}c_{34}c_{24}c_{14}c_{43}c_{33}c_{23}c_{13}c_{42}c_{32}c_{22}c_{12}c_{41}c_{31}c_{21}c_{11} - - - - - - - - - - - - - - - - - -$

Figure 4.1: Linear PA for 4×4 matrix product with nonpipelined functional units.

$$\#PE = 1 + (N-1)(|k_1| + |k_2| + |k_3|) = 10. \tag{4.4}$$

■

4.2 New Constraints for Pipelined Functional Units

The effect of pipelining can be summarized by the following simple observation.

Observation 4.1 *Pipelining with S stages is equivalent to retiming by which the delays on all inputs or all outputs, but not both, are increased by S.*

In a PA, two types of data flows exist.

- *Dependent.* The datum is modified as it flows through the PA (read-write data) ;
- *Transmittent.* The datum is not modified (read-only).

If the processors have PFUs, only the delays of the dependent data and not the transmittent data have to be increased. Figure 4.2 shows the effect of pipelining.

The following lemma characterizes the effect of internal pipelines in PEs as new constraints in GPM. We assume that in general each dependent flow i through the PE is computed using a pipeline with S_i stages.

Lemma 4.1 *Pipelining the functional unit is equivalent to enforcing the following constraints in GPM:*

$$t_i \geq S_i, \quad \textit{for all dependent flows } i, \tag{4.5}$$

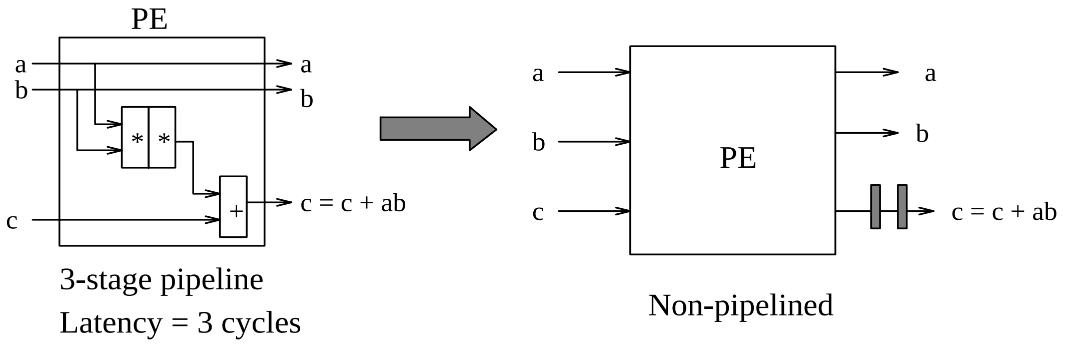


Figure 4.2: Pipelining is equivalent to adding delays to dependent data.

where S_i is the number of stages in the pipeline for dependent flow i.

Proof. An S_i-stage pipeline delays dependent flow i passing through the PE by S_i time steps. Since, by definition, period t_i captures the time difference between two dependent computations of data flow i separated in the index space by vector $\vec{d_i}$, constraint $t_i \geq S_i$ incorporates into GPM the effect of an S_i-stage pipeline to compute the dependent flow i in each PE. ■

Note that pipelining in a PE amounts to a reduction of the search space (as $t_i \geq 1$ for nonpipelined designs). Hence, the computation or completion time (cycle count of the PA) of the optimal two-level pipelined design is more than or at best equal to that of the nonpipelined design. However, the clock rate of a pipelined design can be increased easily to match the same throughput as that of a nonpipelined design. This approach, therefore, provides a trade-off between the degree of internal pipelining in PFUs and the rate at which PEs are clocked.

4.2.1 Minimum initiation interval greater than one

The constraints in Eq. (4.5) assume that two independent operations can be initiated into a PFU in successive cycles without any stage conflicts; that is, MII, the Minimum Initiation Interval, is 1. In general, for pipeline i computing dependent flow i, its reservation table coud have a $MII = \delta_i > 1$. Since in this model, each PE computes the entire body of the innermost loop in each time step, it is necessary to ensure that no two executions of any PE are within δ time steps of each other, where $\delta = \max_i \delta_i$. The following conditions avoid scheduling two index points within δ steps in any PE:

$$\overset{Min}{\forall \vec{\alpha}} \left|\vec{T}.\vec{\alpha}\right| > \delta, \text{ subject to } \mathbf{K}\vec{\alpha} = 0 \text{ and } \mathbf{D}\vec{\alpha} \neq 0\,, \tag{4.6}$$

where $\vec{T} = [t_1 t_2 \ldots t_m]$ is the vector of periods, $\mathbf{K} = [\vec{k_1}\vec{k_2} \ldots \vec{k_m}]$ is the matrix of displacements, and $\mathbf{D} = [\vec{d_1}\vec{d_2} \ldots \vec{d_m}]$ is the matrix of dependencies. These conditions imply that if two distinct computations are assigned to the same PE, then the time between executing them should be at least δ steps. For each feasible solution (satisfying Eq. (4.2)) found in GPM, Eq. (4.6) has to be checked, increasing the time complexity to find the optimal parameter values.

4.3 Pipelined Designs for n-D Meshes

In this section, we present a closed-form condition for deriving pipelined designs from nonpipelined ones for a class of algorithms whose dependence graphs are n-D meshes. For an n-D mesh algorithm, the dependence graph is the identity matrix in n-D (I_n) with $n-1$ transmittent flows of data and one dependent flow. For example, computing a matrix product is a 3-D cube graph algorithm, with **C** as the dependent flow and **A** and **B** as transmitent flows.

Definition 4.1 Let $\vec{I} = (i_1, \cdots, i_k, i_{k+1}, \cdots, i_n)^t$ be the index vector of an n-D mesh dependence graph G. We define a cyclic interchange of distance k, denoted as CI_k, as a reindexing of G as follows:

$$CI_k(G) : (i_{n-k+1}, \cdots, i_n, i_1, \cdots, i_{n-k})^t \longrightarrow (i_1, \cdots, i_k, i_{k+1}, \cdots, i_n)^t. \tag{4.7}$$

Hence, $CI_k(G)$ is a linear transformation of dependence graph G and can be represented by the permutation matrix $\mathbf{P}_k = [\vec{e}_{n-k+1}, \vec{e}_{n-k+2}, \cdots, \vec{e}_n, \vec{e}_1, \vec{e}_2, \cdots, \vec{e}_{n-k}]^t$, where column vector $\vec{e}_i = (\underbrace{0, \cdots, 0}_{i-1}, 1, 0, \cdots, 0)^t$ is the i-th unit vector with a single 1 in the i-th position.

We extend the definition of cyclic interchange CI_k to apply to PAs as well. A PA for an n-D mesh algorithm has n inputs: $\Omega_1, \Omega_2, \cdots, \Omega_n$. A cyclic interchange, CI_k, for a PA is equivalent to renaming the input variables as follows. $CI_k : (\Omega_{n-k+1}, \cdots, \Omega_n, \Omega_1, \cdots, \Omega_{n-k}) \rightarrow (\Omega_1, \cdots, \Omega_k, \Omega_{k+1}, \cdots, \Omega_n)$, i.e., variable Ω_{n-k+1} is renamed as Ω_1, Ω_{n-k+2} as V_2, and so on. Therefore, when referring to dependence graphs, a cyclic interchange CI_k means a reindexing of the index space. In the context of PAs, a cyclic interchange CI_k means a reindexing of the n-input variables. For example, for a PA that computes a matrix product, a cyclic interchange of distance 1 renames variables (C, A, B) to (A, B, C). Thus, in Figure 4.4 (p. 92), variable A is replaced by variable C, variable B by A, and variable C by B.

Since PAs for solving a given algorithm are obtained by a linear transformation $\mathbf{T}$ of dependence graph G, the following lemma shows that PAs for n-D mesh algorithms remain valid under cyclic interchanges of their inputs.

Lemma 4.2 *Given an n-D mesh algorithm with a dependence graph G, and a PA with inputs $\Omega_1, \Omega_2, \cdots, \Omega_n$, the new PA obtained by a cyclic interchange $CI_k, k = 1, 2, \cdots, n-1$ of the existing PA is a valid PA that solves the given n-D mesh algorithm.*

Proof. Since mapping a uniform recurrence on a PA corresponds to a linear transformation, cyclically interchanging the inputs of the PA by CI_k is equivalent to reindexing the dependence graph by CI_k. It is easy to see that reindexing by CI_k maps the dependence vectors into each other, resulting in the same set of dependence vectors. Thus, CI_k maps dependence vector $\vec{d_i}$, $i = 1, 2, \cdots, n$, to vector $\vec{d}_{((i-k-1)\bmod n+1)}$. By applying the same transformation ($\mathbf{T}$) that defines the nonpipelined PA to the cyclically interchanged dependence graph ($CI_k(G)$), we have a new PA whose inputs are a cyclic interchange of inputs in the original PA. In other words, the composite linear transformation $\mathbf{T}\,\mathbf{P}_k$ is also a valid transformation, and PAs remain valid under cyclic interchanges for n-D mesh algorithms. ■

The next lemma establishes the main result for deriving designs of PAs with PFUs from existing nonpipelined ones.

Lemma 4.3 *Assume an n-D mesh algorithm involving inputs $\Omega_1, \Omega_2, \cdots, \Omega_n$. Further, assume that a nonpipelined PA exists for this algorithm with periods $t_1, \ldots, t_n$, where t_1 is the* *dependent data flow. To obtain a PA with S-stage pipelined PEs from the original nonpipelined design we can apply CI_{j-1}, a cyclic interchange of distance $j-1$, if there exists a period $t_j \geq S$.*

Proof. For an n-D mesh algorithm, if $t_1 > S$, then the parameters in GPM correspond to a valid S-stage pipelined PA (Eq. (4.5)). Thus, if there exists some j, $1 \leq j \leq n$, such that $t_j > S$, then by applying CI_{j-1} we obtain a dependence graph whose dependent flow is Ω_j instead of Ω_1. This corresponds to a new PA with its inputs interchanged by the same CI_j. By Lemma 4.2, cyclically interchanging the inputs results in a valid PA that still solves the same n-D mesh algorithm. ■

Lemma 4.3 can be used to derive PAs, given existing nonpipelined designs. In the next section, we apply this result as a pruning condition to find optimal PA designs for the matrix-product application.

4.4 Applications

This section presents the application of GPM to find optimal array designs for the matrix-product and transitive-closure problems. The evaluation metric of the ratio of clock periods of the nonpipelined and pipelined designs is described below in Section 4.4.1.

4.4.1 Evaluation metric

The total time to compute a matrix product is equal to $T \times t_c$, where T is the number of clock cycles, and t_c is the clock period of each clock cycle. Hence, for a given PA design, its performance can be improved only by increasing its clock rate. This has become increasingly difficult, as there are a fixed number of gate delays (corresponding to the computation of the inner loop body by each PE) that have to be incurred within a clock cycle.

For a given clock rate, it is much easier to design pipelined functional units in each PE and subdivide the computations (and hence the fixed gate delays) than to design a nonpipelined PE in which the entire inner loop has to be executed within a clock tick. As a result, it is simpler to scale the clock rate of a pipelined design than that of a nonpipelined one. Equivalently, the clock rate of a pipelined design with ALUs of S stages can be up to S times higher than that of a nonpipelined one. Since it is difficult to predict how many times the clock rate of a pipelined design would be as compared to a nonpipelined one (as it depends on whether the designer can split the operation equally and balance the delays), we evaluate an equivalent metric of the ratio of the clock periods as described below.

Let t_c^1 and t_c^S be the clock periods of the nonpipelined PA and a PA with S-stage PFUs (denoted as S-pipelined), respectively. Further, let T_{comp}^1 and T_{comp}^S be the computation times (i.e., clock cycles) of the optimal nonpipelined PA and the optimal S-pipelined PA, respectively. The metric used for evaluating an S-pipelined design over a nonpipelined design is the ratio of clock periods $R = \frac{t_c^S}{t_c^1}$ in order for both designs to have the same performance i.e., $t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1$.

$$\begin{aligned} t_c^S \times T_{comp}^S &= t_c^1 \times T_{comp}^1 && (4.8) \\ \Longrightarrow R &= \frac{t_c^S}{t_c^1} = \frac{T_{comp}^1}{T_{comp}^S} \leq 1 \ \text{(since } T_{comp}^1 \leq T_{comp}^S). && (4.9) \end{aligned}$$

It is easy to see that for the same clock rate, if the pipelined design has the same performance as a nonpipelined one, then all of the internal pipelines in the ALUs are fully utilized. Hence, GPM aims to find pipelined designs so that $T_{comp}^S = T_{comp}^1$ or $R = t_c^S = t_c^1 = 1$, i.e., all of the stages of the internal pipelines are fully utilized. A value of $R < 1$ implies that the pipelined design has to be clocked faster in order to have the same performance as the nonpipelined one. This also means that some of the stages in the internal pipelines are not being fully utilized.

4.4.2 Matrix product: results

Computing the product of two matrices is a fundamental algorithm in signal and image processing. In this section, we present optimal designs of pipelined PAs for computing the product of two $N \times N$ matrices. Since, there is only one dependent flow $\mathbf{C}$, let S denote the number of stages in the PFU to compute $\mathbf{C}$.

Figure 4.3 shows the 3-D plot of the ratio of clock periods $R = t_c^S = t_c^1$ as a function of the number of pipeline stages (S) and the matrix size (N). The X-Y plane of the 3-D plot also

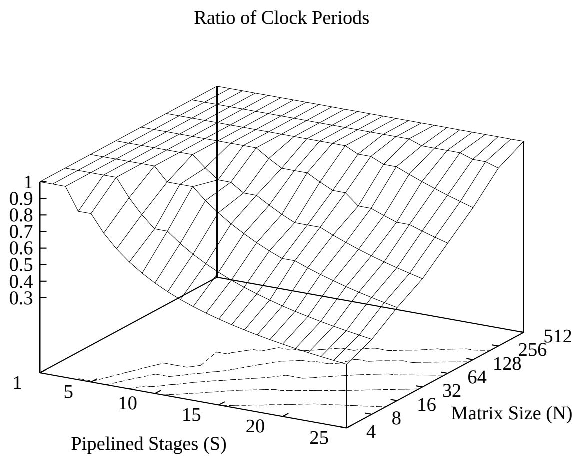
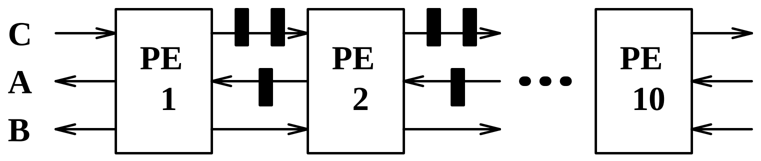


Figure 4.3: Matrix Product: Ratio of clock periods $R = \frac{t_c^S}{t_c^1}$ in order for the the S-pipelined and nonpipelined designs to have the same performance, i.e., $t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1$.

shows the contours for different fixed values of R, as S varies from 1 to 25, and the problem size N varies from 4 to 512 in 8 steps. For any given value of N, R stays at the desired maximum value of 1 until S reaches a critical value ($S_{crit}(N)$), beyond which R starts to fall below 1. For example, for $N = 128$, $R = 1$ for $S \leq 15$, i.e., $S_{crit}(128) = 15$. Similarly, for a given value of S, there is a minimum value of N ($N_{crit}(S)$) such that GPM can find pipelined designs with $R = 1$. Thus, given a 10-stage PFU (typical of an arithmetic pipeline), GPM is able to find a pipelined PA design for the matrix-product problem that can fully exploit the 10-stage pipeline without increasing the cycle count when the matrix size N is at least 64, i.e., $N_{crit}(10) = 64$. Hence, for matrix products, given a value of S, there exists a reasonably small matrix size $N_{crit}(S)$, beyond which the internal pipeline stages can be fully utilized. These results show the ability of GPM to systematically explore the design space to arrive at pipelined designs that optimize the given objective of computation time T_{comp}.

Example 4.2 In the design presented in Figure 4.1, input **C** is the only dependent data flow in computing matrix products. Therefore, if we assume 3-stage PFUs for computing inner



$$c_{44} - - - c_{43}c_{34} - - c_{42}c_{33}c_{24} - c_{41}c_{32}c_{23}c_{14} - c_{31}c_{22}c_{13} - - c_{21}c_{12} - - - c_{11}$$

$$a_{14}a_{24}a_{34}a_{44} - a_{13}a_{23}a_{33}a_{43} - a_{12}a_{22}a_{32}a_{42} - a_{11}a_{21}a_{31}a_{41} - - - - - - - - -$$

$$b_{44}b_{34}b_{24}b_{14}b_{43}b_{33}b_{23}b_{13}b_{42}b_{32}b_{22}b_{12}b_{41}b_{31}b_{21}b_{11} - - - - - - - - - - - - - -$$

Figure 4.4: A PA for matrix multiplication with 3-stage PFUs. Note that this is obtained by a cyclic interchange of the inputs as compared to those shown in Figure 4.1.

products, the constraint $t_1 > 3$ is added to account for the PFUs (Eq. (4.5)). Figure 4.4 shows the data flows for the new PA obtained. The two delays on the C-links in Figure 4.4 indicate that each PE has a 3-stage pipeline. The different stages of the pipeline in each PE are fully utilized to execute independent operations of 3 C elements. This leads to a computation time of this PA of 19 cycles, which is the same as that of the nonpipelined design ($R = 1$). Comparing the data flows in Figures 4.4 and 4.1, we see that the new PA is obtained by replacing the C input by B, B by A, and A by C. ■

Lemma 4.3 has been applied in generating the results in Figure 4.3. For instance, given a matrix size $N = 64$, the optimal nonpipelined design has periods $(t_1, t_2, t_3) = (4, 5, 10)$, resulting in $T_{comp} = 1198$. Using Lemma 4.3, the optimal pipelined designs for S varying from 1 to 10 can be obtained by CI_2 (interchange of distance 2) without any search, as the objective function T_{comp} (which is the sum of the three periods) evaluates to the same value under any cyclic interchange of inputs. From the above parameter values and Lemma 4.3, it is easy to see that all of the optimal designs for $S = 1, \cdots, 10$ have the same computation time of 1198 cycles, i.e., $R = 1$ for $S = 1, \cdots, 10$. Consequently, Lemma 4.3 can be used as an effective pruning condition in GPM to define the flat portion of the surface in Figure 4.3, and can be used in trade-off analysis discussed in the next section.

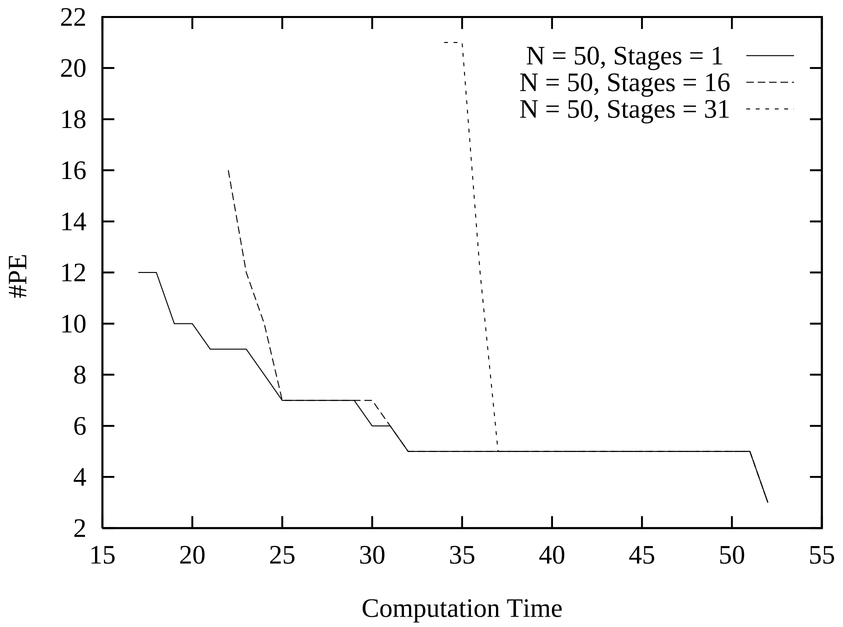


Figure 4.5: Processor-time trade-offs for matrix product: Variation in $\#PE$ with T_{comp}.

Processor-Time Trade-offs

The GPM can be used to generate optimal designs when there are constraints on processor count $\#PE$ or computation time T_{comp} for a given number of stages S in each PFU. Figure 4.5 shows the trade-offs between $\#PE$ and T_{comp} for computing 50×50 matrix products and three values of S. A particular trade-off curve (with a fixed value of S) shows the minimum $\#PE$ (respectively, T_{comp}) required, given an upper bound on the computation time T_{comp}^{ub} (respectively, $\#PE^{ub}$). For example, given a bound $T_{comp} \leq T_{comp}^{ub} = 25$, Figure 4.5 shows that the minimum $\#PE$ needed is 7, if each PE has a PFU with at most 16 stages. Thus, the designer can use this figure to arrive at a final design, given restrictions on resources such as $\#PE$ or T_{comp}. Figure 4.5 also shows that, given a fixed number of PEs, the minimum computation time required increases as the number of stages in the PFU increases. This is true because it becomes harder to utilize the PFUs efficiently. For instance, if $\#PE \leq 12$, then $T_{comp}^{1} \geq 17$, $T_{comp}^{16} \geq 23$, and $T_{comp}^{31} \geq 36$. Further, the plot shows an initial steep

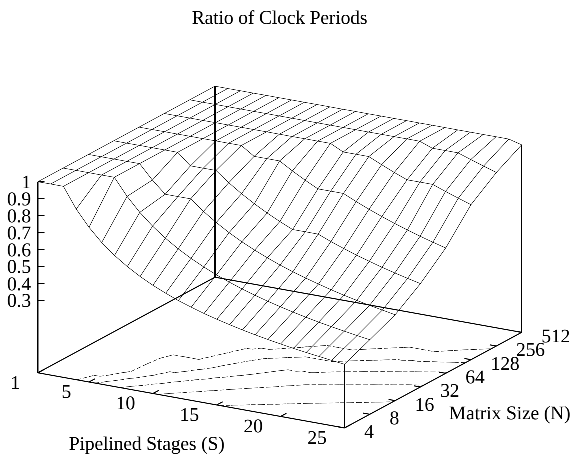


Figure 4.6: Transitive Closure: Ratio of clock periods $R = \frac{t_c^S}{t_c^1}$ in order for the the S-pipelined and nonpipelined designs to have the same performance, i.e., $t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1$.

decline as S is increased. Hence, for larger S, there are more attractive alternatives than the time-optimal or processor-optimal designs.

4.4.3 Transitive closure: results

In this section, we present pipelined linear array designs for the problem of finding the transitive closure of an $N \times N$ matrix (Example 1.4). For transitive closure, the dependent flow is along dependence vector $\vec{d}_3$ (Example 1.4); let S denote the number of stages in the PFU of the PE.

Figure 4.6 shows a 3-D plot of the ratio of clock periods $R = t_c^S = t_c^1$ as a function of the number of pipeline stages (S) and the matrix size (N) for the transitive-closure problem (similar to Figure 4.3 for matrix product). The X-Y plane of the 3-D plot also shows the contours for different fixed values of R, as S is varied from 1 to 25 and the problem size N is varied from 4 to 512. The maximum value of $R = 1$ implies full utilization of internal stages of the pipelined functional unit. For example, when $N = 128$, $R = 1$ for $S \leq 14$

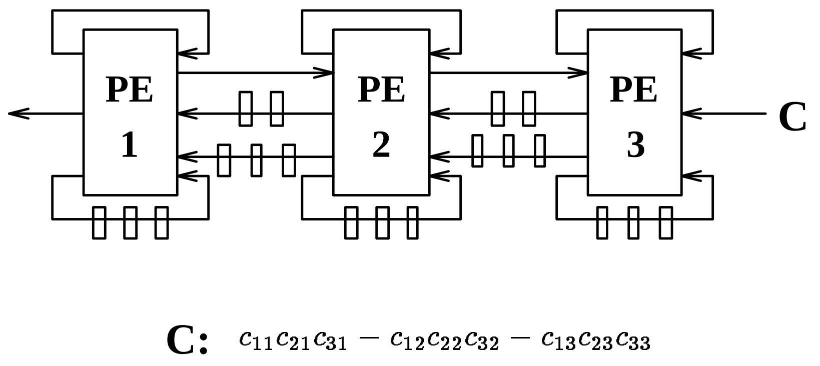

PE 1
PE 2
PE 3
C
C: $c_{11}c_{21}c_{31} - c_{12}c_{22}c_{32} - c_{13}c_{23}c_{33}$

Figure 4.7: Linear PA for transitive closure with 3-stage PFUs with $(t_1, t_2, t_3) = (1, 1, 3)$, $(k_1, k_2, k_3) = (0, -1, 1)$.

($S_{crit}(128) = 14$). Similarly, for a 9-stage PFU, GPM is able to find the best pipelined design when $N \geq 64$ ($N_{crit}(9) = 64$). Compared to the matrix-product problem, the results for transitive closure are slightly worse in the sense that the flat portion of the surface where $R = 1$ is smaller, i.e., R starts to fall off from unity earlier as S or N is increased. This indicates that transitive closure is a more difficult problem to map ono a pipelined PA than the matrix-product problem.

Example 4.3 Figure 4.7 shows the data flows for the pipelined design if we assume a 3-stage PFU to compute the result in each PE. ■

Similar to Figure 4.5, Figure 4.8 shows the processor-time trade-offs for the transitive closure of a 50×50 matrix for three different values of S. These trade-off figures can be used to obtain the best designs when there are constraints on processor count $\#PE$ or computation time T_{comp} for a given number of stages S. For example, given a bound $T_{comp} \leq T^{ub}_{comp} = 25$, Figure 4.8 shows that the minimum $\#PE$ needed is 4, if each PE has a PFU with at most 16 stages. Figure 4.8 also shows that, given a fixed number of PEs, the minimum computation time required increases as the number of stages in the PFU increases. For instance, if $\#PE \leq 6$, then $T^1_{comp} \geq 18$, $T^{16}_{comp} \geq 24$, and $T^{31}_{comp} \geq 37$. This is because it becomes harder to utilize PFUs with more stages efficiently. Further, the sharp decline

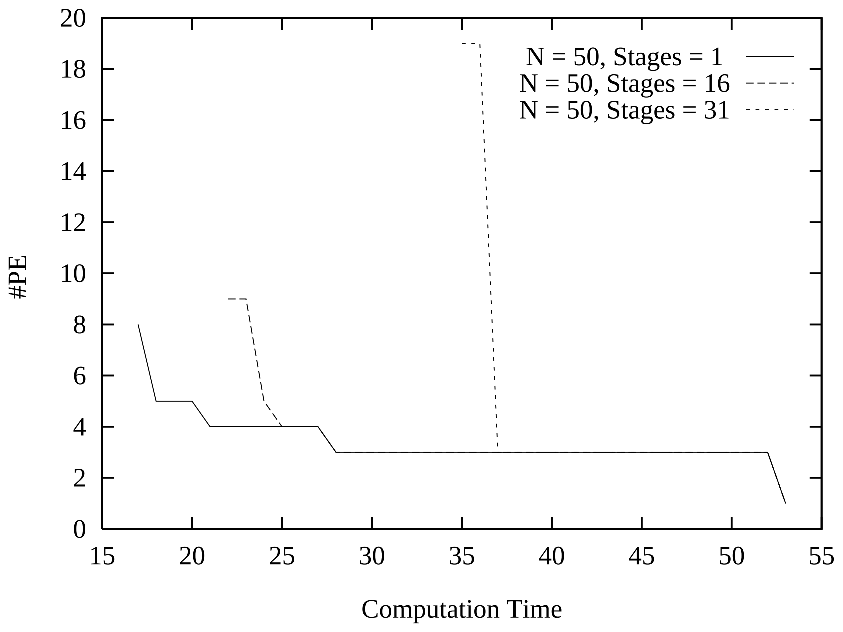


Figure 4.8: Processor-time trade-offs for transitive closure: Variation in $\#PE$ with T_{comp}.

in the curves indicates that there are more attractive alternatives than the time-optimal or processor-optimal designs.

4.5 Comparisons with Existing Work

There have been earlier efforts by Kung and Lam [4] and recently by Valero-Garcia et al., [5] to obtain two-level pipelined PAs. They used a common approach that retimes a PA in order to include additional delays for pipelining. Their approach, however, is restricted to PAs that have already been designed; that is, they took an existing PA and derived a two-level pipelined one by retiming. Since addition of delays increases the clock-cycle count of the retimed PA, they coalesced the PEs in order to obtain better utilization and improved execution time. Valero-Garcia et al. [5], further showed that linear PAs with contra-flow data can be retimed and the PEs coalesced to obtain the same cycle count.

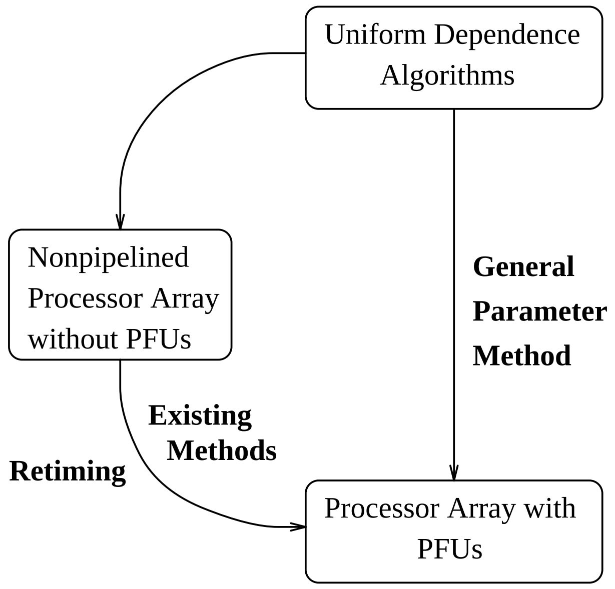

Uniform Dependence Algorithms
Nonpipelined Processor Array without PFUs
General Parameter Method
Existing Methods
Retiming
Processor Array with PFUs

Figure 4.9: Relationship between proposed and existing methods [4, 5] of synthesizing PAs with PFUs.

The drawback with these previous approaches is that coalescing is not always possible. In fact, for lower-dimensional PAs, there is no systematic technique to coalesce PEs by a factor of S. In addition, these methods cannot systematically explore the space of pipelined designs because they start with a particular PA and are restricted only to the data flows defined in it. For instance, the PA shown in Figure 4.4 cannot be obtained by retiming and coalescing, as there is a new data-flow pattern that is a cyclic interchange of the old one. In contrast, GPM can be used as a systematic tool to explore the entire space of pipelined PAs and arrive at a design that optimizes a user-specified objective and satisfies design constraints. Figure 4.9 shows the differences between the two methods. The proposed method incorporates pipelining in the design process and hence can systematically enumerate alternatives.

4.6 Summary

Two-level pipelining in processor arrays (PAs) involves pipelining of operations across processing elements (PEs) and pipelining of operations in functional units in each PE. Although it is an attractive method for improving the throughput of PAs, existing methods for generating PAs with two-level pipelining are restricted and cannot systematically explore the entire space of feasible designs. In this chapter, we extend the General Parameter Method (GPM) to find optimal designs of PAs with two-level pipelines. The basic idea is to add new constraints to include the effect of internal functional pipelines in the PEs. As an illustration, pipelined PA designs for computing matrix product and transitive closure have been presented. For n-dimensional meshes and other symmetric problems, pipelined PA can be obtained from existing nonpipelined ones using a reindexing transformation. For pipelines with a minimum initiation interval (MII) greater than one, additional constraints that ensure correctness of the synthesized PAs are given.

In the next chapter, the design of an coprocessor with limited hardware resources targeted for uniform dependence algorithms is presented. The general parameter method developed is used at the heart of the mapping process to maximize utilization of PEs and achieve high throughputs for the target application domain.

5. COPROCESSOR DESIGN

The first part of this thesis, Chapters 2, 3, and 4, described a systematic synthesis method for special-purpose systolic architectures designed as hardware implementation of a given algorithm. In mass quantities, the production of such fixed-function arrays is manageable and economical. However, when a systolic array was to be used for a new application, the manufacturer has to take the long and costly process of designing and fabricating the application-specific integrated chip (ASIC). Although the cost and risk of ASIC development has decreased in recent years, budget constraints have motivated a trend away from custom hardware development except in cases where the performance required justifies the cost of developing such specialized hardware. Consequently, general-purpose or versatile systolic architectures are more attractive alternatives.

This chapter discusses the design of a parallel VLSI coprocessor that is programmable for an application domain. The coprocessor interfaces with a front end (host) machine that is responsible for data input and control. The overall requirements on the coprocessor are:

- fixed data bandwidth to main memory for easy integration into existing systems;
- scalability, or the ability to increase performance by adding processor/memory modules without increasing the bandwidth to main memory.

Section 5.1 discusses how these requirements affect the choice of architecture for the coprocessor.

The domain of applications targeted for the coprocessor are the nested loop algorithms given in Section 1.2. Such loops are found frequently in signal and image processing, scientific computations, matrix and linear algebra computations, optimization, digital communications and control. Although an application-specific design for each application would result in higher performance, we have chosen a common retargetable architecture that can be reused for a number of applications. The proposed architecture can be visualized as a coprocessor to workstations or as a VLSI pipeline in supercomputers for loop computations, similar to vector functional units for executing vector instructions.

The goal of this research is to develop a general architecture such that the class of uniform dependence algorithms can be mapped and executed efficiently. The trade-offs we consider include the following.

- For a given performance, reduce clock frequency by increasing area;
- For a given clock rate, increase performance by increasing area.

Section 5.5 discusses the impact of the above trade-offs on the overall system metrics of design cost (improvement in yield), performance (speedup), and power.

There have been numerous efforts to develop general-purpose systolic computers in the past few years. These include Warp and iWarp [24, 25], Matrix-1 [26], SLAPP [27], medium-grain architecture for Image and Signal processing [55, 56], VATA [57], pseudo-systolic linear array [7, 58] and a host of others. However, most of these designs have powerful processors with large local memories, and high-bandwidth data interconnect between processors and host/global memory. Hence, the cost of such systems is very high (in thousands/millions of dollars) due to expensive hardware and complex designs. In our approach, we aim at building a simple, resource-limited VLSI array processor that can be attached to standard single-ported main memory (with fixed bandwidth and long latencies for data access) that results in a system with low cost and acceptable performance for targeted applications. This is in contrast to most existing architectures in which the memory bandwidth is increased proportionally as the system is scaled for higher performance. We believe good performance can be obtained by sophisticated mapping and analysis in software (compiler) that can exploit the hardware effectively for the set of applications. This is in contrast to most existing "class-specific" or general-purpose systolic architectures in which it is the programmer's responsibility to determine which computations are performed by which cells and how the data are moved among the cells and memory.

Recently, there have been efforts to develop a systematic approach to partition and map matrix algorithms on mesh-connected arrays [58, 6, 59]. Our approach differs in trying to extract maximum data re-use under the constraint of a low-bandwidth interconnect to main memory in our array processor. This work also differs from the traditional systolic array mapping/partitioning on fixed processor arrays [1, 60, 61, 62, 63, 64, 65, 66] by assuming only a limited storage in the processor array and by considering the effect of main-memory latency due to low-bandwidth interconnection to main memory. The goals of our design that are different from other research efforts are:

- combination of hardware architecture and software mapping methods (Section 5.3),
- reduction in clock rate for the same throughput (Section 5.4), and
- fixed bandwidth to main memory (Section 5.1).

The remainder of this chapter is organized as follows. We first present the proposed array processor (Section 5.1), describe the mapping and partitioning techniques (Section 5.3), evaluate and discuss results using matrix product and transitive closure as examples (Section 5.4), discuss the impact of clock-rate reduction on overall system metrics (Section 5.5, and conclude with a summary of the chapter (Section 5.6).

5.1 Coprocessor Architecture and Rationale

In this section we describe the architectural features of the coprocessor and the rationale behind the choice of these features.

5.1.1 Coprocessor architecture

The architecture proposed in this research is broadly composed of the following components (Figure 5.1):

- an external main memory (MM) to store input and output data,
- an access-unit (AU) to streamline the flow of data between the processors and main memory,

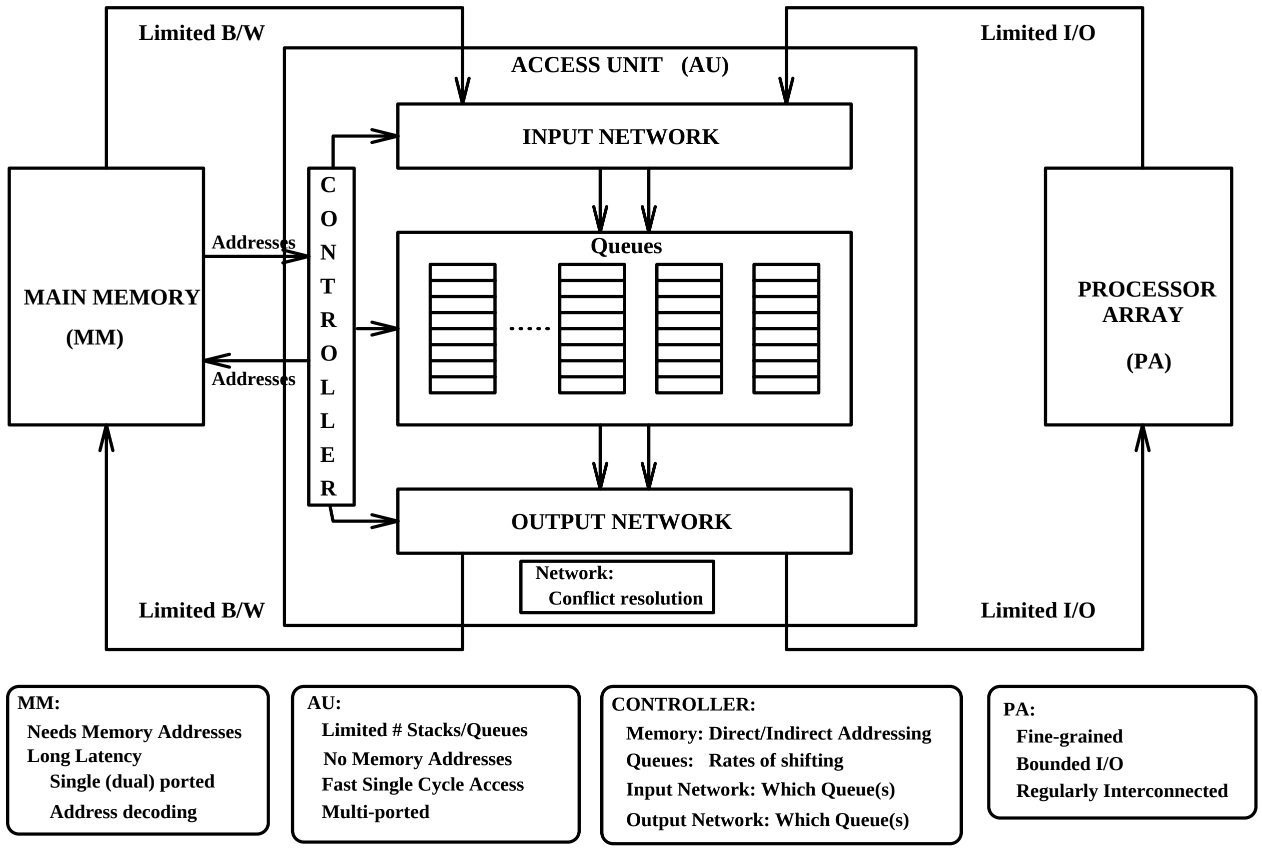

Limited B/W
ACCESS UNIT (AU)
Limited I/O
INPUT NETWORK
C O N T R O L L E R
Queues
MAIN MEMORY (MM)
Addresses
Addresses
PROCESSOR ARRAY (PA)
OUTPUT NETWORK
Network: Conflict resolution
Limited B/W
Limited I/O
MM: Needs Memory Addresses Long Latency Single (dual) ported Address decoding
AU: Limited # Stacks/Queues No Memory Addresses Fast Single Cycle Access Multi-ported
CONTROLLER: Memory: Direct/Indirect Addressing Queues: Rates of shifting Input Network: Which Queue(s) Output Network: Which Queue(s)
PA: Fine-grained Bounded I/O Regularly Interconnected

Figure 5.1: Coprocessor architecture proposed to solve a class of algorithms modeled by uniform recurrences.

- a network that transfers the data between the AU and processors or main memory,
- a dedicated processor array (PA) to execute the computations of a given algorithm.

The remainder of this section describes the individual components in more detail.

Main Memory: (MM) The memory in the architecture is a standard (usually interleaved) memory for storing data involved in the computations. The data in the memory are accessed by supplying a memory address to the address decoder in the memory. The use of memory addresses limits the bandwidth to access the data in the memory (as there are address lines in addition to data lines), and constrains the data to be accessed through a small number of memory ports (usually one or two). Further, the time to access the data is relatively long, because of both the decoding required and limited bandwidth. Hence, a buffer memory called the access-unit is included to store the data accessed from memory and to feed the processor array at a much higher rate. The access-unit is used to recirculate the data back to the processors, and reduce the demand on main memory. An important issue to be resolved is the allocation of chip area between the processors and access-unit.

The assumption of accessing data from standard memory with low bandwidth simplifies the integration of the coprocessor to a conventional host. However, to obtain good performance, this choice complicates the compiler that maps and partitions the computations to mask the memory latency. One of the issues in the compiler is, therefore, how to schedule the computations such that the total amount of data accesses to main memory is bounded.

Access-Unit: (AU) The purpose of the access-unit is to supply data to the processor array and mask the long main-memory latencies. The AU has a fixed amount of storage to buffer the intermediate data that cannot be held in the processor array. The storage in the AU is organized as FIFO queues, and explicit memory addresses (except for queue numbers) are not used. In each cycle, the data present at the head of the queues are sent to the PA or to main memory through the output network (Figure 5.1), and data from the PA or main memory are sent to the tail of the queues. The queues require minimal addressing mechanisms in the AU. In addition to saving memory for storing address bits, the queues enable high rates of data transfer between the AU and the PA. The use of queues further simplifies address decoding as compared to a random access cache memory in existing processors. The area saved from the decoding logic, address bits in instructions, and interconnects can be used to increase the size and/or the number of I/O ports of the AU. The queue structure also permits the size of the AU to be increased without changing the number of address bits, number of ports to main memory, and number of ports to the processor array. Hence, the coprocessor can be scaled to higher performance simply by adding more processors and buffer memory without redesigning other parts of the system.

In addition, the AU can (i) prefetch data from the main memory into its queues to hide the memory latency, (ii) shift the different queues at different rates to reorder the data relative to each other, (iii) perform indirect addressing of main memory, in which a sequence of addresses obtained from the memory is subsequently used to access the data in the main memory. The indirect addressing corresponds to the gather-scatter instructions used to process sparse matrices in a vector supercomputer.

Processor Array: (PA) The processor array obtains data items from the heads of the queues in the AU. The choice of the array configuration is again dictated by the requirement of scalability on the coprocessor. For true hardware scalability, the number of memory ports, number of ports to the PA, and number of address bits should be independent of the size of the architecture and the size of the problem being solved. This implies that the PA should be I/O bounded with a constant number of ports. A possible configuration, then, is a linear array of processors with two boundary processors that communicate with the AU. For instance, if a 2-D square-mesh configuration is chosen, the number of peripheral processors is proportional to the square root of the total number of processors. Hence, as the mesh is scaled to higher sizes, the AU will have to be redesigned to accommodate a higher number of ports to the PA. However, a linear array is more difficult to utilize than a square array because it has only two boundary PEs. We use the General Parameter Method discussed in Chapter 2 at the heart of the mapping process and achieve good performance even for large linear array sizes. Section 5.4.4 shows that the performance degradation is only about 10% for a linear array as compared to a square array. In addition, a linear array with limited I/O is good from the point of view of wafer-scale integration [67].

Each of the PEs is locally connected and can only perform near-neighbor communication. The PA can be clocked at high speeds because of data-driven PEs (i.e., no explicit synchronization) and local connections without long signal delays. The array organization with a small number of boundary (I/O) processors reduces the demand on the AU and simplifies the design of the output network between the AU and PA. For regular, recursive computations, this pipelined processing with bounded I/O is very effective, and the additional complexity/cost of the output network is not justified.

Each PE in the PA has a microprogrammed control that governs its operations. The control specifies the actions to be performed in the PE on receiving data from its neighbors. For example, to solve a matrix product expressed as a 3-D recurrence, the PEs perform an inner-product computation $c = c + a{\cdot}b$, where c, a, b are elements of matrices C, A, and B, respectively. ($C = A{\times}B$).

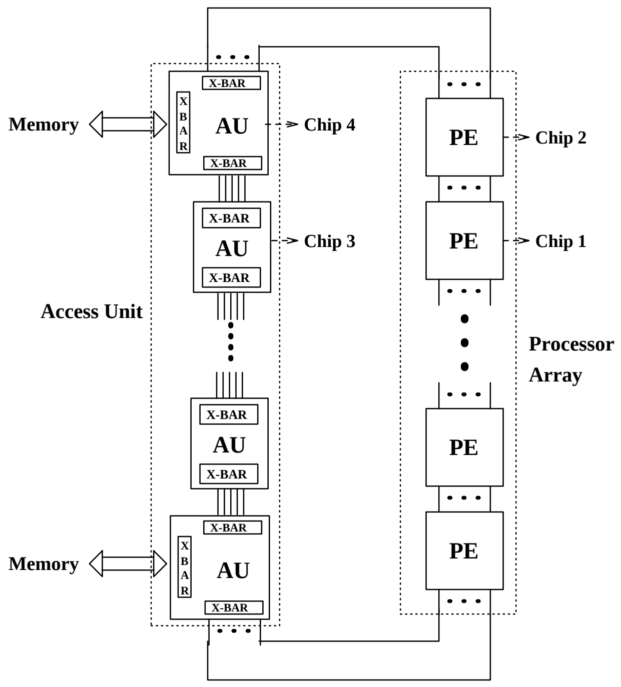


Figure 5.2: Proposed partitioning of the coprocessor architecture into four chip types.

5.1.2 Proposed partitioning of the coprocessor architecture

Figure 5.2 shows the proposed partitioning of the architecture of the coprocessor into four different chip types. The FIFO queues of the access-unit are split over multiple chips as the entire buffer memory required for a large number of PEs cannot be put in a single chip. Hence, we have a linear array of PEs with a linear array of access-unit chips to buffer the data. This layout permits a modular expansion of the system by simply adding more PEs (chip type 1) in the linear array, and more access-unit chips (chip type 3) in the linear buffer chain.

5.1.3 Design approach

Our design procedure for the coprocessor has two major steps: (i) define a model of the architecture in terms of a few parameters (abstract model), (ii) arrive at the final architecture by a trade-off analysis driven by the compiler. The architecture is defined by the following parameters:

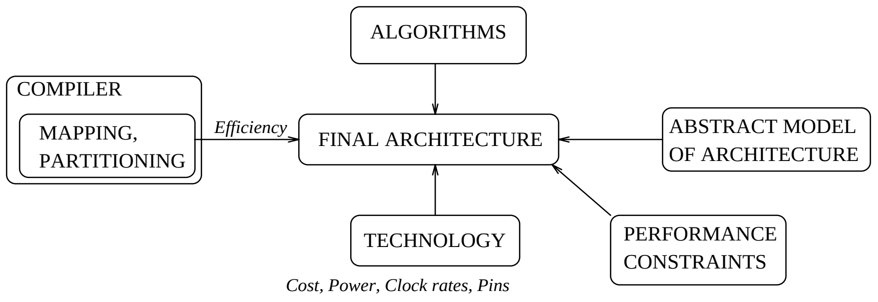


Figure 5.3: Proposed approach for designing the coprocessor architecture.

- Number of PE (#PE),
- Local Memory in each PE (l),
- Bandwidth between PA and AU,
- Interconnection of the PEs (linear, square),
- Size of AU (p),
- Bandwidth between AU and MM (B_{MM}).

The final architecture is obtained after an analysis of the performance versus cost from the point of view of software, algorithms, technology, and other performance constraints (real-time processing, for instance) (see Figure 5.3).

5.1.4 Comparison to existing architectures

In this section, we perform a qualitative comparison of the coprocessor to other existing architectures with limited bandwidth to main memory. The attributes used are:

- Hardware Cost
- Software Cost
- Generality
- Hardware Scalability

Hardware scalability is the ability to increase performance by adding more processing units to the system in a modular fashion. The architectures considered in the comparison are:

- Systolic "General-Purpose" Processors (SGPP). These are programmable general-purpose systolic machines that have been built for a class of applications. Machines in this category include iWARP (in the systolic mode) [25], SLAPP [27], Matrix-1 [26], and medium-grain image processing architectures [56].
- Partitioned Systolic Arrays (PSA). These include a number of research efforts aimed at designing fixed-size systolic arrays [1, 60, 61, 62, 63, 64, 65, 68, 66, 69].
- Systolic Arrays (SA). These refer to traditional, algorithm-specific, problem-size-dependent systolic arrays.
- Coprocessor (CoP). Our proposed coprocessor.

Commercially available shared-memory multiprocessors (SMM) and distributed-memory multicomputers (DMM) do not have a bandwidth limitation to main memory. A SMM is attached through a dedicated interconnection network to a set of memory modules. As a SMM is scaled to higher performance, the number of memory modules and the size of the network must also be increased. Likewise, a DMM with local memory in each processor also has increased memory sizes and bandwidths, as it is scaled to higher system performance.

Figure 5.4 shows the comparison of the different architectures on the chosen attributes. It should be noted that the figure tries to show only the relative ordering between the different architectures and not an absolute evaluation.

In terms of hardware cost (or equivalently, hardware complexity), a SA is the simplest with SGPP being the most expensive. The CoP has about the same amount of hardware complexity as a PSA with an external buffer to recirculate the data.

In terms of software cost or the ease of programming the architecture, a SA is the lowest as there is no programming effort once the hardware is designed. The CoP has the same level of difficulty as a SGPP that accepts high-level sequential programs as input.

For generality, a SA is the most restricted in terms of the application and the problem size it is designed for. A PSA removes the restriction of the application size but is still tied

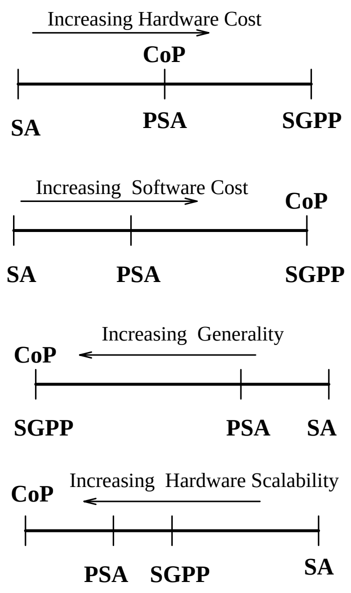


Figure 5.4: Comparing the coprocessor to other architectures with limited memory bandwidth.

to an application. The domain of recursive algorithms targeted for the coprocessor is similar in nature to the domain of a SGPP. Note that although iWARP is a general purpose DMM, we are doing the comparison when it is operating in the systolic mode.

In terms of hardware scalability, the CoP is superior to the other architectures. A SA is the least scalable and a SGPP requires a redesign of the network and controllers as the system is scaled to larger sizes.

To summarize, the coprocessor has low hardware cost, low software cost, reasonable generality, and very good hardware scalability. The software cost is reduced in a SGPP by using complex hardware to achieve high performance. In our approach, we have chosen limited hardware resources, and through our research on mapping and partitioning, have reduced the software cost to achieve good performance.

5.2 Target Algorithms

The application domain of the coprocessor is the set of nested-loop algorithms described in Section 1.2. As indicated there, nested-loop structures have a direct correspondence with recurrence equations, which provide a succinct mathematical representation for them. An n-dimensional recurrence equation is equivalent to n nested loops, where the loop-carried dependencies correspond to the dependencies in the recurrence equation.

Recurrences can be classified as uniform or nonuniform based on the nature of the dependencies [70]. A recurrence equation, $Z(\vec{p}) = \phi[Z(\vec{q_1}), Z(\vec{q_2}), \cdots, Z(\vec{q_r})]$, is called *uniform* if $\vec{q_i} = \vec{p} + \vec{d_i}$, where $\vec{d_i}$ is a constant n-dimensional vector independent of $\vec{p}$ and $\vec{q_i}$. A recurrence equation is called *affine* or *linear* (LRE) if $\vec{q_i} = \mathbf{A_i}\vec{p} + \vec{b_i}$, where $\mathbf{A_i}$ is a constant-coefficient $n \times n$ matrix, and $\vec{b}$ is an n-dimensional vector. A recurrence equation is called *nonlinear* if $\vec{q_i} = \chi(\vec{p})$, where χ is a nonlinear function.

In the remainder of this chapter, the dependence graph of the loop algorithm is used as a graphical tool to discuss the partitioning and mapping procedure. The dependence graph (DG) of an n-nested loop algorithm is defined over an an n-dimensional integer lattice domain, where nodes correspond to the operations inside the nested loops, and arcs correspond to loop-carried dependencies.

In this work, we restrict ourselves to recurrences with uniform dependencies, which involve uniform recurrences, and "uniformized" affine recurrences. Hence, only structural information of the algorithm, i.e., index set and dependence matrix, is needed. Examples 1.3 and 1.4 show the recurrences and nested-loop representations of the matrix-product and transitive-closure algorithms, respectively.

In references [7, 6], Moreno and Lang describe a uniformization or regularization procedure to convert a matrix algorithm to a 3-D cubical mesh called the multimesh graph (MMG). In this approach, a fully parallel data dependence graph of a given matrix algorithm is converted to a regular MMG by performing transformations to remove broadcasting, bidirectional flow of data, and irregular dependencies. Informally, a matrix algorithm is described recursively by an outer loop with a loop body of vector, scalar, and other matrix algorithms.[1] The restriction is that each node of the data-dependence graph can have at most three operands, which allow transformations to regularize the dependence graph to an MMG with a 3-D cube structure. A number of important applications in signal processing

[1]For a formal definition, see Chapter 5 of reference [6].

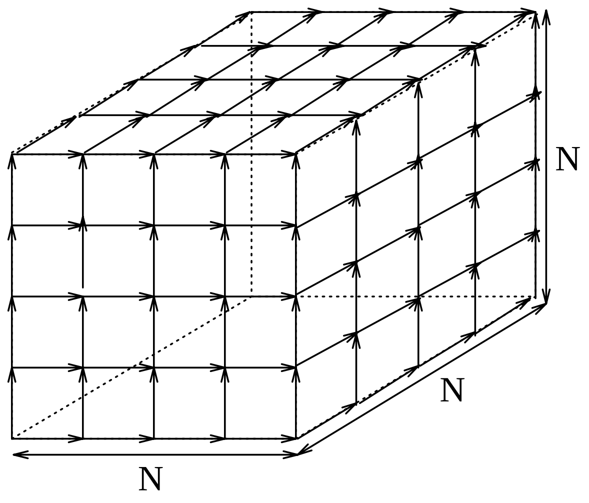


Figure 5.5: Cubical-mesh dependence graph for computing the product of two $N \times N$ matrices.

can be formulated as matrix algorithms and converted to 3-D MMGs. Therefore, the matrix product is a benchmark problem for uniform recurrences from the point of view of MMGs.

For the transitive-closure problem, we use the regularization procedure of Moreno and Lang to obtain a 3-D multimesh graph with a cubical structure. Figure 5.6 (p. 111) shows the 3-D MMG for transitive closure. The darkened nodes in the figure are delay nodes, which are added in the regularization procedure to obtain a cubical structure. Such a 3-D MMG with delay nodes is chosen as it presents a uniform and simple method of executing the MMG. For an algorithm-specific design, the delay nodes would contribute to a significant portion of the execution time; hence, the MMG structure was not used in Chapters 3 and 4. For the coprocessor with a low bandwidth to an external main memory, the regular 3-D MMG structure with additional delay nodes is justified as it simplifies the mapping (using fewer control bits) and the access pattern to the external memory. Hence, in this chapter, we use the 3-D MMG structure of the transitive-closure problem to evaluate the coprocessor.

Example 5.1 Figure 5.5 shows the 3-D dependence graph for the matrix-product algorithm given in Example 1.3. The DG is an $N \times N \times N$ cube with unit vectors along the axes as dependence vectors. ■

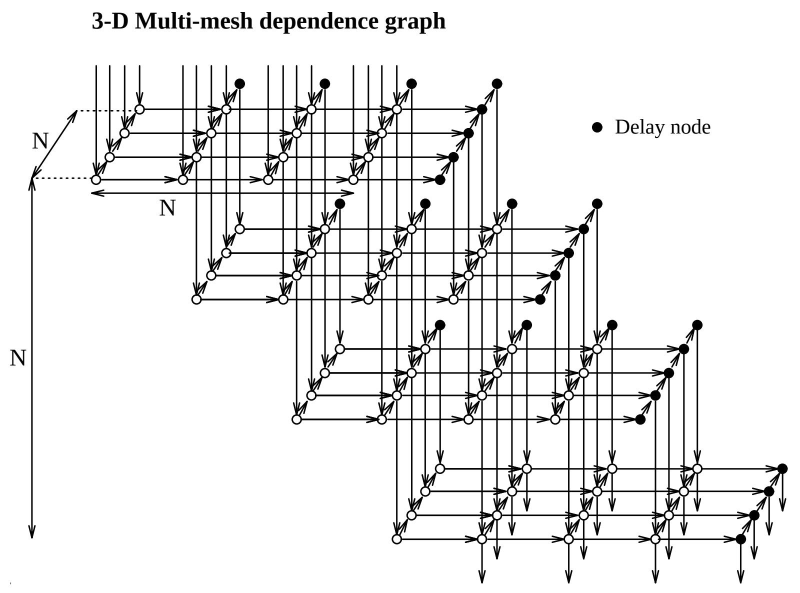


Figure 5.6: Cubical-mesh dependence graph for computing the transitive closure of an $N \times N$ matrix [6, 7]. The figure is drawn for $N = 4$.

Example 5.2 Figure 5.6 shows the cubical-mesh dependence graph [6, 7] for a 4×4 transitive-closure problem shown in Example 1.3. ■

Figure 5.7 shows a pictorial view of the targeted algorithms considered in this chapter. The coprocessor design includes both hardware and compiler design, as good performance is dependent on both the ability of the compiler to exploit existing hardware and the choice of hardware that can be best supported by the compiler.

5.3 Mapping Process

In this section, we describe our method of mapping the high-level loop specification of an algorithm to our target coprocessor, i.e., the compilation procedure from the program to the architecture. The goal of the mapping process is to generate address sequences for the controller in the AU and code sequences that the PEs execute on receiving the data. The mapping process can be broken down into the following five steps:

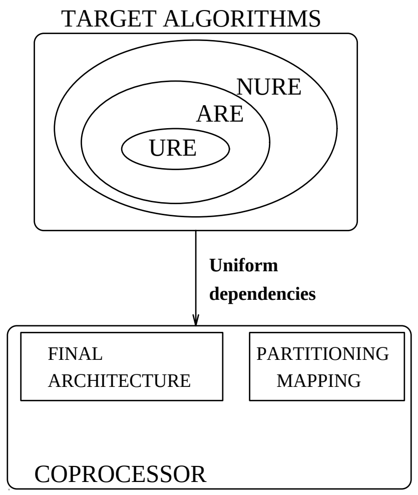


Figure 5.7: Algorithms targeted for the coprocessor. The words URE, ARE, and NURE stand for Uniform Recurrence Equations, Affine Recurrence Equations, and Nonuniform Recurrence Equations, respectively.

1. Partition the DG into blocks that can be executed in a single pass of the data on the PA.
2. Sequence blocks through the AU, i.e., determine which blocks will occupy the AU in each time step.
3. Schedule execution of a single block on the PA.
4. Generate address and code sequences from Steps 3 and 4.

The last step of address and code generation is well-defined once the partitioning, sequencing and scheduling of the DG are known. Hence, in the following description, we assume that the DG is given and proceed with the description of Steps 1,2, and 3 of the mapping process.

5.3.1 Partitioning

The objective in this step of the mapping process is to partition the DG into non-overlapping blocks or chunks that can be processed by the PA in one pass. This step is necessary because the PA has only a limited number of PEs and I/O pins. Hence, the DG is broken down into blocks of maximum size that can executed efficiently in the PA.

Methods to find independent partitions in which the communication between blocks is zero have been proposed before [71, 2]. However, when the original DG has only one connected component (which is the case for the algorithms we consider), independent partitioning results only in one block, i.e., the entire DG. A technique called supernode partitioning has been proposed [72], in which the goal is to partition the nodes that depend on each other and reduce communication between supernodes by propagating results inside the supernode. However, a systematic way to find such partitions is not presented.

In this paper, our approach to partitioning is similar to that in references [1, 73]. For an n-D algorithm, we find n-independent hyperplanes to partition the DG into blocks. Hence, our blocks are "parallelopipeds," and the shape of the blocks can be described by a partitioning matrix $\mathbf{P}$ consisting of n partitioning vectors.

$$\mathbf{P} = [\vec{p}_1\ \vec{p}_2\ \vec{p}_3\ \cdots\ \vec{p}_n]\,.$$

Since we are dealing with uniform dependence algorithms, we restrict the sizes and shapes of all the blocks to be the same, i.e., all of the blocks are identical. This simplifies the address and code generation for PEs. The reason for choosing exactly n partitioning vectors in an n-D domain is that the blocks will not be regular if we have other than n partitioning hyperplanes.

Example 5.3 In Figure 5.8, the index set is a 2-D plane, and the blocks formed by the hyperplanes are normal to the vectors $\vec{p}_1$ and $\vec{p}_2$. Choosing $\vec{p}_1$ or $\vec{p}_2$ alone results in unequal or unbounded blocks. Similarly, choosing $\vec{p}_1$, $\vec{p}_2$ and $\vec{p}_3$ results in unequal blocks. ■

Definition 5.1 Given an n-D dependence graph G of an algorithm and a partitioning matrix $\mathbf{P}$, the block-level dependence graph G_b of G, is a dependence graph in n-D space, where

- nodes in G_b correspond to blocks of G that consists of all of the nodes within the "parallelopiped" defined by n partitioning hyperplanes in $\mathbf{P}$, and
- edges in G_b correspond to dependence vectors crossing the hyperplanes between adjacent blocks in G.

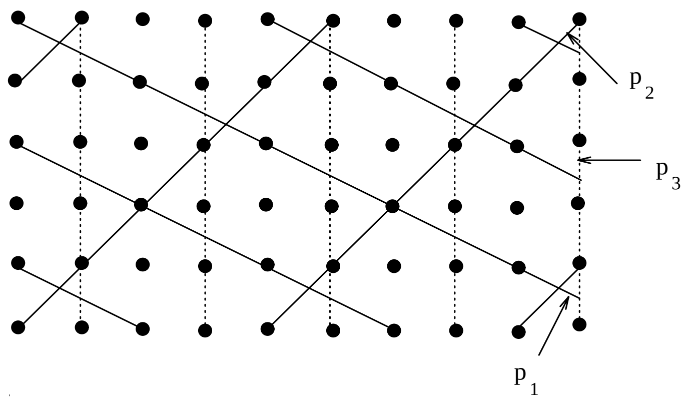


Figure 5.8: Hyperplane partitioning of dependence graphs. The index set is the 2-D plane. The vectors $\vec{p}_1$, $\vec{p}_2$, and $\vec{p}_3$ correspond to a family of regularly spaced hyperplanes normal to them. The vectors determine only the orientation of the hyperplanes, i.e., block shape, and not the spacing between them.

The following lemma presents the conditions for the choice of a valid partitioning matrix **P**.

Lemma 5.1 *The partitioning of a DG by a partitioning matrix* **P** *is valid if and only if* $\mathbf{P}^t\mathbf{D} \geq \vec{0}$ *or* $\mathbf{P}^t\mathbf{D} \leq \vec{0}$, *where* **D** *is the dependency matrix.*

Proof. A given partitioning matrix **P** is valid if the compressed or block-level DG is acyclic. Hence, given a partitioning vector $\vec{p}_i$ all of the dependence vectors $\vec{d}_j$ should cross the hyperplane corresponding to $\vec{p}_i$ in the same direction, i.e., $\vec{p}_i^t d_j \geq 0$ or $\vec{p}_i^t d_j < 0$, $\forall j$. Figure 5.9 shows the case in which $\vec{p}_i$ is not valid (Block 1 depends on Block 2 and vice versa). ■

Next, we present a procedure to choose a good partitioning vector which results in a very small amount of communication between adjacent dependent blocks.

Let $g = rank(\mathbf{D})$, where $\mathbf{D} = \left[\vec{d}_1\vec{d}_2\cdots\vec{d}_k\right]$ is the dependency matrix. Hence, only g of the k dependence vectors are linearly independent. Without loss of generality, assume that the first g columns are linearly independent, and let $\mathbf{D}' = \left[\vec{d}_1\vec{d}_2\cdots\vec{d}_g\right]$ be an $n \times g$ matrix consisting of the g linearly independent vectors of **D**. Let $\mathbf{D}'_i$ be an $n \times g - 1$ matrix derived from $\mathbf{D}'$ by dropping the i-th column vector, i.e., $\mathbf{D}'_i = [d_1 \cdots d_{i-1}d_{i+1}\cdots d_g]$. The number

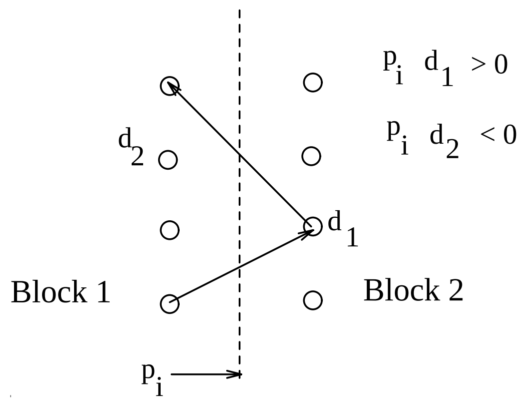


Figure 5.9: A pair of cyclically dependent blocks. Block 2 depends on block 1 through dependence $\vec{d_1}$, while block 1 depends on block 2 through $\vec{d_2}$. Hence, partitioning vector $\vec{p_i}$ is invalid.

of partitioning hyperplanes needed to partition the DG is g (since $g = rank(\mathbf{D})$). Hence, matrix $\mathbf{P} = [\vec{p_1} \cdots \vec{p_g}]$ is chosen such that $\vec{p_i}$ is given by

$$\begin{aligned} \vec{p_i^t}\mathbf{D}_i^{'} &= \overline{0}, \ 1 \le i \le g, \quad \text{and} \\ \vec{p_i^t}\vec{d_i} &> 0. \end{aligned} \tag{5.1}$$

The idea is to choose $\vec{p_i}$ as the basis vector of the left null space of matrix $\mathbf{D}_i^{'}$ and invert the sign of the elements of $\vec{p_i}$ if $\vec{p_i^t}\vec{d_i} < 0$. Hence, by construction, the partitioning matrix $\mathbf{P}$ is feasible.

Corollary 5.1 *If the columns of matrix $\mathbf{D}^{'}$ form a normal basis, i.e., $\vec{d_i^t}\,\vec{d_j} = 0, i \neq j$, then $\mathbf{P} = \mathbf{D}^{'}$ produces a valid partitioning matrix.*

Proof. Consider column $\vec{p_i}$ of partitioning matrix $\mathbf{P} = \mathbf{D}^{'}$, where $\mathbf{D}^{'}$ is an orthonormal matrix. Hence, $\vec{p_i^t}\vec{d_i} = \vec{d_i^t}\vec{d_i} = |\vec{d_i}|^2 > 0$, where $|\vec{d_i}|$ is the magnitude of $\vec{d_i}$. By Lemma 5.1, $\mathbf{P}$ is valid. Also, $\vec{p_i^t}\vec{d_j} = 0, j \neq i$, hence it satisfies Eq. (5.1). ∎

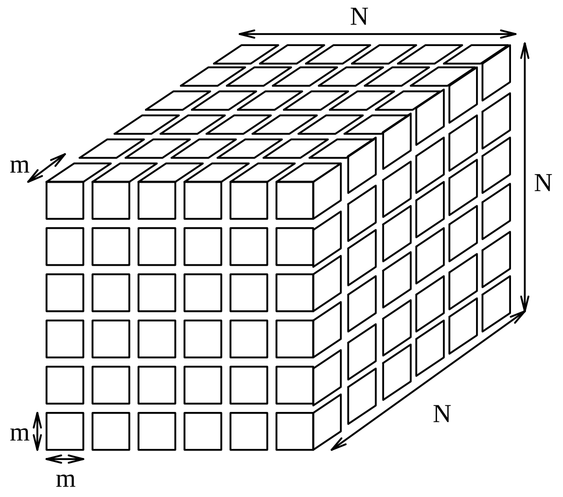


Figure 5.10: Partitioned dependence graph for the matrix-product problem. The partitioning matrix **P** is equal to the dependence matrix **D**.

Example 5.4 For a 3-D matrix product, the dependency matrix is

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.2}$$

According to the procedure above, the partitioning matrix $\mathbf{P} = \mathbf{D}$ is feasible, as $\mathbf{D}^t\mathbf{D} = \mathbf{I}_3 > \overline{0}$ ($\mathbf{I}_3$ is the identity matrix in 3-dimensions). Figure 5.10 shows the partitioning of the DG by the partitioning matrix $\mathbf{P} = \mathbf{D}$. The block-level DG is also a cubical 3-D mesh of size $\frac{N}{m} \times \frac{N}{m} \times \frac{N}{m}$. ■

Example 5.5 For the multimesh dependence graph of the transitive-closure problem, the dependence vectors are $(1,0,0)^t, (0,1,0)^t, (0,0,1)^t$. The partitioning matrix **P** is equal to identity matrix $\mathbf{I}_3$. Figure 5.11 shows the block-level DG of size $\frac{N+m}{m} \times \frac{N+1}{m} \times \frac{N}{m}$ ($N = 4, m = 2$ in the figure) for the transitive-closure problem. Thus, regularization into an MMG provides

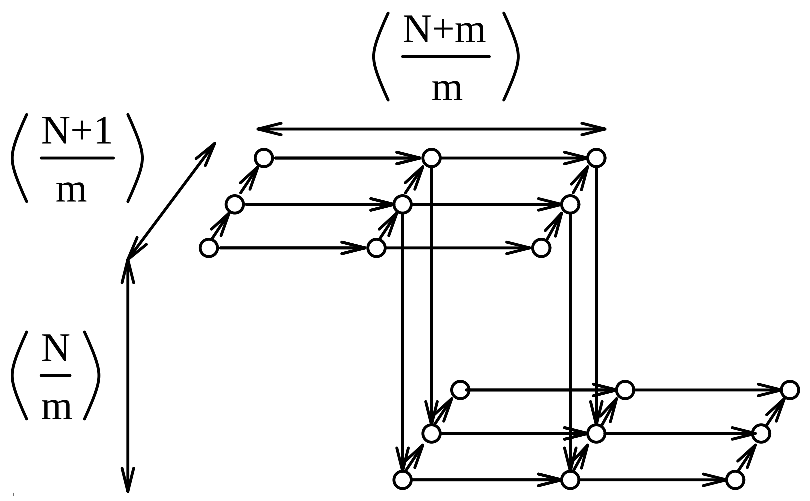


Figure 5.11: Partitioned multimesh dependence graph of the transitive-closure problem. The partitioning matrix is equal to the identity matrix $\mathbf{I}_3$. The figure is drawn for $N = 4, m = 2$.

a uniform way of partitioning the input algorithm. ■

The following lemma shows that the above choice of $\mathbf{P}$ is good in the sense that it reduces the amount of data communication between blocks.

Lemma 5.2 *The choice of partitioning matrix* $\mathbf{P}$ *by the above procedure (Eq. (5.1)) results in the minimum amount of communication between blocks in the partitioned DG.*

Proof. The amount of communication between blocks in the partitioned DG is dependent on the number of dependence vectors crossing the block boundary to adjacent blocks. Since $rank(\mathbf{D}) = rank(\mathbf{P}) = g$, at least one dependence vector must cross each hyperplane $\vec{p_i}$ (else $rank(\mathbf{D}) < g$ if all $\vec{d_i}$ are parallel to p_i). For a partitioning matrix $\mathbf{P}$ obtained using the procedure, exactly one dependence vector d_i crosses the partitioning hyperplane $\vec{p_i}$ (as $\vec{p_i}^t d_i > 0$, and $\vec{p_i}^t \mathbf{D}_i^{'} = \vec{0}$). Other choices of $\vec{p_i}$ will result in equal or increased communication across the hyperplane $\vec{p_i}$. Hence, partitioning with matrix $\mathbf{P}$ results in the minimum total amount of communication among the blocks. The total amount of communication c from a (interior) block for **P** chosen using the procedure is given by

$$c = \sum_{i=1}^{g} \frac{\prod_{i=1}^{g} b_i}{b_i} min(b_i, \vec{p}_i^t \, \vec{d}_i) \qquad (5.3)$$

where b_i is the size of the block along $\vec{p}_i$, i.e., two partitioning hyperplanes perpendicular to $\vec{p}_i$ are placed b_i apart. The value $\vec{p}_i^t \, \vec{d}_i$ is the projection of dependence vector $\vec{d}_i$ along partitioning vector $\vec{p}_i$. Therefore, if $\vec{p}_i^t \, \vec{d}_i \geq b_i$, dependence vectors $\vec{d}_i$ from all of the $\prod_{i=1}^{g} b_i$ index points cross the hyperplane perpendicular to $\vec{p}_i$ resulting in $\prod_{i=1}^{g} b_i$ words of communication along $\vec{d}_i$. ■

The complexity of finding matrix **P** is $\begin{pmatrix} k \\ g \end{pmatrix} O(n.g^2)$ as there are $\begin{pmatrix} k \\ g \end{pmatrix}$ ways of choosing g independent columns of D, and $O(n.g^2)$ is the cost of finding a null-space vector of an $n{\times}g - 1$ matrix.

The size of a block is chosen such that the entire block can be processed by the PA in a single pass. Hence, the size of a block is $b_1 \times b_2 \times \cdots \times b_g$ where b_i, $i = 1, \cdots, g$ is chosen depending on the size of the PA and the local memory per PE.

5.3.2 Sequencing blocks through the access-unit

The execution model of the coprocessor is as follows. Initially, p blocks from main memory (MM) are loaded into AU, and the PA begins executing these p blocks. As the execution proceeds, new blocks are fetched from MM into AU, and some of the existing blocks in AU are written back to MM. As the AU is of limited size, a block of data will have to be fetched multiple times from MM, and the goal in this step is to decide which blocks will be fetched into AU as the execution proceeds. The blocks to reside in AU must be chosen to reduce the traffic between MM and AU; equivalently, the data reuse should be maximized for the blocks in AU. We can think of the AU forming a "storage window" or tile over the block-level DG of the algorithm. The AU stores all of the data needed to compute the blocks in the "storage window," and the output of this phase is to describe how the "storage window" will be moved over block-level DG in a nonoverlapped fashion (else some computations will be redundant).

The sequencing procedure is described below for a block-level DG in the form of a n-dimensional mesh with identity dependence matrix, i.e.,

$$D = I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{5.4}$$

Let the size of the block-level DG be $V = N_1 \times N_2 \times \cdots N_n$, where N_i, $i = 1, \ldots, n$ is the number of nodes along direction $d_i = \left\{\underbrace{0\ldots0}_{i-1}1\underbrace{0\ldots0}_{n-i}\right\}^t$. The "storage window" is an $(n-1)$-D tile in the n-D DG.

The following pseudo-code describes the movement of the "storage window" of p blocks. Without loss of generality, assume that $N_1 \leq N_2 \ldots \leq N_n$ (otherwise the DG can be reindexed).

Procedure 5.1

```
for $i_1 = 1$ to $N_1$ step $\sqrt[n-1]{p}$
  $\ddots$
    for $i_{n-1} = 1$ to $N_{n-1}$ step $\sqrt[n-1]{p}$
      for $i_n = 1$ to $N_n$
        Schedule$(i_1, i_2, \ldots, i_n)$
```

where Schedule$(i_1, i_2, \ldots, i_n)$ schedules all of the p blocks in the $\overbrace{\sqrt[n-1]{p} \times \ldots \times \sqrt[n-1]{p}}^{n-1}$ $(n-1)$-D "storage window" at node $(i_1, i_2, \ldots, i_n)$ of the block-level DG to be brought into the AU. The "storage window" is on a plane perpendicular to vector d_n in the compressed DG, and moves along direction $\vec{d_n}$.

Consider a general URE in an n-D domain, with dependence vectors $\mathbf{D} = [\vec{d}_1, \vec{d}_2, \cdots, \vec{d}_n]$ (if the number of dependencies is larger than n we consider only the n linearly independent ones). Note that a set of n linearly independent dependence vectors $\vec{d}_1, \vec{d}_2, \cdots, \vec{d}_n$ can be converted to a set of n unit vectors (corresponding to an n-D mesh) by a linear transformation or an appropriate basis change. Thus, the UREs are in some sense equivalent to each other from the point of view of dependencies; therefore, the sequencing scheme developed for n-D meshes can be easily extended to other UREs. Consequently, the storage window is an $(n-1)$-D tile formed by the first $(n-1)$ dependence vectors, and the tile is moved along the remaining vector $\vec{d}_n$ in the DG.

The shape of a domain of a given URE is also taken into account in the sequencing scheme as follows. For a URE defined over an arbitrary convex domain, the sequencing procedure traverses the domain by a set of parallel 1-D "lines." The "width" of a line corresponds to an $(n-1)$-D storage tile, and its direction denotes the movement of the $(n-1)$-D tile along the n-th dependence vector. A penalty is incurred each time the storage tile shifts from the tail of a line to the head of another parallel line (the term denoted by "XY plane except (0,0,0)" in Eq. (5.9)). The head, tail and length of these parallel lines are different for different domain shapes. For the n-D mesh, all of the lines are parallel to vector $d_n = (0, \cdots, 0, 1)^t$, and are all of the same length N_n. Thus, the regularization of the given DG into a MMG and the decomposition of the block-level DG into parallel lines present a uniform way of handling general UREs.

Example 5.6 Figure 5.12 shows the storage-window movement for matrix product with $N = 6m$, $p = 9$, where each tile is a square of size 3×3. As the block-level DG is a full 3-D mesh, it is perfectly tiled with 216 nodes covered in 24 tiles of size 9 each. ■

Example 5.7 Figure 5.13 shows the storage-window movement for transitive closure with $N = 4m$, $p = 4$, where each tile is a square of size 2×2. As the block-level DG is not a full 3-D mesh, the 100 nodes in the block-level DG are fitted into 36 tiles of 4 nodes each. ■

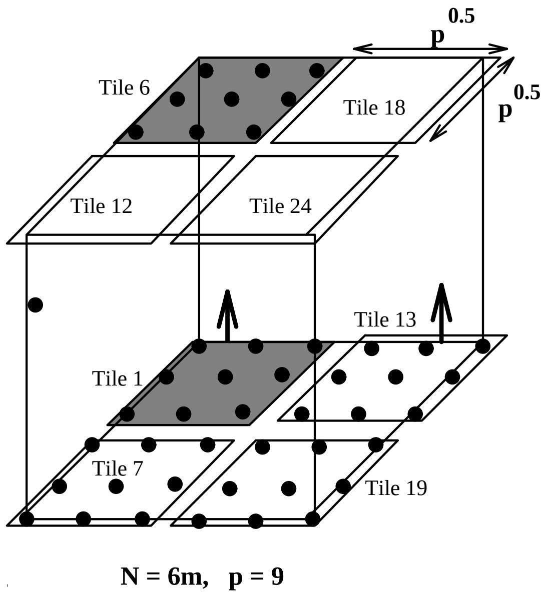


Figure 5.12: Storage-window movement in the block-level DG for the matrix-product problem. The darkened areas show the storage window. For $N = 6m$, $p = 9$, there are 24 tiles.

The size of the storage tile is chosen to minimize the total number of data accesses over the link to MM. Since the size of AU is constant, the number of data accesses is proportional to the perimeter of the tile. It can be easily seen that the shape of the tile should be chosen as an equisided $(n-1)$-D "parallelopiped" to minimize the total number of data accesses to MM.

For the above sequencing scheme with an equisided storage tile, the number of data accesses from the MM (or the I/O complexity denoted by Q) for a block-level DG in the form of an n-D mesh is given by

$$Q = \frac{(n-1)V}{\sqrt[n-1]{p}} + \frac{V}{N_n}. \tag{5.5}$$

Lemma 5.3 establishes that the above sequencing scheme for n-D meshes is asympototically optimal with respect to the number of accesses to the MM.

Lemma 5.3 [74] *For n-dimensional meshes, $Q = \Omega\left(\frac{V}{\sqrt[n-1]{S}}\right)$, where S is the size of the limited memory and Q is the I/O complexity.*

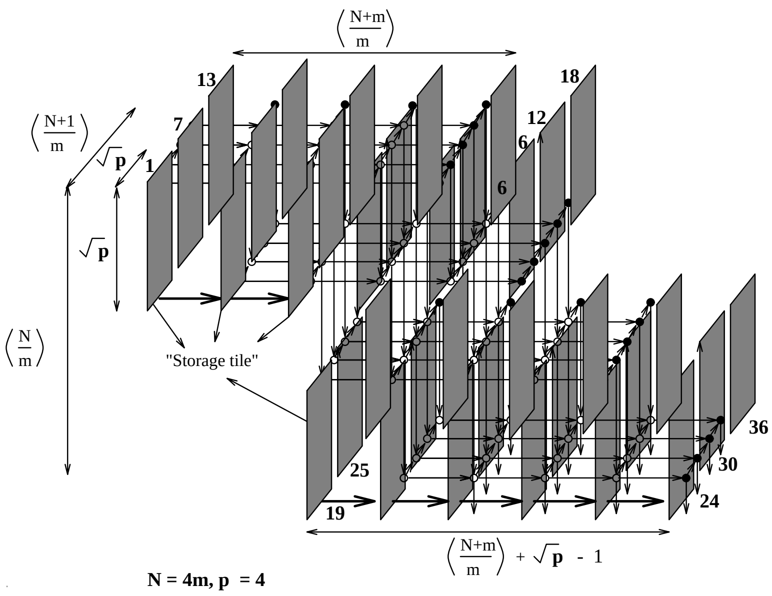


Figure 5.13: Storage-window movement in the block-level DG for the transitive-closure problem. The darkened areas show the storage window. For $N = 4m$, $p = 4$, there are 36 tiles.

In this case, $S = p$, and it is obvious that the number of accesses from MM (Eq. (5.5)) due to the above described scheme (Procedure 5.1) has the same asymptotic complexity as the lower bound given above. Hence, our sequencing scheme is optimal with respect to the number of accesses from MM.

5.3.3 Executing a block in the processor array

In Sections 5.3.1 and 5.3.2, we have discussed techniques to partition the DG and sequence the resulting blocks. In this step of the mapping process, the goal is to map a single block onto a PA so that it can be executed in a single pass.

For this step, the *Generalized Parameter Method* (GPM) developed in Chapter 2 is used to determine the data distributions of the inputs of a block of DG, i.e., which data should be input into the boundary PEs at each time step. The objective used in GPM is to maximize the utilization of the PA. If $\#PE, T_c$, and T_{seq} denote the number of PEs in the processor array, completion time of all of the blocks, and serial time to compute DG, respectively, then

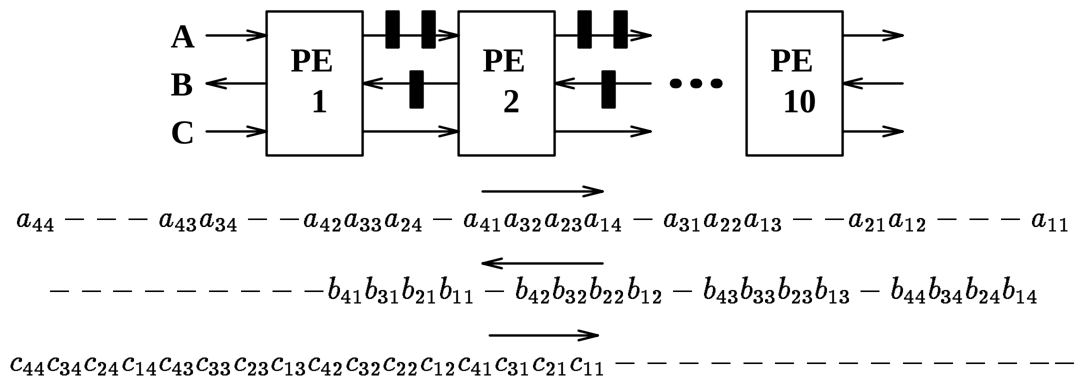

A
B
C
PE 1
PE 2
PE 10
$a_{44} - - - a_{43}a_{34} - - a_{42}a_{33}a_{24} - a_{41}a_{32}a_{23}a_{14} - a_{31}a_{22}a_{13} - - a_{21}a_{12} - - - a_{11}$
$- - - - - - - - - - b_{41}b_{31}b_{21}b_{11} - b_{42}b_{32}b_{22}b_{12} - b_{43}b_{33}b_{23}b_{13} - b_{44}b_{34}b_{24}b_{14}$
$c_{44}c_{34}c_{24}c_{14}c_{43}c_{33}c_{23}c_{13}c_{42}c_{32}c_{22}c_{12}c_{41}c_{31}c_{21}c_{11} - - - - - - - - - - - - - - - -$

Figure 5.14: Data-input patterns for a linear array that computes the product of two 4×4 matrices.

.

the utilization U is given by

$$U = \frac{T_{seq}}{\#PE \times T_c}. \tag{5.6}$$

Hence, for a given algorithm with fixed T_{seq}, maximizing U is equivalent to minimizing the PE-time product $\#PE \times T_c$. This objective tries to reduce the computation time of each block, and increases the overlap between consecutive blocks to reduce the load/drain penalties of the blocks.

Example 5.8 Consider an $m \times m \times m$ block from the 3-D mesh for matrix-product and transitive-closure applications. If the target array is 1-dimensional, Figure 5.14 shows an input distribution that minimizes the $\#PE \times T_c$ product in the array for $m = 4$ (there are 10 PEs in this case). The data distribution of the inputs is as shown in the figure. Similar distributions are possible that minimize other objectives such as completion time and processor-time product. ■

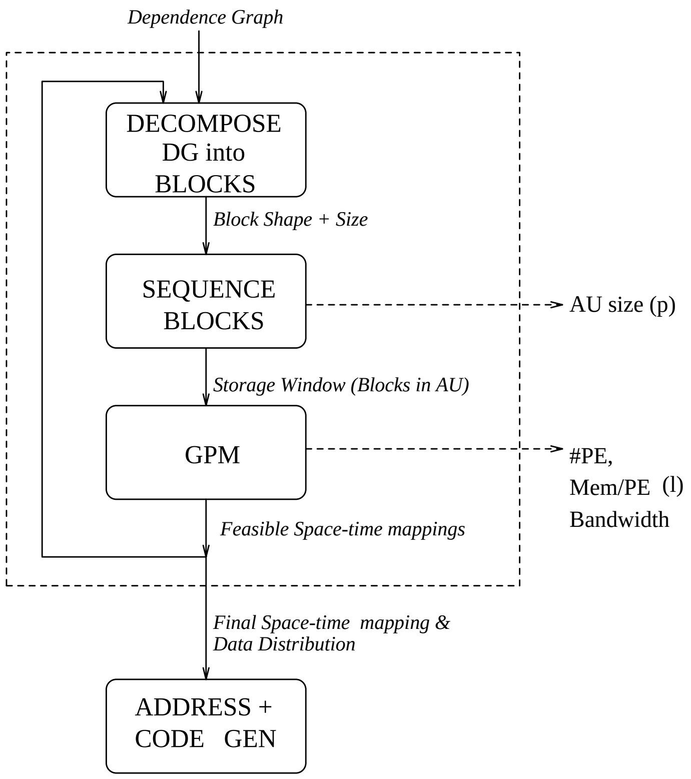

Dependence Graph
DECOMPOSE DG into BLOCKS
Block Shape + Size
SEQUENCE BLOCKS
AU size (p)
Storage Window (Blocks in AU)
GPM
#PE, Mem/PE (l) Bandwidth
Feasible Space-time mappings
Final Space-time mapping & Data Distribution
ADDRESS + CODE GEN

Figure 5.15: Mapping the MMG of an algorithm to the coprocessor architecture.

5.3.4 Overview of mapping process

Having presented a glimpse of the individual steps in earlier subsections, we now present the overview of the entire mapping process from the DG description to the array architecture. Figure 5.15 gives the steps involved in the mapping process. The loop in Figure 5.15 indicates the iteration over the many different ways of partitioning the given DG, sequencing the blocks, and determining a block schedule in GPM.

5.4 Applications: Matrix Product and Transitive Closure

In this section, we present our results in mapping algorithms described by UREs on the coprocessor. Our results are based on the matrix-product and transitive-closure applications.

Let the DG of the 3-D mesh for the matrix product be of size $N \times N \times N$ corresponding to $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are two $N \times N$ matrices. As presented in Section 5.3.1, this DG is partitioned into cubical blocks of size $m \times m \times m$. The AU holds a $\sqrt{p} \times \sqrt{p}$ square tiles of blocks of DG (formed along dependencies $(1,0,0)^t$ and $(0,1,0)^t$), and the movement of the tile is along dependence vector $(0,0,1)^t$. Figures 5.5, 5.10, and 5.12 show the original DG, the block-level DG after partitioning, and the movement of the tile (sequencing blocks through AU) for the matrix product, respectively.

Figures 5.6, 5.11, and 5.13 show the original DG, the block-level DG, and the movement of the storage tile in the block-level DG for transitive closure, respectively. The darkened nodes are delay nodes added to regularize the original DG to a cubical structure. The 3-D MMG for transitive closure is also partitioned into cubical blocks of size $m \times m \times m$ for 1-pass execution on PA. The size of the block-level DG after partitioning is $\frac{N+1}{m} \times \frac{N+m}{m} \times \frac{N}{m}$. Note that block-level DG is identical to the original DG except for its dimensions. The "storage window" is a $\sqrt{p} \times \sqrt{p}$ square set of blocks and is moved along the $(0,1,0)^t$ direction as shown in Figure 5.13 in order to minimize the total number of data accesses.

These examples illustrate the benefit of deriving MMG from the nested-loop algorithm, as it provides for a uniform way of partitioning the MMG, sequencing the blocks of the MMG, and designing the PE array to execute a block of the MMG.

5.4.1 Evaluation metrics

Since one of our goals is to develop a cost-effective design, the fundamental metric used is the performance of the targeted algorithms on a given amount of silicon chip area. Hence, we need an estimate of the area consumed and the completion time of the design in terms of the abstract model parameters.

Area Model

The total area occupied by the array architecture is the sum of the areas of the processors, AU, controller, and the input and output networks in the AU.

$$Area = Area_{proc} + Area_{AU} + Area_{pins} + Area_{controller} + Area_{network}. \qquad (5.7)$$

The dominant terms in the total area are the first two terms: $Area_{proc}$ and $Area_{AU}$. The other 3 terms are lower-order terms, and their contribution is neglected in our simplified model. For example, for linear-array configurations, the number of ports on PA and AU are constants, and, therefore, the area contributions due to the pins (proportional to the number of ports) and the network (proportional to the square of the number of ports) are constant and small. The area of the controller (size of the control memory) is dependent on the mapping process and proportional to the regularity in data-access patterns from MM. For UREs, the access pattern and address generation will be regular and simple, and the size of the control memory will be small. Therefore, the area index, containing the dominant terms, is given by

$$AreaIndex = \underbrace{\#PE(A_{PE} + l)}_{Area_{PA}} + \underbrace{3\sqrt{p}m^2 + pm^2}_{Area_{AU}} \tag{5.8}$$

where A_{PE} is the area of a single PE in memory words, which captures the implementation cost of a PE, and l reflects the local memory per PE. Since, AU holds $\sqrt{p} \times \sqrt{p}$ blocks of the DG forming a square in the 3-D mesh (Figure 5.12), storage is needed in the AU for (i) pm^2 elements of C, (ii) $\sqrt{p}m^2$ elements of A, (iii) $\sqrt{p}m^2$ elements of B, and (iv) additional $\sqrt{p}m^2$ words for the next set of $\sqrt{p} \times \sqrt{p}$ blocks brought in from MM into AU for future processing.

Model of Completion Time

The total completion T_{compl} in PE-cycles is given by

$$\begin{aligned} T_{compl} &= max\left(\frac{2\sqrt{p}m^2}{B_{MM}}, p\ t_{block}\right)\frac{Num\_blocks(N,m)}{p} \\ &+ \underbrace{\frac{2pm^2}{B_{MM}}\left(\frac{Num\_XY\_blocks(N,m)}{p} - 1\right)}_{XY\ plane\ except\ block(0,0,0)} \\ &+ \underbrace{\frac{pm^2 + 2\sqrt{p}m^2}{B_{MM}}}_{block(0,0,0)} + \underbrace{max\left(\frac{pm^2}{B_{MM}} - (p-1)\ t_{block}, 0\right)}_{block(N,N,N)} \end{aligned} \tag{5.9}$$

where B_{MM} is the bandwidth between MM-AU in elements per PA clock tick, t_{block} is the nonoverlapped time taken to execute one block of DG by PA,[2] $Num\_blocks(N, m)$ is the number of blocks of the DG or the number of nodes in the block-level DG, and $Num\_XY\_blocks(N, m)$ is the number of nodes in the projection of the block-level DG to the XY plane parallel to the storage tile of AU, i.e., $Num\_XY\_blocks(N, m)$ is the number of times the storage tile has to change direction.

The first term in Eq. (5.9) is the dominant term and is the product of the time taken for each window ($\sqrt{p} \times \sqrt{p}$) in AU and the number of windows over DG. For each window of size $\sqrt{p} \times \sqrt{p}$, $\frac{2\sqrt{p}m^2}{B_{MM}}$ is the time it takes to fetch the elements needed for the next window, and $p\ t_{block}$ is the time to process p blocks in the current window. The second term in the equation models the additional time required whenever the window changes direction which involves writing and reading pm^2 elements of the output matrix. The third term is the initial latency to load the data corresponding to $\sqrt{p} \times \sqrt{p}$ blocks in the storage window. For a $\sqrt{p} \times \sqrt{p}$ storage window, there are pm^2 elements of the result matrix (corresponding to the area of the tile) and $2\sqrt{p}m^2$ elements of the input matrix (corresponding to half of the perimeter). The final term is the additional time over the time for the final storage tile ($p\,t_{block}$) to write back the results from AU to MM. Note that there are pm^2 elements of the output matrix to be written back to MM, and t_{block} is the earliest time after the start of the last storage window when the first m^2 elements are available for write-back.

Example 5.9 For the matrix-product problem, as shown in Figure 5.10,

$$\begin{aligned} Num\_blocks(N, m) &= \frac{N}{m} \times \frac{N}{m} \times \frac{N}{m} \\ Num\_XY\_blocks(N, m) &= \frac{N}{m} \times \frac{N}{m}. \end{aligned}$$

[2]For any block of the DG, the processing time includes the load, drain, and computation times of the block on the PA. In GPM, the block scheduling is done to overlap consecutive blocks entering the array, thereby reducing the effective load, drain (and maybe the computation time) of a block. The nonoverlapped time for a block refers to those portions of the load, drain and computation times that are not masked by successive blocks.

For the transitive-closure problem with 3-D MMG, as shown in Figure 5.13,

$$\begin{aligned} Num\_blocks(N,m) &= \frac{N+1}{m} \times \frac{N+m}{m} \times \frac{N}{m} \\ Num\_XY\_blocks(N,m) &= \frac{N+1}{m} \times \frac{N}{m}. \end{aligned}$$

■

Size of AU

In order to have efficient processing, AU must be large enough to mask the MM latency fully, i.e., completely overlap the loading of the inputs of the next storage window with the processing of the current one. Therefore,

$$\begin{aligned} \frac{2\sqrt{p}m^2}{B_{MM}} &\leq p\,t_{block} \\ p &\geq \left(\frac{2m^2}{B_{MM}t_{block}}\right)^2. \end{aligned} \tag{5.10}$$

Example 5.10 Let $N = 512$, $m = 8$ and $B_{MM} = 1/5$ (5 cycles per word). The number of processors, $\#PE = m^2 = 64$.

- For a square array, $t_{block} = m = 8$ and the AU size is about $400K$ words or $12.8M$ bits assuming 4 bytes/word.
- For a linear array with $t_{block} = m^2 = 64$ and the AU size is about $8K$ words or $256K$ bits.

Although $12M$ bits of fast memory in the AU is not possible with the current technology, $256K$ bits is very feasible. For a linear array, as $t_{block} = O(m^2)$, p is independent of the block size m. This is true because for a linear PA used for block processing, its I/O bandwidth is constant, independent of the number of PEs and block size, and the size of AU in blocks (p) depends only on the bandwidth to MM. ■

Clock-rate Reduction

A useful way of looking at the coprocessor is as follows. Suppose that we increase the area by a certain factor, what is the reduction in clock rate in order to obtain the same throughput (or completion time)? Therefore,

$$\begin{aligned} T_{compl} \times \frac{1}{Clock_{PA}} &= T_{ref} \times \frac{1}{Clock_{ref}} \\ \frac{Clock_{ref}}{Clock_{PA}} &= \frac{T_{ref}}{T_{compl}} \qquad (5.11) \\ & \qquad (5.12) \end{aligned}$$

where T_{ref} and $Clock_{ref}$ are the completion time and clock rate of the the reference design, respectively. The reference design is chosen as the basic coprocessor design with one PE and the appropriate amount of AU memory to mask the MM latency. Thus, both the reference and current designs have the same bandwidth limitation. The $AreaIndex$ of the reference design in memory words can be obtained from Eq. (5.8) and Eq. (5.10) when $m = 1$, $t_{block} = 1$, and $\#PE = 1$. It is given by

$$AreaIndex_{ref} = A_{pe} + \frac{4}{B_{MM}^2} + \frac{6}{B_{MM}} \qquad (5.13)$$

where A_{pe} is the area of a PE in memory words and B_{MM} is the bandwidth to MM. Section-s 5.4.4 and 5.4.5 present the cost-performance trade-offs of the coprocessor where the cost is measured as the Area Index (Eq. (5.8)), and the performance is measured as the reduction in clock rate over the reference design (Eq. (5.11)). Note that Eq. (5.11) can be interpreted as the speedup over the reference design (with 1 PE) for equal clock rates.

A reduced clock speed is desirable for several reasons. First, and most important, the chip yield would be significantly higher if it were designed for a lower clock rate. For instance, the yield would improve significantly if the chip is designed for a 1-MHz instead of a 50-MHz clock rate. Also, power dissipation is lower at lower clock rates, which leads to better integration and lower packaging costs. Hence, reduced clock rates will lower the system cost. This is intuitively in agreement with existing high-speed processors where there is a steep (possible exponential) variation in cost (and design time) with clock speed.

5.4.2 Relationship between clock-rate reduction and area index

If the size of PA is increased by a factor θ, the block size m will have to be increased by θ for single-pass execution of each block by PA. For a linear array, $t_{block} = O(m^2)$, and the number of blocks in AU (p) is independent of block size m (Eq. (5.10)). Therefore, the area of AU for a linear array increases by θ^2 when m is increased by θ (Eq. (5.8)). The total area ($AreaIndex$) of the coprocessor, which is dominated by the area of AU, increases by a factor θ^2 when m is increased by θ. Hence, $AreaIndex$ grows as the square of the number of PEs in the linear array. The following argument shows that the size of AU has to grow at least as the square of the number of PEs, in order to mask MM latency when processing a cubical block of DG on a linear PA.

Consider a cubical $m \times m \times m$ block of DG to be processed in a linear PA. For a linear PA has constant I/O bandwidth, the time to process a block is $\Omega(m^2)$, as there are $O(m^2)$ input and output elements to be loaded into PA. Therefore, the size of AU has to be $\Omega(m^2)$ as all the $O(m^2)$ elements needed to process a block have to be held in the AU to mask the MM latency. The number of PEs in PA is $O(m)$, as there are m^3 operations to be completed in $O(m^2)$ time. Therefore, to mask MM latency, the size of AU (given by $\Omega(m^2)$) grows at least as the square of the number of PEs (given by $O(m)$).

The completion time T_{compl} (and, hence, the clock-rate reduction) can, at best, increase by θ when the number of PEs is increased by θ (superlinear speedups are not possible for deterministic processing). The area index grows at least as θ^2 when the number of PEs is increased by θ. Thus, clock-rate reduction can grow at best as the square root of $AreaIndex$ when the effect of MM latency is masked completely. However, beyond a certain number of PEs (or area index) for a fixed problem size, the completion time is bounded by the fixed bandwidth to MM, and is equal to the time to read and write the elements of input and output matrices. Hence, clock-rate reduction will flatten out beyond a certain area ratio (A_{crit}) when MM becomes a bottleneck. Thus,

$$\text{Clock-rate reduction } (CRR) = \begin{cases} \frac{Clock_{ref}}{Clock_{PA}} = O(\sqrt{AreaIndex}), & AreaIndex \leq A_{crit} \\ \frac{Clock_{ref}}{Clock_{PA}} = \frac{Volume(N)}{B_{MM}}, & \text{otherwise} \end{cases}$$

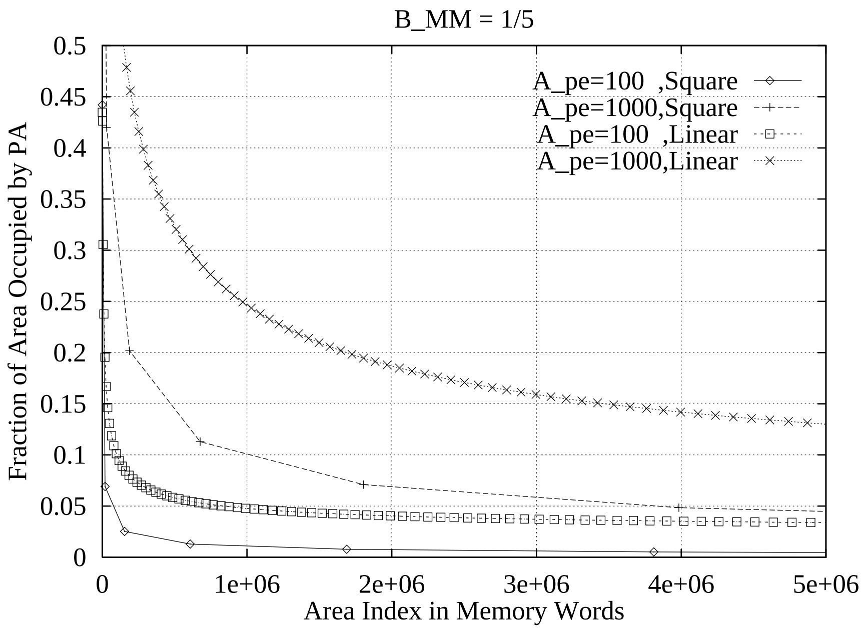


Figure 5.16: Area allocation between AU and PA of the coprocessor for masking MM latency fully. The area of the reference design is equal to 230 words for $A_{pe} = 100$ and 1130 words for $A_{pe} = 1000$ (Eq. (5.13)).

where $Volume(N)$ is the total amount of data to be accessed (including both read and write data) to process the given n-dimensional uniform dependence algorithm. Thus, for $AreaIndex \geq A_{crit}$, $CRR_{max} = \frac{Volume(N)}{B_{MM}}$.

5.4.3 Area allocation between AU and PA

Figure 5.16 shows the variation of the fraction of total area occupied by the PA as we increase the total silicon area of the coprocessor. The size of AU in Eq. (5.8) is computed using p from Eq. (5.10), which is chosen to overlap the memory fetches completely with computations. The x-axis is the area of the chip in memory words. Thus, $AreaIndex$ of 5 megawords corresponds to an area equivalent to 160 megabits of storage assuming each word is 32 bits. The cost of a PE in memory words is denoted as A_{pe}. Thus, $A_{pe} = 100$ means that each PE occupies an area equivalent to 100 words of memory. Hence, for 5000 K words of total area and $A_{pe} = 100$, only 4% of the total area is occupied by PEs for a linear-array configuration with $A_{pe} = 100$. This shows that most of the chip area is taken up by the AU if we design the chip with the optimal balance where memory latency is fully masked. Moreover, for the same total area, a linear PA has more of its area devoted to PEs. The effect of increased area of a PE is to lift the entire plot upwards both for linear and square PAs. Although this figure is for $B_{MM} = 1/5$ (5 cycles to access a word from memory), the same effect is observed for other bandwidths.

5.4.4 Cost-performance trade-offs: matrix product

Figure 5.17 shows the cost-performance trade-offs of the coprocessor on the matrix-product application. Performance is measured as the reduction in clock rate, and the cost is measured as $AreaIndex$. The system is designed at the balance point to mask memory latency fully (Eq. (5.10)). Trade-offs are shown for problem sizes of $N = 1,024$ and $N = 10,240$ when the latency to access a word from MM is 5 cycles. The area cost of a PE in memory words is denoted as A_{pe}. It captures the effect of technology on PE implementations. The amount of local memory in each PE is controlled by parameter l. In our approach, for a given block size, we obtain a virtual array (linear or square), and cluster the virtual PEs to obtain increased local memory per PE and reduced number of PEs. Hence, l virtual PEs are clustered together, and the larger the value of l, the lower the number of PEs and the lower the clock-rate reduction. For $l = 1$, each physical (and virtual PE) has 3 words of storage, one for $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$. The x-axis is again the area index in memory words.

Figure 5.17 shows that for 5 megawords of silicon area, we can reduce the clock rate by a factor of about 175 for a square array and about 160 for a linear array. Therefore, for a $10,240 \times 10,240$ matrix product, if we clock the array at 1 MHz, we will obtain a performance equivalent to that of a reference design (with 1 PE) running at 160 MHz (6.25 ns). For a 66 MHz clock rate (15 ns), we can clock the coprocessor at 400 KHz for the same performance (of about 160 MFLOPS), or run it at 1 MHz and obtain a 2.5 times speedup. The final speed can be chosen from a variety of alternatives depending on the objective of the design. Figure 5.17 shows the square-root relationship between clock-rate reduction and area index (Section 5.4.2). For instance, when the area is increased from 1 megawords to 4 megawords, the clock-rate reduction only doubles from 75 to about 150 for a linear array.

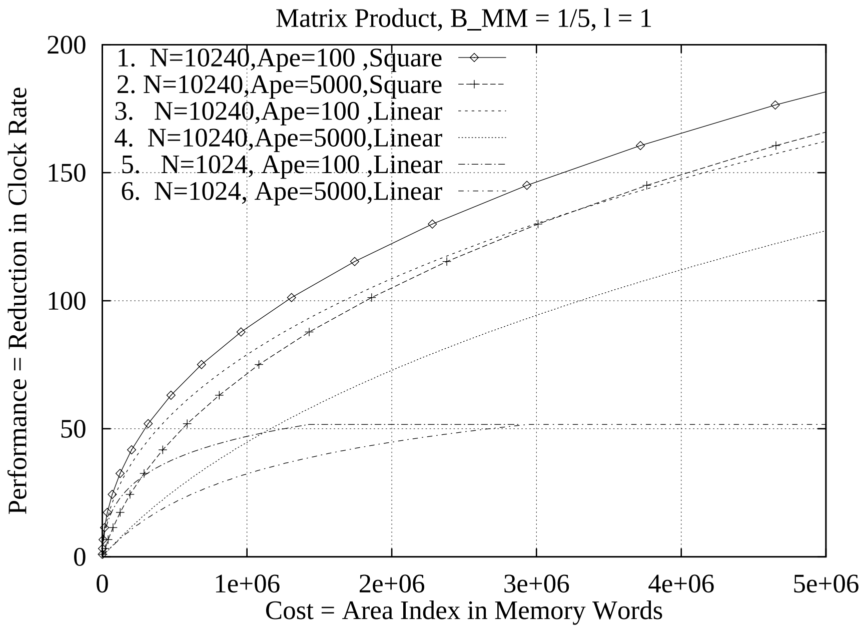


Figure 5.17: Cost-performance trade-offs of the coprocessor on the matrix-product problem. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = CRR^3_{max} = CRR^4_{max} = 512.5$, $CRR^5_{max} = CRR^6_{max} = 51.7$. The values of critical $AreaIndex$ in megawords beyond which clock-rate reduction saturates are: $A^1_{crit} = 105.9$, $A^2_{crit} = 111.4$, $A^3_{crit} = 140$, $A^4_{crit} = 155$, $A^5_{crit} = 1.44$, $A^6_{crit} = 2.97$.

The degradation in performance in using a linear array rather than a square array is about 8% for $A_{pe} = 100$ and 20% for $A_{pe} = 5000$.

Figure 5.17 also shows the cost-performance curves for a smaller problem size $N = 1024$. The performance of the coprocessor saturates because of the bottleneck of limited bandwidth to MM. The completion time of the coprocessor is equal to the time to access the input matrices (**A**, **B**) and the output matrix **C** from MM. Hence, the maximum clock-rate reduction is given by $\frac{Clock_{ref}}{Clock_{PA}} = \frac{T_{ref}}{T_{compl}} = \frac{N^3}{\frac{4N^2}{B_{MM}}} = 51.2$ for $N = 1024$. This maximum value of 51 is for $AreaIndex \geq 1.4$ megawords ($A_{crit} = 1.4$ megawords).

Therefore, a linear PA is an attractive choice for the matrix-product problem and other UREs. Its advantages are its constant I/O bandwidth and modularly expandable layout. The linear PA achieves good performance because of the ability of the mapping algorithm to exploit locality in the high-dimensional loops effectively.

Figure 5.18 depicts the reduction in clock rate for linear and square arrays with $l = 8$. Thus, 8 virtual PEs are coalesced into a physical PE leading to an 8-fold increase in local memory of a PE. As the figure shows, while the corresponding curves for square PAs remain unchanged from Figure 5.17, the curves for a linear array are lower than the corresponding ones in Figure 5.17. Figure 5.18 shows that there is an 100-fold reduction in clock speeds possible (37.5% lower than when $l = 1$) as shown in the figure. Again, the clock-rate reduction grows as the square root of the area index when MM latency is masked.

Figure 5.19 shows the effect of increasing B_{MM} between MM and AU on the cost-performance trade-offs. As shown in the figure, the MM bandwidth is a key factor that influences the performance of the coprocessor. For instance, when *AreaIndex* is 5 megawords, the performance doubles from 150 to 300 when B_{MM} is increased from 1/5 (5 cycles/word) to 1/2 (2 cycles/word). High bandwidths are possible because of the lower frequency at which the coprocessor can be clocked for a given performance level.

Figure 5.20 shows the sensitivity of the performance of the coprocessor to variations in the size of AU. In Figure 5.20, the size of AU is scaled by a factor of ALPHA from the optimal value given in Eq. (5.10). If $ALPHA < 1$, the PEs will be idle between the time the current set of p-blocks is completed to the start of the next set of p blocks. Therefore, the performance measured as clock-rate reduction will decrease as ALPHA is decreased.

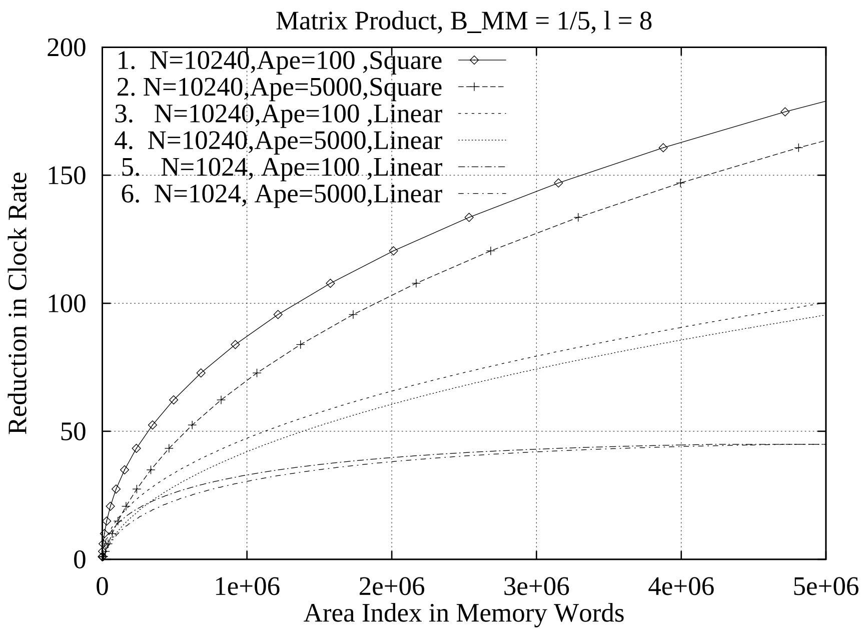


Figure 5.18: Effect of local memory per PE (l) on cost-performance trade-offs for the matrix-product problem. Local memory in the PE is used to simulate a set of virtual PEs reducing the PE-count. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = 512.5$, $CRR^3_{max} = CRR^4_{max} = 445.6$, $CRR^5_{max} = CRR^6_{max} = 44.9$. The values of critical $AreaIndex$ in mega-words are: $A^1_{crit} = 107.7, A^2_{crit} = 112.8, A^3_{crit} = 428.05, A^4_{crit} = 433.1, A^5_{crit} = 4.93, A^6_{crit} = 5.53$.

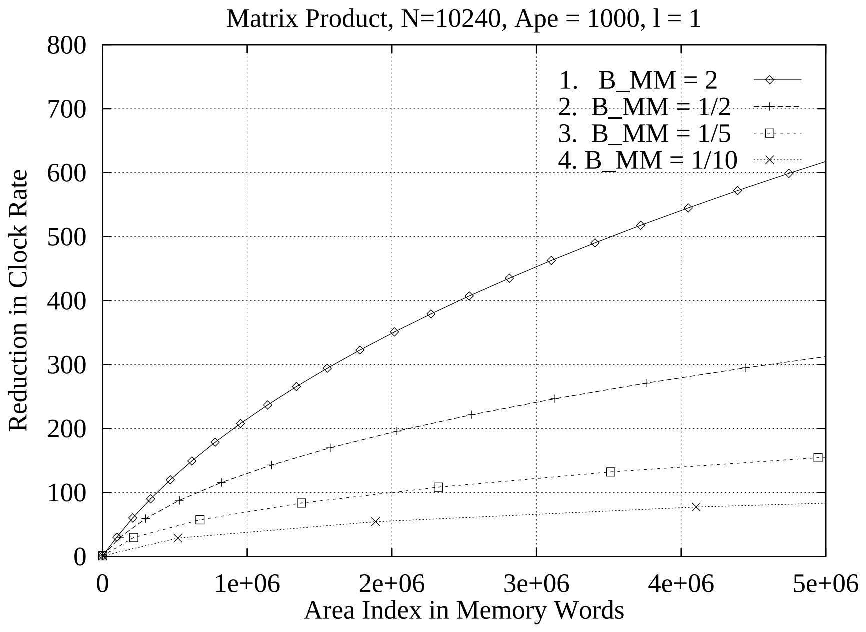


Figure 5.19: Effect of bandwidth (B_{MM}) on cost-performance trade-offs for the matrix-product problem. The values of maximum clock-rate reduction for plots $2, 3, 4$ are: $CRR^2_{max} = 1280.5$, $CRR^3_{max} = 512.5$, $CRR^4_{max} = 256.5$. The values of critical $AreaIndex$ in megawords are: $A^2_{crit} = 211.6$, $A^3_{crit} = 142.8$, $A^4_{crit} = 123$. Plot 1 does not saturate as the bandwidth to MM is higher than the data rate required by PA.

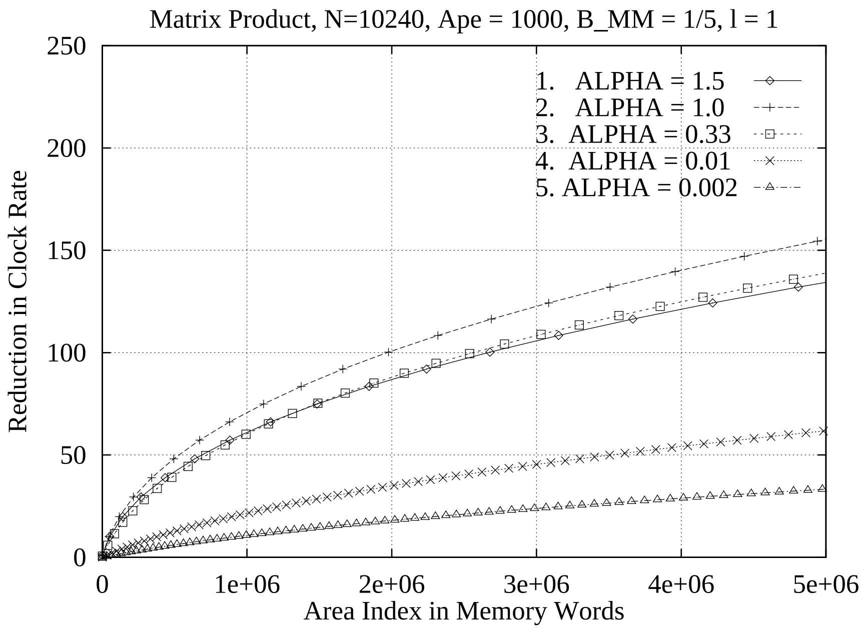


Figure 5.20: Sensitivity of clock-rate reduction of coprocessor to changes in AU size for the matrix-product problem. If ALPHA < 1, the PEs are idle due to insufficient data accessed from MM. If ALPHA > 1, chip area is wasted by the extra memory in AU. The maximum clock-rate reduction for all curves is 512.5. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = A^2_{crit} = 142.8, A^3_{crit} = 175.1, A^4_{crit} = 770.7, A^5_{crit} = 2463.8$.

Figure 5.20 shows that for a 500-fold decrease in AU size (ALPHA = 0.002), the reduction in clock rate drops by a factor of 4. For example, if the area index is 5 megawords, the clock-rate reduction drops from 160 to 40 as ALPHA is moved from 1.0 (optimal value) to 0.002. Hence, the linear-array design shows good resilience to change in the size of AU from its desired optimal value. The low sensitivity to AU size can be used to obtain significant area savings without sacrificing performance. For instance, when $ALPHA = 0.33$, the AU is one-third smaller than when $ALPHA = 1.0$, resulting in a 32% area savings (from Figure 5.16 and 5.20) for only a 7% decrease in performance. Hence, the designer can fine tune the final design to maximize the area-performance trade-off.

5.4.5 Cost-performance trade-offs: transitive closure

Figures 5.21, 5.22, 5.23, and 5.24 show the cost-performance trade-offs of the coprocessor on transitive-closure problems described by 3-D MMGs. The x-axis is the cost or area index in memory words, and the y-axis is the performance measured as clock-rate reduction. Due to the more irregular nature of the transitive-closure algorithm, the performance of the coprocessor on transitive closures is lower than that on matrix products.

For instance, from Figure 5.21, for 5 megawords of silicon area and a $10,240 \times 10,240$ problem, we can reduce the clock rate by a factor of about 135 (compared to 160 for a matrix product) for a linear array to have the same execution time as a reference 1-PE system. For $N = 1024$, the maximum clock-rate reduction is equal to 34.8 and $A_{crit} = 1.45$ megawords. Again, the effect of local memory, $l = 8$, (Figure 5.22) is to scale the performance down to about 80 for $N = 10240$. Figure 5.23 again shows that B_{MM} is important for good performance and shows the improvement due to increased bandwidths. Finally, Figure 5.24 establishes that the cost-performance trade-offs are not very sensitive to the size of the AU required for masking memory latency fully. For instance, when $ALPHA = 0.33$ in Figure 5.24, the AU is one-third smaller than when $ALPHA = 1.0$, resulting in a 28% area savings (from Figure 5.16 and 5.24) for a 15% decrease in performance. The cost-performance trade-offs are more sensitive for the transitive-closure problem than for the matrix-product problem due to irregularities in the algorithm.

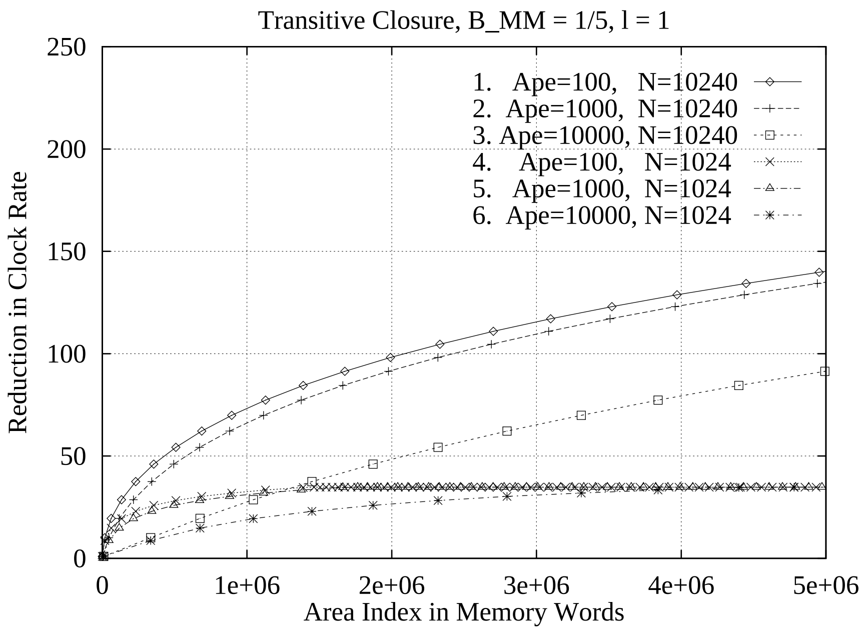


Figure 5.21: Cost-performance trade-offs for the transitive-closure problem. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = CRR^3_{max} = 342$, $CRR^4_{max} = CRR^5_{max} = CRR^6_{max} = 34.8$. The values of critical $AreaIndex$ in megawords beyond which clock-rate reduction saturates are: $A^1_{crit} = 140, A^2_{crit} = 144, A^3_{crit} = 170.6, A^4_{crit} = 1.44, A^5_{crit} = 1.73, A^6_{crit} = 4.6.$

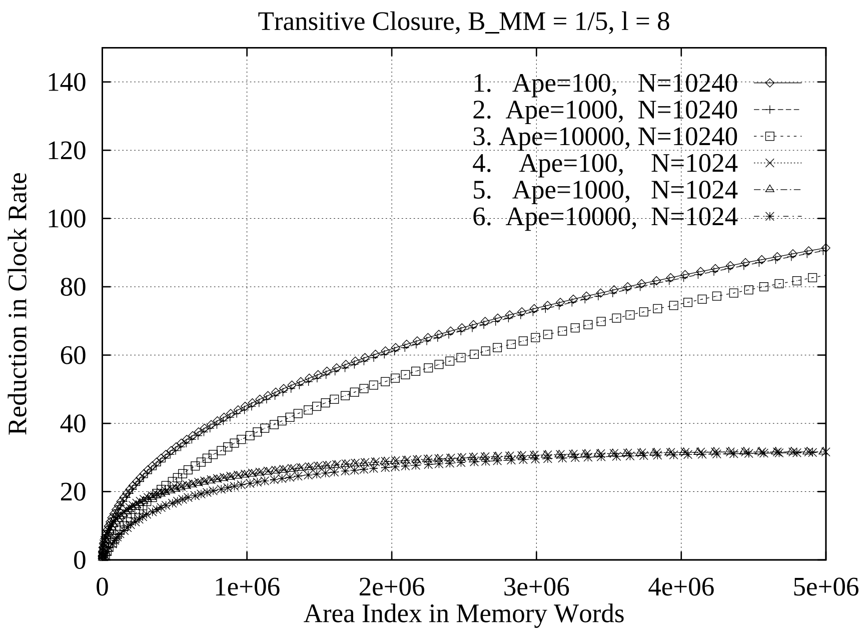


Figure 5.22: Effect of local memory per PE (l) on cost-performance trade-offs for the transitive-closure problem. The values of maximum clock-rate reduction are: $CRR^1_{max} = CRR^2_{max} = CRR^3_{max} = 310.9$, $CRR^4_{max} = CRR^5_{max} = CRR^6_{max} = 31.6$. The values of critical $AreaIndex$ in megawords are: $A^1_{crit} = 426.1$, $A^2_{crit} = 427$, $A^3_{crit} = 436.3$, $A^4_{crit} = 4.23$, $A^5_{crit} = 4.33$, $A^6_{crit} = 5.26$.

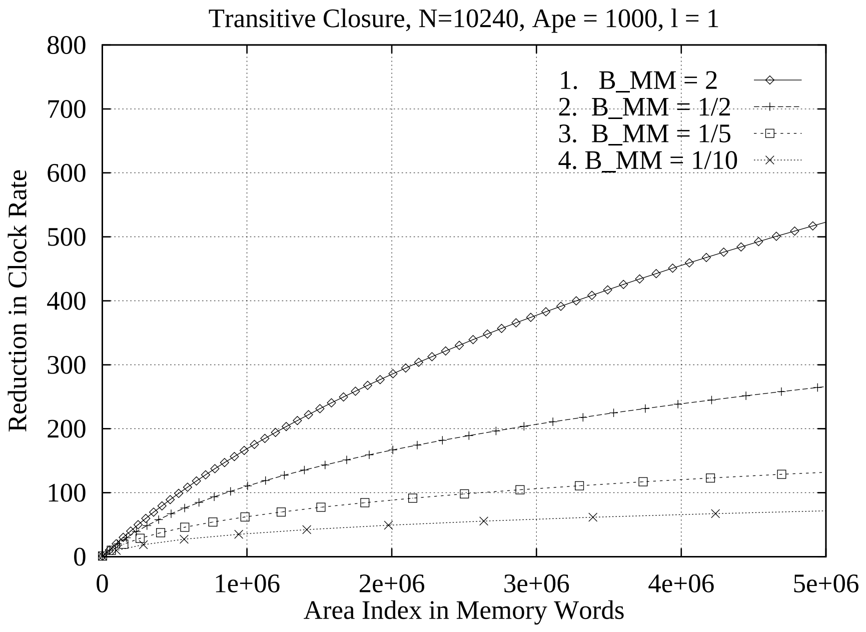


Figure 5.23: Effect of bandwidth (B_{MM}) on cost-performance trade-offs for the transitive-closure problem. The values of maximum clock-rate reduction for plots $2, 3, 4$ are: $CRR^2_{max} = 854$, $CRR^3_{max} = 342$, $CRR^4_{max} = 171.3$. The values of critical $AreaIndex$ in megawords are: $A^2_{crit} = 215.4$, $A^3_{crit} = 144$, $A^4_{crit} = 123$. Plot 1 does not saturate as the bandwidth to MM is higher than the data rate required by PA.

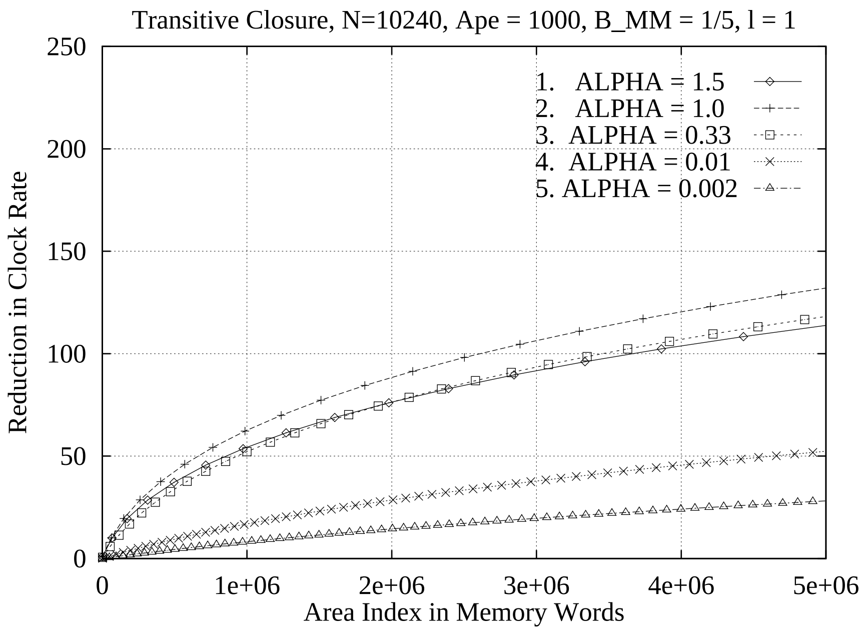


Figure 5.24: Sensitivity of cost-performance trade-offs to changes in AU size for the transitive-closure problem. The maximum clock-rate reduction for all curves is 342. The values of critical $AreaIndex$ in megawords are: $A^1_{crit} = A^2_{crit} = 144$, $A^3_{crit} = 177.8$, $A^4_{crit} = 780$, $A^5_{crit} = 2498.3$.

5.5 Discussion

The main contribution of Section 5.4 is the square-root relationship between the clock-rate reduction and the area of the coprocessor under fixed MM bandwidth. Two applications of matrix product and transitive closure have been used to evaluate the coprocessor, and the effect of the different parameters of the abstract model on the square-root relationship has been presented in Sections 5.4.4 and 5.4.5. In this section, we present the impact of the square-root relationship on the overall system metrics such as yield (considered to be indicative of the cost), performance (throughput), and power dissipation.

5.5.1 Yield

The most important consequence of reducing the clock rate is the improvement on the yield of manufacturing the chips. Although chip yield would improve as the clock frequency is reduced, we need larger silicon area to obtain lower clock frequencies. Hence, the number of chips in the system will be higher as the clock rate is reduced. As an illustration, we compute the rate at which the chip yield should drop with frequency in order to improve the net system yield.

Let y denote the chip yield at some frequency f, and let $\#C$ be the number of chips used in the system. We compute the expected number of chips that have to be tested to find $\#C$ working chips. We assume that chips on a wafer are independent of each other and assume them to be drawn with replacement from a large set, until the required number of working chips are found. Hence, the expected number of draws (chips to be tested), $E(1)$, until one working chip is obtained is given by

$$E(1) = \sum_{i=1}^{\infty} i\,(1-y)^{i}\,y = \frac{1}{y}, \tag{5.14}$$

where y is the chip yield. Therefore, the expected number of chips to find $\#C$ working chips is given as

$$E(\#C) = \#C\,E(1) = \frac{\#C}{y} \propto \frac{AreaIndex}{y}, \tag{5.15}$$

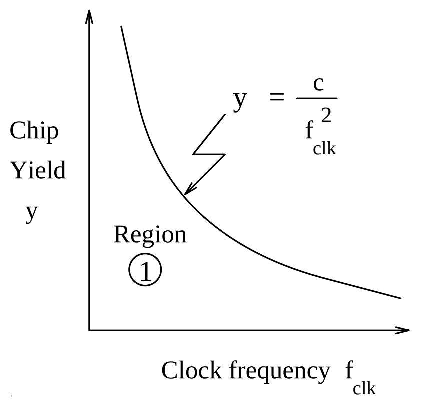


Figure 5.25: Chip yield y versus clock frequency f_{clk}; $y \geq \frac{c}{f_{clk}}$ (Region 1) to reduce the expected number of chips tested (Eq. (5.16)).

as the number of chips used is directly proportional to the *AreaIndex* of the coprocessor. The highest-order term in *AreaIndex* varies as the inverse-square of the clock frequency (f_{clk}), i.e., $AreaIndex \propto \frac{1}{f_{clk}^2}$ (since clock-rate reduction is proportional to the square root of *AreaIndex*). Hence, the chip yield y has to drop at least as the inverse square of the clock frequency in order to reduce the expected number of chips $E(\#C)$ in Eq. (5.15). Therefore, for some constant c,

$$y(f_{clk}) \geq \frac{c}{f_{clk}^2} \text{ for } E(\#C) \text{ to decrease.} \tag{5.16}$$

Hence, there would be a net reduction in the number of chips tested (improved system yield) if chip yield drops at least as the inverse square of the clock frequency (Figure 5.25). Therefore, as the area of the coprocessor is increased, the expected number of chips to be tested would decrease, thus lowering the design cost.

5.5.2 Speedup

The metric of clock-rate reduction in Eq. (5.11) can be interpreted as the speedup over the reference design of 1 PE for equal clock rates. Thus, clock-rate reduction can be thought of as the improvement in throughput for a given clock rate, or as the reduction in clock frequency for a given throughput. Hence, speedup over the reference design increases as the square root of area index for a fixed clock rate. It should be noted that speedup is linear in the number of PEs since $AreaIndex$ is proportional to the square of the number of PEs (Section 5.4.2).

5.5.3 Power

There are three major sources of power dissipation in digital CMOS circuits, which are summarized in the following equation [75] :

$$P = p_t(C_L.V.V_{dd}.f_{clk}) + I_{sc}.V_{dd} + I_{leakage}.V_{dd}. \tag{5.17}$$

The first term represents the switching component of power, where C_L is the loading capacitance, f_{clk} is the clock frequency, V_{dd} is the supply voltage, V is the voltage swing (usually equal to supply voltage V_{dd}), and p_t is the activity factor, i.e., the probability that a power-consuming transition occurs. The second term is due to direct-path, short-circuit currents I_{sc}, and the final term is due to leakage currents $I_{leakage}$. The dominant term in a "well-designed" circuit is the switching component. Therefore, as the clock frequency f_{clk} is reduced, P decreases in a linear fashion, and as the area of the coprocessor is increased, P decreases as the square root of the area.

The total power dissipated by the coprocessor P_{total} is proportional to $P \times AreaIndex$, where P is given by Eq. (5.17) and $AreaIndex$ is the area of the coprocessor given by Eq. (5.8). As the area is increased, the total power P_{total} *increases* as the square root of the area (since $P \propto \frac{1}{AreaIndex}$).

$$P_{total} = P \times AreaIndex \propto \sqrt{AreaIndex} \text{ (as } P \propto \frac{1}{\sqrt{AreaIndex}}). \tag{5.18}$$

Hence, total power dissipated increases in a square-root fashion as the area is increased. However, the power density (measured as power per unit area) decreases as $AreaIndex$ is increased. This could simplify the design of the heat sinks and potentially reduce the cost of a chip.

For low-power designs, the popular approach is to reduce power P (Eq. (5.17)) by decreasing the supply voltage V_{dd}. For a given performance, the supply voltage cannot be lowered as it leads to slower operation of the circuit. Hence, multiple PEs can be used to lower the clock rate, so that the longer clock period can accommodate the slower operation of the circuit at lower supply voltages. Lowering the clock rate with a linear array of PEs allows power P (Eq. (5.17)) to be reduced by reducing the supply voltage while maintaining the required performance. The linear array organization, in addition, does not increase the loading capacitance, thereby allowing P (Eq. (5.17)) to be reduced from both a lower clock frequency and a lower supply voltage for fixed performance.

5.6 Summary

This chapter describes an approach to designing a coprocessor for executing loop computations described by uniform dependence algorithms. Our results show that a modularly expandable linear array of PEs with constant main-memory bandwidth achieves high performance and is a good choice for the processor array. We achieve this good performance because our General Parameter Method can map the high-dimensional loops on the processor array efficiently.

An important result we have obtained is the square-root relationship between clock-rate reduction and area of the coprocessor under fixed main-memory bandwidth. Two applications of matrix product and transitive closure have been used to evaluate the coprocessor and to study the effect of the different parameters on the square-root relationship. Section 5.5 presented the impact of the square-root relationship on system yield, speedup, and power dissipation. We have found that the system yield improves with the area of the coprocessor when chip yield decreases at least as the inverse square of clock frequency.

6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This thesis has addressed a number of issues in the design of fine-grained processor arrays for regular, recursive, uniform dependence algorithms. Such algorithms are commonly found in a number of applications in digital signal processing, image, speech, text processing, matrix algorithms, scientific computing, digital communications, pattern matching, graph theory and others. The contributions made in this thesis are as follows:

General Parameter Method (GPM). A major contribution of this thesis is a systematic parameter-based approach to design optimal, lower-dimensional processor arrays for algorithms with uniform dependencies. Chapter 2 discusses the parameter definitions and the constraints developed for valid space-time mapping of the algorithms to processor arrays. An efficient polynomial-time search procedure to find the optimal array parameters for any user-specified nonlinear and nonmonotonic objective expressed in terms of the parameters is given. Existing synthesis techniques such as the dependency method (DM) have exponential complexity, in the worst case, in finding optimal designs.

Chapter 2 also presents a set of equivalencies by which a designer familiar with the dependence-based methods can utilize the efficiency of the GPM to obtain optimal array designs. This establishes and extends the importance of the general parameter method which can then be used in a "black-box" fashion in the dependence methods.

Chapter 3 presents detailed results of applying GPM to two important benchmark problems: finding the transitive closure of a matrix (Algebraic Path Problems) and finding the product of two matrices (3-D cube graph algorithms). Optimal array designs for minimizing computation time, completion time (with load and drain times), and processor count are given. The formulation of the design as a constrained optimization problem allows a designer to incorporate additional real-life constraints into the design process. Continuous processor-time trade-offs which arise when there are bounds on either processors or time or both are also presented. Using these trade-offs, a designer can arrive at a design that best meets his or her design objectives.

In Chapter 4, the extension of GPM to synthesize two-level pipelined processor arrays with pipelines of processors, and pipelined functional units in each processor is described. Compared to existing approaches that can retime only an existing design, the extended GPM can explore the search space systematically for alternative designs and find the optimal one. For algorithms whose data-dependence graphs are n-D meshes, a closed form solution for obtaining a two-level pipelined design from a nonpipelined one is given. Also, results of applying the extended GPM to matrix product and transitive closure are presented.

Coprocessor Design In the second part of the thesis (Chapter 5), the design of a scalable VLSI coprocessor for the class of regular, recursive, uniform dependence algorithms is dicussed. This work complements the GPM by focusing on issues of algorithm partitioning and block sequencing for fixed-size arrays and a fixed amount of buffer memory, instead of mapping to an algorithm-specific size-dependent processor array. An architecture with a linear array of processors and a linear array of access-unit chips connected to a standard, slow memory is proposed. The access-unit chips in turn contain a set of queues or shift registers to buffer the data between the processors and main memory. Such an architecture is useful from the view of wafer-scale integration (fixed number of I/O ports) and can be easily scaled for higher performance by simply adding extra processors and memory chips. In addition, a linear array architecture is suitable for low-power designs as described in Section 5.5.3.

We also present the main steps in the design of the compiler to map high-level language descriptions to the coprocessor architecture. There are three steps in the compilation process:

block partitioning (as the number of PEs is fixed), block sequencing (as the size of access-unit is fixed), and block scheduling using the General Parameter Method. The hyperplane partitioning method proposed in Section 5.3.1 minimizes the amount of communication between adjacent parallelopiped blocks. The block sequencing method minimizes the number of data accesses to main memory. The GPM is used in the block scheduling onto the linear array to maximize the utilization.

Results on matrix-product and transitive-closure applications indicate that the coprocessor can achieve an order of magnitude improvement in clock-rate or throughput over a reference 1-PE design. An important result obtained is the square-root relationship between clock-rate reduction and area of the coprocessor under fixed main-memory bandwidth. From the square-root relationship, it can found that the system yield improves with the area of the coprocessor when chip yield decreases as the inverse square of the clock frequency. The proposed coprocessor can be used as a back-end accelerator in desktop workstations or as a VLSI pipeline in supercomputer architectures for loop algorithms.

6.2 Future Work

In this section, some possible avenues of future research are indicated for an aspiring researcher.

This thesis has tried to promote application-specific computing as a promising approach for high-performance designs. Such special-purpose systems can now be inexpensively built using custom designs or commercially available field programmable gate arrays (FPGA). Moreover, recent advances in design and simulation tools have decreased the cost and risk of ASIC development. An important research area would be to extend the class of algorithms that can be executed on the coprocessor to nonlinear recurrences. Currently, there exist good techniques to transform affine recurrences into equivalent uniform ones. A number of important problems in optimization and NP-hard problems can be formulated as nonlinear recurrences or loops with nonuniform dependencies. Hence, a scheme to systematically regularize nonlinear dependencies would allow a number of synthesis techniques to be applied to them.

It is well-known that processors are an order of magnitude faster than memories and will probably continue to be that way in future. Hence, it is important to develop schemes to amortize the cost of memory accesses over many operations. The idea of access-unit used in this thesis, i.e., a set of FIFO queues organized as a stream buffer as opposed to a cache, can be exploited to improve the memory system performance for certain access patterns found commonly in many applications.

Power efficiency is going to be a critical issue for the success of future portable information systems. Voltage scaling and clock rate reduction are two approaches to designing low-power systems. Voltage scaling has to be dealt with at the circuit-design level, while clock rate reduction without performance loss can be achieved by exploiting parallelism. A possible research area could be to apply and combine such system-level and circuit-level techniques to design low-power systems into a unified framework, and develop a CAD tool that considers power consumption as one of the metrics at the algorithmic, architectural, and logic levels. This would allow the designer to trade-off power in a systematic manner with other metrics such as cost and performance.

There are a number of commercially available distributed memory machines with thousands of processors such as Intel Paragon, CM-5, Ncube, and Cray MPP. However, programming these machines for high-performance seems to be a challenging and arduous task. There are a number of research efforts aimed at developing compilers that can automatically extract the parallelism from sequential programs and map them on to these machines. But the key is to improve the efficiency of mapping these scientific programs onto such distributed memory machines. In this regard, the idea of linear transformations of dependence graphs at the heart of many array synthesis techniques can be adapted to develop efficient data alignment, data partitioning, and computation partitioning techniques for a compiler for distributed memory machines starting from a specification written in a language such as High Performance Fortran (HPF).

Microprocessors with superscalar or VLIW hardware have started appearing in commercial workstations. The key to the performance of these machines would be compiler methods to exploit the multiple functional units (by issuing multiple instructions in each cycle), and effective cache management to hide the long memory latencies. There exist compiler optimization techniques such as, software pipelining to utilize the multiple functional units, and loop blocking (or tiling) to improve cache performance. However, methods for software pipelining are mainly local techniques that exploit parallelism only across consecutive loop iterations. For certain common classes of applications, more global iteration-level transformations can be developed to detect independent iterations in the dependence graph and issue them to the multiple functional units. In this thesis, some of these techniques have been developed in the context of VLSI array architectures, and extending these methods to a compiler for the superscalar/VLIW architectures would be a promising research direction.

REFERENCES

[1] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–12, Jan. 1986.

[2] W. Shang and J. A. B. Fortes, "On mapping of uniform dependence algorithms into lower dimensional processor arrays," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 350–363, May 1992.

[3] P.-Z. Lee and Z. M. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Transactions on Computers*, vol. C-37, pp. 1578–1597, Dec. 1988.

[4] H. T. Kung and M. Lam, "Wafer-scale integration and two-level pipelined implementations of systolic arrays," *Journal of Parallel and Distributed Computing*, no. 1, pp. 32–63, 1984.

[5] M. Valero-Garcia, J. J. Navarro, J. M. Llaberia, and M. Valero, "Systematic hardware adaptation of systolic algorithms," *International Symposium on Computer Architecture*, pp. 96–104, 1989.

[6] J. H. Moreno, "Matrix computations on mesh arrays," Ph.D. dissertation, University of California, Los Angeles, June 1989.

[7] J. H. Moreno and T. Lang, "Matrix computations on systolic-type meshes: An introduction to multi-mesh graph (MMG) method," *IEEE Computer*, vol. 23, p. 23, Apr. 1990.

[8] H. T. Kung and C. Leicerson, "Systolic arrays (for VLSI)," *Sparse Matrix Proceedings, Society for Industrial and Applied Mathematics*, pp. 256–282, 1978.

[9] H. T. Kung, "Why systolic architectures ?," *IEEE Computer*, vol. 15, pp. 37–46, Jan. 1982.

[10] D. Heller, "Partitioning big matrices for small systolic arrays," in *VLSI and Modern Signal Processing*, S. Y. Kung, H. J. Whitehouse, and T. Kailath, Eds., Englewood Cliffs, NJ: Prentice Hall Inc., 1985, pp. 185–199.

[11] K. T. Johnson, A. R. Hurson, and B. Shirazi, "General purpose systolic arrays," *IEEE Computer*, pp. 20–31, Nov. 1993.

[12] J. A. B. Fortes and B. W. Wah, "Systolic arrays: From concept to implementation," *IEEE Computer*, vol. 20, pp. 12–17, July 1987.

[13] P. Quinton and Y. Robert, *Systolic Algorithms and Architectures.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[14] Z. Chen and W. Shang, "On uniformization of affine dependence algorithms," *Proceedings Fourth Symposium on Parallel and Distributed Processing*, pp. 128–137, Dec. 1992.

[15] P. Quinton and V. V. Dongen, "The mapping of linear recurrence equations on regular arrays," *Journal of VLSI Signal Processing*, vol. 1, pp. 95–113, Kluwer Academic Publishers, 1989.

[16] T. Tzen and L. Ni, "Data dependence analysis and uniformization of doubly nested loops," *Proceedings International Conference on Parallel Processing*, vol. II, pp. 91–99, Pennsylvania State University Press, 1992.

[17] Y. Wong and J. M. Delosme, "Broadcast removal in systolic arrays," *Proceedings International Conference on Systolic Arrays*, pp. 403–412, ACM/IEEE, 1988.

[18] Y. Yaccoby and R. Cappello, "Scheduling a system of affine recurrence equations onto a systolic array," *Proceedings International Conference on Systolic Arrays*, pp. 373–381, ACM/IEEE, May 1988.

[19] S. V. Rajopadhye, "Synthesizing systolic arrays with control signals from recurrence equations," *Distributed Computing*, vol. 3, pp. 88–105, Springer-Verlag, 1989.

[20] L. W. Tucker and G. G. Robertson, "Architecture and applications of the connection machine," *IEEE Computer*, pp. 26–38, Aug. 1988.

[21] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing.* Redwood City, CA: Benjamin/Cummings, 1989.

[22] U. Banerjee, *Dependence Analysis for Supercomputing.* Kluwer Academic, 1988.

[23] H. B. Ribas, "Obtaining dependence vectors for nested-loop computations," *Proceedings International Conference on Parallel Processing*, vol. II, pp. 212–219, Pennsylvania State University Press, 1990.

[24] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp computer: Architecture, implementation and performance," *IEEE Transactions on Computers*, vol. C-36, pp. 1523–1538, Dec. 1987.

[25] S. Borkar, "iWarp : An integrated solution to high-speed parallel computing," *Proceedings Supercomputing*, pp. 330–339, IEEE Computer Society Press, Nov. 1988.

[26] D. E. Fouler and R. Schreiber, "The Saxpy Matrix-1: A general purpose systolic computer," *IEEE Computer*, vol. 20, p. 35, July 1987.

[27] B. L. Drake, F. T. Luk, J. M. Speiser, and J. J. Symanski, "SLAPP: A systolic linear algebra computer," *IEEE Computer*, vol. 20, p. 45, July 1987.

[28] K. N. Ganapathy and B. W. Wah, "Optimal design of lower dimensional processor arrays for uniform recurrences," *Proceedings Application Specific Array Processors*, pp. 636–648, IEEE Computer Society Press, Aug. 1992.

[29] K. N. Ganapathy and B. W. Wah, "Synthesizing optimal lower dimensional processor arrays," *Proceedings of International Conference on Parallel Processing*, pp. 96–103, Pennsylvania State University Press, Aug. 1992.

[30] K. N. Ganapathy and B. W. Wah, "Optimal synthesis of algorithm-specific lower-dimensional processor arrays," submitted to *IEEE Transactions on Parallel and Distributed Systems*, Nov. 1993.

[31] J. A. B. Fortes, B. W. Wah, W. Shang, and K. N. Ganapathy, "Algorithm-specific parallel processing with linear processor arrays," in *Advances in Computers*, M. Yovits, Ed., Academic Press, 1994 (to appear).

[32] K. N. Ganapathy and B. W. Wah, "Optimal synthesis of processor arrays with pipelined arithmetic units," *Parallel Processing Letters, Special issue on parallelization techniques for uniform algorithms*, Spring 1994 (to appear).

[33] K. N. Ganapathy and B. W. Wah, "Designing a coprocessor for regular recurrent computations," *Fifth IEEE Symposium on Parallel and Distributed Systems*, pp. 806–813, Dec. 1993.

[34] M. Valero, S. Y. Kung, T. Lang, and J. A. B. Fortes *Proceedings 1991 Application Specific Array Processors*, (Los Alamitos, CA), IEEE Computer Society Press, 1991.

[35] J. A. B. Fortes, E. Lee, and T. Meng *Proceedings 1992 Application Specific Array Processors*, (Los Alamitos, CA), IEEE Computer Society Press, 1992.

[36] R. H. Kuhn, "Optimization and interconnection complexity for parallel processors, single stage networks and decision trees," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, Oct. 1980.

[37] D. I. Moldovan, "On the analysis and synthesis of VLSI algorithms," *IEEE Transactions on Computers*, vol. C-31, pp. 1121–1126, Nov. 1982.

[38] W. Shang, "Scheduling, partitioning and mapping of uniform dependence algorithms on processor arrays," Ph.D. dissertation, Purdue University, Indiana, May 1990.

[39] J. A. B. Fortes, K.-S. Fu, and B. W. Wah, "Systematic design approaches for algorithmically specified systolic arrays," in *Computer Architecture: Concepts and Systems*, V. M. Milutinovic, Ed., North Holland, 1988, pp. 454–494.

[40] P.-Z. Lee and Z. M. Kedem, "Mapping nested loop algorithms into multidimensional systolic arrays," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 64–76, Jan. 1990.

[41] V. P. Roychowdhury and T. Kailath, "Subspace scheduling and parallel implementation of non-systolic regular iterative algorithms," *Journal of VLSI Signal Processing*, vol. 1, Kluwer Academic Publishers, 89.

[42] G.-J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Transactions on Computers*, vol. C-34, pp. 66–77, Jan. 1985.

[43] M. T. O'Keefe, J. A. B. Fortes, and B. W. Wah, "On the relationship between systolic array design methodologies," *IEEE Transactions on Computers*, vol. C-41, pp. 1589–1593, Dec. 1991.

[44] J. Xue, "A new formulation of the mapping conditions for the synthesis of linear systolic arrays," *Proceedings Application Specific Array Processors*, pp. 297–308, IEEE Computer Society Press, 1993.

[45] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1975.

[46] M. Gondran, M. Minoux, and S. Vajda, *Graphs and Algorithms.* New York: Wiley, 1984.

[47] B. A. Carre, *Graphs and Networks.* Oxford University Press, 1979.

[48] U. Zimmermann, "Linear and combinatorial optimization in ordered algebraic structures (chapter 8: Algebraic path problems)," *Annals of Discrete Mathematics*, vol. 10, pp. 1–380, 1981.

[49] D. H. Lehmann, "Algebraic structures for transitive closure," *Theoretical Computer Science*, vol. 4, pp. 59–76, 1977.

[50] R. E. Tarjan, "A unified approach to path problems," *Journal of the ACM*, vol. 28, pp. 577–593, 1981.

[51] S. Y. Kung, S. C. Lo, and P. S. Lewis, "Optimal systolic design for transitive closure and shortest path problems," *IEEE Transactions on Computers*, vol. C-36, no. 5, pp. 603–614, 1987.

[52] G. Rote, "A systolic array for algebraic path problem," *Computing*, vol. 34, pp. 192–219, Springer-Verlag, 1985.

[53] O. H. Ibarra and M. Palis, "VLSI algorithms for solving recurrence equations and applications," *IEEE Transactions on Acoustics, Speech, Signal Processing*, vol. ASSP-35, pp. 1046–1064, July 1987.

[54] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," *Proceedings of CALTECH Conference on VLSI*, pp. 509–525, Jan. 1979.

[55] J. G. Nash, K. W. Przytula, and S. Hansen, "The systolic/cellular system for signal processing," *IEEE Computer*, vol. 20, p. 96, July 1987.

[56] K. W. Przytula, "Medium grain parallel architecture for image and signal processing," in *Parallel Architectures and Algorithms for Image Understanding*, V. K. P. Kumar, Ed., Academic Press, 1991, pp. 95–119.

[57] J. Symanski and K. Bromley, "Video analysis transputer array (VATA) processor," *SPIE Real-Time Signal Processing XI*, Aug. 88.

[58] J. H. Moreno and M. E. Figueroa, "A decoupled access/execute processor for matrix algorithms: Architecture and programming," *Proceedings Application Specific Array Processors*, pp. 281–295, IEEE Computer Society Press, 1991.

[59] D. Le, M. Ercegovac, T. Lang, and J. Moreno, "MAMACG: A tool for mapping matrix algorithms on to mesh connected processor arrays," *Proceedings Application Specific Array Processors*, pp. 511–525, Aug. 1992.

[60] S. Y. Kung, *VLSI Processor Arrays.* Englewood Cliffs, NJ: Prentice Hall, 1988.

[61] K. Jainandunsing, "Optimal partitioning schemes for wavefront/systolic array processors," Technical Report, Delft University of Technology, Delft, The Netherlands, Apr. 1986.

[62] J. Bu and E. F. Deprettere, "Processor clustering for the design of optimal fixed-size systolic arrays," *Proceedings Application Specific Array Processors*, pp. 402–413, IEEE Computer Society Press, Sept. 1991.

[63] J. J. Navarro, J. M. Llaberia, and M. Valero, "Partitioning: An essential step in mapping algorithms into systolic array processors," *IEEE Computer*, vol. 20, pp. 77–89, July 1987.

[64] A. Suarez, J. M. Llaberia, and A. Fernandez, "Scheduling partitions in systolic algorithms," *Proceedings Application Specific Array Processors*, pp. 619–633, IEEE Computer Press, Aug. 1992.

[65] P. Kuchibhotla and B. D. Rao, "Efficient scheduling methods for partitioned systolic algorithms," *Proceedings Application Specific Array Processors*, pp. 649–663, IEEE Computer Press, Aug. 1992.

[66] V. V. Dongen, "Mapping uniform recurrence onto small size arrays," *Proceedings of PARLE*, pp. 191–208, 1991.

[67] E. E. Swartzlander, "Advanced technology for improved signal processor efficiency," *Proceedings 1992 Application Specific Array Processors*, (Los Alamitos, CA), pp. 257–268, IEEE Computer Society Press, 1992.

[68] W. P. Burleston, "Partitioning problem on VLSI arrays: I/O and local memory complexity," *Proceedings of ICASSP*, (Toronto, Canada), pp. 1217–1220, May 1991.

[69] X. Zhong and S. Rajapadhye, "Deriving fully efficient systolic arrays by quasi-linear allocation functions," *Proceedings of PARLE*, pp. 219–235, 1991.

[70] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrences," *Journal of the ACM*, vol. 14, pp. 563–590, July 1967.

[71] J. K. Peir and R. Cytron, "Minimum distance: A method for partitioning recurrences for multiprocessors," *Proceedings International Conference on Parallel Processing*, pp. 217–225, 1987.

[72] F. Irigoin and R. Triolet, "Supernode partitioning," *Proceedings of the Fifteen annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 319–329, Jan. 1988.

[73] Z. Chen and W. Shang, "Mapping of uniform dependence algorithms onto fixed size processor arrays," *International Parallel Processing Symposium*, Apr. 1993.

[74] J.-W. Hong and H. T. Kung, "The I/O complexity: The red blue pebble game," *Proceedings of 13th annual ACM Symposium on Theory of Computing*, pp. 326–333, May 1981.

[75] A. P. Chandrakasan, S. Sheng, and R. W.Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-state Circuits*, vol. 27, pp. 473–484, Apr. 1992.

VITA

Kumar Nanjunda Ganapathy received a Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology at Madras in 1987, and a Master of Science degree in Electrical and Computer Engineering from the University of Massachusetts at Amherst in 1989. He is currently a candidate for Doctor of Philosophy in Electrical Engineering at the University of Illinois at Urbana-Champaign. After completing his doctoral dissertation, he will take up a position as a Design Engineer with the Telecommunications Division of Rockwell International at Newport Beach, California.