UNIVERSITY OF CALIFORNIA
Los Angeles

A SYNTHESIS SYSTEM FOR APPLICATION SPECIFIC ARRAYS IMPLEMENTING MATRIX COMPUTATIONS

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy
in Computer Science

by

James Jianhuang Liu

1994

Contents

List of Figures

List of Tables

ABSTRACT OF THE DISSERTATION

A SYNTHESIS SYSTEM FOR APPLICATION SPECIFIC ARRAYS IMPLEMENTING MATRIX COMPUTATIONS

by

James Jianhuang Liu
Computer Science Department
University of California, Los Angeles
Professor Miloš D. Ercegovac, Chair
1994

We propose a theoretical and methodological approach for synthesis of application specific arrays. The system, called ALIAS (ALgorithms Into Arrays Synthesizer), is intended to be an interactive design environment facilitating mapping between matrix algorithm specifications and hardware implementations.

The input to the system is the single-assignment form of the algorithms and the output from the system is a synthesized array or a mapping of the algorithm on a user-specified array. A priori knowledge is not required for the user since the symbolic transformations are automatically done by the tool. ALIAS provides automated transformations of input matrix algorithms and interactive mapping to processor arrays.

A key feature of the proposed approach is *MGD (Mesh Graph Descriptor)* notation, used to formally characterize the design transformations in both the behavioral and structural domains. This notation is designed to be transparent to the user and, more importantly, to allow denotational transformations to take place.

ALIAS consists of three main modules: *MAP (Matrix Algorithm Parser), DIP (Display Interface Program)*, and *DM (Descriptor Manipulator)*. MAP takes inputs which are the single-assignment equivalents of matrix algorithms and produces MGD specifications for the dependency graph corresponding to that algorithm. DIP displays the *MMG (Multi-Mesh Graph)* structure to the user in interactive manner. DM module then applies *transformers* selected by the user to the MGD descriptors in deriving the array implementations of the algorithm. Given the design parameters and constraints, DM can be

selectively used to obtain optimal designs. The user can choose to display various screens showing the MMG, node scheduling and assignment, or the array design along with the number of execution steps.

Chapter 1

Introduction

Matrix computations are frequently used in image and real-time signal processing techniques as can be found in applications such as radar, speech processing, weather forecasting, medical diagnosis, and nuclear engineering. The increasing demand in these areas necessitates state-of-the-art methodologies in transforming computation specifications into processing arrays. Consequently, many tools have been constructed to automatically generate the designs that realize matrix computations in these application areas. Most of these tools adopt little or no formal methodology. We propose a formal and integrated technique of mapping matrix algorithm specifications into designs of parallel processing arrays and demonstrate usefulness of our approach by implementing it as a tool.

1.1 Motivation

In the mid 80s, synthesis tools were mature at the logic level which were primarily used to either optimize a set of predefined netlists or convert intermediate form into gate-level logic. Synthesis is gradually evolving into higher levels of abstraction. Eventually, starting at the behavioral synthesis level, the entire target system can be created from the predetermined constraints such as types of hardware resources and their performance.

One of the common characteristics of previous synthesis tools is the use of a transformational approach, namely, the architectures are derived by transforming the original algorithm descriptions that are unsuitable for direct implementation. As pointed out in a survey work [FFW88], two primary limitations in the state-of-the-art transformational systems are the lack of powerful *systematic semantic* transformations and the inability to systematically achieve optimality in the resulting designs. We propose a method that takes specification of algorithms in the original form and applies systematic transformations to get the satisfactory designs. Many of the previously proposed approaches are not amenable to the physical implementation as tools for designing application specific arrays. Our method, which incorporates and integrates ideas from past studies as well as new transformational notations, provides a basis for fast design of efficient and useful arrays.

Many methods have been proposed in the literature to synthesize parallel arrays. However, few researchers incorporated their methods into practical design environment. To implement a synthesis tool, we usually require an appropriate language to be the specification of the synthesized functional components. The importance of this type of language can be intuitively justified by the VHDL in the circuit synthesis environment (e.g., a silicon compiler). Moreover, the lack of appropriate, integrated, and fully-automated tools suggests a need for further work on methodologies which should be efficient as well as practical. A good methodology should be practical in the sense that the targeted application areas have to reflect the actual needs. A great deal of the past work aims at relatively trivial examples and therefore contains less value to practicality, though most of them are distinctive and self-contained. A good rationale of devising a technique is, rather, to take the existing and useful algorithms and aim to solve the problem without any compromise in adapting or diverting the technique to specially tailored examples. A good methodology also should be able to seek optimal solutions. The desired features in a practical and efficient design environment, addressed in this dissertation, are:

- Capability of behavioral input to the synthesis system.
- Provision of a systematic and robust transformation system.
- Minimal requirement in user's intervention in the synthesis system and in knowledge about technical detail of the transformation process.
- Flexibility in deriving the optimal (or suboptimal) designs which suit the user's needs.

Very few of the systems, proposed previously, can satisfy all the the above requirements.

1.2 Related Work

We begin by examining properties of the existing methods in transforming the specifications into VLSI designs beforehand. Basically, for most methods, there are two stages involved in the design process. First, the extraction of the partial ordering in an algorithmic description yields a dependence graph (DG). Second, since it is not practical to assign of each node in a DG to a processor, it is often desirable to cluster the nodes in the DG and map them to a processor. The use of transformational methods is a trait commonly found in most previously proposed approaches. For example, the original algorithmic specification is not amenable for direct systolic implementation. As a result, final systolic designs are derived by a series of transformations on the input to the system. There are other key issues, in addition to transformation, on which the synthesis system can be distinguished. These issues include the format of the input algorithm, the specification of the final designs, and the model of the synthesis environment. We primarily focus on the transformational model of the design flow and use it to classify the currently known approaches. In general, the proposed approaches can be classified into the following three categories.

Index transformation Most proposed methods fit in this category. Typically, index sets and data dependence vectors are used to mathematically map algorithms into arrays. Dependence vectors are modified through linear transformation to obtain different structures that are suitable for parallel array implementation. Work of this nature can be found in [Mol82], [Mol83], [CS83], [MW84], [Qui84], [Rao85].

Graphical transformation Illustrative and mechanical methods are used to accomplish similar objective as in the index transformation. Design approaches based on this type of transformations are suitable for local regularizations and have a potential for obtaining better cell bandwidth and utilization. Lack of formal manipulation of the dependence graphs is their key limitation. Examples belonging to this category are [Kun88], [Omt88], [Mor89] [Le92].

Symbolic transformation The method proposed in this thesis fits in this category [LJ88b], [LJ88a], [Luk92], [LE93b], [LE93a]. The transformation stages in the synthesis procedure are systematically characterized by symbolic formalism, instead of being characterized mathematically or mechanically like in the above two approaches.

Now we review some of the proposed systems.

Moreno proposes a graphical method that realizes matrix computations on mesh arrays [Mor89]. The mesh arrays are two-dimensional structures with nearest-neighbor connected cells. There are two stages in the design methodology: *algorithm regularization* and *derivation of arrays*. The first stage involves the work to derive a fully-parallel data-dependency graph (FPG) of the input algorithm and manually transforms the graph into a 3-D one with uni-directional nearest-neighbor dependencies. The second stage is to transform the mesh graph into a 2-D G-graph which is realized as an algorithm-specific array or is mapped onto a class-specific array. In this work, the design transformations are not formalized.

Le [Le92] develops an interactive tool that automates Moreno's design methodology. Algorithms like matrix multiplication, LU decomposition, and Warshall-Floyd algorithm are implemented on an X-window-based environment. However, the partitioning of the input algorithms into limited-sized arrays and the performance analysis are not addressed in the work.

ARREST is a CAD tool for VLSI arrays [BJ92]. It accepts a high-level descriptions in terms of Affine Recurrence Equations (AREs) which are equivalent to single-assignment algorithms. After parsing the input, ARREST generates the corresponding dependence graph (DG), on which user modifications are allowed. By user-specified scheduling and projection vectors, the system can also generate signal flow graphs (SFG). ARREST provides cost/performance analysis in terms of latency, processor count, and pipelining period. Projections on the DGs to obtain processor arrays are implemented. By adopting the design techniques for systolic/application specific arrays, this system provides a good integrated interface for VLSI array design to and from VERILOG. However, the problem of partitioning and mapping a large DG into a fixed-sized array is not addressed. The underlying methodology can be further formalized. The user does not get much feedback from the system on the input/output pattern and execution time for the the synthesized arrays.

SYSTARS is a CAD tool for synthesizing and analyzing VLSI systolic and wavefront arrays [Omt88]. SYSTARS also employs graphical approaches to derive the processing arrays on a user-interactive tool. Given a regular iterative algorithm, it displays 3-D dependence graphs which can be manipulated by user. This design tool starts the design of the system architecture of a systolic array with the specification of the algorithm in single assignment form. After the dependence graph is generated, simulation of execution on the dependence graph can be visualized through display. Scheduling vectors, derivable from an integer programming problem, can be entered to the system by user. Application examples discussed are matrix-matrix multiplication, Given's algorithm and Schur's algorithm. The SYSTARS system has the following limitations. The attributes of the nodes in the dependence graph are not explicitly illustrated, neither are the data input and output of the array. Furthermore, the user may have to enter the scheduling vector which implies that the user needs certain knowledge on technical details like the cluster vectors.

In [MF86], methods are also proposed to partition compute-intensive algorithms and map them into fixed size systolic arrays. Initializations on some of the arrays could potentially introduce overhead to the execution. From the same researcher, ADVIS [Mol87] is a software package that facilitates the design of systolic arrays. It is used to find out all the integer solutions in simultaneous equalities of transformation. Accordingly, ADVIS was not targeted as a user interactive design environment. In addition, ADVIS system conducts exhaustive search of all possible solutions and the designer has to select one that suits his needs.

CRYSTAL [Che85] is a parallel language which describes the algorithm executed by systolic architecture as a set of time-space equations. The compiler of CRYSTAL takes these recursive equations and generates implementations. A formalized model of systolic arrays is also given in terms of the algorithmic point of view. There are two major steps in CRYSTAL environment concerning synthesis: a) algorithm transformation, and b) mapping into STREQ (space-time recurrence equations). What is called concrete structure is similar in nature to the multi-mesh graphs described in our work, and the dependence graph in Rao's work [Rao85]. Abstract structure shows analogous characteristics as our designs do. This work emphasizes the language design rather than the synthesis aspect. Mapping algorithms of large sizes into partitioned/limited array processors is not considered in this work.

Rao's method [RK88] includes a broader class of algorithms called RIAs (Regular Iterative Algorithms) than systolic ones. The arrays implementing these algorithms are called Regular Iterative Arrays[1]. One case found that is RIA but not systolic is Gaussian elimination with pivoting. The corresponding processing array requires register pipelines and Last-In-First-Out (LIFO) buffers, thus can't be classified as a systolic array. It is our feeling that some RIAs, by being incorporated with features from multi-processor arrays, form a broader class than the one formed by systolic arrays. In this work, a linear algebra approach is employed to find valid linear schedules and processor arrays sizes. However, no tools adopting this technique have been built so far. Despite formalism provided, the array designs in the paper are direct results from projection (coalescing). Formalized definitions of RIA (regular iterative algorithm) and systolic arrays are discussed in [RK88].

In Jagadish's work [JRK87], an example of matrix-multiplication is provided. Many of the derived implementations are also covered by other researchers. New arrays can be simply found by taking different projection angles. Attempts have been made to allow RIA to be mapped into fixed-sized mesh-connected processor arrays. Several examples have been demonstrated. Nevertheless, the approach of finding RIA equivalents is not covered. The proposed technique is similar to other techniques reported in the past [Mol83], [LW85].

Miranker and Winkler [MW84] give space-time representations of computational structures. It could be the first paper to introduce the concept of algorithm parameterization, which is essentially the addition of more index variables to array variables. The result is a set of refined algorithms with the broadcasting paths eliminated. The algorithms after modifications are close in nature to single-assignment forms[2]. The physical process of computation is interpreted in terms of a graph in physical time and space. A universal planar systolic structure is introduced in this work.

Lam and Mostow [LM85] implemented a synthesis environment called *Sys*. Sys accepts algorithms and applies successive transformations to produce functional-level systolic arrays. Module descriptions can be generated as Sys parses the algorithms. The descriptions are procedure-like with spec-

[1]It is also acronymed as RIAs in the paper, making it confusing.

[2]These parameterized algorithms together with single-assignment algorithms were introduced earlier than, and analogous to, Regular Iterative Algorithms.

ifications of ports, memory registers and detailed computational behavior. Sys is one of the early-stage systolic array synthesizers, and, notably, fully-integrated in the sense that it accepts algorithms and produces arrays. In Sys, the algorithmic inputs have to provide information as to what implementation style should be chosen. Therefore, the designer/user should have in mind a roughly topological design picture before the design can be synthesized. In ALIAS the user merely input algorithms with no connection to design style. Secondly, there is only one design style for each algorithm input to Sys. This excludes various design alternatives from further considerations.

Cappello [CS83] starts with algorithm descriptions and produces a canonical representation[3] by augmenting the index set in the algorithm with time t. In terms of linear transformation, the canonical representation can be converted into geometric representations in three-dimensional space. Many of previous convolver designs have been captured by this method. Note that the method is somewhat similar to approaches in [Mol83] and [MW84]. There exists a limitation in these methods, i.e., the structures in nodes of the dependence representation are not explicitly specified and differentiated. This over-generalization may not produce efficient designs for the class of algorithms whose structures of dependence graphs are not homogeneous. The same author also proposed a processor-time-minimal systolic array for cubical mesh algorithms [Cap92]. In this work, directed acyclic graphs, which are nearly synonymous to mesh graphs, are used as generalized data dependence graphs. The complete graphs are first coalesced to form a partial mapping to a planar array of processors. It is found that a few processors that compute earlier input data can be overlapped with some other processor that handle latter stages of the entire computation. Therefore, these processors are combined together in an effort to reduce hardware costs. Symmetrically, when the trailing processors are handling their computations, the processors that handle the earlier operations will be idle. These trailing processors can be eliminated and their work can be assigned to other processors to enhance utilization and reduce the number of processors implemented. The idea seems to be simple and useful. However, the following three limitations are easily observed: a) The resultant designs contain interconnection irregularities. It is certainly not a desirable feature in the event that hardware is to be imple-

[3]Actually, a synonym of dependence graph embedded in 3-D with one axis tagged as time.

mented. b) The input to the trapezoidal array is not housed at the boundary PEs. This adds difficulty to the implementation, too. c) The introduced design is not amenable to handle more than one instance of data since there are many overlappings in execution which violates the principle of systolic arrays.

Li and Wah [LW85] propose an optimal mapping of algorithms that are parametrized by the velocities of data flows, spatial distributions, and periods of computation. The design problem can be modeled as a set of corresponding systolic processing equations and the constraints on the values of the parameters. The next step is to find the parameter values that minimize the objective function by exhaustive search on the design space. The paper assumes that optimal planar systolic arrays are of hexagonal topology. Hence, the proposed six vector constraints of data moving are based on the same topology. This assumption results in that all the 2-D arrays designed by the method have hexagonal connectivity. An evident drawback of this method is the lack of considering other design possibilities such as mesh-connected and linear arrays. For some applications, mesh-connected arrays provide higher utilization rates, lower connection complexity, and shorter latency. In this paper, it seems unnecessary for the execution time formula of matrix multiplication to distinguish node computation time into t_i and t_j since all nodes should be identical. An observation is that the proposed method is applicable to merely highly regular algorithms.

Hartenstein and Lemmert [HL89] present a CHDL-based CAD system, SYS^3, for systolic architecture synthesis. The high level description language is called $KARL$. The proposed approach uses space-time transformations to map algorithmic descriptions specified in $KARL$ into systolic architecture. There are similarities in nature in generating systolic arrays as found in a high-level synthesis system for hardware. The essential set of all basic systolic cells in hexagonal and mesh interconnections is provided. Simple linear transformations are applied to obtain the final implementations. The generated architectures can be specified in procedural descriptions as well. However, the example used in the paper is somewhat trivial. A tabular comparison between the approach and ADVIS system is also provided.

Lincoln [Lin89] proposes a dependence graph mapping method which is essentially similar to the index dependency approach in Rao's work. The method applies to practical algorithms such as Kalman filtering, discrete Fourier transform, and fast Fourier transform. In this work, it is shown that dependence graph mapping techniques can be applied to real-time signal processing algorithms of higher complexity. The input to this method is also the single assignment algorithm. The output is the systolic or processing array design. Some arrays developed hereby are claimed to have higher throughput than the ones found in other works.

Johnsson [JHM89] discusses a data parallel implementation of the multiplication of matrices of arbitrary shapes on the Connection Machine CM-2. The mapping is done in two phases, alignment and systolic multiplication. The alignment task is performed using the router which is basically an interconnection switching network. As opposed to the fully-parallel implementation on a machine that is dedicated to the similar application, this commercial machine implementation tends to have lower performance due to the overhead of the router. In this paper, no comparisons to other similar work are stated.

Sheu and Chang [SC91] present a nonlinear transformation approach to execute for-loop algorithms in parallel form. This work also considers mapping algorithms into fixed size systolic arrays. It is claimed, for one particular case, the total execution time can be reduced to one sixth of linear method. However, two possible limitations of this nonlinear method is that only model of n-nested loops can be applied and the indices of innermost loops have to be formulated in a continuous manner. The practicality of the especially chosen algorithms may be questionable, even though an extensive and formalized analysis of the nonlinear method is presented.

Among the aforementioned approaches, many can be classified as index-dependency methods with linear transformations [CS83], [Mol83], [Rao85], [Mol87]. These methods, despite their elegance in mathematical derivations, have a tendency to oversimplify the nature of most practical algorithms. Many algorithms are not uniform as matrix-vector or band-matrix multiplication. By applying index dependency method to these algorithms, solutions obtained may not be satisfactory. Furthermore, index-dependency methods are more amenable to designing systolic arrays. As it turns out that not many successful processing arrays of this type have been built to handle practical applications, other favorable options should be explored. For instance, iWARP array processor is a hybrid configuration of systolic arrays and multiprocessors [Bor88].

1.3 Objectives of the Research

We propose a system called ALIAS (ALgorithms Into Arrays Synthesizer), which is an interactive design environment facilitating automatic design of processing arrays for matrix computations. These matrix computations are mainly for real-time signal/image processing applications. ALIAS accepts as input matrix algorithms in their single-assignment forms and produce application specific processing arrays along with interconnectivity of the PEs, input data distribution, computations inside each nodes, mapping assignment, and performance measures on display. Each of the steps in the synthesis procedure is performed in a systematic and formalized manner. Another appropriate name for ALIAS design system would be, in view of the above functionality, a *systolic compiler*, in analogy to *silicon compiler* which generates layout from higher-level description.

Typically, there are three levels associated with an array compiler environment from top to down: array layer, data/control layer, and implementation layer. The array layer takes the behavioral description (e.g. algorithmic specification) as input and produces array processors in the form of functional blocks and interconnections. The input to the next layer – data/control – is a set of functional requirements of processor elements and their interconnections at the array layer. The output is a set of components such as data path and control path, which are fed into implementation layer. The implementation layer deals with physical issues such as area consideration (partition, placement and routing), wiring delay analysis, power dissipation, fabrication, packaging, configuration, etc.

The ALIAS design system fits in the first layer of the three, as illustrated by Figure 1.1. This is different from most traditional high-level synthesis systems which most likely fit in the lower two layers. The user of these synthesis systems usually has fundamental knowledge about what the floor plan and functionality of components would look like. For example, a user needs to elaborate the design requirements in the form of the specifications such as VHDL, or Verilog. Hence, our ALIAS synthesis system also provides higher level of abstraction to the user by not having user to translate system requirements into cumbersome language specifications.

The objectives of ALIAS are as follows.

- To show how parallel processing arrays can be generated from high

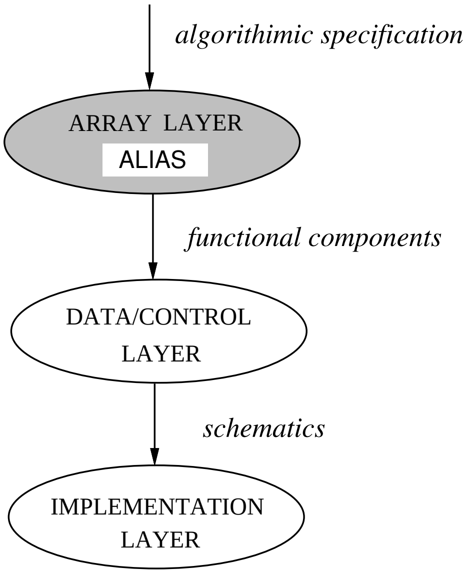

Figure 1.1: Layers of a typical array compilation environment and the standing of ALIAS.

level descriptions and how design alternatives can be obtained given only a single specification.

- To show that, via a formal approach, designs generated need less or no verification at the higher design level.
- To introduce a new coordinate-free notational tool as the basis for dependence graph transformations and conformity check upon system requirement constraints.
- To propose a machine-independent methodological framework which can be readily portable and built upon.

1.4 Methodology

The work presented here is based on a systematic method for deriving mesh arrays for matrix computations by Moreno [Mor89]. This method derives a data-dependency graph based on an input algorithm. The graph is subsequently transformed into a 3-D multi-mesh graph (MMG). Next, the MMG is transformed into a 2-D G-graph which can be subsequently realized as an algorithm-specific or class-specific array. Some of the design rationales and features of this method have been adopted in our approach, i.e., coalescing mesh graphs and constructing computational nodes. There are some changes, too. Instead of taking algorithms with loops in regular mathematical form, in ALIAS, single-assignment algorithms are the input to the system since it is shown that all broadcasts can be systematically transformed into propagations [WD92]. Namely, the data dependences can be "localized" and "regularized". Bidirectional and irregular transmittent arcs problem can be eliminated. In [Rao85], it is shown that if an algorithm can be implemented on a systolic array then the algorithm can be described as a regular iterative algorithm.

As far as the operational issues are concerned, users of ALIAS provide interactive decision-making parameters in order for the design systems to proceed. The parameters are the projection directions (X, Y, Z, diagonal, skewed) and the size of the grouping prisms prior to the projection stage. These options are displayed in the form of a menu to obtain from the interactive user the parameters as required in the system synthesis process.

A conceptual graph depicting the design environment ALIAS is illustrated Figure 1.2.

In our research, we introduce a notation tool called *MGD (Mesh Graph Descriptors)*. MGD is an intermediate and hidden-from-user specification of MMG created by *MAP (Matrix Algorithm Parser)* after the input to our design system, i.e., the matrix algorithm having been parsed. There are two types of single-assignment algorithms considered in our synthesis system: a) *index-independent*, b) *index-dependent*. Index-independent algorithms refer to those single-assignment algorithms without conditional statements inside the loop structure. Examples in this category are matrix-vector multiplication, polynomial multiplication, skewed outer product, matrix-matrix multiplication, and convolution. Each statement inside the loop of an index-independent single-assignment algorithm gets executed exactly once for every iteration. Index-dependent algorithms refer to these single-assignment algorithms with conditional statements conditioned by its indices. In other words, the flow of execution may depend upon the values of the indices. For instance, LU decomposition, QR decomposition, and algebraic path problem (APP) are of this type. The execution trace for an index-dependent algorithm may have different computation pattern for different values of loop variables. Interpretation on algorithms of index-independent type can be done more easily than index-dependent ones. The formulation of MMG is incremental as the interpreting process proceeds, with the predetermined incremental pattern called *DRS (Default Replication Sequence)*. To generate MMG graphs corresponding to single-assignment algorithms without conditional structure inside loop is straightforward. The dimensionality of MMGs is equivalent to the number of iterative statements in algorithms parsed by MAP. Creating MMGs for algorithms with conditional assignments needs slightly more complicated procedures. The allocation of nodes in this case is no longer a replicating process. The algorithm being dealt with needs to be symbolically executed so that the MGD can be formulated and from which nodes in MMG can be identified and allocated.

There are works reported to derive systolic arrays from regular recurrence loops automatically [Mol82], [Qui84]. However, none of them has a completely automated procedure in producing array designs from high-level algorithms. The intervention of human designers cannot be lifted. Designers are required to have certain knowledge about manipulating loop dependencies and about solving the projection problems. The existing tools have very

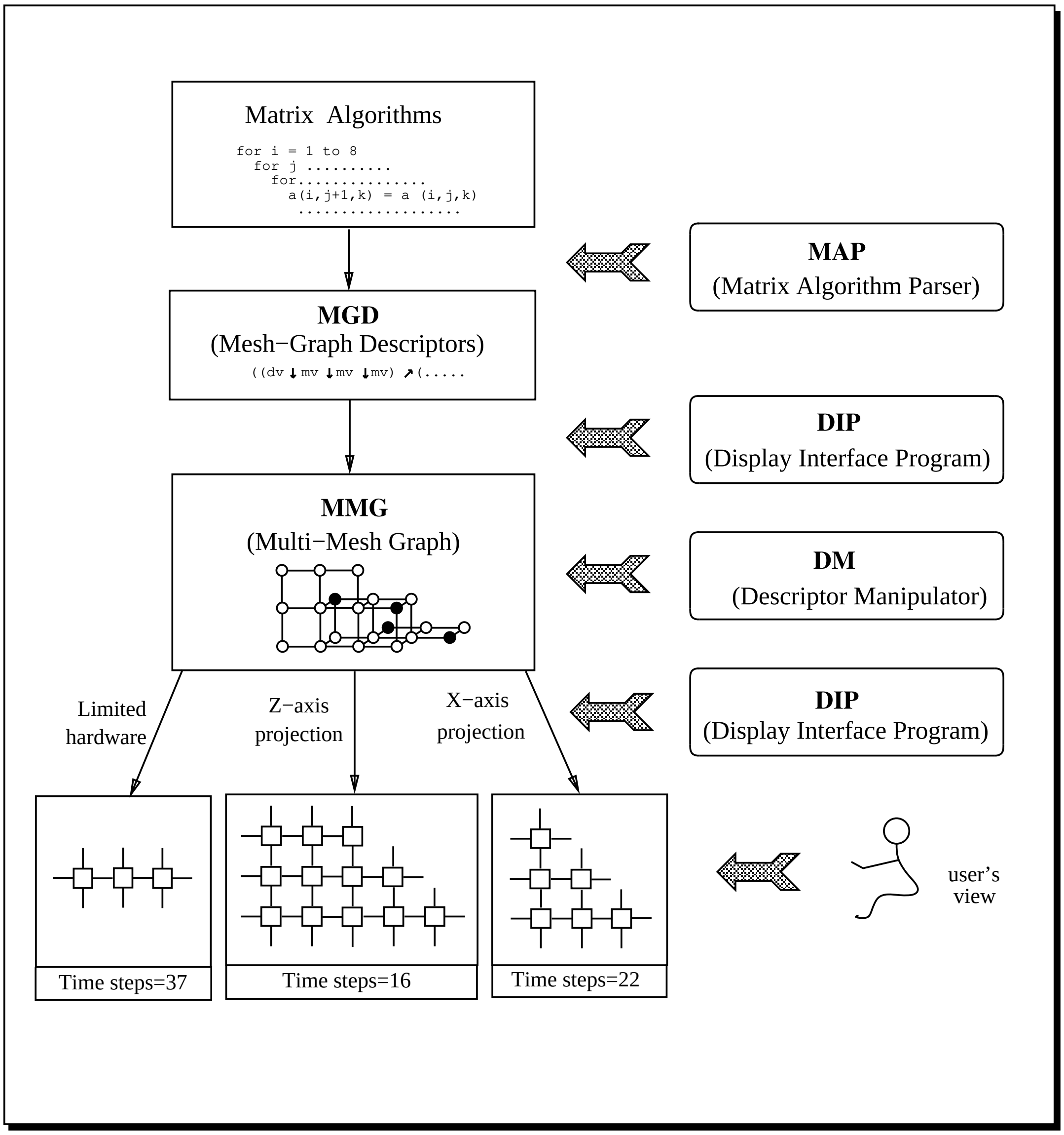

Figure 1.2: An overview of ALIAS design environment.

limited capability of providing users with sufficient design transparency.

Our proposed approach takes recursive single-assignment algorithms as inputs. The interpreter symbolically interprets the input algorithms to build up the intermediate representations. The final useful arrays can be obtained after several steps of refinement. The users merely play an assistant role to the tool. This does not preclude the users from having control over synthesis steps.

ALIAS is designed for automating processing array designs and one objective is to minimize user intervention. ALIAS also provides formal tool as the foundation for array synthesis. As far as limitations is concerned, ALIAS handles a limited class of matrix computation algorithms. However, a broader class of algorithms can be accommodated by modifying the MAP module. Since many recursive algorithms have flattenable data dependencies, it is believed that ALIAS system potentially can be expanded to a powerful and practical design tool.

A common characteristic can be found in most of the previously proposed methods, that is, the use of transformations [FFW88]. In these methods, systolic architectures are derived by transforming the original algorithm descriptions. There are three main issues in a typical transformational system: 1) how algorithms are described, 2) what formal models are used, 3) how systolic architectures are specified, and 4) what types of transformations are used on and between these representations. As a result, we can draw a Y-chart with three axes which are algorithm representation, algorithm model, and architecture specification [Gaj88]. This diagram is shown in Figure 1.3.

1.5 Designing Application Specific Arrays

In this chapter, we concentrate on the general design approach for parallel array processors, as well as issues like architectural models, mapping methods, and partitioning techniques. Since there are works reported to have these aspects of design issues as the basis [NLV87], [JRK87], [Kun88], [JK89], [Mor90], we feel that it should be of general interests to give an introduction.

Our main objective of the research is to map single-assignment algorithms into application specific processor arrays. The single-assignment algorithms are considered as in a refined form of matrix algorithms in the sense that the broadcasting and cross-node data propagation paths have been trans-

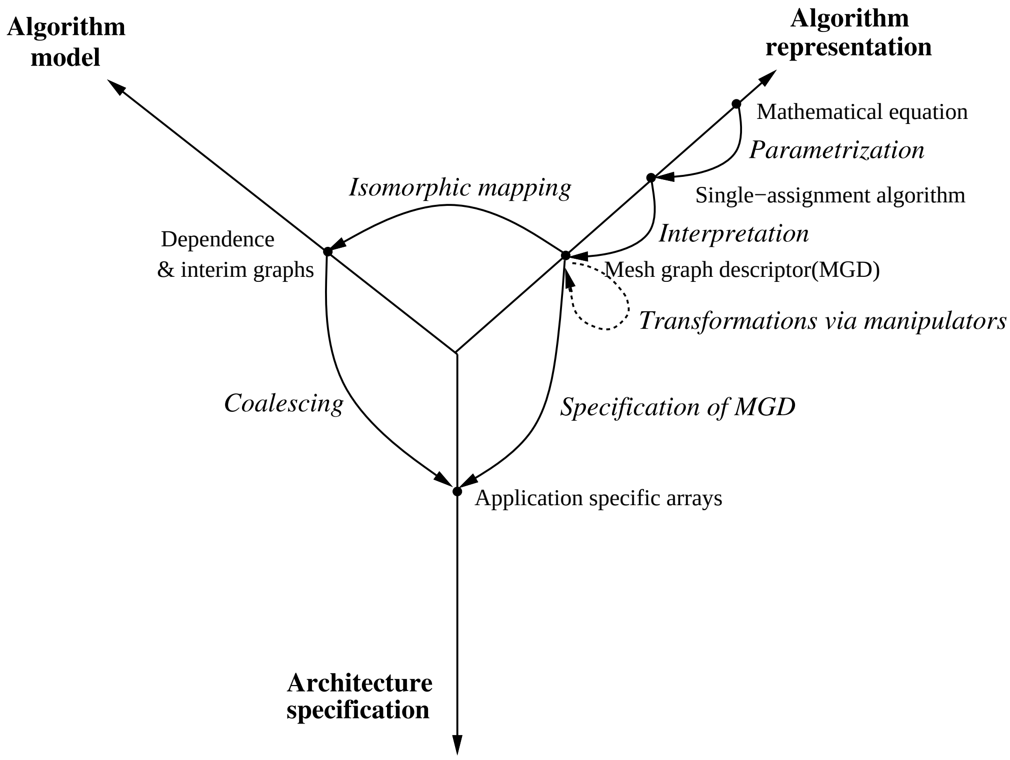

Figure 1.3: Y-chart for ALIAS system.

formed into unit-distance communication links. The following are properties of matrix algorithms [Mor89]:

- They constitute a compute-intensive class, with matrix operands/results.
- They exhibit sufficient fine-grain parallelism that is well suited for implementation in systolic-type arrays.
- Small problems with fixed-size data are suitable for multiple-instance execution in arrays, while problems with large-size data are suitable for partitioned execution in arrays.

Matrix algorithms have close nature to the so-called *systolic algorithms* in some reported work. In fact, matrix algorithms like matrix multiplication, convolution, Given's rotation, Fadeev Algorithm, and QR decomposition are systolic algorithms since they can be successfully realized by systolic arrays. A systolic algorithm is defined to be a system of first order recursion equations (FOREQ) [Che85]. FOREQ is defined on a processor structure of 2, 3, or occasionally 4 dimensional space. A system of FOREQ is the intermediate form in the synthesis and design methodology which can be viewed as a subset of the single-assignment algorithm family. In [NLV87], an intuitive definition of SA (systolic algorithm) is given. Algorithms that are efficiently executed on SAPs (systolic array processors) are called *systolic algorithms*. An SAP is a regular array of simple processors elements (PEs) that have a nearest-neighbor interconnection pattern.

Given a systolic algorithm, it needs to be converted to single-assignment format so that ALIAS system can recognize it. A single-assignment code is a form where each variable is assigned only one value during the execution of the algorithm [Kun88]. From a single-assignment algorithm, the parallelism information can be manifested by extracting the data dependence graph from the algorithm. For instance, consider the following matrix algorithm of matrix-vector multiplication and the loop form.

Algorithm:

$$C_i = \sum_{j=1}^{3} A_{ij} B_j$$

Algorithm before Conversion:

for i = 1 to 3 do
  c(i)=0
  for j = 1 to 3 do
    $c(i) = c(i) + a(i,j) * b(j)$

The above loop-form algorithm is not single-assignment code in that $c(i)$ has occurrences both as an operand and as a result. To convert the algorithm into the required form, a new index dimension is introduced for vector c. This process is called *pipelining* [MW84], or *algorithm transformation* [Che85], or *adjoining time index* [CS83]. The newly parametrized single-assignment algorithm is

Algorithm in Single-assignment Format:

for i = 1 to 3 do
  c(i,1)=0
  for j = 1 to 3 do
    $c(i, j+1) = c(i,j) + a(i,j) * b(j)$

in which each element of vector c gets assigned only one value, hence it is now a single-assignment code.

1.5.1 Regular Arrays

Topology and Attributes

Common characteristics that can be found in the application specific processor arrays are *regularity* and *local connectivity*. There are similarities between network and regular array topologies. Regular arrays considered in our work are of the following types: linear, triangular, hexagonal, mesh, and skewed mesh. Figure 1.4 shows the structures typically seen in our environment, in which the hexagonal array is topologically equivalent to the mesh array with diagonal links.

Regular arrays are suitable for VLSI fabrication for they can be easily replicated and the implementation cost can be reduced. Regular arrays are also easy to expand to conform to the problem size or to meet a certain performance requirement. Verification of the functionality on the arrays comprised of regular modules is a task more tractable than that on irregular modules.

The key attributes commonly observed in regular arrays are nearest-neighbor connection, unidirectional connection and regularity. The regular

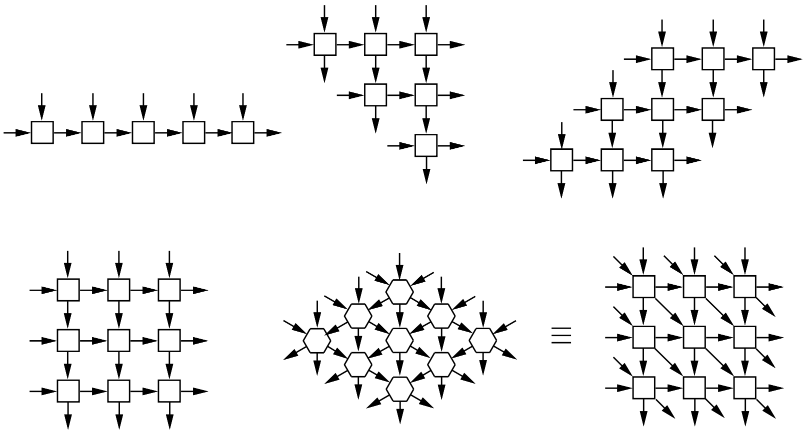

Figure 1.4: Typical regular arrays.

arrays do not have the broadcasting property or any cross-PE connectivity. These regular arrays can be synthesized by ALIAS with the single-assignment algorithms as the input objects. These algorithms, with their fine-tuned local reference pattern among vector variables, can be symbolically interpreted by ALIAS. Intuitively, the regular arrays generated from coalescing operations on the dependence graphs from the interpretation consist of only nearest-neighbor connections. Although arrays of multi-directional flow can be synthesized, we do not consider them in our work because uni-directional designs are much more amenable to realistic implementation.

1.5.2 Classification of Processor Arrays

It would be useful to analyze and categorize the arrays.

By Functionality

From the perspective of the functionality, processor arrays can be classified as either special-purpose or multi-purpose. Our research focuses on the special-purpose category.

Special-purpose Each of these arrays is for the realization of a specific algorithm. The objective is to get the optimal design possible given certain constraints. The control inside each cell is simpler than that of the multi-purpose type since only one kind of algorithm is accommodated. Software examples to synthesize these arrays are such as CRYSTAL [Che85], SYS [LM85], ADVIS [Mol87], SYSTAR [Omt88], and HIFI [AD88].

Multi-purpose In contrast to special-purpose arrays, arrays of this type are designed to execute a class of algorithms. One of the trade-offs is diversity versus performance. Since the arrays are not customized for a particular type of algorithms, the problems of longer communication delay and lower utilization would result in average performance. These multi-purpose systolic-like processors have their own instruction set which implies programmability. Hardware examples like iWARP [Bor88], [Tse90], and API15C [FGL90] fall into this category.

By Pipelinability

We know that the regular arrays execute in a pipelined (rhythmic) fashion among input data tokens for a given algorithm. However, not all arrays are able to be pipelined at the level of different algorithm instances. We distinguish algorithms based on this observation.

Single-instance An array of this type has to finish one algorithm prior to another's initiation. The pipelining behavior can only observed inside one set of input, not among sets of input. Examples are iWARP and ADVIS.

Multi-instance For this type of arrays, pipelinability is at its maximum since the second input set can be issued to the array immediately after the issuance of the first set. Examples are API15C, SYS, SYSTAR, and CRYSTAL.

The multi-instance arrays are more desirable since a potentially higher throughput can be attained.

By Size

Given an algorithm, there is a wide spectrum in the design space where the design candidate can be located. The designer or the synthesizer (the automated case) has to determine, from the input algorithm and the given constraints, which design best suits the needs and with less cost. If the problem size is reasonably small and implementation cost is not a main concern, the designer may choose *algorithm-size-dependent* arrays. When the size of a computing problem is significantly larger, algorithm-size-dependent arrays are no longer feasible. This is justified by the fact that no systolic arrays have been ever put into real implementation. Therefore a compromised way must be figured out to handle the case in which limited hardware resources are available and large computational problems still need to be carried out.

Algorithm-size-dependent Algorithm-size-dependent arrays refer to, as the name suggests, the parallel arrays whose dimensions are in proportion to the size of the algorithm. They can be further classified into the following two categories.

- **Systolic** In this case, all the nodes along one axis, either X or Y or Z, are coalesced into one processor. The size of the prism that is involved in the coalescing procedure would be $1 \times 1 \times m$. This procedure forms the systolic arrays, which are like those earliest versions proposed by Kung [KL78].
- **Pseudo-systolic** Multiple coalescing operations are possible. A 3-D mesh graph can be projected into a linear array. Another possibility is to choose the prism size to be $1 \times m \times n$. That is to say, the grouping prisms are planes parallel to each other. All the nodes in the mesh graph covered by the plane are coalesced into a processor node of the parallel array. This pushes the implementation somewhat off the end of the systolic version at the gain of hardware cost and the loss of speed. A small-size local memory inside each processor node is required in contrast to the memoryless nodes in the systolic implementation.

The design environments synthesizing algorithm-size-dependent arrays are SYS, SYSTAR, CRYSTAL, and API15C.

Algorithm-size-independent In this scheme, given a computation-intensive problem with sizable dependence graph, we map more nodes in the graph into one processor than the algorithm-size-dependent scheme. That is to say, we have to design a *fixed-size* regular array since the data dependence graphs have to be partitioned to fit into arrays with limited communication links, I/O ports, processors, and local memory. Examples are iWARP and ADVIS.

By Node-similarity

By the likeliness of the nodes that comprise the parallel arrays, we classify them into:

Homogeneous The processor nodes in an array of this type has the same functional structure in the implementation. They are more amenable for VLSI implementation since it would cost less in the design layout phase due to ease of replication. However, the mesh graphs of the algorithm to be synthesized may contain different types of primitive nodes. Examples are arrays for realization of matrix-vector operations.

Heterogeneous More than one type of nodes can be found in the parallel arrays in this category. Relatively less parallelism can be anticipated in heterogeneous arrays. An example in this category is the array that implements the transitive closure algorithm.

1.5.3 Partitioning

It is relatively easy to derive algorithm-size-dependent arrays than algorithm-size-independent ones because to derive the latter more complicated operations on the dependence graphs need to be realized. These operations involve mapping a large dependence graph into a much smaller regular processor arrays. This is the so-called the partitioning problem [Kun88]. There are two operations called cut-and-pile and projection that can contribute to this result.

Coalescing In this scheme, the dependence graph implied by the algorithm is first partitioned into several blocks in two dimensional case or prism in three dimensional case. Then a block or a prism is mapped to a processor. This method is straightforward and simpler than cut-and-pile, though the utilization rate is potentially lower.

This type of partitioning technique is also known as *LSGP (locally sequential globally parallel), projection, coalescing.* So from now on we

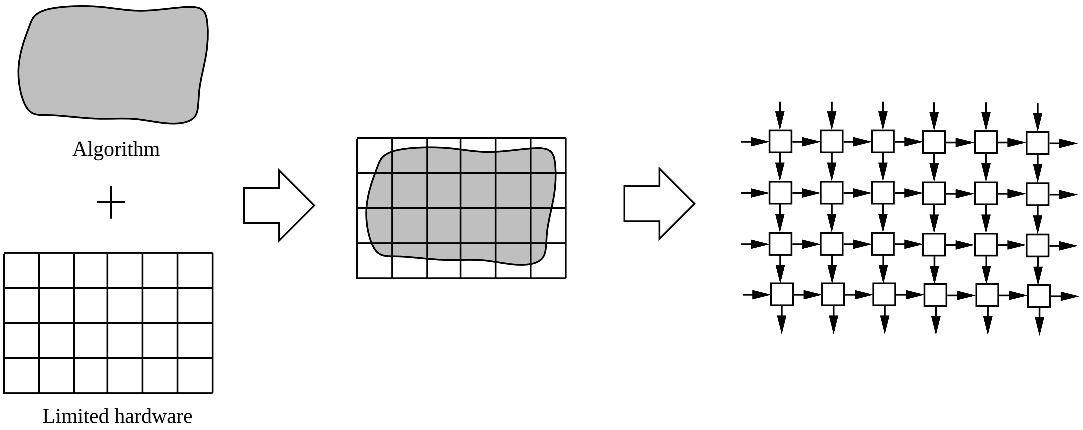

Figure 1.5: Coalescing partitioning technique.

will use these names interchangeably to refer to this scheme. Figure 1.5 depicts this scheme.

Cut-and-pile First step in this method is similar. The dependence graph is partitioned into blocks or prisms. However, each block or each prism is mapped to the entire processor space. The second block or prism can only be executed after the first one is initiated to the array. Therefore this partioning technique is not suitable for achieving multi-instance computations. The sequence of execution of the entire algorithm must be coordinated in a way so that the partial ordering in the dependence graph won't be violated. Figure 1.6 depicts this scheme. Cut-and-pile, also known as *LSGP (locally sequential globally parallel)*, usually requires an external FIFO to the array to store the intermediate data during the computation.

1.5.4 Two Key Issues in Mapping

There are two main considerations in the procedure of mapping algorithms into arrays [Kun88]. One consideration is how to assign the nodes in the data dependence graphs into the physical processor space and the other is to resolve the sequence of the execution of the dependence nodes on each

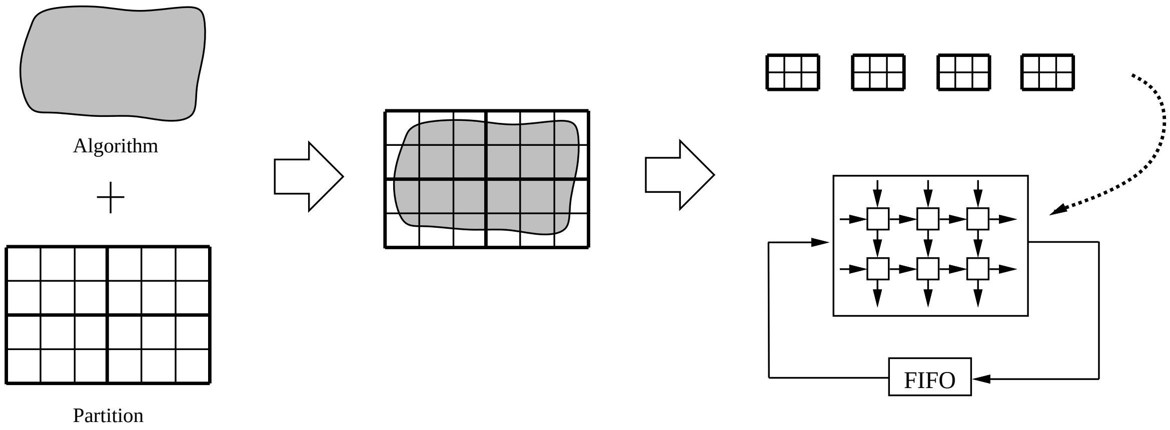

Figure 1.6: Cut-and-pile partitioning technique.

processor. In short, *processor assignment* and *scheduling* are the two key considerations during mapping procedure.

The cut-and-pile and coalescing methods in previous sections are two methods to accomplish the task of processor assignment. The scheduling can be viewed as a task to map the dependence nodes in N-dimensional space onto time axis. A commonly method of scheduling the execution is to draw iso-temporal planes or lines on a 3-D or 2-D dependence graph, respectively. All the nodes fall on the line or plane are scheduled to be executed at the same time instance on the processor space. Note that there are nearly infinitely many combinations for processor assignment and scheduling. Hence, system requirements have to be analyzed and evaluated prior to the implementation of the design aid tool. The design objectives are usually to:

- Minimize the execution time.
- Maximize the utilization rate.

While the overall system goal is achieved, usually there are constraints specified by designers or customers. These constraints, like pre-conditions, have to be satisfied throughout the entire synthesis process. Common constraints are:

- Internal memory inside each cell.

- Communication links among cells.
- Control circuitry.
- Limited processor I/O ports.
- Design time.

1.6 Organization of the Dissertation

Chapter 1 presents an introduction which includes motivations, objectives, and a literature review. It also gives fundamentals of designing application specific arrays with coverage of issues such as architectural model, mapping methods, partitioning techniques, etc. Chapter 2 introduces a new graphical language called Mesh Graph Descriptor (MGD). This language is expressly designed to capture design requirements. Issues involved in synthesizing systolic and pseudo-systolic arrays are covered. Terms such as *grouping* and *view* are also defined. The correlation of view and the descriptors are also elaborated in the same chapter. Chapter 3 deals with one of the powerful characteristics in MGD, i.e., its transformation capabilities. After introducing a key language construct called *transformer*, examples are given to demonstrate how transformers are able to transform the MGD descriptors. A multi-level abstraction concept is also introduced which is a nice feature provided by MGD as well as required by the procedures in the synthesis of processing arrays. Orthographical and diagonal coalescings of the MMG graphs in terms of formalized MGD transformations are addressed. Chapter 4 focuses on the physical issues of mapping procedures. The algorithms to determine scheduling and performance parameters during the synthesis are discussed. How to determine cell functionalities and input arrangement is addressed. Chapter 5 shows how the methodology of ALIAS can be effectively applied to LU decomposition, matrix-matrix multiplication examples, and the Algebraic-Path Problem (APP). A new parsing concept called *inside-out scanning* for index-independent single-assignment algorithm is introduced in an attempt to show how the application algorithm can be interpreted stepwise to create its corresponding 3-D multi-mesh graph. Afterwards, the systolic/pseudo-systolic arrays implementing the same algorithm can be created through systematic projections along appropriate directions. Chapter 6 gives a description of ALIAS system modules, MAP, DM, and DIP. Chapter 7 concludes the dissertation by giving a summary and pointing out the future directions of the research. The appendices include a guide for using ALIAS and the design of this tool.

Chapter 2

A Graph-based Description Language – MGDL and an MGDL-based Transformation System

In this chapter, the definition and structure of Mesh Graph Descriptors Language (MGDL) will be introduced. It is specifically designed to suit our needs in the ALIAS design environment. To automate the synthesis of processing arrays, mesh graphs are manipulated by the software design tool. This symbolic formalism plays a key role in accomplishing one of the principal research objectives, which is to provide a design tool that requires little or no user knowledge on how the final designs can be synthesized.

2.1 Background and Features of MGDL

The Mesh Graph Descriptor Language (MGDL) is a graphical language specifically designed for describing mesh graphs and systolic/application specific arrays. Mesh graphs referred in this work resemble MMG graphs in [Mor89], with two differences being that delay nodes are not used in our work and ϵ nodes (no-op nodes) are added to facilitate transformation. An Multi-Mesh Graph (MMG) is a fully-parallel data-dependence graph in 3-D space with unidirectional arcs. For simplicity, a dependence graphs on 2-D plan also an MMG since it can be treated as a degenerate case of a 3-D MMG. The MMG is produced in the regularization stage of Moreno's graph-based design methodology.

The structure in MGDL demonstrates features such as conciseness, expandability, verifiability, and best of all, transformability. The design of this language was greatly inspired by Functional Programming (FP) [Bac78]. One feature distinguishing FP as a unique language lies in the provision of algebraic laws, which can transform FP programs into behavior-preserving equivalents. There are several hardware specification and design environments based on FP. μFP and νFP are two examples. In [She84], [She85], [LJ88b], [LJ88a], a language called μFP, a variation of FP, is introduced. Many transformation theorems are also provided, based on Leiserson's Retiming rule [LS83] and topological manipulations at the functional block level. The language and theorems are primarily used to derive and retime regular synchronous circuits. νFP extends FP with additional functional forms and primitives [PSE85] [Sch86] [Pat90]. In contrast to μFP, which extends the semantics of FP to operate on streams, the semantics of νFP are the same as those in FP when it is used to specify algorithms. The layout from algorithmic specifications in νFP can be synthesized by the system through functional mappings. The targeted application area proposed in ALIAS is different from νFP, in which examples such as multiplier, FFT, and tally circuit are synthesized. In an attempt to meet the needs in manipulating mesh graphs in our design environment, many useful transformation laws are defined. These laws help derive the processor arrays formally and systematically.

Many similar works are reported which take algorithms as input and generate data dependence graphs accordingly. Through delay manipulation and node merging techniques, these dependency graphs can be turned into array-level implementations that realize the algorithms [JRK87], [Kun88], [Mor89]. None of the above methods provides formal specification and derivation for dependence graph manipulations. The MGDL has been constructed expressly to capture both behavioral and geometrical domains of data dependence graphs. It is a very practical notation in the sense that many of graphic manipulations like rotation, folding, and grouping can be facilitated by *transformers*. In simple words, most of the treatments on the dependence graphs in the literature can be formally specified as a textual language. Therefore, manipulating mesh graphs can be easily done in the language domain without actually touching any of pictorial parts. This methodology not only simplifies the transformation process but also exemplifies a useful underlying tool to automating the procedure of obtaining the processor array implementations.

2.2 Defining MGDL

The elements of the MGDL are introduced below.

Definition 1 (MGDL) *The MGDL is a 4-tuple* $\{N, B, C, T\}$*, where*

- N *is a finite set of* **computation nodes**. *A computation node is associated with a function.*
- B *is a set of six* **composers** $\{\nearrow, \swarrow, \uparrow, \downarrow, \rightarrow, \leftarrow\}$. *They specify both sequentiality in time and directionality in space.*
- C *is a set of four* **constructors** $\{\ (,\ ),\ [,\ ]\ \}$. *The parentheses specify the grouping of nodes in mesh graphs. A pair of brackets specifies a cluster of nodes to be mapped to a processor.*
- T *is a finite set of* **transformers**. *Transformers are in fact functions to perform transformations that map an instance of MGDL to another.*

We now name any instance of MGDL to be MGD. An MGD can be regarded as a program of the language.

Transformers are behavior-preserving, i.e., both pre-applied and post-applied MGDs have common implications on all the parameters in the computational domain. Transformers are imposed on an MGD to obtain another MGD which satisfies design constraints.

Functional form is an expression formed by a node or nodes interconnected by composers and constructors. For instance, a node a is a functional form and $a \rightarrow b$ is another.

Composers in Mesh Graph Descriptors

Three basic composers that connect nodes in a 3-D MMG are shown in Figure 2.1. These composers account for three essential connectivity patterns in any 3-D MMG since there are up to six nodes adjacent to any node and these nodes are connected either along the X, Y, or Z axis. A node is allowed to communicate in one of six directions, $\{X^+, X^-, Y^+, Y^-, Z^+, Z^-\}$. As will be seen in a later section, composers on 2-D plan can be likewise defined.

These composers form a superset of conventional dependency notations for they not only capture the ordering semantics of execution, but they also imply the interconnecting structure in space. Thus both the behavioral and structural domains of the elements in a mesh graph can be characterized by the composer notation.

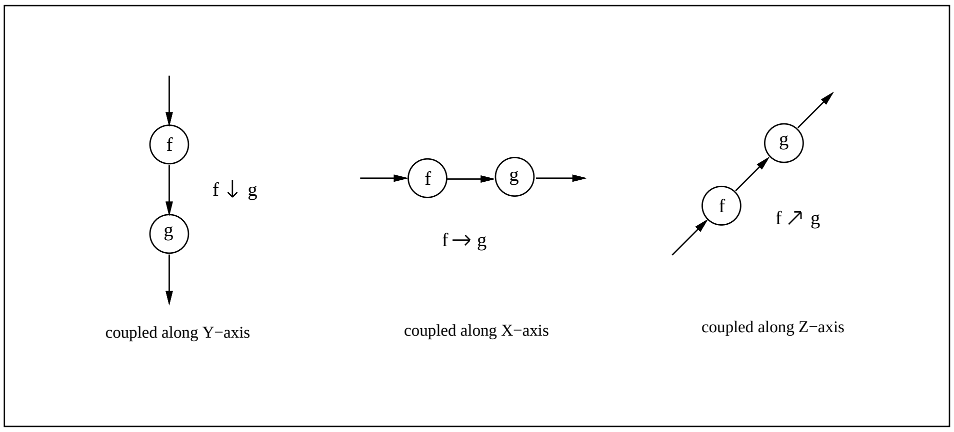

Figure 2.1: Three basic composers in mesh-graph descriptors.

A mesh graph may also be viewed as a lattice of the cube in which partial ordering is implied by the geometric structures of the graph. It may also be regarded, in the behavioral domain, as a dataflow graph since a node cannot be executed (fired) unless all its data inputs particular node are present. However, considering this dataflow analogy would not be so appropriate since in systolic arrays all data arrive at processors synchronously in a predefined way. Dataflow models are more suitable in describing wave-front processing arrays [Kun88].

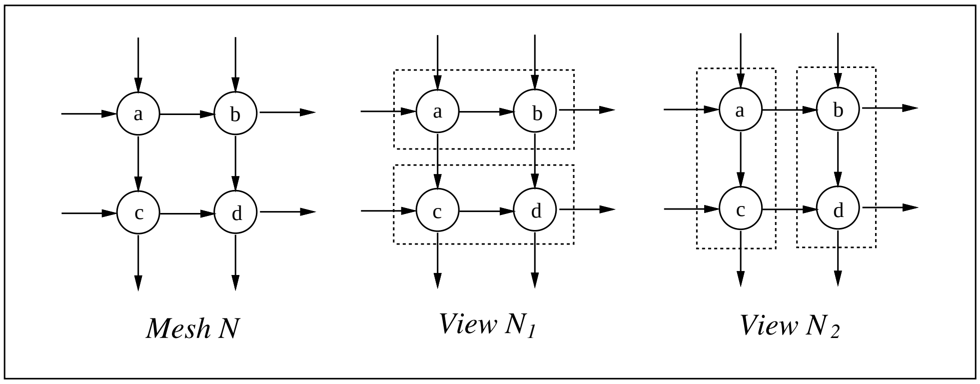

Figure 2.2: A 2-D MMG example.

2.3 Relations between Mesh Graphes, MGDs, Views, and Contours

Definition 2 (View) *A view is a hierarchical and graphical representation of an MMG which is implied by the composition structure of the corresponding descriptors.*

An MMG can have multiple views, which are formally specified by Mesh-Graph Descriptors (MGDs). Now we use the MGDs defined in the previous section to specify mesh graphs. For instance, mesh graph N in Figure 2.2 can be specified by the MGD,

$$(a \rightarrow b) \downarrow (c \rightarrow d),$$

which implies view N_1, or it can be specified by

$$(a \downarrow c) \rightarrow (b \downarrow d),$$

which implies view N_2.

There are two ways to specify this 2-D mesh N with 4 nodes. Since there is usually more than one MGD expression to specify views of any MMG, we need to have a unique way of specifying the MMG.

Definition 3 (Canonical MGD (CMGD)) *A Canonical MGD is the MGD of any MMG with nodes composed along the X-, then Y-, then Z-axis.*

For any view that created from a particular mesh graph, its CMGD is unique. The uniqueness of CMGD can be justified as follows. For any node in a view the ordered set of successor nodes are unique along any X, Y, or Z coordinate. Moreover, since CMGD (and any MGD) always starts from the node which is not a successor of any nodes and the CMGD is composed along X, Y, then Z which is unique, CMGD must be unique. In the above example, the CMGD for Mesh N would be $(a \rightarrow b) \downarrow (c \rightarrow d)$.

Definition 4 (Grouping) *Grouping is an operation of structurally combining nodes together within mesh graphs. Grouping can be orthographical – parallel to coordinates – or nonorthographical. The grouped nodes are mapped onto a physical processor.*

Definition 5 (Projection) *Projection is an operation of mapping all the grouped nodes of an MMG onto a set of of physical processors, with each group corresponding to a processor.*

Grouping is a prerequisite for the operation of projection on mesh graphs. The grouping sizes on one mesh graph (width of the grouping band or prism) need not be equal.

For instance, a mesh graph embedded in 2-D space can be orthogonally grouped along either X- or Y-axis. The grouping of the mesh graph N can be done in the domain of descriptors as the parentheses in the above MGD are replaced by brackets.

There are many nonorthographical groupings. We are interested in orthographical and nonorthographical groupings along bisectors of two coordinates since other groupings, in most cases, result in inefficient designs.

Definition 6 (Contour) *A contour is a* grouped *representation of a Multi-Mesh Graph (MMG) or its descendants which provides an information hiding mechanism for lower level MMG or any graphs derived from MMG with respect to higher operation entities.*

Contours can be obtained from the views of MMG by the grouping operation. To specify mesh graphs in views, all elements in MGDs except constructors "[" and "]" are used since no concept of view is involved in plain mesh graphs. With constructors "[" and "]" placed in the MGD, views can be turned into contours which are suitable for the final parallel array realization. Constructors "(" and ")" in th MGD can be mapped to "[" and "]" respectively by applying transformer function group to the descriptor.

Transforming a 3-D MMG into final design(s) with fewer processor elements in satisfaction of certain performance requirements is the key objective in the design process. Defining different views of MMG graphs by means of descriptors makes it much easier to manipulate complicated MMG graphs. In Figure 2.2, there are two views of the same mesh graph with four nodes, namely N_1 and N_2. Define a Γ mapping function

$$\Gamma : View \rightarrow CMGD.$$

Accordingly, we have

$$\Gamma(N_1) = [a \rightarrow b] \downarrow [c \rightarrow d] \text{ and } \Gamma(N_2) = [a \downarrow c] \rightarrow [b \downarrow d].$$

The CMGD for N_2 has a composition starting along the Y axis instead of, according to the original definition, X because the grouping is along Y. Also Note that FP function "transpose" is applicable to these descriptors, namely, $trans \circ [a \rightarrow b] \downarrow [c \rightarrow d] = [a \downarrow c] \rightarrow [b \downarrow d]$, or simply $\Gamma(V_2) = trans \circ \Gamma(V_1)$.

Definition 7 (Rank) *Rank is the nonnegative integer value associated with the composer. The composers delimited by the innermost constructors have rank 0. The rank increases by 1 for each additional level of constructors.*

In $[a \rightarrow b] \downarrow [c \rightarrow d]$, $\rightarrow$ has rank 0 and $\downarrow$ has rank 1. Rank values range from 0 to 2 since there are at most three dimensions used in practice. A composer with rank value more than 0 implies that there is more than one link between the two functional forms the composer is interconnecting.

V_1 is the view of the four-node MMG suitable for the projection operation along X-axis, whereas V_2 is the view ready for the Y-axis projection. In the descriptor of view V_1, $[a \rightarrow b]$ implies that node a and node b are grouped first and so are node c and node d. Therefore, two new lumped nodes can be generated with one from grouping a and b and the other from grouping c and d. With function *trans*, the descriptor set for the projection along one axis can be easily transformed into another descriptor set for projection along another axis.

For any 3-D MMG graph, the composition may be initiated along any axis. For example, $X \diamond Y \diamond Z$ is the notation for *composition sequence* in

MGD(compose along X then Y then Z). For example, $X \diamond Y \diamond Z$ is the notation for *composition sequence* in MGD (compose along X then Y then Z). Hence, there are 6 ways for specifying the MGD for any 3-D MMG. The following are examples three composition sequence for Mesh M in in Figure 2.3.

1) $((a \rightarrow b) \downarrow (c \rightarrow d)) \nearrow ((e \rightarrow f) \downarrow (g \rightarrow h)) \quad (X \diamond Y \diamond Z)$
2) $((a \nearrow e) \rightarrow (b \nearrow f)) \downarrow ((c \nearrow g) \rightarrow (d \nearrow h)) \quad (Z \diamond X \diamond Y)$
3) $((a \nearrow e) \downarrow (c \nearrow g)) \rightarrow ((b \nearrow f) \downarrow (d \nearrow h)) \quad (Z \diamond Y \diamond X)$.

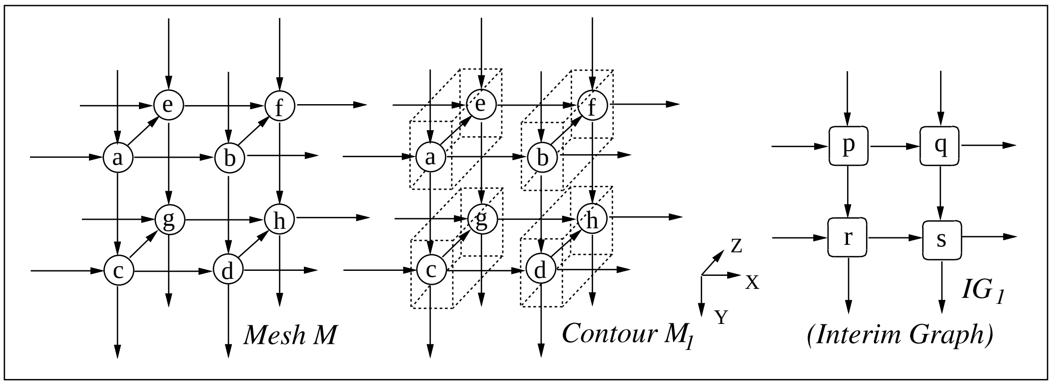

Figure 2.3: A contour of mesh M.

2.4 Power Forms of the MGD

In this chapter MGD has not been applied to real applications. They have been relatively short. There are potentially some sizable MGD corresponding to application examples. It is desirable to alleviate this problem of specification by taking advantage of the characteristics of MMG graphs. Owing to the uniformity of most mesh graphs or their part, a curtailed or folded representation of MGD can be adopted to specify MMG succinctly. In this section we introduce the power form of MMG descriptors. If all the nodes in Figure 2.3 were of type f rather than $a, b, \cdots, h$. the MGD could be shortened in the power form as follows,

$((f \rightarrow f) \downarrow (f \rightarrow f)) \nearrow ((f \rightarrow f) \downarrow (f \rightarrow f))$

$$= (f^{2\rightarrow} \downarrow f^{2\rightarrow}) \nearrow (f^{2\rightarrow} \downarrow f^{2\rightarrow})$$
$$= (f^{2\rightarrow})^{2\downarrow} \nearrow (f^{2\rightarrow})^{2\downarrow}$$
$$= ((f^{2\rightarrow})^{2\downarrow})^{2\nearrow}$$
$$\stackrel{\text{def}}{=} f^{2.2.2}$$

where the last expression denotes that the mesh graph consists of nodes of type f and of dimension $2 \times 2 \times 2$. The power form $((f^{p\rightarrow})^{q\downarrow})^{r\nearrow}$ is denoted as $f^{p.q.r}$.

2.5 Classification of MMG Graphs

Every single-assignment algorithm can be parsed and the MMG graph corresponding to that algorithm can be created by means of the MAP program. Different matrix algorithms lead to different MMG graphs, hence their descriptors. The attributes of the structures of MMG affect the syntax of descriptors in that the descriptors are the textual representations of MMG geometry. It would be advantageous to classify MMG graphs and make an attempt to capture their characteristics prior to the implementation of the physical design system.

Homogeneous This category encompasses the MMG graphs which are strictly uniform. Strict uniformity has the implication that each MMG graph in this family is either linear in 1-D, or rectangular in 2-D (including square), or cubic in 3-D. Furthermore, each node in the MMG performs exactly identical function. The descriptors of these MMG are shortest because they can be reduced into power forms. The MMG facilitating matrix-vector and matrix-matrix multiplication, for instance, fit into this category.

Quasi-homogeneous/scalable The uniformity of the MMG structures is somewhat relaxed to an extent with respect to homogeneous type. The forms of this type of MMG in 2-D are polygons and in 3-D are polyhedrons. In 3-D for instance, they can be viewed as the results of a cube under one or many skewed cuts. The MMG structure is not strictly uniform; however, they are scalable in the sense that a much larger mesh graph can be reproduced simply by examining and replicating the mesh graph in a smaller scale. The descriptors for MMG identified hereby can also be reduced into their power form, although not so concise as those in preceding category. As a result, the descriptors do not grow way out of hand as the magnitude of the mesh graphs increases. Examples of algorithms are QR-decomposition, LU-decomposition, Fadeev algorithm, and so forth.

Non-homogeneous No single trace can be found about the uniformity for non-homogeneous matrix algorithms. The descriptors by no means can be written in power form, i.e., they have to be completely unrolled. This gives rise to a disadvantage of having lengthy descriptors which are in proportion to the corresponding mesh graphs. One conceivable example is the multiplication of sparse matrices.

2.6 MMG Regularization

So far in the examples all the given MMGs are of homogeneous type. A potential question is how to characterize MMGs that are of irregular configuration. Instead of applying brute force in augmenting MGD syntax, we adopt an alternate approach, namely, we patch up the irregular MMGs and make them look regular by adding dummy (no-op) nodes, denoted by ϵ.

Definition 8 (ϵ node) *An ϵ node is an identity (id) node to be distinguished from other computational nodes. If an input λ is seen along direction $\mathcal{R}$, $\mathcal{R} \in \{X^+, X^-, Y^+, Y^-, Z^+, Z^-\}$ at time t, λ will also be the output seen at time t along the same direction $\mathcal{R}$.*

From the above definition, it is understood that ϵ can be thought of as a no-delay node that is between delay node D and anti-delay node D^{-1}. In addition, the incoming data to an identity node leaves the node without altering its direction.

Definition 9 (Regularization) *A regularization is a process of adding the minimal number of ϵ nodes to a non-homogeneous MMG so that the newly derived MMG is homogeneous. The newly derived MMG is called the regularized MMG (RMMG).*

An MMG regularization example is illustrated in Figure 2.4. The same procedure of adding these ϵ nodes can be applied to three-dimensional mesh graphs as well. For the sake of clarity, two-dimensional example is shown. This is similar to adding zeroes to a sparse matrix. These dummy nodes help hold the regular computational nodes in place during transformations. With the help of these seemingly excessive nodes, MMG of any type and any shape can be characterized by the MGD and thus are amenable to systematic transformations, including projections, rotations, transpositions, and mirrorings.[1].

Incidentally, not all 3-D embedded graphs can be regularized without altering its connectivity semantics. In here, 3-D embedded graphs refer to any graphs that are formed by nodes interconnected with arcs (no dangling nodes) and should be a superset of the MMG graphs generated by the matrix algorithms. If there was a node, say h right on the top of node e in part (a) of Figure 2.4, the regularized graph for (a) would have implied that the eastbound output of node b was fed into node h according to the definition of ϵ. Fortunately, none of the matrix computation algorithms we have seen in the literature implies similar shape like this. From our understanding, the dependence graphs that realize matrix algorithms are of homogeneous and quasi-homogeneous types. For those quasi-homogeneous MMGs, as pointed out in previous section, they can be visualized as a resultant graph after cutting a cube at skewed angles. This leads to a fact that the nodes on the boundary along one direction are *monotonously increasing* and therefore, regularizations can be justified as a *connectivity-preserving* process on MMG graphs.

A corresponding MGD of the MMG in (b) of Figure 2.4 is as follows,

$$(a \to b \to \epsilon \to \epsilon) \downarrow (\epsilon \to c \to d \to e) \downarrow (\epsilon \to \epsilon \to f \to g).$$

Without *regularization*, our work would have been more difficult since some certain class of MMG graphs cannot be properly specified by MGD, or the MGD syntax would have been extended to describe the big MMG class consisting of different types.

[1]Transformations of MMGs will be covered in the next chapter.

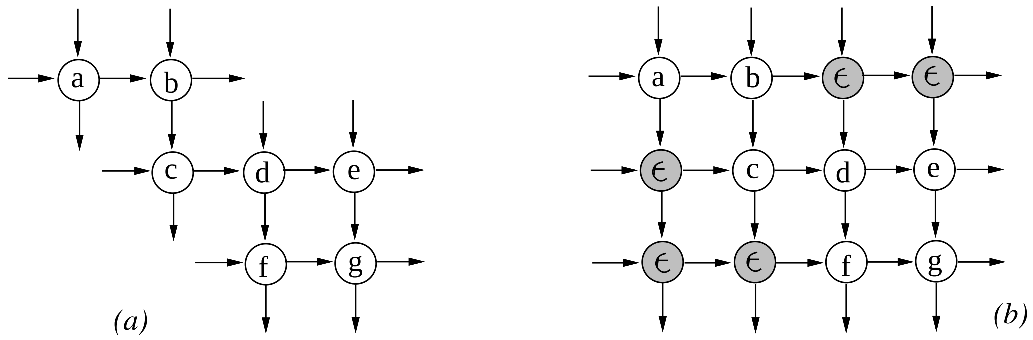

Figure 2.4: (a) A non-homogeneous MMG, (b) the corresponding regularized MMG.

2.7 An Algorithm Transformation System

ALIAS automated design system is based on the MGD, the graph-based description language proposed in this chapter. The relationships between the MGD and ALIAS will be discussed in detail in this section.

Definition 10 (Algorithm Transformation System) *An algorithm transformation system is an 8-tuple* $\{\mathcal{A}, \mathcal{G}, \sigma, \mathcal{D}^*, \Gamma, \mathcal{V}, \delta, \mathcal{P}\}$*, where*

- $\mathcal{A}$ *is a finite set of* **matrix algorithms** *in their single-assignment form.*
- $\mathcal{G}$ *is a finite set of* **mesh graphs** *which are data-dependence graphs of* $\mathcal{A}$*.*
- σ *is a* **transition function** *mapping* $\mathcal{A}$ *into* $\mathcal{G}$*.*
- $\mathcal{D}^*$ *is a finite set of* **MGD***, and* $\mathcal{D}$ *refers to the corresponding set of CMGD.*
- Γ *is a* **transition funtion** *mapping* $\mathcal{G}$ *or* $\mathcal{V}$ *into* $\mathcal{D}$*.*
- $\mathcal{V}$ *is a finite set of* **views***.*
- δ *is a finite set of* **transition functions** *mapping* $\mathcal{D}$ *into* $\mathcal{D}$*.* **Grouping function** g*, a member of* δ*, maps* $\mathcal{D}$ *into* $\mathcal{P}$*.*

- $\mathcal{P}$ *is a finite set of* **parallel arrays**.

From the above definition, we have the following properties that hold for the algorithm transformation system.

1. Since $\sigma : \mathcal{A} \rightarrow \mathcal{G}$, therefore $\forall \mathcal{A}_i \in \mathcal{A}$, $\exists \mathcal{G}_i \in \mathcal{G} \Longrightarrow \mathcal{G}_i$ realizes $\mathcal{A}_i$.
2. $\forall \mathcal{G}_i \in \mathcal{G}$, $\exists \mathcal{D}_i$ in $\mathcal{D}$ that specifies $\mathcal{G}_i$. $\forall \mathcal{G}_i \in \mathcal{G}$, $\exists!$ $\mathcal{D}_j \in$ CMGD that specifies $\mathcal{G}_i$.
3. The inverse funtion of $\Gamma : \mathcal{V} \rightarrow \mathcal{D}$ exists, i.e., $\Gamma^{-1} : \mathcal{D} \rightarrow \mathcal{V}$.
4. $\mathcal{V}_1 = \Gamma^{-1}(\delta_1(\Gamma(\mathcal{G}_i))), \mathcal{V}_2 = \Gamma^{-1}(\delta_2(\Gamma(\mathcal{G}_i)))$, ..., $\mathcal{V}_n = \Gamma^{-1}(\delta_n(\Gamma(\mathcal{G}_i)))$, $\{\delta_1, \ldots, \delta_n\} \in \delta$, $\{\mathcal{V}_1, \ldots, \mathcal{V}_n\} \in \mathcal{V}, n \geq 1$.
5. $\mathcal{V}_j = \Gamma^{-1}(\delta_i(\Gamma(\mathcal{V}_i))), \{\mathcal{V}_i, \mathcal{V}_j\} \in \mathcal{V}, \delta_i \in \delta$.
6. Define a function $\Theta : \mathcal{D}^* \rightarrow \mathcal{G}$, then we have $\Theta(\delta_i(\Gamma(\mathcal{G}_i))) = \Theta(\delta_j(\Gamma(\mathcal{G}_i)))$, for all i, j.
7. The synthesis process by the transformation system can be characterized by

$$
\begin{aligned}
&\mathcal{P}_1 = \Gamma^{-1}(g(\delta_{1n} \cdots \delta_{11}(\Gamma(\sigma(\mathcal{A}_k))) \cdots)) \\
&\mathcal{P}_2 = \Gamma^{-1}(g(\delta_{2n} \cdots \delta_{21}(\Gamma(\sigma(\mathcal{A}_k))) \cdots)) \\
&\cdots\cdots \\
&\mathcal{P}_n = \Gamma^{-1}(g(\delta_{nn} \cdots \delta_{n1}(\Gamma(\sigma(\mathcal{A}_k))) \cdots)),
\end{aligned}
$$

   where $\{\mathcal{P}_1, \ldots, \mathcal{P}_1\} \in \mathcal{P}$, $\{\delta_{11}, \ldots, \delta_{nn}\} \in \delta$, and $\mathcal{A}_k \in \mathcal{A}$.

Property 1 states that for every matrix algorithm there exists at least one mesh graph which is a data dependence graph corresponding to the algorithm. Property 2 is equivalent to say that for every mesh graph a least a mesh graph descriptor can be found to be its specification. Also, there exists a unique mesh graph descriptor – canonical mesh graph descriptor – for every mesh graph. Property 3 asserts that the inverse function of Γ exists, i.e., the mapping between view and CMGD is bijective. Property 4 claims that a number of views of a dependence graph can be created by applying different transformation functions. Property 5 says that transformation functions can also map one view to another. Property 6 argues that the application of transformation functions in δ is structure-preserving, i.e., the transformed descriptors (thus views) still refer to the same original mesh graph. This is important since we need to be assertive that the mechanism of our transformation system guarantees that the synthesized parallel arrays can correctly implement the given algorithms as the input to the system. Property 7 gives a view of the synthesis process in terms of equations. Various arrays can be obtained by applying successive transformers to the MGD which are notational representations of the input algorithm.

Figure 2.5 illustrates the model of how ALIAS system symbolically translates one view into another by applying mapping functions Γ, δ, and Γ^{-1}.

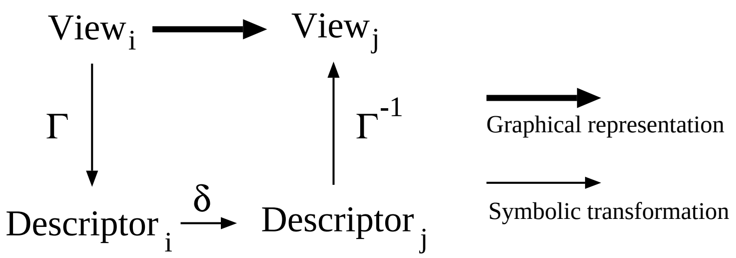

Figure 2.5: How the synthesis system internally transforms $View_i$ to $View_j$

2.8 Summary

In this chapter we introduced a graph-based description language, mesh graph descriptor. The design of MGD shares properties of formalism and simplicity found in functional programming. MGD was made intentionally concise in an effort to accomplish the synthesis task only. The four elements in MGD are sets of *nodes, composers, constructors*, and *transformers*. *Functional form* is defined as an expression formed by a node or nodes interconnected with composers and constructors. The correspondence between views and MGD is also discussed. Also the model of our synthesis system ALIAS is described in formalism. The properties of the system are discussed in detail.

Chapter 3

MGDL Transformations

In the previous chapter, the graph language MGDL was introduced. Now we try to identify some properties of MGDL. Examples will be shown such as how an MGD can be transformed, how multi-level abstraction can be achieved, how orthographical/diagonal coalescing can be done, and how a complicated MGD can be combined to a reduced concise form. Attempts will be made in this chapter to categorize the MMG graphs and the relationships between these graphs and the graph descriptors will be addressed.

3.1 Transformers

During the process of obtaining the designs of the systolic or pseudo-systolic arrays [Mor89] that realize the goal algorithms, different kinds of operations are needed to convert the preliminary MMG into various "offsprings". The MGDL constructs that do these operations are *transformers*.

The *trans* function brought up in the previous chapter is one of MGDL transformers. The *mirror* function is another type of transformers, denoted as $\bowtie$. $\bowtie$ takes one parameter as the direction of mirroring (reflection). For one case,

$$\bowtie_X\colon (a \rightarrow b) \downarrow (c \rightarrow d) \equiv (a \leftarrow b) \downarrow (c \leftarrow d),$$

as shown in (a) of Figure 3.1.

Mirror function also maps different MGDs of an MMG to the expression that specify the same MMG. A clarifying example follows. Let

$$D_1 = ((a \rightarrow b) \downarrow (c \rightarrow d)) \nearrow ((e \rightarrow f) \downarrow (g \rightarrow h))$$

and

$$D_2 = ((a \nearrow e) \downarrow (c \nearrow g)) \rightarrow ((b \nearrow f) \downarrow (d \nearrow h)).$$

If we define a mapping function Θ

$$\Theta : \mathit{Descriptor} \rightarrow \mathit{MMG},$$

we have $\Theta(D_1) = \Theta(D_2)$ in the sense that they both refer to the same MMG. Let $D_3 = \bowtie_X : D_1$ and $D_4 = \bowtie_X : D_2$. Then we have

$$D_3 = ((a \leftarrow b) \downarrow (c \leftarrow d)) \nearrow ((e \leftarrow f) \downarrow (g \leftarrow h))$$

and

$$D_4 = ((a \nearrow e) \downarrow (c \nearrow g)) \leftarrow ((b \nearrow f) \downarrow (d \nearrow h)).$$

$\Theta(D_4) = \Theta(D_3)$ also holds since both of them denote the same computational graph which is the mirroring image of the graph specified by D_1 and D_2. Also applying twice the same mirroring function to an MMG results in the same MMG. Notationally, this can be expressed as $V \equiv \bowtie \circ \bowtie : V$, where V is a *view*.

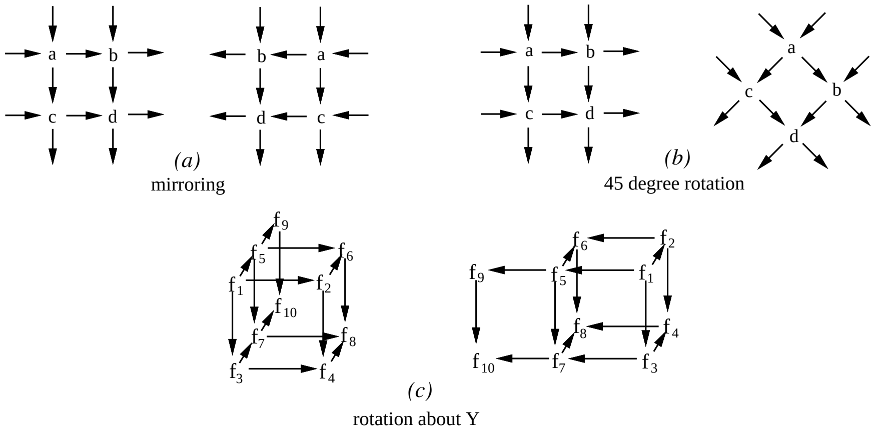

Figure 3.1: How transformers transform MMG.

We are to see a list of transformer examples. In addition to those, in a later section there are some others to be introduced which are evolved from FP (functional programming) [Bac78]. In the following transformers specifications, c_1 and c_2 are composers, $\Longrightarrow$ denotes left-to-right mapping, and $\Longleftrightarrow$ denotes the bidirectional mapping (either left mapped to right or vice versa).

$$
\begin{array}{rcl}
trans & ::= & (\alpha\ c_1\ \beta)c_2(\gamma\ c_1\ \delta) \Longleftrightarrow (\alpha\ c_2\ \gamma)c_1(\beta\ c_2\ \delta) \\
\bowtie_X & ::= & \rightarrow \Longleftrightarrow \leftarrow \\
\bowtie_Y & ::= & \downarrow \Longleftrightarrow \uparrow \\
\bowtie_Z & ::= & \nearrow \Longleftrightarrow \swarrow \\
\theta_{XY} & ::= & c \Longrightarrow c_{\theta^\circ\ clockwise} \\
90^\circ_Y & ::= & \{\rightarrow, \nearrow, \leftarrow, \swarrow\} \Longrightarrow \{\nearrow, \leftarrow, \swarrow, \rightarrow\} \\
90^\circ_X & ::= & \{\downarrow, \nearrow, \uparrow, \swarrow\} \Longrightarrow \{\nearrow, \uparrow, \swarrow, \downarrow\} \\
90^\circ_Z & ::= & \{\rightarrow, \downarrow, \leftarrow, \uparrow\} \Longrightarrow \{\downarrow, \leftarrow, \uparrow, \rightarrow\}
\end{array}
$$

The above representation of transformers adopts the concept of text matching and rewriting. Once a transformer is to be applied to an MGD specification, all text lines contained in that particular MGD will be invariably affected and converted by the specifications.

The lines of the transformer specifications with angles are of *rotator* type. These are functions which rotate the composers in the MGD and meantime, coincidentally, turn the corresponding MMG around. The line with θ specifies 2-D MMG rotation on XY plane. One illustrative case in (b) of Figure 3.1 is that

$$45^\circ_{XY} : (a \rightarrow b) \downarrow (c \rightarrow d) \equiv (a \tilde{\searrow} b) \tilde{\swarrow} (c \tilde{\searrow} d),$$

in which the *tilde* symbol is used to denote planar composer as opposed to 3-D composers.

The specification for 90°_Y, for instance, means that if an MMG needs to be rotated about Y axis for ninety degrees, four arcs $\{\rightarrow, \nearrow, \leftarrow, \swarrow\}$ existing in that particular MMG have to be mapped to $\{\nearrow, \leftarrow, \swarrow, \rightarrow\}$. Rotating –90 degrees, 180 degrees or larger angles can be easily respecified or rederived from current specifications. The left MMG shown in part (c) of Figure 3.1 has the descriptor of

$$((f_1 \rightarrow f_2) \downarrow (f_3 \rightarrow f_4)) \nearrow ((f_5 \rightarrow f_6) \downarrow (f_7 \rightarrow f_8)) \nearrow (f_9 \rightarrow f_{10}).$$

Let it be denoted as D_p. If $D_q = 90^\circ_Y : D_p$, then we have

$$D_q = ((f_1 \nearrow f_2) \downarrow (f_3 \nearrow f_4)) \leftarrow ((f_5 \nearrow f_6) \downarrow (f_7 \nearrow f_8)) \leftarrow (f_9 \nearrow f_{10}),$$

which is shown as the MMG on left of part(c) in Figure 3.1.

Justified by the above examples, one evident virtue of adopting the MGD along with transformers is that, without examining any part of MMG's pictorial representations, the MGD can be modified by software internally to achieve various topological manipulations. As we see, the manipulations are done in a coordinate-free manner. This is one of the distinguished features in ALIAS. The transformers are *composable*, i.e., they can be successively applied to the MGD and the resultant MGD is the corresponding description of the MMG after these successive operations.

Incidentally, in the real ALIAS implementation, the composers are internally represented in symbolic form. They appearance in the text as the arcs is intended for promoting legibility. $\{\rightarrow, \downarrow, \nearrow\}$ are textually coded as {.ea.,.so.,.ne.}. Another appropriate representation of greater expressive power would be to code them as $\{\vec{ea}, \vec{so}, \vec{ne}\}$.

3.2 Multi-level Abstraction

Multi-level abstraction is a highly desirable feature in any synthesis environments. It is also provided in νFP design environment [PSE85]. A νFP algorithm can be mapped into a structure corresponding to a combinational network by passing symbolic inputs to functions which in turn generate symbolic outputs. The unit of information represented by a symbolic atom can be anything corresponding to the level of abstraction. By tagging the appropriate functions, the layout may be generated at any desired hierarchical level. A layout of a full adder using either *and, or, xor* gates or composed of *half adders* is one example of multi-level abstraction.

Another example of multi-level abstraction can be found in [LJ88a], in which regular synchronous circuits are derived through applying divide-and-conquer method on the hardware specifications. The method decomposes the specification into components which can be implemented by known architectures, and combine the architectures by circuit combinators. Circuits are transformed by substitution of equals using theorems that are FP-like. *Data refinement* is a proposed technique that allows *bit level* architecture to be developed from the corresponding *word level* architectures.

The mesh descriptors being proposed also have property for the users to view the mesh graphs from different levels. This characteristic also allows the

ALIAS system to synthesize the target arrays based on different projection angles or under the constraint of limited implementation components.

3.3 Coalescing

The MMG coalescing is basically a process of lumping nodes and thus turns the MMG being examined into another with fewer nodes and possibly of lower dimension. In this section, we will see some more examples as how to do orthographical coalescing, namely, lumping nodes along either X, Y, or Z axis, and diagonal coalescing, namely, lumping nodes along any axis that is a bisector of any two or any three of the X, Y, or Z axis. We introduce the concept of coalescing because it is a very common case and is desirable, in a typical synthesis problem, to combine the computational nodes into fewer nodes which can be physically mapped to processor elements. In a physical design problem, the size of the input matrices is usually too large to fit into the limited available implementation resources. Therefore, a good design approach should be able to wisely break down the enormous data dependence graph and to balance the mapping to the processors so that design goals like minimal turnaround time under implementation resource constraint can be achieved. To fulfill these requirements, the possibilities of mapping computational nodes implied by the algorithm should be explored. It is a rule of thumb that, by the process of coalescing nodes along three coordinate axes and diagonal directions, optimal designs can be synthesized although coalescings along other directions also lead to design possibilities.

3.3.1 Orthographical Coalescing

With the attribute of multi-level abstraction in our notation, contour M_1 in Figure 2.3 can be redefined as

$$\Gamma(M_1) = (p \rightarrow q) \downarrow (r \rightarrow s), \text{ where}$$

$$p \stackrel{\text{def}}{=} a \nearrow e \quad q \stackrel{\text{def}}{=} b \nearrow f \quad r \stackrel{\text{def}}{=} c \nearrow g \quad s \stackrel{\text{def}}{=} d \nearrow h.$$

Another contour of mesh M can be created by lifting the contour in Figure 2.3 to a higher level. For instance, $\Gamma(M_2) = v \rightarrow w$ is another contour of MMG M, where

$$v \stackrel{\text{def}}{=} (a \downarrow c) \nearrow (e \downarrow g) \qquad w \stackrel{\text{def}}{=} (b \downarrow d) \nearrow (f \downarrow h).$$

Contour M_2 is illustrated in Figure 3.2. Note that the composition sequence for the canonical MGD is followed. The descriptors are composed along the X-, then Y-, then Z-axis.

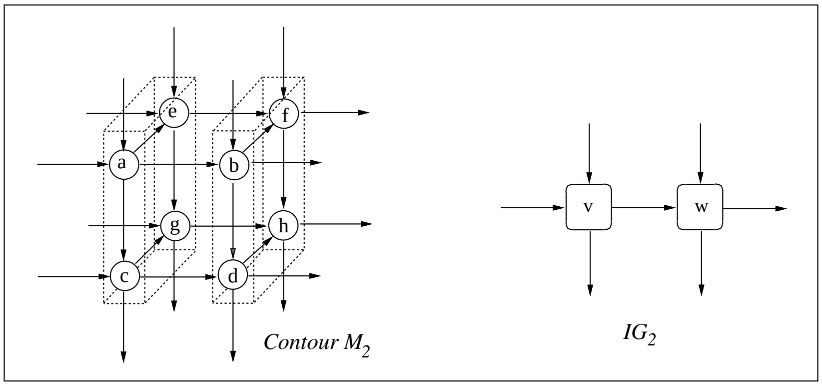

Figure 3.2: Yet another contour of mesh M.

A method of deriving mesh arrays for matrix computation is proposed in [Mor90]. This is a data-dependency graph-based transformational method consisting of two stages: algorithm regularization and derivation of arrays. In the this process of array derivation, projection along X, Y, or Z axis (orthographical coalescing) is employed to reduce the size of MMG and, thereby, arrays for implementation can be obtained. Creating different views on MMG not only produces multi-level abstraction, but provides a sound theoretical groundwork for all orthographical coalescings. These potential coalescings (projections) can be applied in sequence to a particular MMG until an as-small-as-desired MMG is obtained.

Definition 11 (Interim Graph) *An Interim Graph (IG) is a coalescent graph from a larger-scale view or another larger-scale interim graph, with fewer number of nodes and/or less dimensionality with respect to its ancestor.*

The shrinking process of the sizes of the IG can lead to the physical array designs as a result of limitation from implementation costs. As will be seen from examples in latter sections, a problem usually starts from a 3-D mesh graph and gradually reduce its sizes through grouping process until a array with desired size is obtained.

It would be interesting to see how *quasi-homogeneous* MMGs can be manipulated by our methodology. Now we take a practical dependence graph from a band matrix and vector multiplication example and do both the orthographical and diagonal coalescings. In Figure 3.3, the dependence graph is shown (a), as well as the *regularized* graph (b). (c) is the coalesced graph from the regularized graph. The ϵ nodes inserted in (b) are intended to make the graph homogeneous and do not imply any computations within. They are meant to be retained throughout the entire transformation process and to be removed at the stage where the design is to be extracted from the MGD. It is an analogy that the ϵ nodes are like the polyethylene balls inside a box that hold the object in place so that it won't tilt. The ϵ nodes should not be deleted from the MGD, therefore, the sequence of scheduling of the design can still be implied by the descriptors. To see clearly, we do not distinguish the computation nodes and label them as c in the following MGD of the regularized dependence graph,

$$(c \rightarrow c \rightarrow \epsilon \rightarrow \epsilon) \downarrow (c \rightarrow c \rightarrow c \rightarrow \epsilon) \downarrow (\epsilon \rightarrow c \rightarrow c \rightarrow c) \downarrow (\epsilon \rightarrow \epsilon \rightarrow c \rightarrow c)$$

3.3.2 Diagonal Coalescing

As pointed out in a previous paragraph, diagonal coalescings often result in efficient designs. It is interesting to see how orthographical coalescings can be done through MGDs. However, projections along any nonorthographical direction (diagonal for instance) are not as straightforward. An example will be given shortly on how this new type of projection can be dealt with by our method.

An MMG is shown in part (a) of Figure 3.4. Let the corresponding MGD be denoted as D_r. We have

$$\begin{aligned} D_r &= ((g_1 \rightarrow g_2) \downarrow (g_3 \rightarrow g_4)) \nearrow ((g_5 \rightarrow g_6) \downarrow (g_7 \rightarrow g_8)) \nearrow \\ &\quad ((g_9 \rightarrow g_{10}) \downarrow (g_{11} \rightarrow g_{12})). \end{aligned}$$

It is assumed that the diagonal coalescing to be performed is along the direction of $X^+Y^+Z^+$. In this case, nodes g_1 and g_5 would be combined with

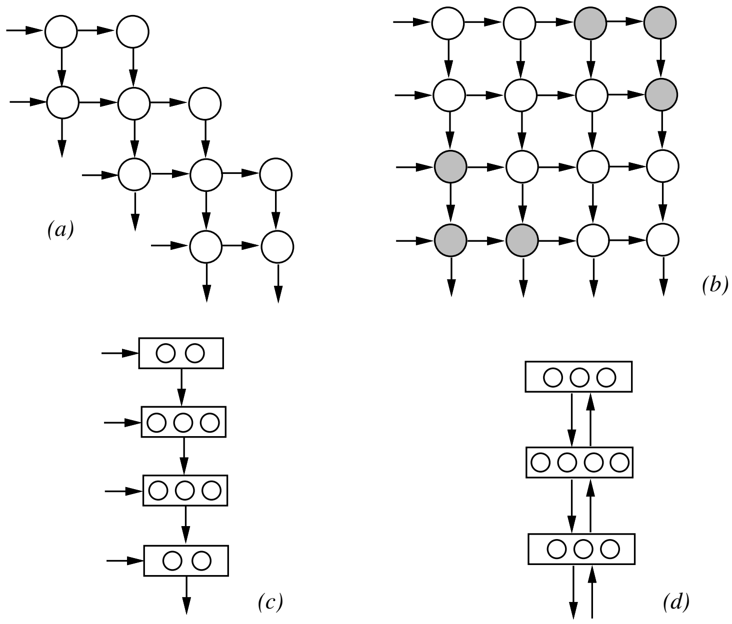

Figure 3.3: Orthographical and diagonal coalescings on a band matrix and vector multiplication example: (a) the original data dependence graph, (b) the regularized graph, (c) coalescing along X^+ axis, (d) coalescing along X^+Y^+ axis.

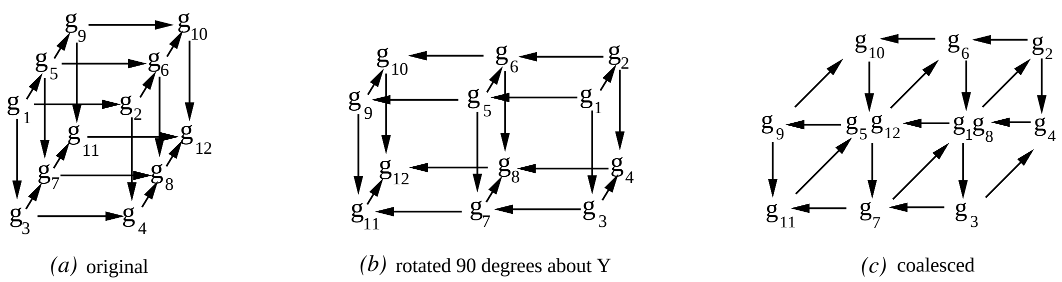

Figure 3.4: Diagonal coalescing.

nodes g_8 and g_{12}, respectively. In an effort to see without obstruction on the design screen, D_r has to be rotated 90 degrees about Y axis before any coalescing operations are done. Let D_s denote the description after rotation. Then

$$\begin{aligned} D_s &= ((g_1 \nearrow g_2) \downarrow (g_3 \nearrow g_4)) \leftarrow ((g_5 \nearrow g_6) \downarrow (g_7 \nearrow g_8)) \leftarrow \\ &\quad ((g_9 \nearrow g_{10}) \downarrow (g_{11} \nearrow g_{12})), \end{aligned}$$

which corresponds to (b) of Figure 3.4.

To facilitate the merging of MMG nodes along a diagonal, a new term *grouping expression* is defined to be a sequence of composers that are used to traverse MGD descriptors so that nodes along that diagonal can be merged. A simple translation from the direction of coalescing into grouping expression can be easily found. The grouping expression for projection direction of $X^+Y^+Z^+$ would be $|\rightarrow^n\ !\ \downarrow^n\ !\ \nearrow^n|$. Note the arcs are generated by a straightforward mapping. The "!" notation denotes random permutations among members they delimit. What the previous grouping expression absorbs (accepts) is strings of any permutations of these three arcs with the number of each type being equivalent to n. A simple and clarifying case is that the string set absorbed by $\mid a^2!b \mid$ is $\{aab, aba, baa\}$ [1]. To enhance the visibility of the MMG, it has been turned 90 degrees. Accordingly, the new grouping expression should be $|\leftarrow^n\ !\ \downarrow^n\ !\ \nearrow^n|$.

Now the synthesis system may use the grouping expression as a tool to locate all the nodes that should be grouped together. It starts from g_1 in this case. Since g_8 is the only one found, it should be grouped with g_1. The synthesis system continues to scan the next untraced node which is g_2. All the nodes in the MGD ought to be traced so that the coalesced MMG may be created. The traced MGD is

$$(([g_1, g_8] \tilde{\nearrow} g_2) \downarrow (g_3 \tilde{\nearrow} g_4)) \leftarrow (([g_5, g_{12}] \tilde{\nearrow} g_6) \downarrow (g_7 \tilde{\nearrow} [g_1, g_8])) \leftarrow ((g_9 \tilde{\nearrow} g_{10}) \downarrow (g_{11} \tilde{\nearrow} [g_5, g_{12}])),$$

which corresponds to the coalesced MMG in (c) of Figure 3.4. Note that this MMG is on a 2-D plane. Again the order of applying rotation and grouping won't affect the resultant MMG. Rotating the MMG which has already been grouped from the original would give the same result.

[1] Grouping expressions are not only effective on orthographical/diagonal coalescing, but also they are amenable to coalescing at any other angles.

Let's revisit the band matrix and vector multiplication example in the previous section. To make a X^+Y^+ direction coalescing, the grouping expression would be $|\rightarrow^n ! \downarrow^n|$. To illustrate this operation, the previous MGD is reproduced here with c nodes labeled for identification.

$(c_1 \rightarrow c_2 \rightarrow \epsilon \rightarrow \epsilon) \downarrow (c_3 \rightarrow c_4 \rightarrow c_5 \rightarrow \epsilon) \downarrow (\epsilon \rightarrow c_6 \rightarrow c_7 \rightarrow c_8) \downarrow (\epsilon \rightarrow \epsilon \rightarrow c_{10} \rightarrow c_{11})$

Again, the grouping expression can be used as a tool to locate groups of nodes, each of which corresponds to a processor node. Similarly, the traced MGD will be

$([c_1, c_4, c_7, c_{10}] \tilde{\nearrow} [c_2, c_5, c_8] \tilde{\nearrow} [\epsilon, \epsilon] \tilde{\nearrow} \epsilon) \tilde{\swarrow} ([c_3, c_6, c_9] \tilde{\nearrow} [c_1, c_4, c_7, c_{10}] \tilde{\nearrow} [c_2, c_5, c_8] \tilde{\nearrow} [\epsilon, \epsilon]) \tilde{\swarrow} ([\epsilon, \epsilon] \tilde{\nearrow} [c_3, c_6, c_9] \tilde{\nearrow} [c_1, c_4, c_7, c_{10}] \tilde{\nearrow} [c_2, c_5, c_8]) \tilde{\swarrow} (\epsilon \tilde{\nearrow} [\epsilon, \epsilon] \tilde{\nearrow} [c_3, c_6, c_9] \tilde{\nearrow} [c_1, c_4, c_7, c_{10}])$,

in which $\tilde{\nearrow}$ and $\tilde{\swarrow}$ are, respectively, the transformation of the the original $\rightarrow$ and $\downarrow$ arcs affected by the projection angle X^+Y^+. This MGD form is not the final form in specification of the coalesced graph since it is not structurally linear. After pruning out the ϵ nodes and redundant expressions, the resultant MGD is

$[c_3, c_6, c_9] \tilde{\nearrow} \tilde{\swarrow} [c_1, c_4, c_7, c_{10}] \tilde{\nearrow} \tilde{\swarrow} [c_2, c_5, c_8]$,

which corresponds to part (d) in Figure 3.3.

Note that the diagonal coalescing schemes are bisectorial with respect to the coordinates. There can exist non-bisectorial (non-45°) coalescing directions. Arrays of different interconnecting pattern can be generated accordingly. However, as Figure 3.5 suggests, two adjacent nodes in MMG may be mapped to non-adjacent processors. As a result, the data token to be passed from node p has to get across another processor in order to arrive at node q. Or else, to avoid the crossing to happen, another separate link has to be created between the two processors where p and p are allocated.

Skewed diagonal coalescing schemes in three dimensional case result in extra cost in communication. An obvious example can be found in [JRK87], [JK89], where the MMG from matrix-matrix multiplication, if taken skewed diagonal projections, may contribute to planar processor arrays with complicated interconnecting patterns. Although these patterns seem to look regular, the implementation costs would be much higher than arrays derived from orthographical and strictly diagonal projections. In our work, for that reason, we only consider both orthographical and diagonal coalescings.

Projection technique can be applied more than once to some MMGs. It is straightforward to see that the projection image from a 3-D MMG is

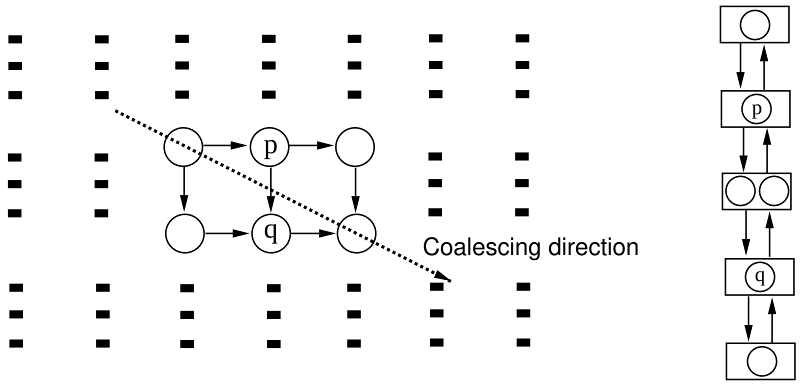

Figure 3.5: Skewed diagonal coalescing.

planar, whereas projecting a 2-D MMG would result in a linear structure. In the design stages for some applications, applying the projection once may not provide a satisfactory design, mostly due to very high implementation cost. *Multi-projection* is a technique to overcome this difficulty by, during design process, compressing the MMG structure more than once. Not all MMGs may be suitable for multi-projection. Conflicts exist in cases in which timing and clocking problem cannot be resolved by newly projected graph. Should the further projection be allowable, the MGD would be in effect along the process since it possesses the property of muti-level abstraction and is amenable for projection along arbitrary directions.

3.3.3 Some Remarks on Coalescing Directions

In preceding sections we mentioned two types of coalescings: orthographical and diagonal. The reason of grouping nodes in MMG to form another graph is that, when the new graph being mapped to parallel processors, parallelism can be achieved and the implementation cost can be reduced. Hence, as various coalescent directions being worked out, we should try to apply only practical directions. Some directions of coalescing are trivial since they contribute to arrays with very little or no parallelism. It is mentioned in Chapter 1 that the two key issues in mapping algorithms into parallel arrays are processor assignment and scheduling. What is considered here is an effect of the scheduling imposed upon the processor assignment.

Taking the scheduling issue into account, we can draw iso-temporal lines (planes) across the primitive nodes in the MMG. All the nodes marked by a particular line (plane) will be executed at the same time step. Generally speaking, coalescing directions parallel to these lines (planes) are undesirable since same-execution-time nodes will be coalesced into one processor and then the parallelism is lost. Therefore, orthographical coalescings usually are useful mappings from which parallel arrays can be created. This is because the iso-temporal lines (planes) are in general not parallel to the composition directions. Conceivably, diagonal coalescings may not always lead to useful parallel arrays and should be avoided.

3.4 DM Module

DM (Descriptor Manipulator) module in ALIAS consists of many rules which resemble FP programs. Its objective is to perform transformations on the MGD to obtain a more suitable structure for implementation. By examining the current structure of the MGD, DM determines the appropriate rule to apply and repeats this until the desirable MGDs are synthesized (which satisfy the given design constraints specified by the on-line user). We believe that a systematic and formalized rule-based system provides ALIAS with robustness in transformation. We also believe it is relatively error-free in synthesis of hardware designs as opposed to a more ad hoc approach.

It would be interesting to review some concepts and properties regarding functional programming proposed by Backus [Bac78] prior to the introduction of rules in DM.

FP is a member in the functional languages family and it is the one with most succinct syntax and powerful transformations. To accomplish a particular programming task, a verbose narration is usually inevitable if an imperative language is used. In FP, however, it can be simplified to a great extent.

In FP, "programs" are simply functions without variables. An FP system is founded on the use of a fixed set of combining forms called *functional forms*. Functional forms along with concise definitions serve as the only means of building new functions from existing ones; they use no variables or substitution rules. All the functions of an FP system are of one type: they map objects and always take a single argument [Bac78].

A concrete example will be helpful in realizing the advantages of functional programming style. Not like tedious loop specification and variable usage in traditional programming language, the FP program to do inner product, for instance, would be

$$Def \quad IP \equiv (/+) \circ (\alpha\times) \circ trans$$

in which functions $trans$, $\alpha\times$, and $/+$ form a sequence to be applied to input *object*. If the input is defined as $<< a_1, a_2, \ldots, a_k >, < b_1, b_2, \ldots, b_k >>$, we have the following derivation,

$IP :<< a_1, a_2, \ldots, a_k >, < b_1, b_2, \ldots, b_k >>$
$= (/+) \circ (\alpha\times) \circ trans :<< a_1, a_2, \ldots, a_k >, < b_1, b_2, \ldots, b_k >>$
$= (/+) \circ (\alpha\times) :<< a_1 b_1 >< a_2 b_2 >, \ldots, < a_k, b_k >>$
$= (/+) :< \times :< a_1 b_1 > \; \times :< a_2 b_2 >, \ldots, \times :< a_k, b_k >>$
$= (/+) :< a_1, b_1 \;\; a_2 b_2, \ldots, a_k b_k >$
$= a_1 b_1 + a_2 b_2 + \ldots + a_k b_k$

Incidentally, α in the definition of FP program IP is a functional form. It is formally defined as

$$\alpha\, f : x \equiv \quad x = \phi \rightarrow \phi; \quad x =< x_1, \ldots, x_n > \rightarrow < f : x_1, \ldots, f : x_n >; \quad \bot \,.$$

This is *apply-to-all*, which means each element in an object sequence gets applied by *f* (the functional argument of α). *Right Insert* / is another functional form, defined as

$/f : x \equiv x =< x_1 > \rightarrow x_1; \quad x =< x_1, \ldots, x_n > \;\&\; n \geq 2 \rightarrow f :< x_1, /f :< x_2, \ldots, x_n >>; \bot \,.$

Note that this is recursively defined.

Using the inner product function, the matrix-matrix multiplication takes the following form

$$Def \quad MM \equiv \alpha\alpha\, IP \circ \alpha\, distl \circ distr \circ [1, trans \circ 2].$$

In other words, to perform matrix-matrix multiplication, the second matrix should be transposed, then row-column pairs are formed and the IP operation are applied to each pair. Admittedly, this is a more elegant way in writing algorithms. We adopt the nature of FP to our transformational system so that this symbolic formalism can make our design system more flexible and have a higher degree of automation.

Definition 12 (Driver) *A driver comprises one or more transformers which are cascaded (serialized) via the composition function $\circ$. The transformers may be prefixed by FP functional forms apply-to-all (α).*

Drivers are used to create different views on the MGD so that further reductions on the size of mesh graph lead to practically mapped arrays. They are actually transformation rules for the MGD. Drivers can be applied successively to the MGD until designs are obtained. There are some properties about drivers which are worth being discussed. Some of them resemble the characteristics of functional programming since the driver is in sense a descendant from FP.

- Applying drivers to the MGD should be a *behavior-preserving* operation in the sense that the transformed MGD still inherits the computational characteristics from the original.

- The execution of a driver on the MGD is *nonrepetitive* and *deterministic*. Its structure is helpful in understanding without complicated control structure and looping construct. Determinism implies that the results of the application upon a given MGD at different times will always be the same.

- Drivers are completely *general* in the sense that it works for any well-defined mesh graph descriptors.

- Drivers are *hierarchical*. They are composed by simple language construct, i.e., functions and functional forms.

- Drivers are said to be *dimension-free*. Each of the drivers retains its functionality, regardless the dimensionality of the MGD which are applied by the drivers. For instance, a driver that coalesces two dimensional descriptors to one dimensional should also be able to reduce three dimensional to one.

With the basic concepts of FP construct in mind, we are ready to see how drivers can be applied to MGD descriptors in order to obtain coalesced graphs from the original. For the sake of clarity, illustration on 2-D will be given and later a generalization on 3-D graphs will be addressed.

3.4.1 Applying Drivers to a 2-D Mesh Graph

Figure 3.6 part (a) shows a 4×6 planar mesh graph. The corresponding MGD, in one out of six ways, can be specified as

$\mathcal{K} = (k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3 \xrightarrow{t} k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6) \downarrow (k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9 \xrightarrow{t} k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12}) \downarrow (k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15} \xrightarrow{t} k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18}) \downarrow (k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21} \xrightarrow{t} k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24})$,

where composer $\xrightarrow{t}$ is used to denote links that pass transmittent data.

We are about to see how the following drivers can be applied to $\mathcal{K}$ and the corresponding mesh graph can be transformed.

(i)	$\alpha\texttt{group}$
(ii)	$\alpha\texttt{group} \circ \texttt{trans}$
(iii)	$\alpha\texttt{group} \circ \texttt{pair}$
(iv)	$\alpha\texttt{group} \circ \texttt{pair} \circ \texttt{trans}$
(v)	$\alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ \texttt{pair} \circ \alpha\texttt{pair}$
(vi)	$\alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ \texttt{split} \circ \alpha\texttt{split}$

Definition 13 (Dual) *On 2-D mesh graph manipulation, two drivers are said to be dual to each other if they are of the same functionality but are of different angle (axis) in arranging the graph.*

Note that rules *(i)* and *(ii)* are dual, so are *(iii)* and *(iv)*, as we will see clearly in the following examples on how to apply drivers to the MGD[2].

$\boxed{\text{Driver (i)}}$

$\alpha\texttt{group} \circ \mathcal{K}$

$$\Longrightarrow [k_1, k_2, k_3, k_4, k_5, k_6] \downarrow [k_7, k_8, k_9, k_{10}, k_{11}, k_{12}] \downarrow [k_{13}, k_{14}, k_{15}, k_{16}, k_{17}, k_{18}] \downarrow [k_{19}, k_{20}, k_{21}, k_{22}, k_{23}, k_{24}]$$

which corresponds to part (a) in Figure 3.6.

[2] The outermost parenthesis of the MGD has been omitted for a clearer view. Logically speaking, all MGDs should have come with the outermost parenthesis for the sake of derivation consistency.

$\boxed{\text{Driver (ii)}}$

αgroup $\circ$ trans $\circ\, \mathcal{K}$

$\Longrightarrow$ αgroup $\circ\, (k_1 \downarrow k_7 \downarrow k_{13} \downarrow k_{19}) \xrightarrow{t} (k_2 \downarrow k_8 \downarrow k_{14} \downarrow k_{20}) \xrightarrow{t} (k_3 \downarrow k_9 \downarrow k_{15} \downarrow k_{21}) \xrightarrow{t} (k_4 \downarrow k_{10} \downarrow k_{16} \downarrow k_{22}) \xrightarrow{t} (k_5 \downarrow k_{11} \downarrow k_{17} \downarrow k_{23}) \xrightarrow{t} (k_6 \downarrow k_{12} \downarrow k_{18} \downarrow k_{24})$

$\Longrightarrow$ $[k_1, k_7, k_{13}, k_{19}] \xrightarrow{t} [k_2, k_8, k_{14}, k_{20}] \xrightarrow{t} [k_3, k_9, k_{15}, k_{21}] \xrightarrow{t} [k_4, k_{10}, k_{16}, k_{22}] \xrightarrow{t} [k_5, k_{11}, k_{17}, k_{23}] \xrightarrow{t} [k_6, k_{12}, k_{18}, k_{24}]$

which corresponds to part (c) in Figure 3.6.

$\boxed{\text{Driver (iii)}}$

αgroup $\circ$ pair $\circ\, \mathcal{K}$

$\Longrightarrow$ αgroup $\circ\, ((k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3 \xrightarrow{t} k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6) \downarrow (k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9 \xrightarrow{t} k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12})) \downarrow ((k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15} \xrightarrow{t} k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18}) \downarrow (k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21} \xrightarrow{t} k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24}))$

$\Longrightarrow$ $[(k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3 \xrightarrow{t} k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6) \downarrow (k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9 \xrightarrow{t} k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12})] \downarrow [(k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15} \xrightarrow{t} k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18}) \downarrow (k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21} \xrightarrow{t} k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24})]$

which corresponds to part (d) in Figure 3.6.

$\boxed{\text{Driver (iv)}}$

αgroup $\circ$ pair $\circ$ trans $\circ\, \mathcal{K}$

$\Longrightarrow$ αgroup $\circ$ pair $\circ\, (k_1 \downarrow k_7 \downarrow k_{13} \downarrow k_{19}) \xrightarrow{t} (k_2 \downarrow k_8 \downarrow k_{14} \downarrow k_{20}) \xrightarrow{t} (k_3 \downarrow k_9 \downarrow k_{15} \downarrow k_{21}) \xrightarrow{t} (k_4 \downarrow k_{10} \downarrow k_{16} \downarrow k_{22}) \xrightarrow{t} (k_5 \downarrow k_{11} \downarrow k_{17} \downarrow k_{23}) \xrightarrow{t} (k_6 \downarrow k_{12} \downarrow k_{18} \downarrow k_{24})$

$\Longrightarrow$ αgroup $\circ\, ((k_1 \downarrow k_7 \downarrow k_{13} \downarrow k_{19}) \xrightarrow{t} (k_2 \downarrow k_8 \downarrow k_{14} \downarrow k_{20})) \xrightarrow{t} ((k_3 \downarrow k_9 \downarrow k_{15} \downarrow k_{21}) \xrightarrow{t} (k_4 \downarrow k_{10} \downarrow k_{16} \downarrow k_{22})) \xrightarrow{t} ((k_5 \downarrow k_{11} \downarrow k_{17} \downarrow k_{23}) \xrightarrow{t} (k_6 \downarrow k_{12} \downarrow k_{18} \downarrow k_{24}))$

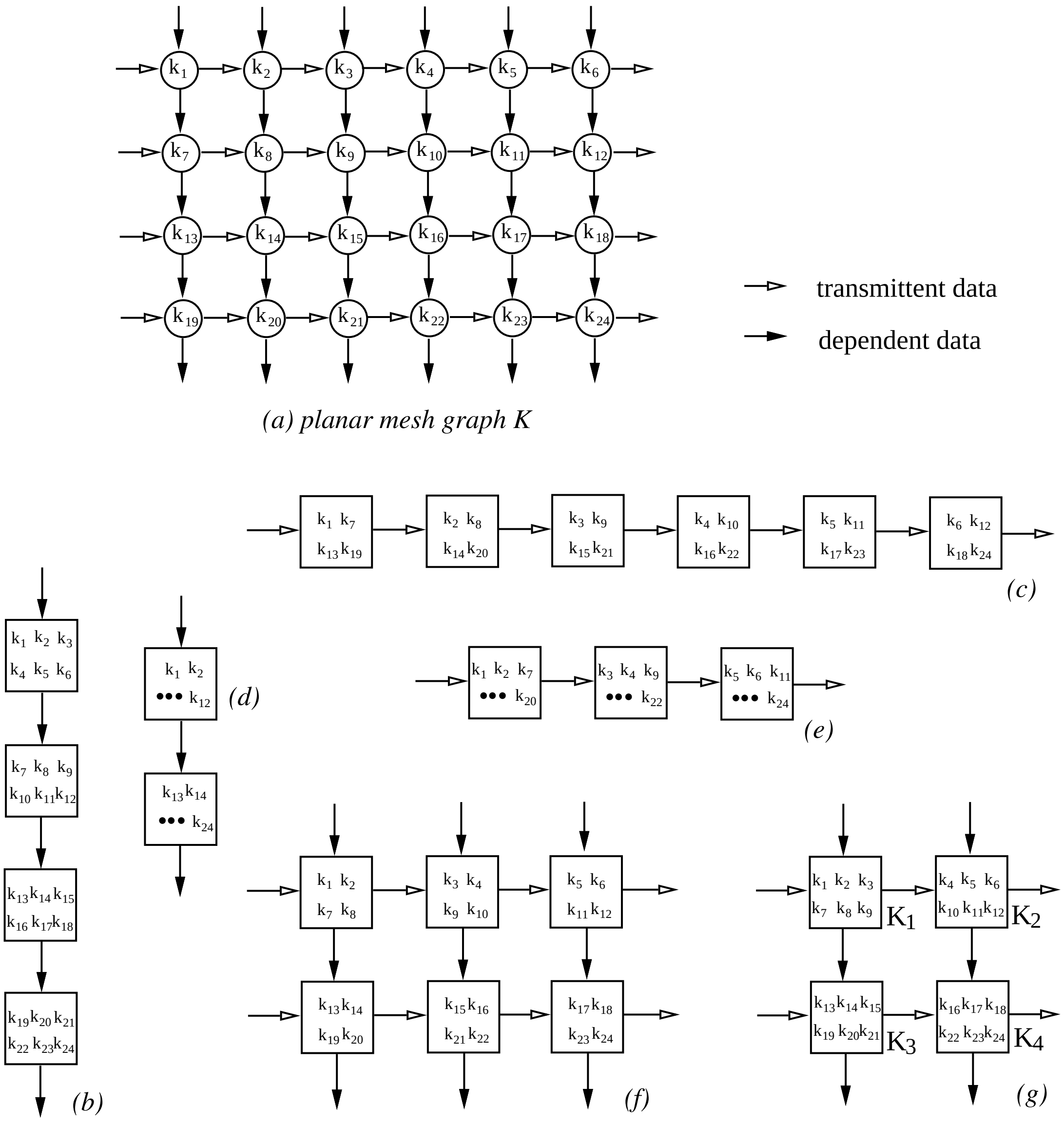

Figure 3.6: A planar mesh graph $\mathcal{K}$ and the effect of applying drivers to $\mathcal{K}$: (a) graph $\mathcal{K}$, (b) driver (i) applied, (c) driver (ii) applied, (d) driver (iii) applied, (e) driver (iv) applied, (f) driver (v) applied, (g) driver (vi) applied.

$$\Longrightarrow [(k_1 \downarrow k_7 \downarrow k_{13} \downarrow k_{19}) \xrightarrow{t} (k_2 \downarrow k_8 \downarrow k_{14} \downarrow k_{20})] \xrightarrow{t} [(k_3 \downarrow k_9 \downarrow k_{15} \downarrow k_{21}) \xrightarrow{t} (k_4 \downarrow k_{10} \downarrow k_{16} \downarrow k_{22})] \xrightarrow{t} [(k_5 \downarrow k_{11} \downarrow k_{17} \downarrow k_{23}) \xrightarrow{t} (k_6 \downarrow k_{12} \downarrow k_{18} \downarrow k_{24})]$$

which corresponds to part (e) in Figure 3.6.

$\boxed{\text{Driver (v)}}$

$\alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ \texttt{pair} \circ \alpha\texttt{pair} \circ \mathcal{K}$

$$\Longrightarrow \alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ \texttt{pair} \circ ((k_1 \xrightarrow{t} k_2) \xrightarrow{t} (k_3 \xrightarrow{t} k_4) \xrightarrow{t} (k_5 \xrightarrow{t} k_6)) \downarrow ((k_7 \xrightarrow{t} k_8) \xrightarrow{t} (k_9 \xrightarrow{t} k_{10}) \xrightarrow{t} (k_{11} \xrightarrow{t} k_{12})) \downarrow ((k_{13} \xrightarrow{t} k_{14}) \xrightarrow{t} (k_{15} \xrightarrow{t} k_{16}) \xrightarrow{t} (k_{17} \xrightarrow{t} k_{18})) \downarrow ((k_{19} \xrightarrow{t} k_{20}) \xrightarrow{t} (k_{21} \xrightarrow{t} k_{22}) \xrightarrow{t} (k_{23} \xrightarrow{t} k_{24}))$$

$$\Longrightarrow \alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ (((k_1 \xrightarrow{t} k_2) \xrightarrow{t} (k_3 \xrightarrow{t} k_4) \xrightarrow{t} (k_5 \xrightarrow{t} k_6)) \downarrow ((k_7 \xrightarrow{t} k_8) \xrightarrow{t} (k_9 \xrightarrow{t} k_{10}) \xrightarrow{t} (k_{11} \xrightarrow{t} k_{12}))) \downarrow (((k_{13} \xrightarrow{t} k_{14}) \xrightarrow{t} (k_{15} \xrightarrow{t} k_{16}) \xrightarrow{t} (k_{17} \xrightarrow{t} k_{18})) \downarrow ((k_{19} \xrightarrow{t} k_{20}) \xrightarrow{t} (k_{21} \xrightarrow{t} k_{22}) \xrightarrow{t} (k_{23} \xrightarrow{t} k_{24})))$$

$$\Longrightarrow \alpha^2\texttt{group} \circ (((k_1 \xrightarrow{t} k_2) \downarrow (k_7 \xrightarrow{t} k_8)) \xrightarrow{t} ((k_3 \xrightarrow{t} k_4) \downarrow (k_9 \xrightarrow{t} k_{10})) \xrightarrow{t} ((k_5 \downarrow k_6) \xrightarrow{t} (k_{11} \downarrow k_{12}))) \downarrow (((k_{13} \xrightarrow{t} k_{14}) \downarrow (k_{19} \xrightarrow{t} k_{20})) \xrightarrow{t} ((k_{15} \xrightarrow{t} k_{16}) \downarrow (k_{21} \xrightarrow{t} k_{22})) \xrightarrow{t} ((k_{17} \downarrow k_{18}) \xrightarrow{t} (k_{23} \downarrow k_{24})))$$

$$\Longrightarrow ([(k_1 \xrightarrow{t} k_2) \downarrow (k_7 \xrightarrow{t} k_8)] \xrightarrow{t} [(k_3 \xrightarrow{t} k_4) \downarrow (k_9 \xrightarrow{t} k_{10})] \xrightarrow{t} [(k_5 \downarrow k_6) \xrightarrow{t} (k_{11} \downarrow k_{12})]) \downarrow ([(k_{13} \xrightarrow{t} k_{14}) \downarrow (k_{19} \xrightarrow{t} k_{20})] \xrightarrow{t} [(k_{15} \xrightarrow{t} k_{16}) \downarrow (k_{21} \xrightarrow{t} k_{22})] \xrightarrow{t} [(k_{17} \downarrow k_{18}) \xrightarrow{t} (k_{23} \downarrow k_{24})])$$

which corresponds to part (f) in Figure 3.6.

$\boxed{\text{Driver (vi)}}$

$\alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ \texttt{split} \circ \alpha\texttt{split} \circ \mathcal{K}$

$$\Longrightarrow \alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ \texttt{split} \circ ((k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3) \xrightarrow{t} (k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6)) \downarrow ((k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9) \xrightarrow{t} (k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12})) \downarrow ((k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15}) \xrightarrow{t} (k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18})) \downarrow ((k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21}) \xrightarrow{t} (k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24}))$$

$$\Longrightarrow \alpha^2\texttt{group} \circ \alpha\texttt{trans} \circ (((k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3) \xrightarrow{t} (k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6)) \downarrow ((k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9) \xrightarrow{t} (k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12}))) \downarrow (((k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15}) \xrightarrow{t} (k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18})) \downarrow ((k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21}) \xrightarrow{t} (k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24})))$$

$$\Longrightarrow \alpha^2\texttt{group} \circ (((k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3) \downarrow (k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9)) \rightarrow ((k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6) \downarrow (k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12}))) \downarrow (((k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15}) \downarrow (k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21})) \rightarrow ((k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18}) \downarrow (k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24})))$$

$$\Longrightarrow ([(k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3) \downarrow (k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9)] \rightarrow [(k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6) \downarrow (k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12})]) \downarrow ([(k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15}) \downarrow (k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21})] \rightarrow [(k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18}) \downarrow (k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24})])$$

, or equivalently,

$$\Longrightarrow (K_1 \xrightarrow{t} K_2) \downarrow (K_3 \xrightarrow{t} K_4), \text{ where}$$

$$\begin{aligned}
K_1 &\stackrel{\text{def}}{=} (k_1 \xrightarrow{t} k_2 \xrightarrow{t} k_3) \downarrow (k_7 \xrightarrow{t} k_8 \xrightarrow{t} k_9)\\
K_2 &\stackrel{\text{def}}{=} (k_4 \xrightarrow{t} k_5 \xrightarrow{t} k_6) \downarrow (k_{10} \xrightarrow{t} k_{11} \xrightarrow{t} k_{12})\\
K_3 &\stackrel{\text{def}}{=} (k_{13} \xrightarrow{t} k_{14} \xrightarrow{t} k_{15}) \downarrow (k_{19} \xrightarrow{t} k_{20} \xrightarrow{t} k_{21})\\
K_4 &\stackrel{\text{def}}{=} (k_{16} \xrightarrow{t} k_{17} \xrightarrow{t} k_{18}) \downarrow (k_{22} \xrightarrow{t} k_{23} \xrightarrow{t} k_{24}).
\end{aligned}$$

which corresponds to part (g) in Figure 3.6.

As mentioned, the process of applying drivers to MGDs is deterministic. A result is obtained by applying any driver to any MGD. However, a question may arise as a dual to the above observation: Are there any cases in which applying more than one driver to a particular MGD can lead to the same result? The answer is positive in that another driver, in addition to driver (vi), can provide the same outcome. If we apply "$\alpha^2\texttt{group}\circ\alpha\texttt{split}\circ\alpha\texttt{trans}\circ\texttt{split}$" to $\mathcal{K}$, exactly the same coalesced mesh graph can be derived.

In preceding paragraphs we have demonstrated how multi-level abstraction and coalescing can be facilitated through the use of MGDs. In this

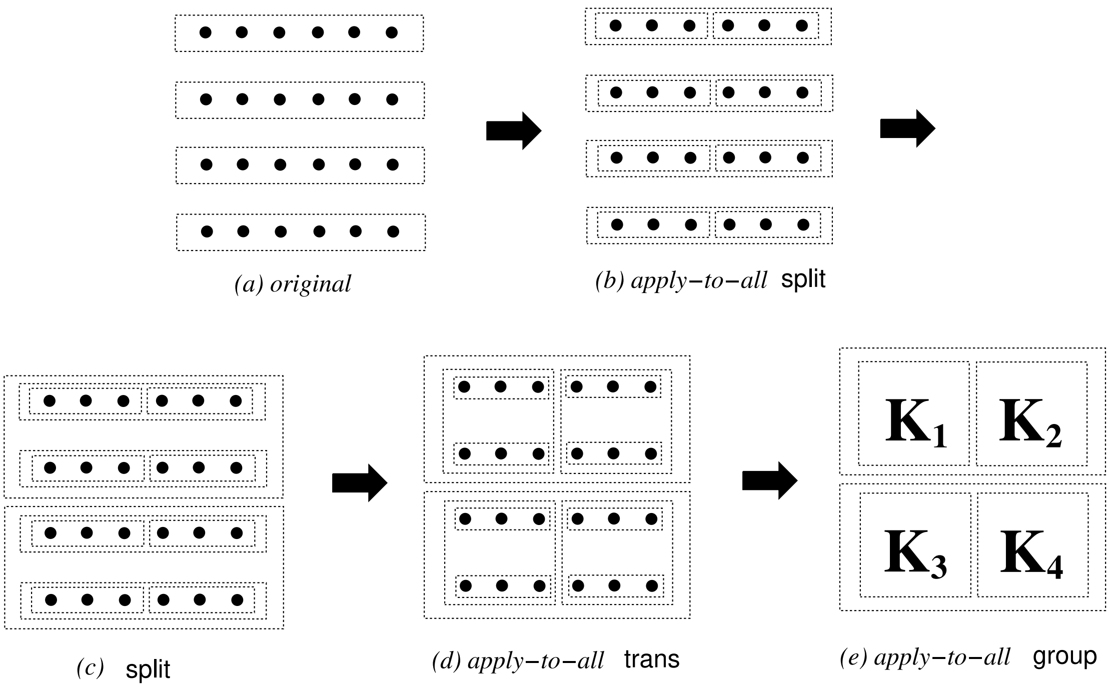

Figure 3.7: (a) - (d): Different views of MMG $\mathcal{K}$ by applying driver (vi) to its MGD, (e) is the contour corresponding to the resultant MGD.

section drivers which are essentially a series of transformers have been applied to the MGD to obtain different topological representations of mesh graphs. It would be of interests to take snap shots of the process starting from the original mesh graph to the concluding graph. Figure 3.7 illustrates the views (graphical semantics) of MMG $\mathcal{K}$ which correspond to the MGDs derived by successively applying the transformers in driver *(vi)*. Note that the contours demonstrated here are of different meaning as to concept of *view* introduced earlier. The view created for mesh graph $\mathcal{K}$ is the graph with 4 nodes K_1, K_2, K_3, and K_4, each of which comprises 6 primitive nodes.

3.4.2 Applying Drivers to 3-D MMGs

An example of applying drivers to a planar mesh graph was provided in the previous section. Same rationale can be employed for the three dimensional mesh graphs. Instead of the complicated derivations, a generalized form will be given in this section as well as the size measures of possibly coalesced implementations. If a 3-D mesh graph, say $\mathcal{L}$, of dimension $p \times q \times q$ is under consideration, the corresponding MGD of $\mathcal{L}$ will be as follows. The composition sequence for this descriptor is $Y \diamond X \diamond Z$.

$$\begin{array}{rcl}\mathcal{L} & = & ((l_1 \downarrow \cdots \downarrow l_q) \rightarrow \cdots \rightarrow (l_{pq-q+1} \downarrow \cdots \downarrow l_{pq})) \nearrow \\ & & ((l_{pq+1} \downarrow \cdots \downarrow l_{pq+q}) \rightarrow \cdots \rightarrow (l_{2(pq-q+1)} \downarrow \cdots \downarrow l_{2pq})) \nearrow \\ & & \cdots\cdots\cdots \\ & & ((l_{pqr-pq+1} \downarrow \cdots \downarrow l_{pqr-pq+q}) \rightarrow \cdots \rightarrow (l_{pqr-q+1} \downarrow \cdots \downarrow l_{pqr})).\end{array}$$

This form is a generic representation of any *homogeneous* 3-D MMG because a matrix algorithm with three loops, after the symbolic interpretation, generates a 3-D MMG which can be regularized and turned into a homogeneous MMG if it is not originally in homogeneous form.

A generalized form of drivers is also possible. We need a slight modification on the apply-to-all (α) function on the previous list of drivers. For instance,

$$\alpha^p \texttt{group} \circ \alpha^{p-1} \texttt{trans}$$

is a driver to coalesce the MGD of dimension q to dimension p, where $p \leq q$. If we reconsider applying the six drivers in the 2-D case to the aforementioned 3-D MGD with dimension $p \times q \times r$, the resulting diagram is as shown in Figure 3.8. The derivations like in the 2-D case are omitted here. They should be similar to the 2-D case.

To present the snapshots for 3-D in detail like in Figure 3.7 would be more complicated. However, the contour of the transformation can still be sketched in a way, as demonstrated in Figure 3.9, so that the process of mapping graph $\mathcal{L}$ can be visualized. This diagram shows the process of applying driver (vi) to $\mathcal{K}$. It is interesting to see that the very same driver, no matter being applied to 2-D or 3-D mesh of any dimension, would guide and project the MMG into an array of only 4 nodes.

3.5 Remarks on Building the MGDs

The MGD is designed in an effort to capture the structure of data dependence graphs and the processor arrays to which they are mapped. With its concise and simple construct, the MGD can be applied by drivers to meet

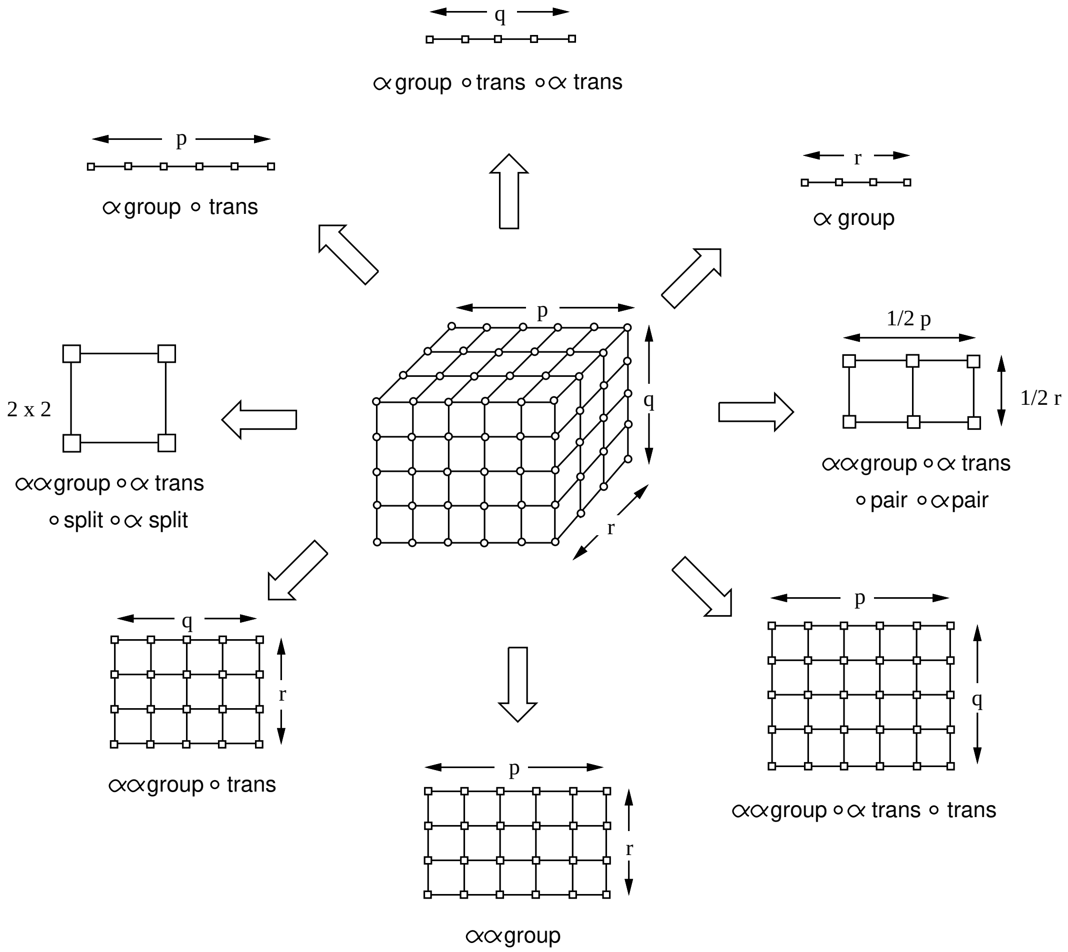

Figure 3.8: The effect of applying drivers on the MGD of $\mathcal{K}$.

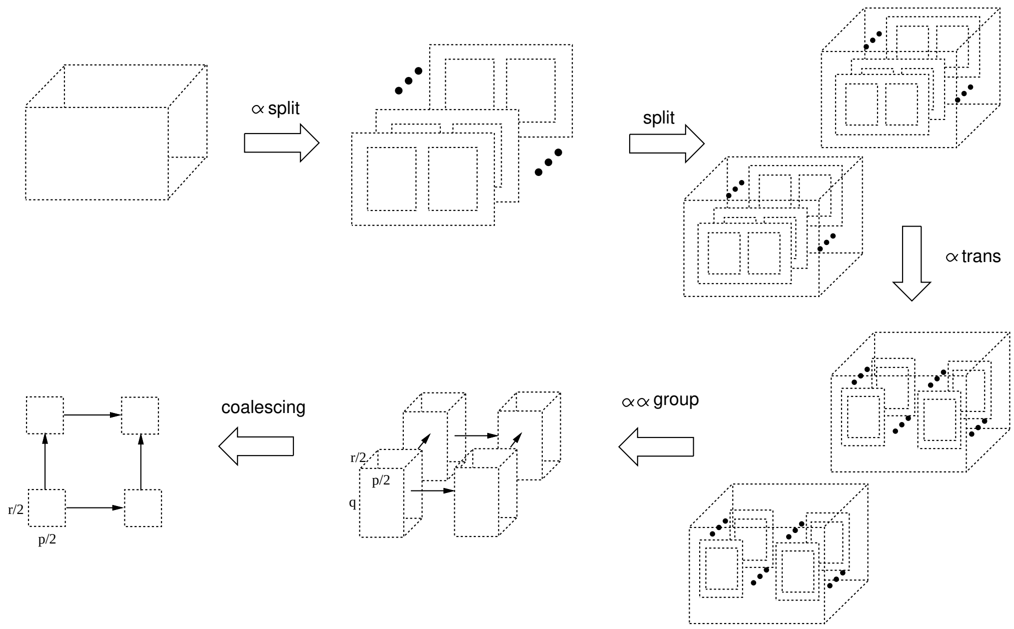

Figure 3.9: Views and contour of MMG $\mathcal{K}$, applied by driver (vi).

our needs in obtaining different types of array designs. The MGD has explicit specifications of the nodes and the links that interconnect them. I/O data are not included in the MGD construct since otherwise the transformations would not work as neatly as they do now. We chose to associate the i/o data specifications within the boundary nodes instead of explicitly expanding these specifications in terms of literals in the descriptors.

3.6 Summary

The transformation properties of the MGD were introduced. These properties together with the definition of the MGD provide symbolic formalism in fulfilling the objective of automating the processing array construction. We believe that, by using concise graph expressions described here, a powerful tool based on the language of simplicity can serve as a better synthesizer for arrays carrying out matrix computations. In next chapter, we discuss two physical cases to which the methodology is applied.

Chapter 4

Physical Issues of Mapping

In previous chapters, we have seen the theoretical basis of our methodology with focus on the descriptors introduced in this work. In this chapter, we put our attention to physical issues of the mapping process, for example, the scheduling and performance issues of the processing arrays being synthesized. We also give the relationship between the implementation attributes of the synthesized arrays and the mesh graphs that have been partitioned by the prisms and tagged by the time steps of execution. It was mentioned in Chapter 1 that two key issues in mapping an algorithm to arrays are *processor assignment* and *scheduling*. We have discussed extensively the assignment issue in previous chapters, i.e., the application of transformations to the MGD descriptors and the derivation of the mapped processing arrays. However, not until we finish scheduling task can the performance of the synthesized arrays be obtained and analyzed. The primitive nodes in the original MMG graph, when assigned to processors in distinct manner, can result in different processor designs. In an effort to single out a set of designs to our interests, the performance of synthesized arrays need to be analyzed.

4.1 Scheduling

The scheduling task can be visualized as a tagging process of each of the MMG nodes by an integer which indicates the time step for that particular node to be executed. The *execution* of algorithms on processor arrays can be visualized as the integration of the allocation of primitive nodes to the processor and the scheduling of these nodes. Generally speaking, there are two constraints which have to be met by the scheduling approach,

- No processor computes two primitive nodes at any time step.
- Every primitive node has to wait for its immediate predecessors to finish before it can be computed.

Types of Scheduling

There are two types of scheduling of the nodes in an MMG.

Linear All the nodes in MMG are to be executed at different time steps; all the nodes can be grouped in different sets based on these time steps, with each set corresponding to a time step. The nodes to be executed at the same step coexist on a line for the 2-D case, a plane on the 3-D case. There exists only one optimal scheduling for each MMG. This optimal scheduling is linear. All other linear scheduling possibilities are non-optimal since they would take longer to complete with respect to the optimal case. The majority of ongoing research focus on this method. The index transformation category mentioned earlier adopts this approach [Mol82], [Mol83], [CS83], [MW84], [Qui84], [Rao85].

Nonlinear In the nonlinear scheduling approach, all the nodes to be fired at the same time step may not coexist on a same line or plane. Only the precedence relationships implied from the MMG are obeyed as the nodes are assigned of the time of execution. Nonlinear scheduling may be desirable under some cases since not all the nodes may take the same amount of time to finish. For instance, there may be "division" nodes as well as "addition" nodes in an MMG. In such case, taking a nonlinear scheduling approach may end up with a better overall performance than the linear approach.

In our symbolic transformation approach we can adopt either the linear or nonlinear approach in pursuit of the optimal performance, depending on the type of MMG we are dealing with.

To demonstrate the scheduling process, we take a small 2-D MMG and see how it can be schedule-wise mapped to its processor array space. In (a) of Figure 4.1, we see a typical 2-D mesh graph. It is unlabeled, with data

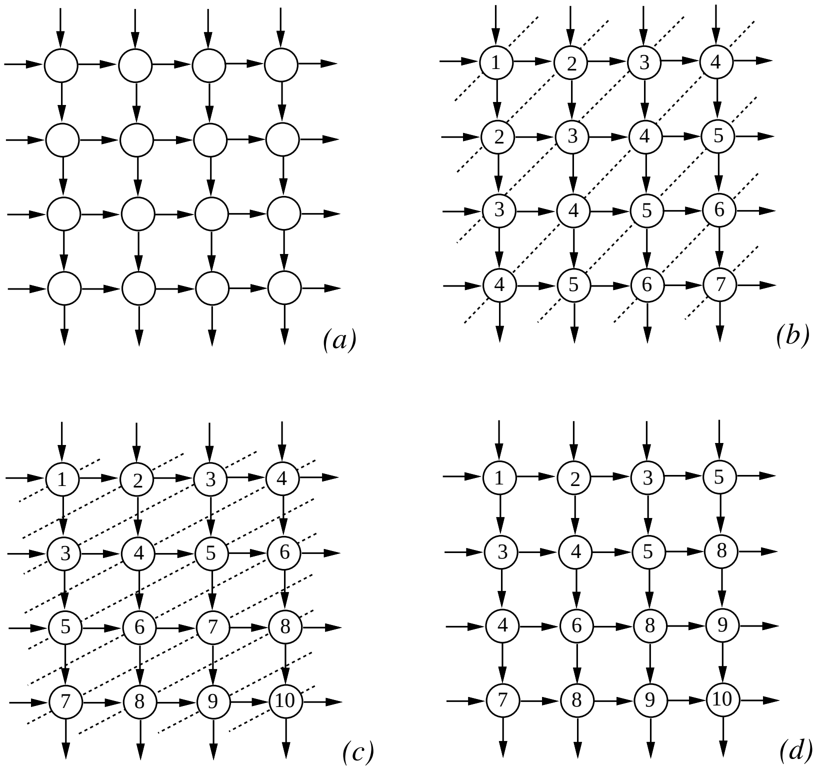

Figure 4.1: The formation of iso-temporal lines: (a) a 2-D MMG, (b) optimal linear scheduling, (c) a nonoptimal linear scheduling, (d) a nonlinear scheduling.

input flows along X^+ and Y^+ directions. To schedule this dependence graph optimally, we have the following considerations. At time step 1, all nodes except the top left one can be executed due to the dependence relationships. At time step 2, two nodes can be labeled which are immediate successors to the top left node. Continuing in this manner, we may label all the nodes in the diagram. If we draw lines that cut through all nodes with the same label, the result will be in part (b) in the same figure. These lines are called iso-temporal lines, which indicate the set of nodes that will be executed at the same time step. This diagram denotes the optimal scheme since all the nodes are scheduled to be executed in the *earliest possible manner*. Part (c) of Figure 4.1 is another example of linear scheduling, yet not optimal. At time step 2, instead of executing two immediate successors, only one of them is picked. Each set of the nodes to be executed at same step is still colinear. In part (d), no specific pattern of execution can be found. This is an example of nonlinear scheduling. Only the precedence relationships of the arcs is followed.

4.2 Cell Functionalities and Determining Input

From the past knowledge we know that the topology of the mapped processor can be invariably obtained, given the MMG and the coalescing direction. Two more attributes amid our interests now are the input data progression and the functionalities of the processor cells.

The computations (functionalities) of the coalesced cell will be the union of the all the functionality of the primitive nodes of which the cell is made. For instance, if cell n_{11}, n_{12}, n_{13} perform $\{+\}$, $\{\times\}$, $\{+\}$, respectively, the functionality of the cell comprising of these three nodes would be $\{+,\times\}$. The implication of this on the coalescing angle is that it may be desirable to combine nodes with similar or same computational attributes into one cell so that the implementation costs can be minimized.

With the knowledge of scheduling strategy, we can calculate the number of steps (time) for the synthesized processor to finish the task as well as identify the set of nodes at each time step. Looking at the MMG diagram, we notice that some of the nodes on the boundary have input from the module to which the array is connected. At the time steps during which these nodes are executed, the inputs have to arrive in time to the cell. Since the scheduling of the primitive nodes has been done, the associated inputs to these nodes can be determined and listed at the input ports of the array processor accordingly. We call the method of determining the input *reverse tracing*. It is called reverse because the direction of numbering the steps to place the input data token is against the data input flow.

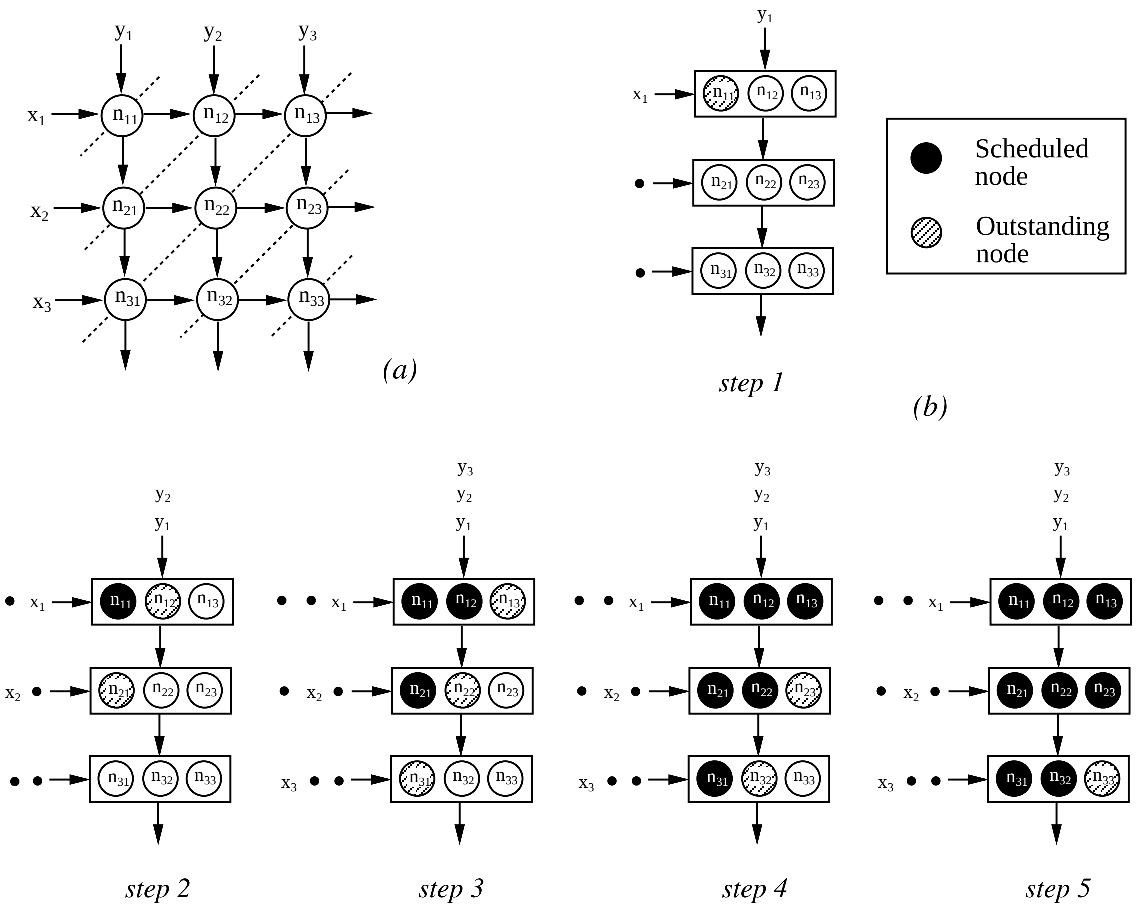

Figure 4.2: Scheduling of inputs: (a) a 3×3 2-D MMG, (b) orthographical coalescing along X^+.

4.2.1 An Example of Optimal Linear Scheduling

Orthographical Coalescing

Figure 4.2 is an example which shows how the input for the orthographically coalesced array can be generated, in this case, from the optimal scheduling information. Part (a) shows a 3×3 2-D MMG. In part (b), step 1 shows that node n_{11} will be scheduled. We define ρ as the distance of input from the array($\rho = 1, 2, 3...$). Also, let the cells coalesced from MMG's row 1, 2, and 3 be denoted as cell 1, 2, and 3, respectively. Since node n_{11} is on boundary with x_1 and y_1 as its inputs, both inputs have to show at the input ports of the coalesced cell at the closest position to the array($\rho = 1$). We notice that after scheduling node n_{11}, x_1 and y_1 appear at the input ports of cell 1, whereas two null data tokens show at the left input ports of cells 2 and 3. The data input corresponding to $\rho = 2$ are the inputs to the primitive nodes in MMG which will be executed at time step 2. At this time step, n_{12} and n_{21} will be outstanding. y_2 and x_2 are the input tokens to these two nodes; therefore, they should be placed at position $\rho = 2$ so that exact execution can be finished. Having finished all five steps, we can obtain all the input data tuples. Note that the entire execution time lapse is dependent solely on the scheduling scheme, that is, it is independent of the data inputs.

In an effort to build up a tool that facilitates automatic generation of processor array of this type, we need to find threads which can be abided by in our synthesis procedures. This concept of reverse tracing is an important part in our synthesis tool.

Diagonal Coalescing

The 2-D example in previous section is used again for the demonstration of diagonal coalescing. In comparison to ortho-coalescing, this case needs a little bit more work since the first primitive node to be executed is not located at the boundary cell.

We define the cell which contains the first node to be executed to be *lead cell*. The cells of the processor array can be labeled by η (nonnegative integers). For lead cell, $\eta = 0$. The adjacent two cells with respect to lead cell are labeled with $\eta = 1$, and so forth. We also define the term *hops* that associated with each primitive node to be the set of composers that walk the lead node to that particular node. For instance, $hops_{n_{23}} = \{\rightarrow, \rightarrow, \downarrow\}$. The *norm* of composer is defined as the number of composers in hops. We can easily see that $\|\rightarrow\|_{n_{23}} = 2$, while $\| \downarrow \|_{n_{23}} = 1$. η can be calculated as

$$\eta = | \ \|\rightarrow\| \ - \ \| \downarrow \| \ |.$$

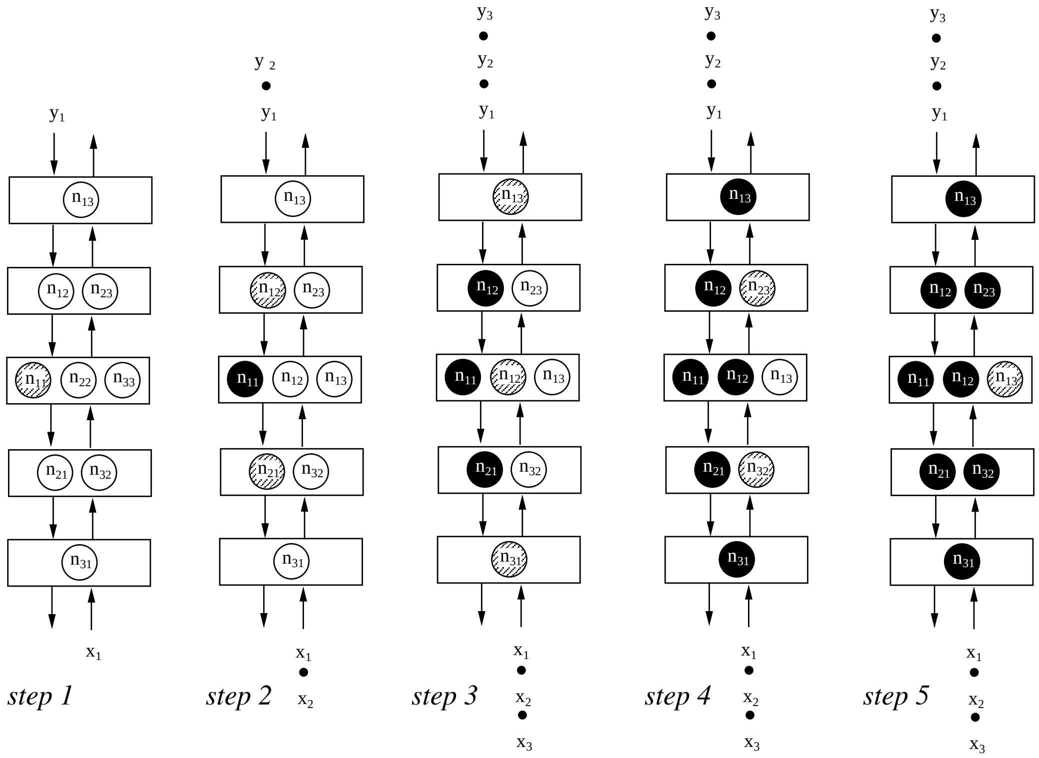

Figure 4.3: How input can be generated: diagonal coalescing along X^+Y^+.

The procedure of deriving the input pattern for y is as follows(x is similar). The lead cell contains y_1 as the northern input. We place y_1 at the top position, namely, $\rho_{y_1} = 1$. Note that it would take 3 steps in this case for y_1 to reach the center cell. n_{12} is a node amid other cells to be governed by next dotted line. n_{12} has one element of y stream as its input as well. We define another function c which gives a particular cell that contains the primitive node which has input from the host. That is

$$c : input \longrightarrow cell.$$

We have clearly now $\eta_{c(y_2)} = 1$. Since the ρ value of current input y_i depends on the distance between cells c_{y_i} and $c_{y_{i-1}}$, we have the following formula to determine the position of the input, if any, associated with current scheduling step.

$$\begin{aligned}\rho_{y_i} &= \rho_{y_{i-1}} + 1 + \{\text{distance between } c(y_i), c(y_{i-1})\} \\ &= \rho_{y_{i-1}} + 1 + \eta_{c(y_i)} - \eta_{c(y_{i-1})},\end{aligned}$$

where the constant 1 is due to the increment in step from pipelining.

From the above equation, we have the following calculations,

$$\rho_{y_1} = 0 + 1 + 0 - 0 = 0$$

$$\rho_{y_2} = 1 + 1 + 1 - 0 = 3$$

$$\rho_{y_3} = 3 + 1 + 2 - 1 = 5.$$

Accordingly, input token y_2 should be placed at the third slot from the array, and so forth. A null token is now inserted between inputs y_1 and y_2. Having traversed all the scheduling lines, we may obtain the set of y stream, as well as the x stream if necessary.

4.2.2 An Example of Non-optimal Linear Scheduling

This approach also applies to other non-optimal linear scheduling cases. In Figure 4.4 a non-optimal linear example can be found. After applying reverse tracing, we can obtain the input stream sets, with x different from the optimal case.

4.2.3 An Example of Non-linear Scheduling

For some MMG with non-homogeneous distribution in execution time, we may need non-linear scheduling for the sake of efficiency. For instance, node n_{21} takes 2 steps to finish, so nodes n_{31} and n_{23} cannot be issued until step time 4, thus the non-linear scheduling. Since our method accomplishes the scheduling task from a node-to-node perspective rather than an overall uniform perspective as in linear approach, it promises higher untilization and better performance in processor arrays being synthesized. Figure 4.5 shows a non-linear scheduling example.

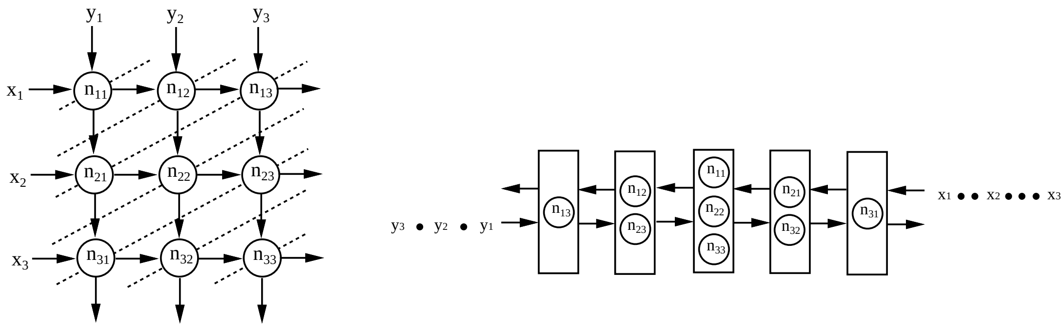

Figure 4.4: Non-optimal linear scheduling.

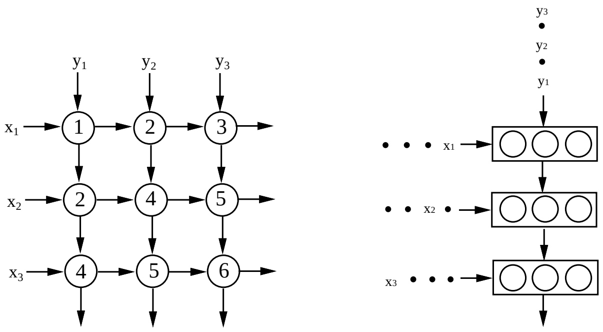

Figure 4.5: Non-linear scheduling.

4.3 The Effect of Coalescing Directions on Performance

We have seen that many different designs can be obtained from the same mesh graph when the graph is treated by different drivers, namely, different projection angles taken. It is desirable to consider many designs in the family that suit the design needs since trade-off is commonly seen with respect to the implementation cost/performance issues. Note that it is also possible to find more than one design under the same implementation constraint. These designs may have different characteristics in the execution behavior, thus different performance measures. Within the class of the designs of the same implementation cost, there is no reason to choose the design with longer execution time.

In Figure 4.6, a non-uniform 2-D mesh graph with unbalanced nodes as well the scheduling chart are shown. The nodes are labeled with tuples in which the first element indicates the node and the second element indicates the time steps for that particular node to finish execution. The transmittent data paths are represented by bold arcs. In part (a) of the same figure, the coalescing is done along the direction that is orthogonal to the transmittent data path. Although node a takes 5 steps to finish, node b may start execution as soon as processor P_1 got the data and passes it down to processor P_2, so b is scheduled at time slot 1. Proceeding in the same manner, we may obtain the scheduling chart at the right side of part (a). It takes 11 times steps to accomplish the entire algorithm. In contrast, part (b) shows the coalescing direction along the transmittent data path. Since the output from node a passed down to node d is not a transmittent datum, node d has to wait until a finishes being put on the execution row. Therefore, node d has to be scheduled at time slot 5. The corresponding timing is also shown. It takes 19 steps in total to finish.

Now we see that, under similar implementation constraint, the total execution time for different coalescing schemes may also vary significantly. Clearly, it is favorable to do coalescing along the orthogonal direction to the transmittent data since early execution of the node in the adjacent processor is possible.

The sequencing control on the processor arrays that realize the MMG with nodes requiring non-uniform execution in time would be more compli-

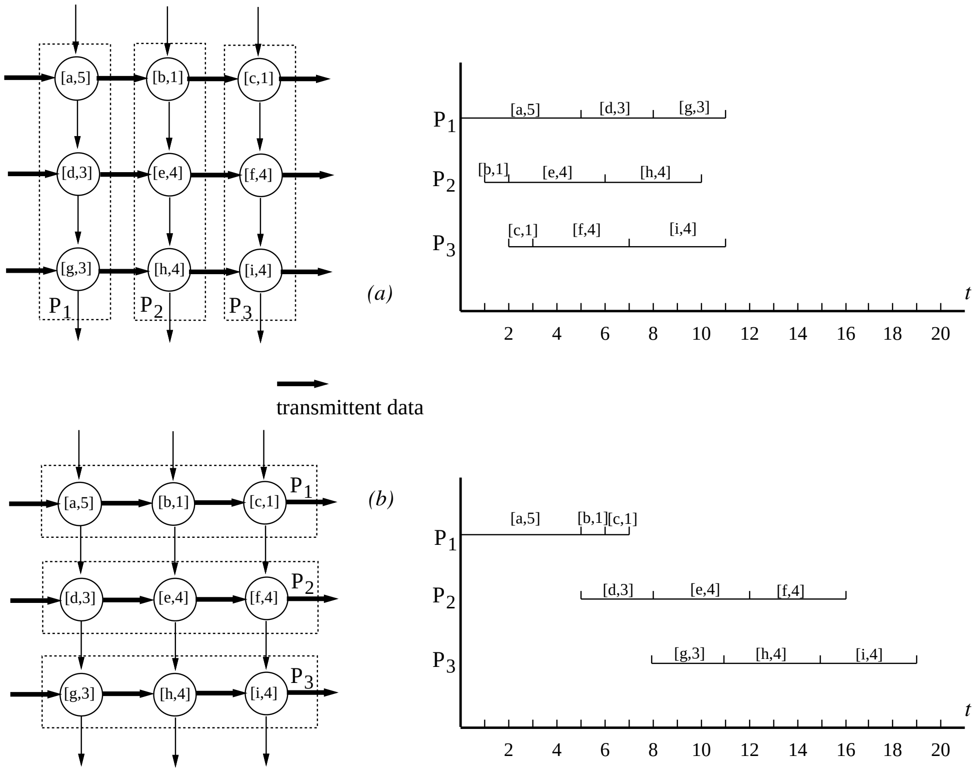

Figure 4.6: Performance influenced by coalescing directions.

cated than MMG with homogeneous nodes in time. One way to simplify the sequencing of MMG with non-homogeneous nodes is to let all the nodes take up the same time as required by the node with longest execution time. This would make the synthesized processor more "systolic-like" at the expense of degraded performance.

4.4 The Mapping of Processor Arrays Coalesced from a 3-D MMG

Given a 3-D MMG, we can synthesize different linear processor arrays by dividing the 3-D MMG into slices. Each of the slices is mapped to a physical processor. If the MMG is partitioned into prisms, the mapped result would be a rectangular or a square array of processors each of which comes from one prism. Let the dimension of the prism be $p \times q \times r$. If $p = q = 1$, then it is a straight projection. By applying projection at different angles, we may derive various types of systolic arrays. It is interesting to note that virtually all of the systolic arrays found in the past can be produced in this manner [JRK87].

In Figure 4.7, the relationship of the scheduled time steps for a primitive node in a 3-D MMG and the input data arriving time is illustrated. The same rationale we found for the 2-D array applies as well to the 3-D MMG. If the scheduled time for a given node is t_k, the corresponding input i_k should arrive at the node at time step t_k in order to be executed. Similarly, after labeling all the nodes in the MMG, we can determine the input marching pattern for the synthesized array.

4.5 Speed-delivery

Given a 2-D or 3-D MMG, we are able to schedule the execution of the primitive nodes as described in previous sections. These scheduling schemes are linear since the assignment of the time steps to the nodes is uniformly distributed over the entire MMG graph. To achieve shorter completion time for an MMG, however, the aforementioned scheduling scheme may not be the best. We may take advantage of a characteristic in the MMGs generated

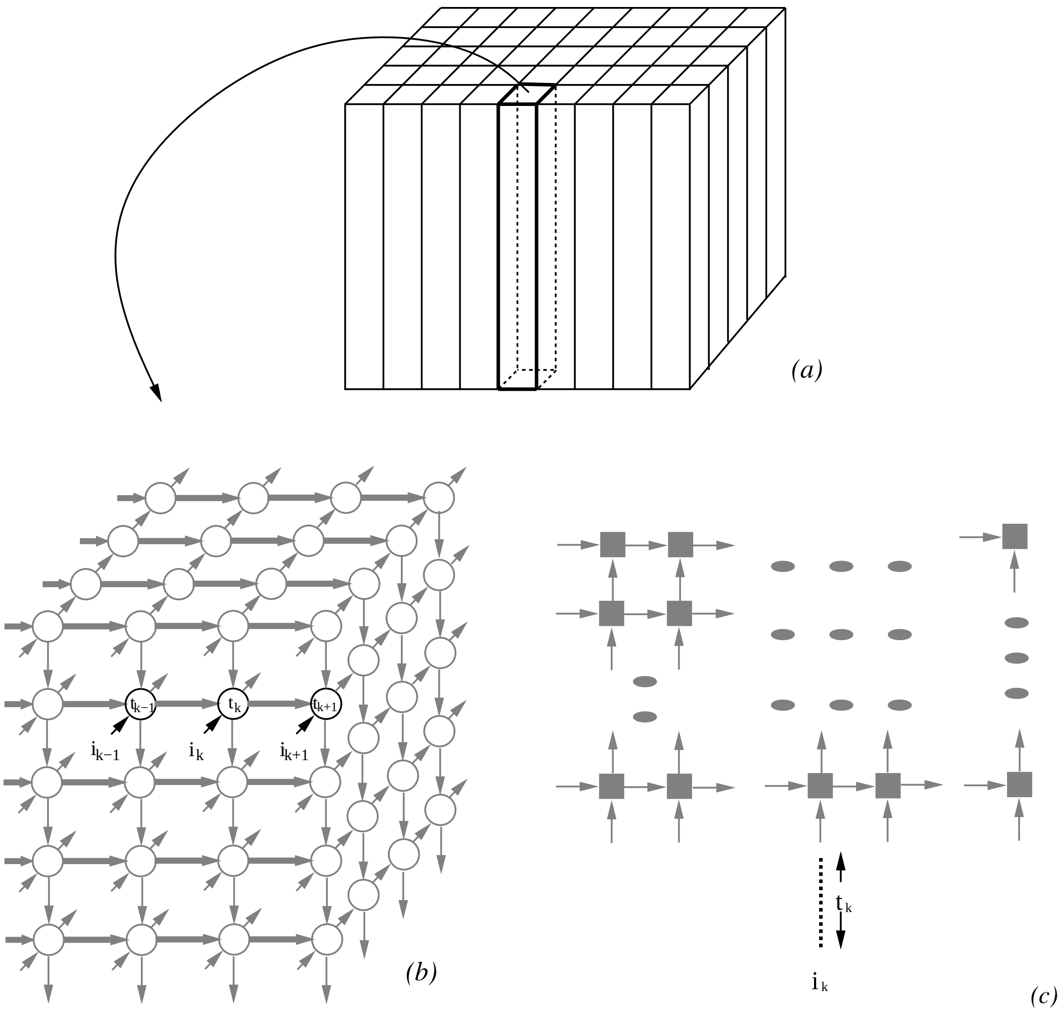

Figure 4.7: Correspondence between the scheduled time step assigned to a node and the input:(a) a typical 3-D MMG partitioned into prisms, (b) the magnified diagram of a particular prism, (c) the input i_k should arrive at time slot t_k.

by most matrix algorithms which is the presence of the transmittent data paths.

There is at least one matrix in every matrix algorithm that needs to be passed (transmitted) through the entire algorithm-implied mesh graph. The boundary nodes along the prisms, therefore, can be placed on the execution row right after the previous prism had received it. We utilize this characteristics and name this approach as "speed-delivery." We found many cases where the speed-delivery approach synthesize more efficient arrays than those produced by traditional linear scheduling approaches. The speedup is even more dramatic for synthesized arrays in larger sizes. Physical examples will be given in chapter 5. We now compare our approach with the DBT approach (Dense to Band matrix transformation by Triangular-block partition) introduced by Navarro [NLV87]. DBT approach is another partitioning approach which allows the input matrix to be partitioned into a band matrix which can be subsequently mapped to a much smaller-sized array.

Below is an example demonstrating how the methods can be applied to matrix-by-vector multiplication. The following equation is a triangular-block partitioning of this problem.

$$\begin{bmatrix} A_{DU11} & A_{DU12} \\ A_{L11} & A_{L12} \\ A_{DU21} & A_{DU22} \\ A_{L21} & A_{L22} \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ \\ X_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ \\ B_2 \end{bmatrix} = \begin{bmatrix} Y_1 \\ \\ Y_2 \end{bmatrix}$$

The above equation can be transformed into the following equation with two extra vectors Y_1^1 and Y_2^1 introduced. Now this matrix-by-vector multiplication problem can be solved in terms of the band-matrix problem. Band-matrix problem can be implemented by systolic arrays with limited number of processors [KL78]. Hence, the proposed method can partition a much larger algorithm and map it to small size arrays.

$$\begin{bmatrix} A_{DU11} & A_{L12} & & & \\ & A_{DU12} & A_{L11} & & \\ & & A_{DU21} & A_{L22} & \\ & & & A_{DU22} & A_{L21} \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_2 \\ X_1 \\ X_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ Y_1^1 \\ B_2 \\ Y_2^1 \end{bmatrix} = \begin{bmatrix} Y_1^1 \\ Y_1 \\ Y_2^1 \\ Y_2 \end{bmatrix}$$

However, despite the nice mathematical formalism, the two intermediate vectors need to be computed before the desired results can be obtained.

The overhead introduced hereby causes longer execution time lapse. In the example give by the same paper, a 6×6 matrix-by-vector example needs a total of 28 steps to finish. In contrast, if the speed-delivery approach were adopted, it would need only a total of 16 steps to finish (14 steps for multiplication, 2 steps for addition) if the mesh graph is to be coalesced into a 3-node linear processor.

4.6 Summary

In this chapter, we have discussed some physical issues in our design methodology. Two types of scheduling have been considered, namely, linear and nonlinear. Our method is capable of synthesizing processing arrays for both scheduling schemes. The decisions of input and cell functionality also were introduced in this chapter. Some examples of 2-D case were illustrated in good detail. Formula to determine input sequencing are also given. The effect of choosing different coalescing directions on the performance has been depicted. It would be favorable to project the mesh graphs along directions which are orthogonal to the transmittent data direction.

Chapter 5

Application Examples

The objective of this chapter is to demonstrate how to apply the proposed methodology to physical examples - matrix-matrix multiplication, LU decomposition, and the Algebraic Path Problem. A 3-D MMG graph can be created after the each of the input algorithms having been interpreted. The next stage of ALIAS system is to perform coalescing operations so that parallel arrays realizing the algorithm can be synthesized.

5.1 Synthesis Steps

Though it appears that it is a fairly long process to map an algorithm into a design consisting of processor array, there is a definitive sequence of steps to achieve this goal. ALIAS basically follows the steps to analyze a given algorithm and transform it into manipulatable MGD formulations from which the implementation can be derived. Below are the descriptions of the steps to obtain the application specific arrays in our design environment.

1. **Determine MMG dimension** The dimension of MMG can be decided by examining the number of loop levels in the given algorithm. For a typical matrix computation algorithm, the number should be either two or three, which contributes to MMG of dimension two or three, respectively. Algorithms with only one loop variable are too trivial to be considered.

2. **Assign coordinates** The coordinates that span either the 2-D or 3-D space now are tagged with the loop variables. The flows of the incoming matrix (matrices) and outgoing matrix (matrices) are determined in this stage. For viewing convenience, an acceptable pattern of flow is usually adopted.

3. **Applicative interpretation** Now the MAP module is ready to symbolically interpret the algorithm. For index-dependent algorithms, MAP exhaustively goes through the loops and provide a list of the execution traces. Meanwhile, the corresponding MGD should be generated which conforms to the behavior of the input algorithm. For index-independent algorithms, only the inner statements of the algorithm need to be interpreted and then instantiated to produce the MMG graph

4. **Coalescing** In an effort to obtain a practical and low-cost implementation, the MMG usually has to be reduced to a much smaller scale in a lower order dimension. This is done through coalescing (covered in Chapter 2 and 3). Coalescings through orthographical and diagonal perspective provide viable implementations.

5.2 Matrix Multiplication

In this section, the examples in both algorithm-size-dependent arrays and algorithm-size-independent arrays that realize the matrix-multiplication algorithm will be demonstrated.

5.2.1 Algorithm-size-dependent Arrays

In the following we will see the derivation of systolic array for matrix-matrix multiplication. We first rewrite the algorithm from mathematical expression into single-assignment recursive algorithm.

Algorithm:

$$C_{ij} = \sum_{k=1}^{3} A_{ik} B_{kj}$$

Recursive Single-assignment Algorithm:

initialization: $a(i,1,k) = A_{i,k}; b(1,j,k) = B_{k,j}; c(i,j,1) = 0$
S1: *for* $i = 1$ *to* 3 *do*
S2: *for* $j = 1$ *to* 3 *do*
S3: *for* $k = 1$ *to* 3 *do*
S4: $a(i,j+1,k) = a(i,j,k)$
S5: $b(i+1,j,k) = b(i,j,k)$
S6: $c(i,j,k+1) = c(i,j,k) + a(i,j,k) \times b(i,j,k)$

In our approach, the derivation of the dependence graph is incremental. Namely, the nodes are allocated as the execution traces of the interpretation. Unless special cases are encountered, we try to regulate the formulation of dependence graphs by following a pattern which is called *DRS (Default Replication Sequence)*, as shown in Figure 5.1.

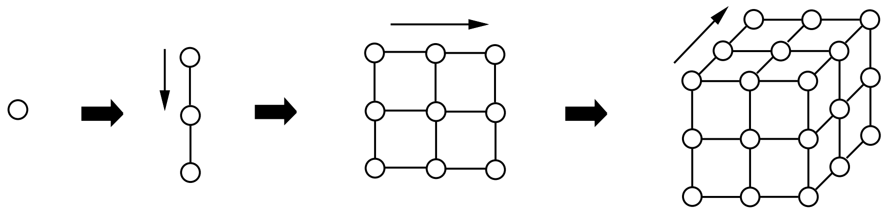

Figure 5.1: Default replication sequence.

Now we apply our interpreter to the above algorithm. The MMG (*Dependence Graph*) can be created which is shown in Figure 5.2.

In Figure 5.2, index space i, j, k of the algorithm is mapped to X, Z, Y axes of our predefined Cartesian coordinates.

Projection

The last step is the *projection* of the MMG from the symbolic interpretation of the algorithm. Projection along Y-axis and X-axis are chosen here, doing Z-axis projection will result in a similar design as the one derived by X direction projection. The resulting designs are in figure 5.3.

For design D_1, the input matrices enter the array from its northern and western ports. However, after computation done, the results end up residing in the array cells. This potentially causes extra output stage and severely slows system turn-around time. It's even unfavorable for multi-instance computation in which each new iteration would have to wait for clearance from

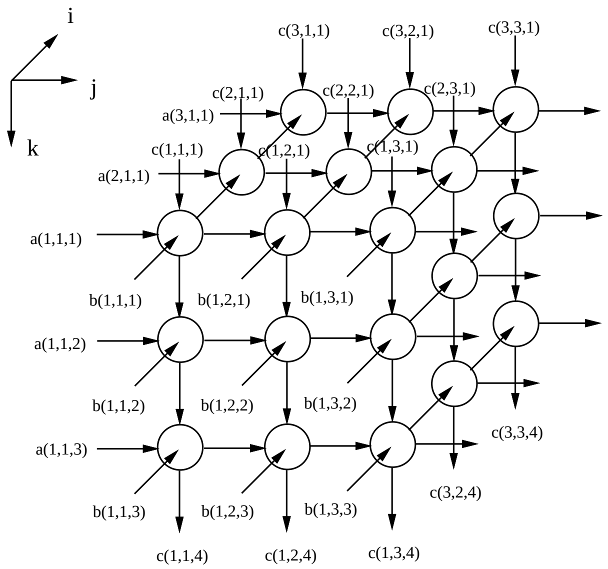

Figure 5.2: The MMG created after the algorithm is parsed.

previous computation. Design D_2 seems to have solved this problem since matrix C would be departing the array from southern port. Another disadvantage is that matrix A has to be "injected" into processor cells beforehand. Searching for new implementation possibilities should not be terminated after two mesh arrays being found. Engaging another projection angle on the MMG would provide a fresh design. Diagonal projection at 45 degrees is a better bet since non-45-degree projections could produce designs with longer latency in data progression.

There are totally eight diagonal projection possibilities for part (d) of Figure 5.2 and ALIAS has to choose a projection angle so that the first-executed MMG node would be placed on the boundary cell of the array. Furthermore, a more balanced configuration concerning matrix inputs should be given high priority. This suggests projection along $X^-Y^+Z^-$. To make the projection direction to be $X^-Y^+Z^-$ so that merged MMG can be clearly examined, the MMG has to be rotated -90 degrees about Y axis. $\{\rightarrow, \nearrow\}$ in the MGD will be, accordingly, turned into $\{\swarrow, \rightarrow\}$. The original MGD is omitted here. The newly rotated MGD is

$$(((w_1 \swarrow w_2 \swarrow w_3) \downarrow (w_4 \swarrow w_5 \swarrow w_6) \downarrow (w_7 \swarrow w_8 \swarrow w_9)) \rightarrow ((w_{10} \swarrow$$

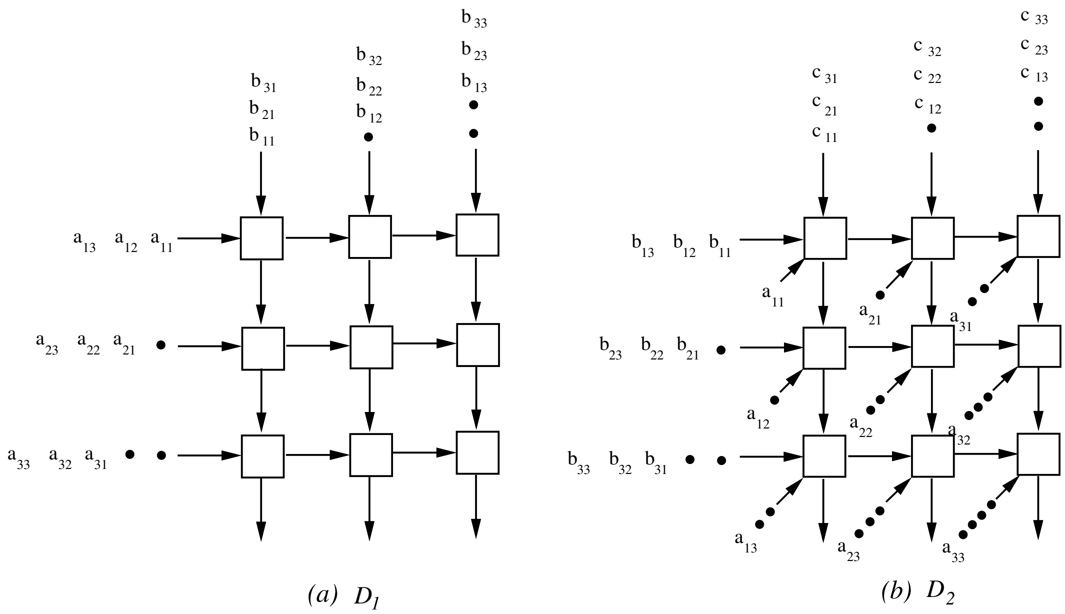

Figure 5.3: Final systolic designs: (a)From projection along Y-axis, (b) From X-axis projection.

$w_{11}\ldots)) \rightarrow ((\ldots w_{27})))$

which specifies the MMG in part (a) of Figure 5.4.

Before nodes can be merged, grouping expression has to be found by the system. Since the projection direction for the original graph is $X^{-}Y^{+}Z^{-}$, it should be mapped to a grouping expression which is $|\rightarrow^{-n} \; ! \downarrow^{n} \; ! \nearrow^{-n}|$. Negative signs denote reverse directions. In other words, instead of traversing forward along the arcs, it should be traversed against the arcs. After -90 degrees rotation about Y, the new grouping expression is $|\swarrow^{-n} \; ! \downarrow^{n} \; ! \rightarrow^{-n}|$. Having been applied by the new grouping expression, the already-turned MGD now has the following form with some nodes combined.

$(((w_1 \tilde{\swarrow} w_2 \tilde{\swarrow} w_3) \downarrow ([w_4, w_{11}] \tilde{\swarrow} [w_5, w_{12}] \tilde{\swarrow} w_6) \downarrow ([w_7, w_{14}, w_{21}] \tilde{\swarrow} [w_8, w_{15}] \tilde{\swarrow} w_9)) \rightarrow ((w_{10} \swarrow [w_4, w_{11}] \swarrow [w_5, w_{12}]) \downarrow ([w_{13}, w_{20}] \ldots) \ldots (\ldots w_{27})))$ which implies the synthesized array as demonstrated in part (b) of figure 5.4.

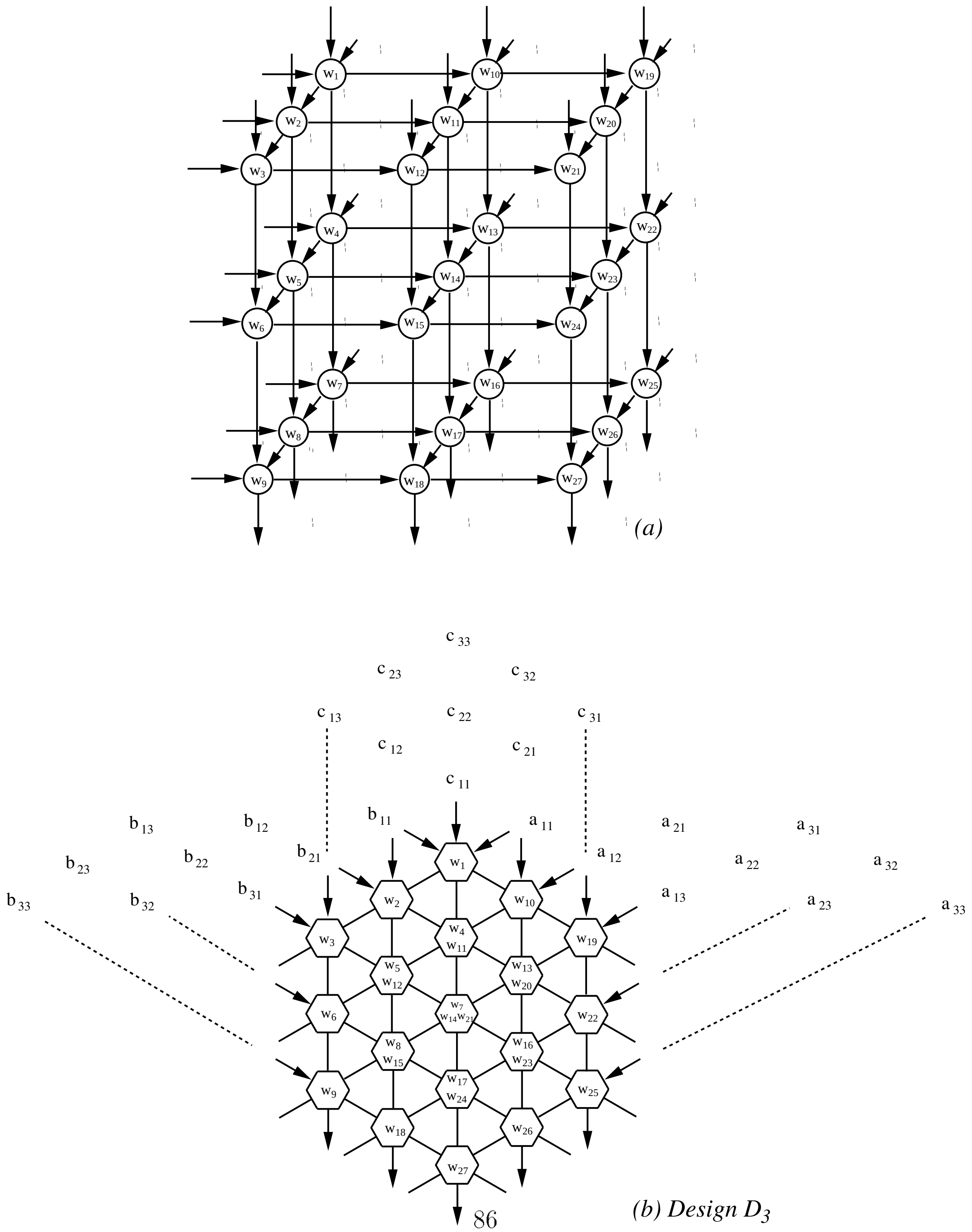

Figure 5.4: (a)MMG after - 90 degrees rotation about Y axis, (b) Corresponding array design.

The ALIAS Design Screens for the Matrix Multiplication Example

The MMG screen for the matrix multiplication algorithm (6×6) is shown in in Figure 5.5. From the MMG screen the user may wish to probe each of the nodes about its computations and iteration space, as shown in Figure 5.6. If the user clicks the box marked with "Apply Drivers to Architecture", the user can get the screen as shown in Figure 5.7. The drivers screen shows several options for the user to choose to project the MMG.

From the drivers screen, the user may click the architecture he wishes to implement. The parallel array processor design can be displayed on the screen together with the input pattern and the optimal scheduling time steps to execute the given algorithm. Figure 5.8 shows an array implementation example for the projection option "Y".

MGD Enhanced

To give a lucid and succinct demonstration, the input components in the dependence graph were not explicitly specified in the MGD. It is highly desirable to incorporate input streams in the descriptor so as to accurately synthesize systolic arrays with data progression illustration. It also has the advantage of letting the throughput and critical path be easily visualized.

The input tuple for a particular node in the dependence graph is concatenated to the right of that node. No changes will be made to cells with no input. Since the degree of dimension will be, at most, three, we specify the tuple to be of cardinality 3 for 3-D mesh descriptors. The convention in preceding paragraph is still followed on the order of the tuple, of which the first, second, and third elements will be input from axes X, Y, and Z, respectively. The plus or minus signs in the data designate toward which direction they are marching. The neutral data element (without sign) refers to the data which reside in the local cell. They are usually the intermediate operands or the final results. Other constructs of the original MGD remain the same in the enhanced, so all the transformations are still valid. One thing to be kept in mind is that the associated input tuples do not necessarily stay the same when a driver or transformer is applied to this particular MGD. Based on the implication from the nature of the transformation, deterministic algorithms can be introduced to correctly transform MGD with input tuples.

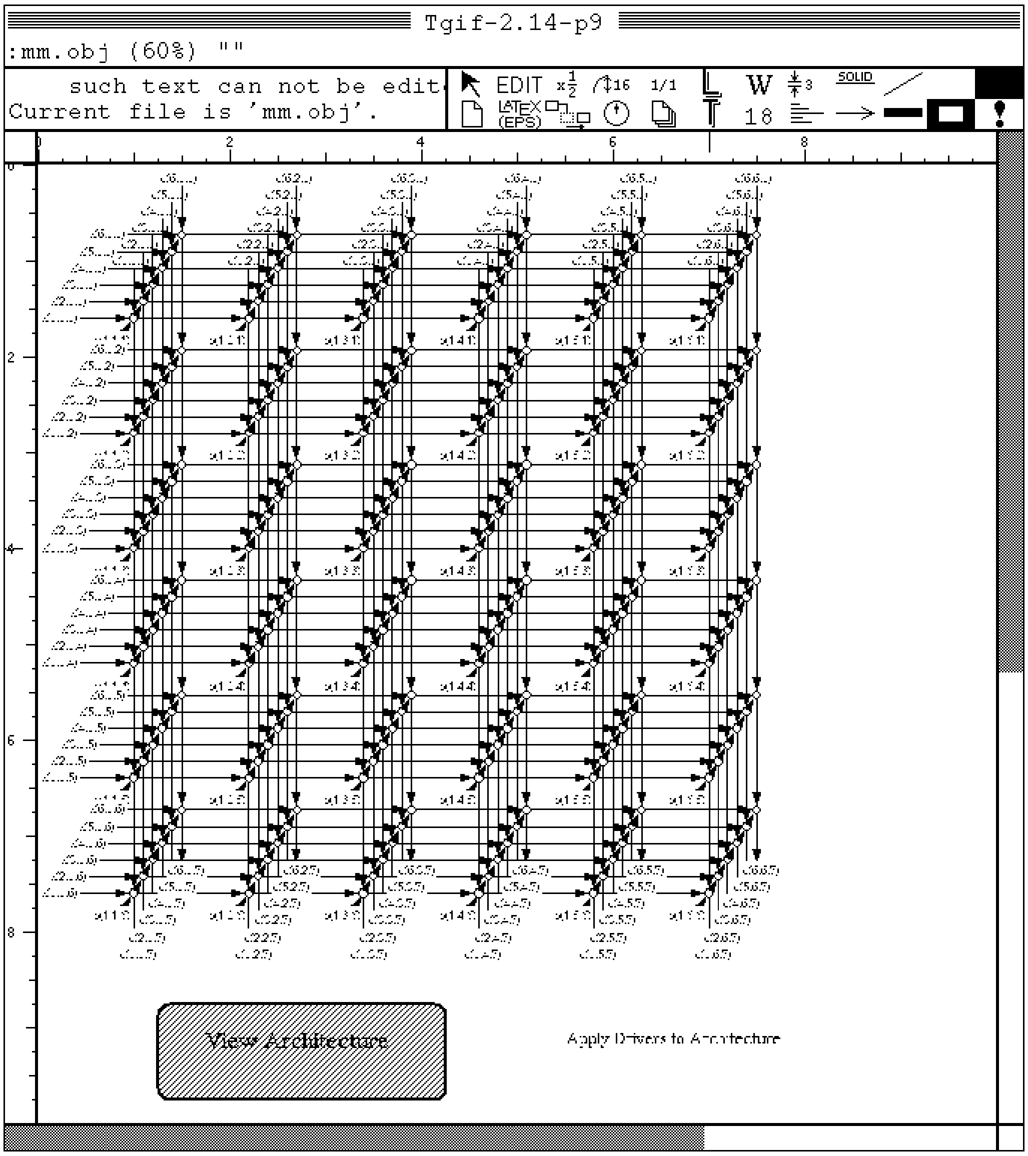

Figure 5.5: An MMG screen for a matrix multiplication example.

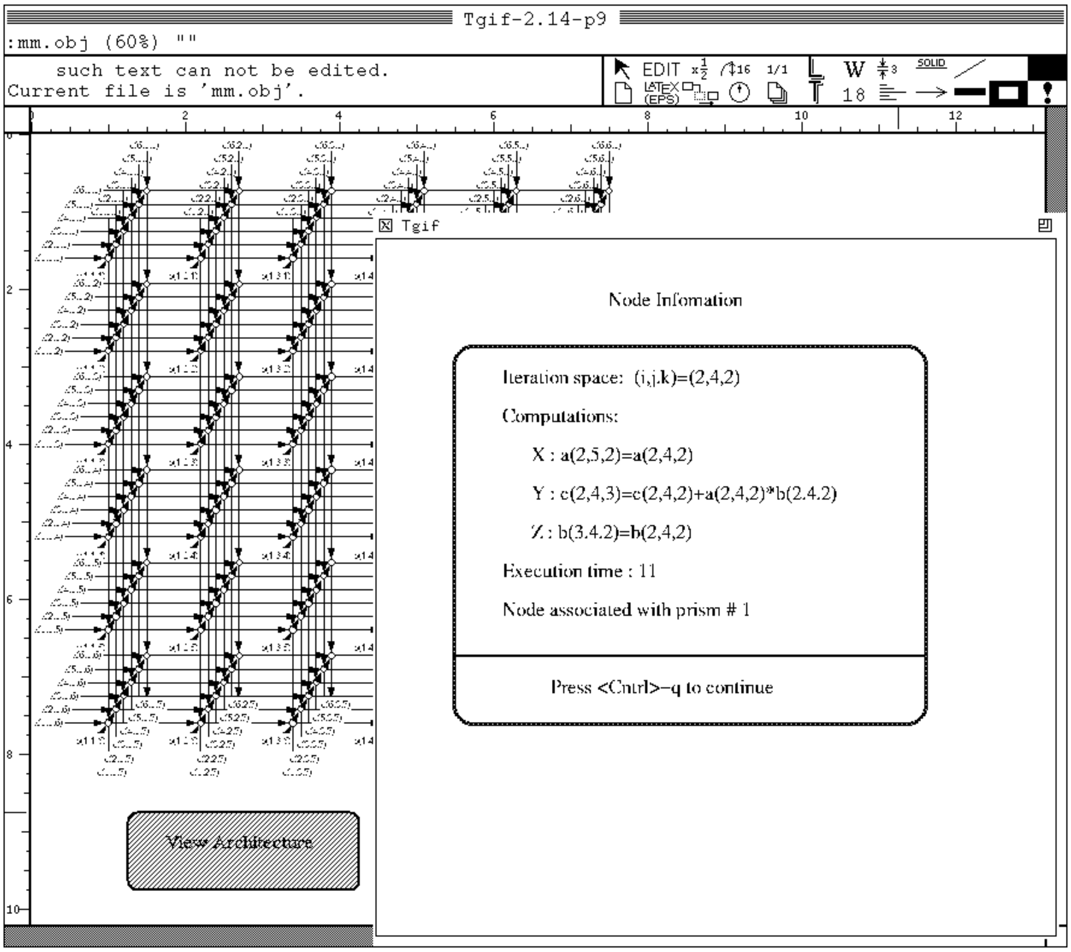

Figure 5.6: A node screen for a matrix multiplication example.

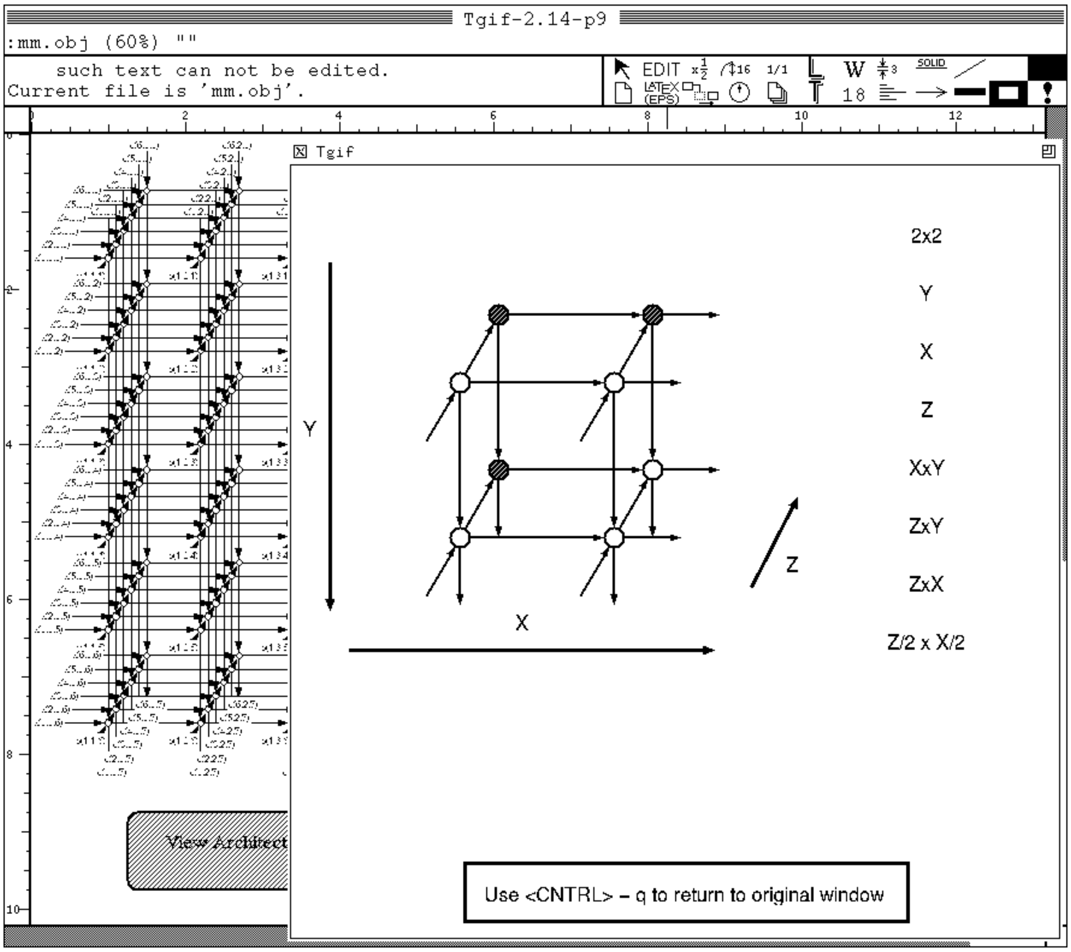

Figure 5.7: A drivers screen for a matrix multiplication example.

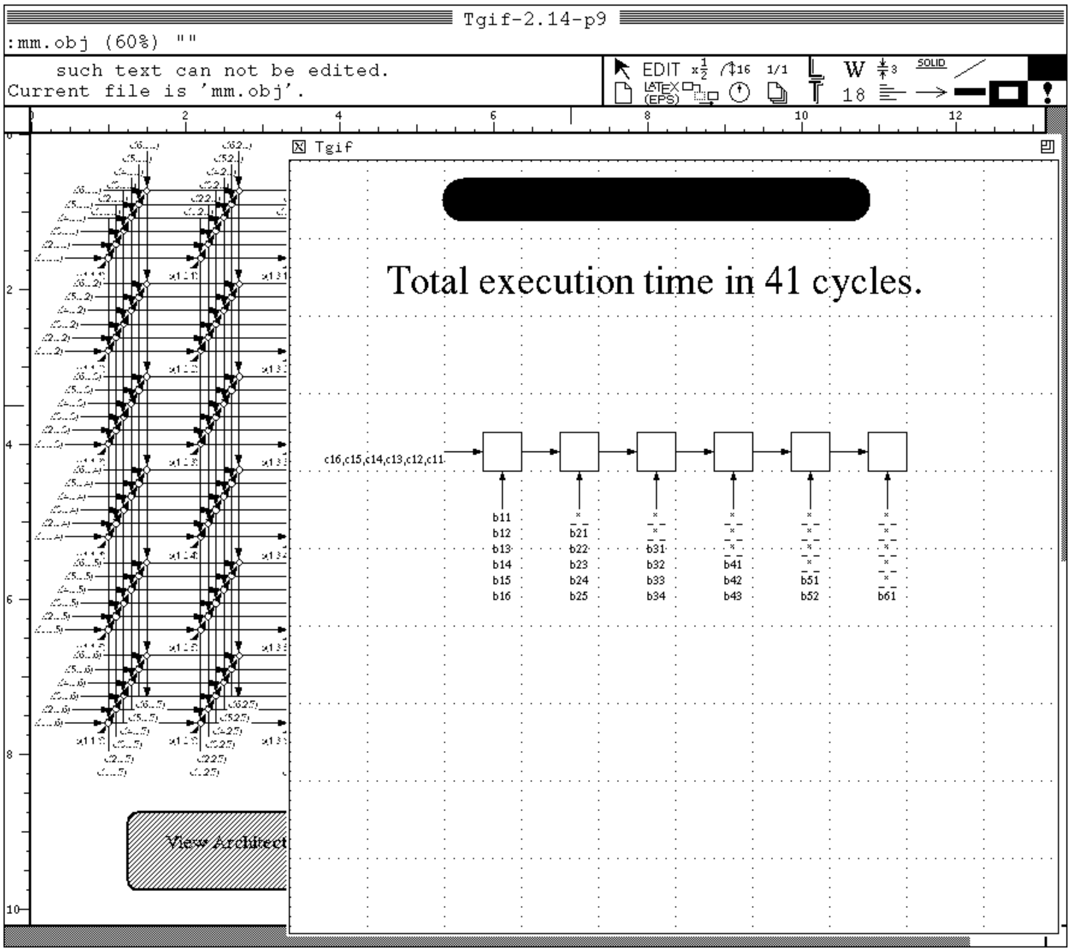

Figure 5.8: An array processor screen for a matrix multiplication example.

To show how to augment the MGD with the input streams, we reconsider the the matrix-multiplication case. The enhanced MGD for the original $3 \times 3 \times 3$ dependence graph would be

$$G_4 = ((w_1 \circ [a_{11}^+, c_{11}^+, b_{11}^+] \rightarrow w_2 \circ [-, c_{12}^+, b_{12}^+] \rightarrow w_3 \circ [-, c_{13}^+, b_{13}^+]) \downarrow (w_4 \circ [a_{12}^+, -, b_{21}^+] \rightarrow w_5 \circ [-, -, b_{22}^+] \rightarrow w_6 \circ [-, -, b_{23}^+]) \downarrow (w_7 \circ [a_{13}^+, -, b_{31}^+] \rightarrow w_8 \circ [-, -, b_{32}^+] \rightarrow w_9 \circ [-, -, b_{33}^+])) \nearrow ((w_{10} \circ [a_{21}^+, c_{21}^+, -] \rightarrow w_{11} \circ [-, c_{22}^+, -] \rightarrow w_{12} \circ [-, c_{23}^+, -]) \downarrow (w_{13} \circ [a_{22}^+, -, -] \rightarrow w_{14} \rightarrow w_{15}) \downarrow (w_{16} \circ [a_{23}^+, -, -] \rightarrow w_{17} \rightarrow w_{18})) \nearrow ((w_{19} \circ [a_{31}^+, c_{31}^+, -] \rightarrow w_{20} \circ [-, c_{32}^+, -] \rightarrow w_{21} \circ [-, c_{33}^+, -]) \downarrow (w_{22} \circ [a_{32}^+, -, -] \rightarrow w_{23} \rightarrow w_{24}) \downarrow (w_{25} \circ [a_{33}^+, -, -] \rightarrow w_{26} \rightarrow w_{27}))$$

Among all the alternative arrays for matrix-multiplication, D_1 and D_2 need local storage and should not be classified as systolic design, strictly speaking. Design D_3 is systolic design with one virtue of having all of its inputs entering from side ports. However, as opposed to D_1 and D_2, it has relatively lower utilization, high implementation cost due to more cells, longer latency. Now it seems more clear why diagonal projection may lead to good designs and definitely should not be left out in the synthesis process.

Incidentally, the operations performed above on the matrix multiplication can be generalized to other cases [Cap92]. Beyond matrix-product, cubical mesh is the dependence dag of a variety of recurrences over three variables. Other computations include a three-pass transitive closure, matrix triangularization, matrix inversion, and two-dimensional tuple comparison.

There are, of course, many other ways to project the mesh graph of matrix multiplication and obtain different types of implementations of the algorithm. In the cases just mentioned, we adopted the grouping prism of dimension $1 \times 1 \times m$ and the results are two dimensional systolic arrays. If we let the grouping prism be of size $1 \times m \times n$, that is, we group primitive nodes falling on a "plane" instead of grouping primitive nodes on a "stick", the synthesized results would be linear arrays. A detail discussion of mapping matrix multiplication algorithm onto linear arrays can be found in [VR86].

At this point, one advantage of adopting MGD becomes more clear. That is, MGD plays a role as switching expressions do in circuit design. A circuit designer can represent a circuit as a set of switching expressions and then apply the laws of Boolean algebra to transform this representation in an effort to find a more efficient circuit.

5.2.2 Algorithm-size-independent Arrays

In the previous section we have seen the steps of turning the input specification of matrix multiplication into systolic arrays. From our understanding, systolic arrays may not be amenable to practical applications since the implementation cost is potentially tremendous. As a result, there are few systolic arrays having been constructed of which an example can be found in [FGL90].

A generalization of application of drivers to 3-D MMG is given in Section 3.4.2. The results of having applied various drivers to the mesh graph descriptor of a 3-D MMG are the descriptors corresponding to the new processor array family implementing the given algorithm. Figure 3.8 gives a nearly full spectrum of the driver application results. We are about to see in this section the algorithm-size-independent arrays which are implementations of the matrix multiplication algorithm.

The aforementioned synthesized examples are systolic versions of the implementation. Suppose the given algorithm is of greater dimension, our method is also capable of mapping a sizable MMG into small processors. Figure 5.9 shows the result of mapping a 6×6 case into 4 processors.

We have discussed the assignment issues , i.e., the application of transformations to the MGD descriptors and the derivation of the mapped processing arrays. However, not until we finish scheduling task can the performance of the synthesized arrays be obtained and analyzed. As we have seen, the nodes in the original MMG graph, when assigned to processors in distinct manner, can result in different processor designs. To find out the total time steps for the parallel array to execute the algorithm, we tag the nodes with numbers assigned to each processor in a sequential order since there is only one computation unit inside each processor. Computations among different processors can be carried out in parallel. However, if an input data for a particular node in a processor is the result coming out of another processor, this node has to wait for execution until the previous computation is finished. In some circumstances, this input data is passed by the previous processor without processing. We call this type of data which passes through a node without being modified *transmittent data*.

By taking advantage of the transmittent data, we don't have to necessarily schedule the primitive nodes in a uniform manner, like linear scheduling schemes usually provide. Once a processor finds that the incoming data is again to be used unchanged by next adjacent processor, it should be able

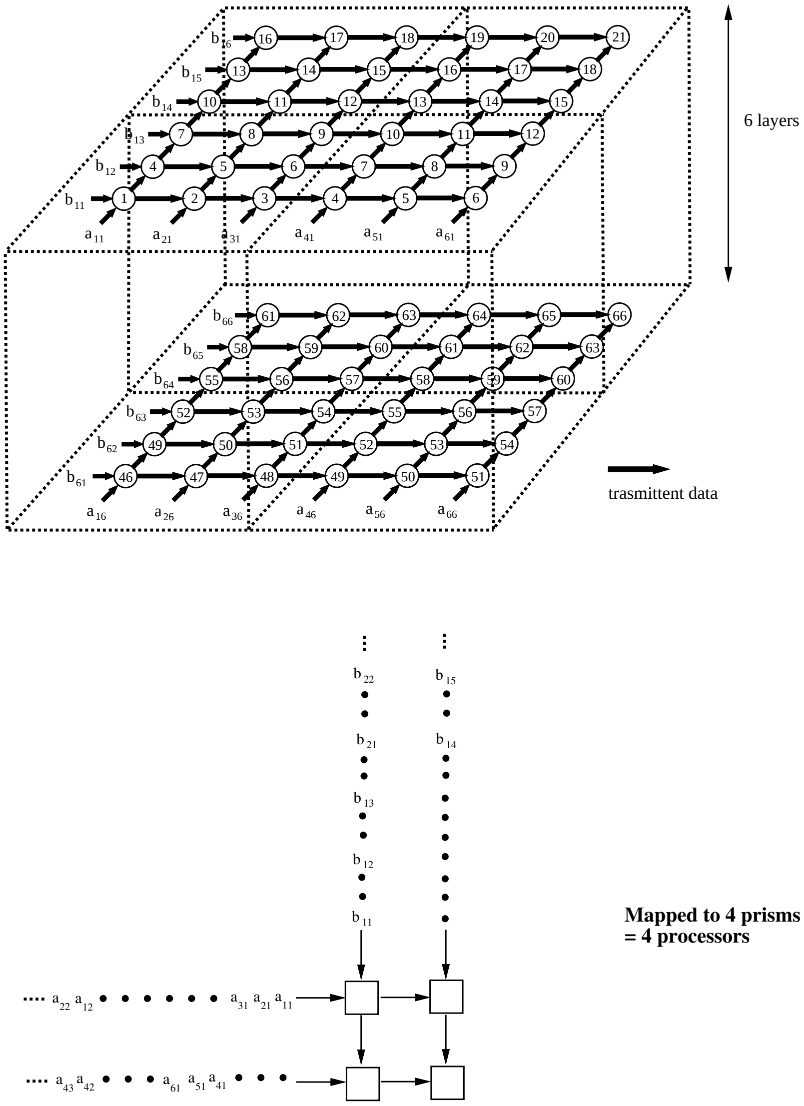

Figure 5.9: Conventional method: map a 6×6 matrix multiplication example to 4 processors.

Scheme	6×6 (to 4 proc.)	9×9 (to 9 proc.)	100×100 (to 10 proc.)
Linear scheduling	66	105	1990
Speed-delivery scheduling	56	85	1018

Table 5.1: Performance comparison of linear and speed-delivery scheduling.

to pass the data immediately after receiving it without having to wait for the current computation to finish. We call this type of scheduling "speed-delivery". By adopting speed-delivery approach, the same MMG can be scheduled more efficiently as can be illustrated by Figure 5.10, in which only 56 steps is needed for 4 processors to finish the same job. The linear scheduling approach would have taken 66 time steps in total. By now one drawback associated with linear scheduling approach is that it may not be thorough enough to utilize some of the useful properties implied by the given algorithms. The approach we adopt, on the contrast, can be fine-tuned to take into account these properties so as to achieve better performance. MGD is good tool for the implementation of the stated methodology since it is fully unfolded and covers more details of the input algorithm.

The primitive nodes in the original MMG graph, when assigned to processors in distinct manner, could result in different processor designs. By taking advantage of the *transmittent data* in the MMG graph, we are able to make early scheduling for the nodes that have been assigned to different processors. This effect is shown in Figure 5.10 in which only a total of 56 steps are required to finish matrix multiplication of dimension 6×6. We call this scheme "speed-delivery". Most of the matrix algorithms come with at least one input matrix that runs through the MMG as the transmittent data. For instance in the matrix multiplication case, our speed-delivery scheme shows the speedup over conventional linear scheduling scheme(see Table 5.1).

As mentioned, there are essentially two approaches to map the mesh graph into algorithm-size-independent array, namely, coalescing and cut-and-pile. In Figures 5.9 and 5.10, we have introduced the scheme of coalescing for matrix multiplication. Figure 5.11 demonstrates the application of cut-and-pile scheme to the example of 6×6 matrix multiplication and map it to a

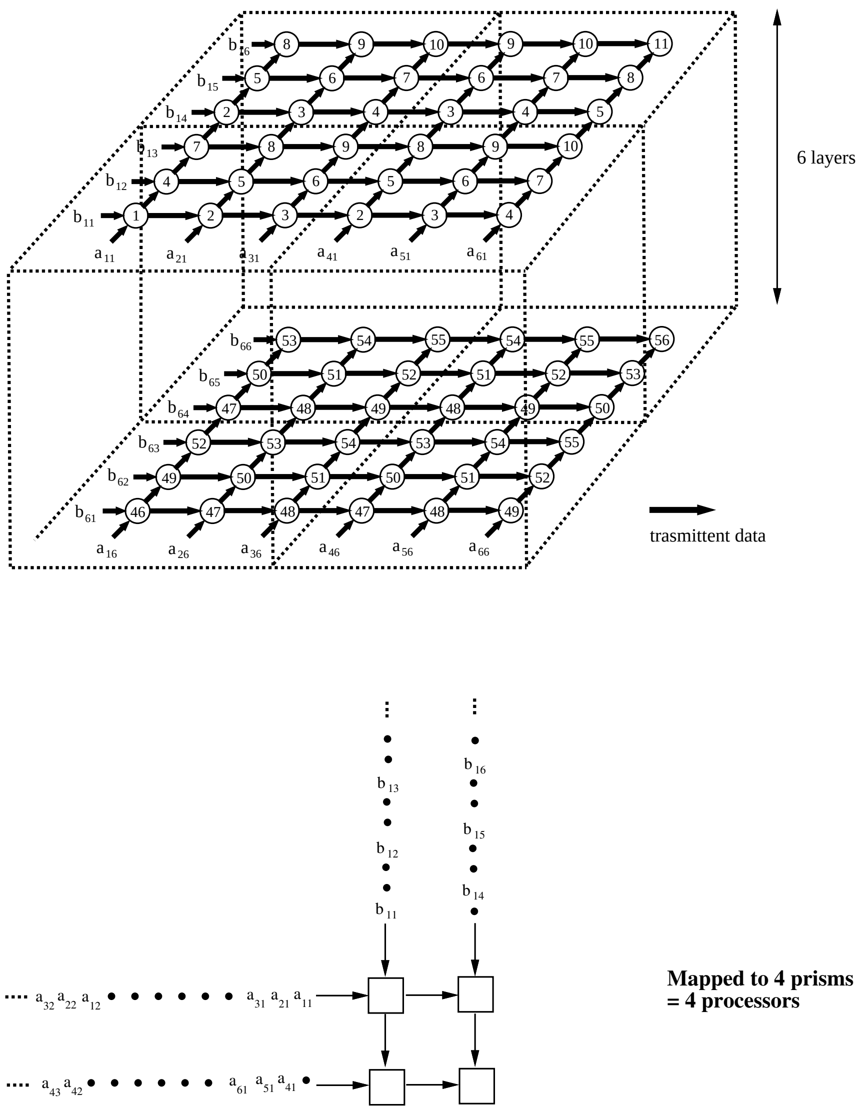

Figure 5.10: Map a 6×6 matrix multiplication example to 4 processors using speed-delivery approach.

3-node linear processor. The mesh graph is partitioned into two large prisms. Unlike the coalescing scheme in which each prism is mapped to a processor, each one of the prisms in the cut-and-pile approach is mapped to the entire processor array. A prism can only be scheduled until its predecessor has been finished by the array processor, thus the name "cut-and-pile." Note that FIFO (first-in-first-out) queues are usually needed for processors synthesized via cut-and-pile approach.

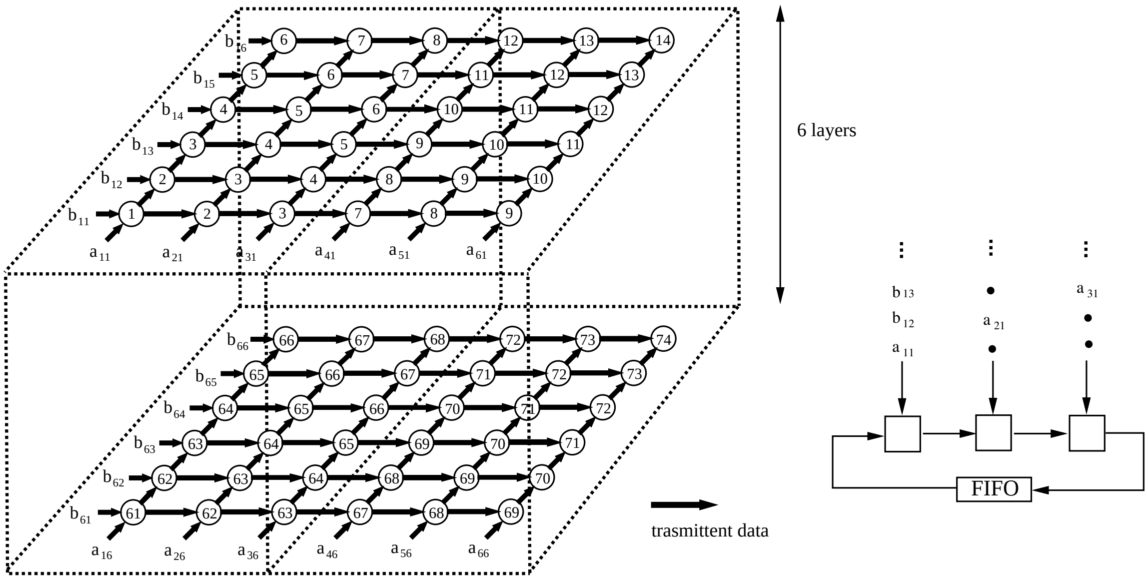

Figure 5.11: Map a 6×6 matrix multiplication example to 3 processors by cut-and-pile method.

Table 5.2 gives a comparison of the coalescing and cut-and-pile approaches over this 6×6 matrix multiplication example. The array processor figures are omitted for cut-and-pile approach since it is not practical to use cut-and-pile approach in mapping mesh graphs into rectangular arrays. This is so because the utilization rate for the array-type cut-and-pile scheme cannot achieve over 50%. Suppose a large MMG is to be mapped to a 2×2 array using cut-and-pile method, only up to 2 processors at any point of time would be kept busy. The utilization rate is even lower for a larger synthesized array.

Scheme	2-node linear	3-node linear	4-node array	6-node array	9-node array
Coalescent	110	74	56	39	28
Cut-and-pile	109	74	–	–	–

Table 5.2: Execution steps for arrays synthesized by coalescing and cut-and-pile approaches based on a 6×6 matrix multiplication example.

5.3 LU Decomposition

LU Decomposition is an important method in linear algebra for obtaining the solutions of simultaneous equations. Suppose we are able to write the matrix A as a product of two matrices,

$$L \cdot U = A$$

where L is in lower triangular form and U in upper triangular form. For the case of a 4×4 matrix A, for example, the above equation would look like this:

$$\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

We can use a decomposition to solve the linear set

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b$$

by first solving for the vector y such that

$$L \cdot y = b$$

and then solving

$$U \cdot x = y$$

There is another version of LU decomposition algorithm from which single-assignment algorithm can be also derived [Mol83]. We didn't choose this version because it only calculates the reciprocal of the diagonal of the U matrix. This was to save the division operations in the MMG graph and to leave the computation to the host to which the array is connected. We use the following recursive single-assignment algorithm for LU decomposition, which is an operationally self-contained version.

Recursive Single-assignment Algorithm:

$$
\begin{aligned}
&\textit{initialization: } a(i,j,0) = A_{ij} \\
&\text{S1: } \textit{for } k = \textit{1 to 4 do} \\
&\quad \text{S2: } \textit{for } j = k \textit{ to 4 do} \\
&\quad\quad \text{S3: } \textit{for } i = k \textit{ to 4 do} \\
&\quad\quad\quad \text{S4: } u(i,j,k) = \begin{cases} a(i,j,k-1) & \text{if } i = k \\ u(i-1,j,k) & \text{otherwise} \end{cases} \\
&\quad\quad\quad \text{S5: } l(i,j,k) = \begin{cases} a(i,j,k-1)/u(i,j,k) & \text{if } j = k \\ l(i,j-1,k) & \text{otherwise} \end{cases} \\
&\quad\quad\quad \text{S6: } a(i,j,k) = a(i,j,k-1) - l(i,j,k) \times u(i,j,k)
\end{aligned}
$$

The first synthesis task is to determine the dimension of the algorithm, which is three in this case. The second task to be done by MAP module upon examining the input algorithm would be to determine the assignment of the coordinates in accordance with the loop variables. Both the given and resultant matrices are assigned in the order of Z^+, X^+, and Y^+. If there is one matrix as input, it flows into Z^+ direction and the two resultant matrices get cranked out along X^+ and Y^+ directions. If there are two input matrices, one goes into Z^+ and the other into X^+ with the result comes out of Y^+.

From the above single-assignment algorithm, the A matrix as the input must be going the direction along coordinate tagged with k since $a(i,j,k)$ has one of its operands as $a(i,j,k-1)$. The Z^+ axis should be, therefore, labeled with k. Similarly, i and j should be situated at Y^+ and Z^+ directions, respectively.

The third step is the symbolic interpretation of the algorithm which results in the list in Table 5.3.

There are three loops in this algorithm, each of which corresponds to a plane in the MMG graph. The first iteration of k ($k = 1$) contributes to the first plane which is made up of 16 nodes. The second iteration of k implies the second plane which consists of only 9 nodes. Hence the MAP module should automatically insert 7 $(16 - 9)$ no-op nodes on the second plane in order

k	j	i	Execution statements
1	1	1	$u(1,1,1) = a(1,1,0)$ $l(1,1,1) = a(1,1,0)/u(1,1,1)$
			$a(1,1,1) = a(1,1,0) - l(1,1,1) * u(1,1,1)$
1	1	2	$u(2,1,1) = u(1,1,1)$ $l(2,1,1) = a(2,1,0)/u(2,1,1)$
			$a(2,1,1) = a(2,1,0) - l(2,1,1) * u(2,1,1)$
1	1	3	$\cdots\cdots$
1	1	4	$\cdots\cdots$
1	2	1	$u(1,2,1) = a(1,2,0)$ $l(1,2,1) = l(1,1,1)$
			$a(1,2,1) = a(1,2,0) - l(1,2,1) * u(1,2,1)$
1	2	2	$u(2,2,1) = u(1,2,1)$ $l(2,2,1) = l(2,1,1)$
			$a(2,2,1) = a(2,2,0) - l(2,2,1) * u(2,2,1)$
1	2	3	$\cdots\cdots$
	$\cdots$		$\cdots\cdots$
1	4	4	$\cdots\cdots$
2	2	2	$\cdots\cdots$
	$\cdots$		$\cdots\cdots$
2	4	4	$\cdots\cdots$
3	3	3	$\cdots\cdots$
	$\cdots$		$\cdots\cdots$
4	4	4	$\cdots\cdots$

Table 5.3: Execution traces for an LU decomposition example.

to make it *homogeneous*. Similarly, until the end of interpretation on the given algorithm, the constructed MMG should look like what is illustrated in Figure 5.12.

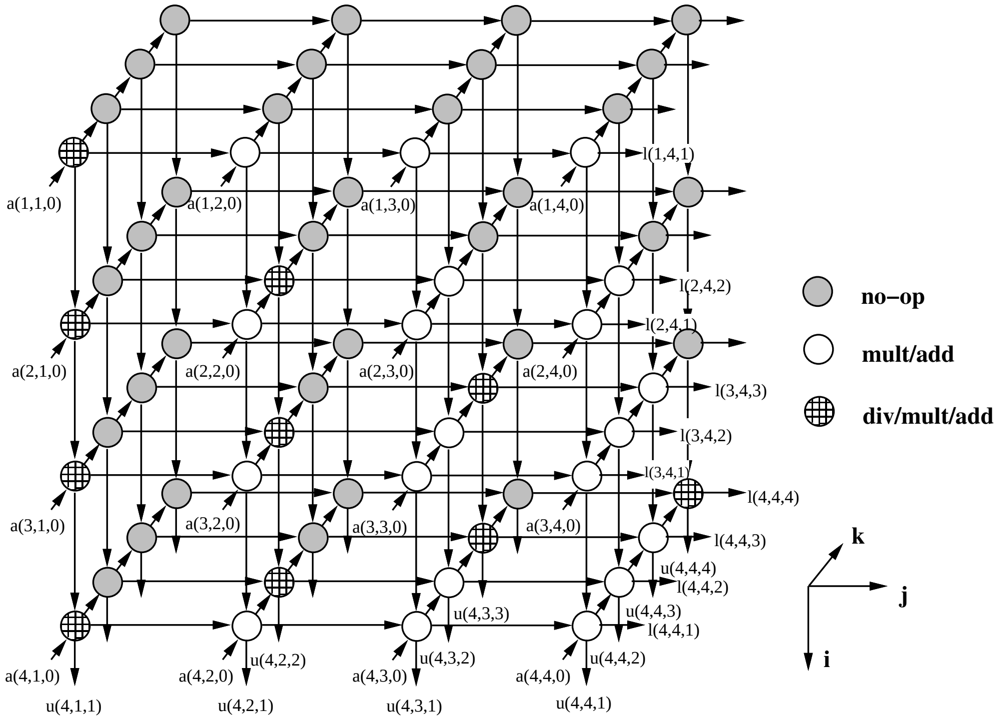

Figure 5.12: A regularized MMG of LU decomposition.

Many application specific designs potentially can be derived from this MMG. Three designs can be obtained from direct orthographical coalescing, including two triangular arrays and one 4×4 square arrays. The procedure is close to what was done in the matrix multiplication example, so no elaboration is needed.

To see clearly how the prisms of coalescing formulated, we prune off the no-op nodes in Figure 5.12 and the result is demonstrated in Figure 5.13.

The MGD generated by MAP for LU decomposition is as follows.

$$\Gamma(LU) = ((dma \downarrow dma \downarrow dma \downarrow dma) \rightarrow (ma \downarrow ma \downarrow ma \downarrow ma) \rightarrow (ma \downarrow ma \downarrow ma \downarrow ma) \rightarrow (ma \downarrow ma \downarrow ma \downarrow ma)) \nearrow ((\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow dma \downarrow dma \downarrow dma) \rightarrow (\epsilon \downarrow ma \downarrow ma \downarrow ma) \rightarrow (\epsilon \downarrow ma \downarrow ma \downarrow ma)) \nearrow$$

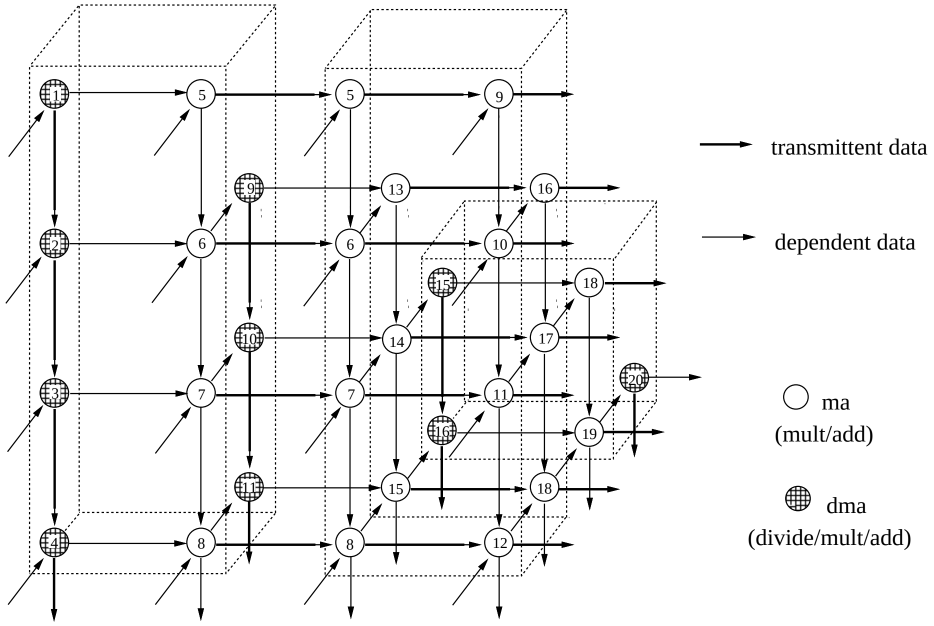

Figure 5.13: A 2×2 prism grouping of LU decomposition.

$((\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow dma \downarrow dma) \rightarrow (\epsilon \downarrow \epsilon \downarrow ma \downarrow ma)) \nearrow ((\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow dma))$

This MGD can be folded into *power form* as this LU MMG is of scalable type.

$$\Gamma(LU) = (dma^{4\downarrow} \rightarrow ma^{4\downarrow} \rightarrow ma^{4\downarrow} \rightarrow ma^{4\downarrow}) \nearrow (\epsilon^{4\downarrow} \rightarrow (\epsilon \downarrow dma^{3\downarrow}) \rightarrow (\epsilon \downarrow ma^{3\downarrow}) \rightarrow (\epsilon \downarrow ma^{3\downarrow})) \nearrow (\epsilon^{4\downarrow} \rightarrow \epsilon^{4\downarrow} \rightarrow (\epsilon^{2\downarrow} \downarrow dma^{2\downarrow}) \rightarrow (\epsilon^{2\downarrow} \downarrow ma^{2\downarrow})) \nearrow (\epsilon^{4\downarrow} \rightarrow \epsilon^{4\downarrow} \rightarrow \epsilon^{4\downarrow} \rightarrow (\epsilon^{3\downarrow} \downarrow dma))$$

After further combining the terms, we have

$$\Gamma(LU) = ((dma^{4\downarrow} \rightarrow (ma^{4\downarrow})^{3\rightarrow}) \nearrow (\epsilon^{4\downarrow} \rightarrow (\epsilon \downarrow dma^{3\downarrow}) \rightarrow (\epsilon \downarrow ma^{3\downarrow})^{2\rightarrow}) \nearrow ((\epsilon^{4\downarrow})^{2\rightarrow} \rightarrow (\epsilon^{2\downarrow} \downarrow dma^{2\downarrow}) \rightarrow (\epsilon^{2\downarrow} \downarrow ma^{2\downarrow})) \nearrow ((\epsilon^{4\downarrow})^{3\rightarrow} \rightarrow (\epsilon^{3\downarrow} \downarrow dma))$$

Considering the four plans correspond to the values of k, $k = 1$ to 4, we have the following final power form of the MGD,

$\Gamma(LU) = ((\epsilon^{4\downarrow})^{(k-1)\rightarrow} \rightarrow (\epsilon^{(k-1)\downarrow} \downarrow dma^{(5-k)\downarrow}) \rightarrow (\epsilon^{(k-1)\downarrow} \downarrow ma^{(5-k)\downarrow})^{(4-k)\rightarrow})^{4\nearrow}$

For algorithms that belong to homogeneous and quasi-homogeneous classes, their descriptors can be also coalesced into power form. In other words, the sizes of the descriptors are fixed even for sizable mesh graphs. Should the LU decomposition grow to 1000×1000, the descriptor for that could be obtained from previous one simply replacing power 3 by 999.

The proposed notation tool–MGD (mesh graph descriptors) are a useful and formal tool to justify the regularity of mesh graphs (be it 1-D, 2-D, or 3-D). The power forms of the descriptors serves as a basis for arbitrarily expanding the mesh graph to any magnitude.

To see how LU decomposition of matrix 4×4 can be done in limited implementation with only 4 processors allowed, we apply Driver (vi) depicted in Chapter 3 to the following MGD of LU decomposition. Suppose we take prisms of size 2×2 along Y^+ direction. For instance, the MGD after being applied by Driver (vi) is as follows.

$(P_1 \rightarrow P_2) \nearrow (P_\epsilon \rightarrow P_3)$, where

$P_1 = ((dma \downarrow dma \downarrow dma \downarrow dma) \rightarrow (ma \downarrow ma \downarrow ma \downarrow ma)) \nearrow ((\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow dma \downarrow dma \downarrow dma))$

$P_2 = ((ma \downarrow ma \downarrow ma \downarrow ma) \rightarrow (ma \downarrow ma \downarrow ma \downarrow ma)) \nearrow ((\epsilon \downarrow ma \downarrow ma \downarrow ma) \rightarrow (\epsilon \downarrow ma \downarrow ma \downarrow ma))$

$P_\epsilon = ((\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon)) \nearrow ((\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon))$

$P_3 = ((\epsilon \downarrow \epsilon \downarrow dma \downarrow dma) \rightarrow (\epsilon \downarrow \epsilon \downarrow ma \downarrow ma)) \nearrow ((\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow \epsilon) \rightarrow (\epsilon \downarrow \epsilon \downarrow \epsilon \downarrow dma))$

A *boundary* node defined to be a node with at least one input from other prism. For instance, in Figure 5.13, the node with input $a(1,3,0)$ is a boundary node. The *execution predecessors* of a node (say, n) is defined to be the nodes that must be executed before n can be scheduled for execution. The execution predecessors for a non-boundary node are the nodes pointing directly to that node. We call these nodes *immediate predecessors*. An extra procedure is needed to find the execution predecessors for a boundary node (say, b). If an immediate predecessor for b is from another prism and the data path between the two nodes is transmittent, the execution predecessor for b in this prism would be the node that produces the transmittent data.

To schedule the nodes in a mesh graph, only one rule is applied. That is, a node can be scheduled for execution after all its execution predecessors have been finished.

In Figure 5.13, the scheduling for the nodes are also illustrated. The number associated with each node is the time step to execute that particular node, with the assumption that each node takes one time step to finish. The node pointed by $a(1,3,0)$ should be executed at step 5 due to the transmittent path from prism P_1. Processor P_3 will be idle at step 17 since a node in it has to wait for one of its execution predecessors in P_2 to finish. As a result, execution time would take a total of 20 steps for this LU decomposition example. An advantage of adopting symbolic synthesis is that the procedure of scheduling can be done by the task of tagging the MGD. All the ϵ nodes are skipped during tagging.

By tagging the MGD descriptors, the scheduling parameters can be more accurately derived, which is a feature not found in most existing index transformation approaches. In addition, we may even assign uneven execution time to different types of primitive nodes. For instance, if we assume it takes 3 steps for *dma* nodes to finish and 1 step for *ma* nodes, it would take a total of 27 steps for the synthesized 3 processors to finish this 4×4 LU decomposition example.

We have only three physical processors P_1, P_2, and P_3. The synthesized array is shown in Figure 5.14. Note that one prism (named P_ϵ) is not shown since it only consists of ϵ nodes and it certainly won't be mapped to a physical processor due to the definition of ϵ nodes.

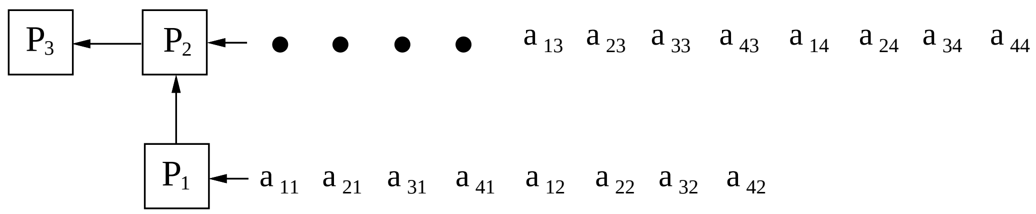

Figure 5.14: The synthesized three-processor array for 4×4 LU decomposition.

The ALIAS Array Screen for an LU Decomposition Example

An array screen for the LU Decomposition algorithm (4×4) is shown in in Figure 5.15.

Minimal Formulation Sets

With the aid of mesh descriptors, a new concept can be formally defined which is a useful reasoning basis for our synthesis environment.

Definition 14 (Minimal Formulation Set) *Minimal Formulation Set (MFS) exists for each and every mesh graph in homogeneous and scalable classes. For any MMG of these, it is the smallest MMG in size that can be, by means of its descriptors, generalized to arbitrarily large MMG which performs exactly the same function for larger data set. The greatest length along either X, Y, or Z axis is defined as the Minimal Formulation Measure (π).*

Accordingly, the minimal formulation measure for LU decomposition is $\pi_{LU} = 3$. Intuitively, an LU-decomposition MMG graph of dimension 3×3 is the smallest possible graph a user or a program may perceive the homogeneity of LU-decomposition. For matrix-matrix multiplication, $\pi_{mm-mult} = 2$. This is because an MMG for this particular operation of size 2×2 is the starting point for further growth up to any size.

It is pointed out in [Rib90] that in the graph-based method [Mor89] one drawback is the use of data structures whose sizes increase with the data for the algorithm. That is, in the case of nested loops, for instance, the greater the loop bounds the larger the data structures. The concept of *Minimal Formulation Set* is useful in reducing the computational complexity that increases with data sizes. In many occasions as the needs arises for analyzing algorithms with large data sets, scaling up the statistics derived from working on MFS is an efficient way of obtaining numbers for the large algorithms.

5.4 The Algebraic-Path Problem (APP)

The APP problem unifies several useful algorithms in computer science, including transitive closure, shortest path, and matrix inversion. The dependence graph corresponding to the original algorithm contains multiple

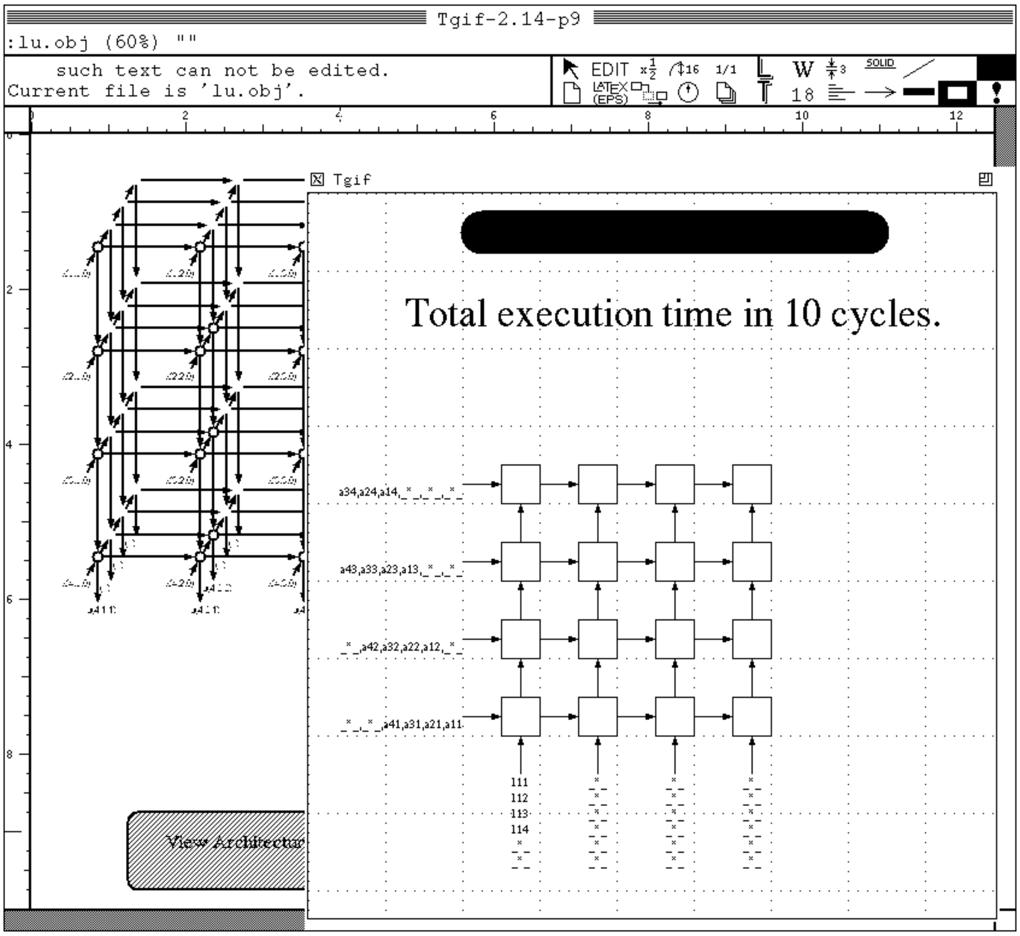

Figure 5.15: An array processor screen for a LU example.

loops. It is not suitable for the implementation in our system. However, by padding the input matrix with two copies of the identity matrix, a new single-assignment algorithm with corresponding input matrix $2n \times 2n$ can be obtained [BR90]. The algorithm is

Recursive Single-assignment Algorithm:

Initialization: $x(i,j,0) = X_{i,j}$
for $k = 1$ *to* 3 *do*
  for $i = k$ *to* $k + 3$ *do*
    for $j = k$ *to* $k + 3$ *do*

$$a(i,j,k) = \begin{cases} a(i,j-1,k) & \text{if } i \neq k, j \neq k \\ x(i,k,k) & \text{if } i \neq k, j = k \end{cases}$$

$$b(i,j,k) = \begin{cases} b(i-1,j,k) & \text{if } i \neq k \\ x(k,j,k) & \text{if } i = k \end{cases}$$

$$x(i,j,k) = \begin{cases} x(i,j,k-1) & \text{if } i = j = k \ \ or \ \ i = k, j \neq k \\ x(i,j,k-1) + a(i,j,k) \times b(i,j,k) & \text{if } i \neq k, j \neq k \\ x(i,j,k-1) \times b(i,j,k) & \text{if } i \neq k, j = k. \end{cases}$$

The drivers screen for the APP algorithm is shown in in Figure 5.16. The nodes in the MMG are of two types. The filled ones are dummy nodes. They are added to make the 3-D MMG homogeneous, thus amenable for transformations. The non-filled nodes are data dependence nodes, created directly from the APP algorithm. The drivers screen shows several options for the user to choose to project the MMG.

From the drivers screen, the user may click the architecture he wishes to implement. The user also may see the scheduling of the nodes by clicking on a particular node (see Figure 5.17). The parallel array processor design can be displayed on the screen together with the input pattern and the optimal scheduling time steps to execute the given algorithm (see Figure 5.18).

5.5 Sufficiency

In this section, we provide a few justifications for our approach. There seemingly are some drawbacks to ALIAS methodology. However, there are good reasons to believe that these drawbacks are merely minor limitations.

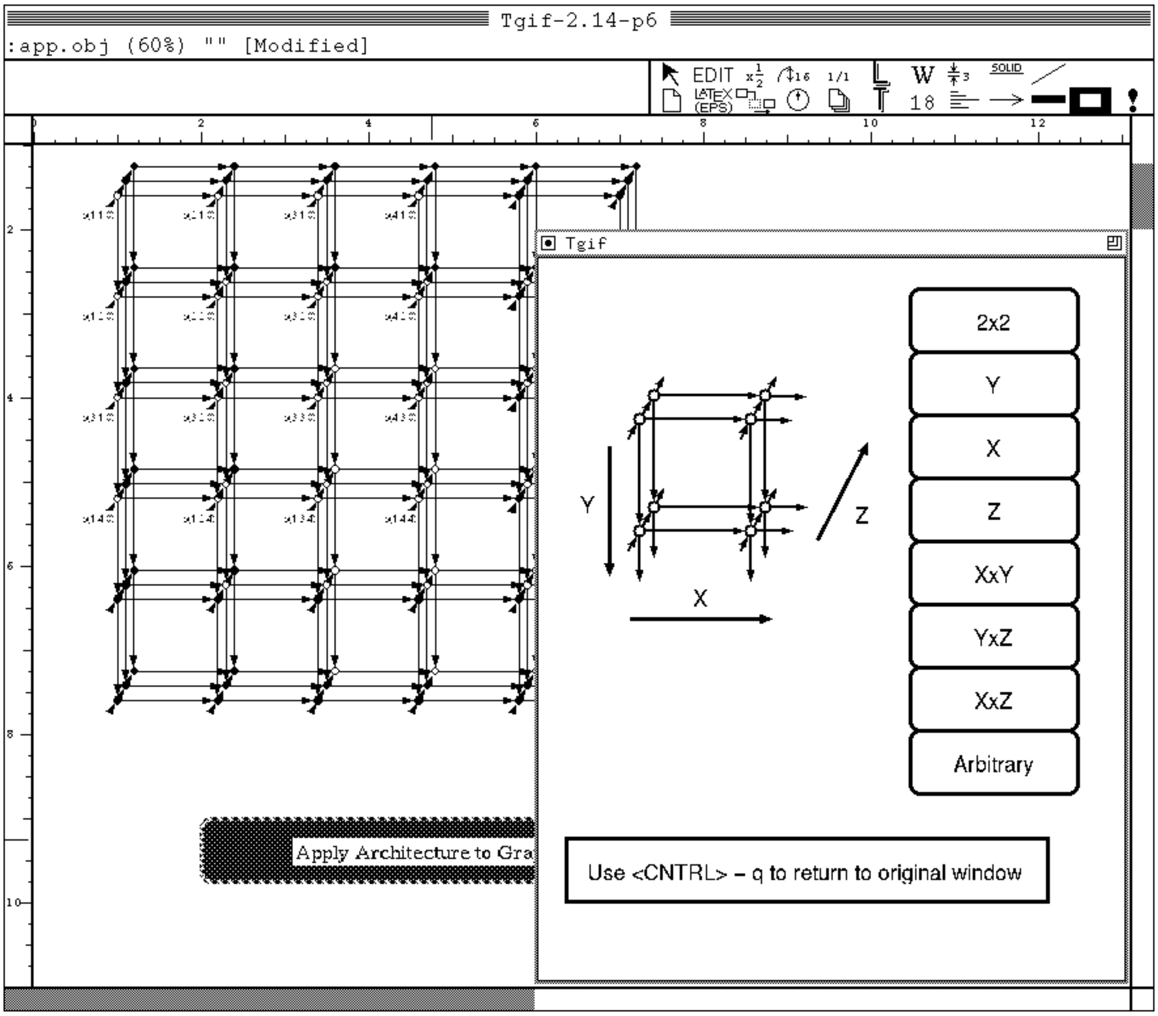

Figure 5.16: A drivers screen in ALIAS design system.

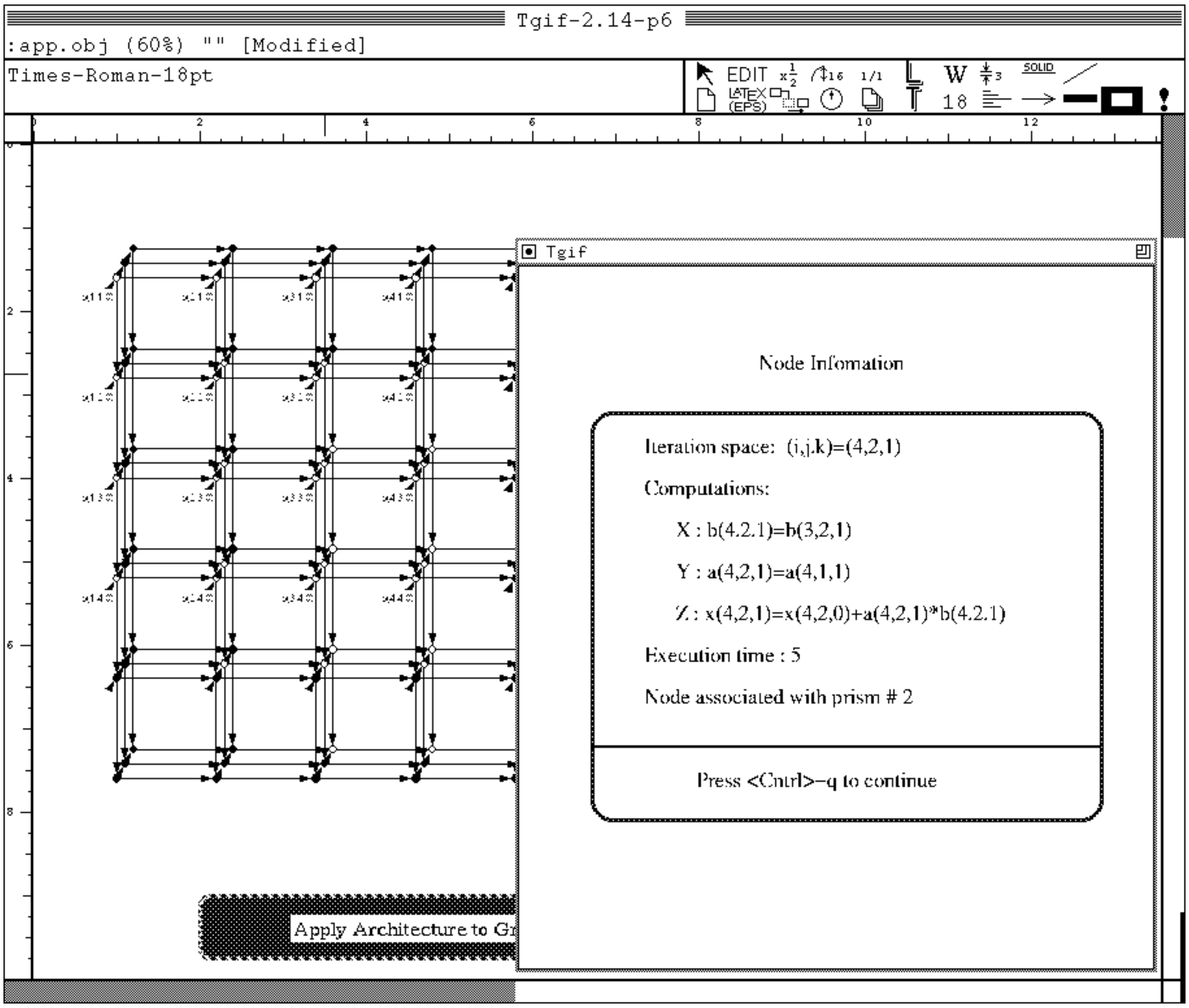

Figure 5.17: A node information screen in ALIAS design system.

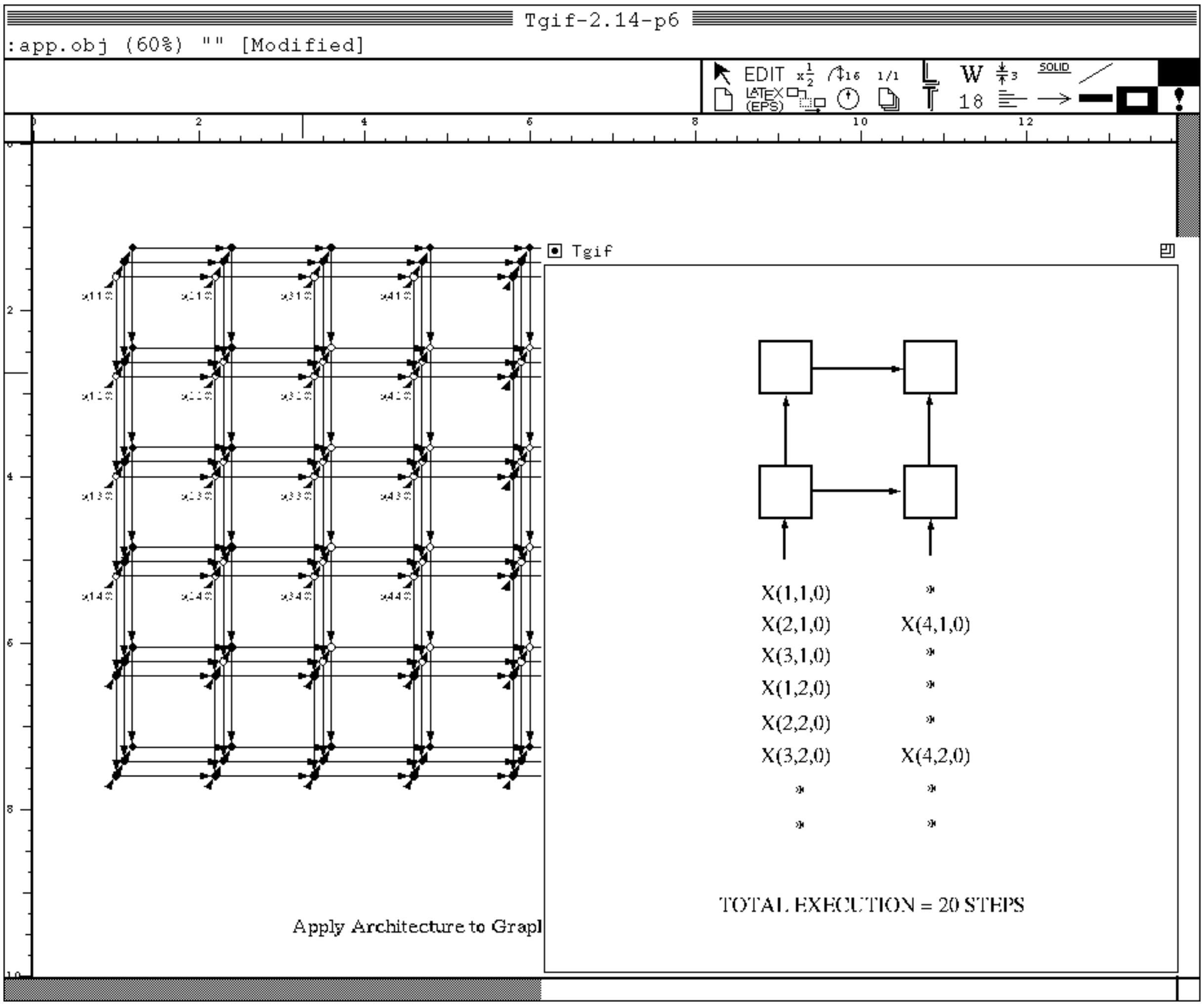

Figure 5.18: An array processor screen in ALIAS design system.

From the observation, it is concluded that the proposed methodology is self-contained and is practically a good foundation for synthesizing VLSI parallel processing arrays.

Algorithms with Higher Dimensionality

A potential question is "How can algorithms with loop levels over three be representable by multi-mesh graph? How are projections to be done?". Fortunately, the vast majority of algorithms to be mapped to array processors have up to three loop levels. We do not have to worry about how to represent higher dimensional algorithms in 3-D space. Suppose that there is an algorithm that needs to be realized in a 2-D mesh array or a 1-D linear array, one way to get around with it is to convert the algorithm into mesh graph descriptors. Then the MGD can be manipulated to produce 3-D mesh graph, which is similar to coalesce a 3-D MGD to a 2-D counterpart. If we follow the previously addressed procedure, an array in lower dimension with respect to that implied in the given algorithm can be readily obtained.

Unidirectional Designs are Sufficient

All MMG graphs specified by MGD descriptors appear to be unidirectional up to this point. Hence, after the projection direction and size of prisms being chosen, the resultant systolic array designs would end up in a unidirectional form, the reason being that the synthesized arrays are grouped macros of the MMG graphs.

Some potential questions may be raised – "Why not take bidirectional designs into account? What are the losses resulting from ignoring the subclass of bidirectional array designs?" From the literature and our design experience, there are a number of reasons why only unidirectional arrays are considered in our design methodology:

- Let's take matrix multiplication algorithm as an example. There are many systolic arrays realizing the same algorithm, some of which have bidirectional communication channels between processors [JK89]. There are also systolic designs with feedbacks. However, those designs with bidirectional links can also be realized by systolic arrays without feedback [KL84].

- Unidirectional implementations usually exhibits more favorable features in the aspects of fault tolerance, and problem decompositions [KL84].
- The design approach dealing with unidirectional designs has the merits of providing multi-level pipelining once the design level is to be refined down to bit-level. Most of the architectural mechanisms supporting pipelining are realized in an acyclic manner. Examples are such as arithmetic pipelines inside math-coprocessors, and vector processors for intensive scientific computations.
- With data contra-flow (bidirectional communication pattern), only one of every two or three consecutive cells is active during each step. As a result, high utilization (efficiency) of PE execution which is a very desirable feature in systolic or quasi-systolic design is difficult to achieve [NLV87].

Generality

We have seen many design tools or methodologies proposed in the literature that are not general enough (ad hoc) since conceivably limited cases can be applied. We started our work by building an adaptable and generalized framework in hope that it is not only innovative but also useful. Being a synthesis tool designer, one should ask oneself the following questions:

- Is the proposed method general enough to cover the class of algorithms that would interest array processor designers? Or is it just confined to the type of algorithms that were especially targeted for the approach without too much practicality?
- How good are the designs that are synthesized from the proposed system?

In the reported theoretical work on the topic of algorithm mapping, few examples are demonstrated to be working and sometime these cases are not good to public interests. We believe that a different perspective should be taken in constructing a practical synthesis tool since the synthesized designs should be at a comparable level as do those designs mechanically coming from human designers.

Being a designer of a synthesis tool, one may ask a question "How good are the designs my system can generate?". A preferable design aid should not only produce the designs that fulfill the goal but also to be able to provide optimal ones. Furthermore, a justification should be given. Naturally, designs synthesized by a particular tool are dictated by its principles in the background.

Our method starts with the algorithm and produces one and the only one MMG that specifies the intrinsic data dependencies originated from the given algorithm. The MMG also can be viewed as a partial order graph which also specifies the sequence of the execution in the given algorithm. In view of the fact the execution sequence can be somehow embedded in components, various systolic arrays were proposed to implement algorithms for matrix computations or other types. As a matter of fact, each and every systolic array is a space-time rearranged structure and thus can be synthesized by our proposed approach.

We believe that our tool can provide optimal implementations of systolic array for the goal algorithm. Hence, ALIAS is said to be *systolic complete* since every systolic structure in the class will be considered by the tool at the synthesis stage. ALIAS is capable of producing more favorable systolic designs carrying out the system requirements.

5.6 Summary

This chapter covers two typical matrix computation cases to which our proposed methodology has been applied. In the more complete matrix-multiplication example, the entire flow of obtaining the final systolic designs starting from the original algorithm has been demonstrated. First, the inside out scanning approach is applied to the parametrized algorithm in which index space has been increased one fold. A 3-D multi-mesh graph illustrating data dependencies as well as its associated MGD can be readily created following the systematic symbolic interpretation. Secondly, the Descriptor Manipulator module of ALIAS can apply the transformers to the MGD in an effort to obtain different systolic designs. Designs of mesh type are results from orthographical projections. Hexagonal designs can be derived from diagonal projection. The method is said to be *systolic complete* for all the systolic designs realizing the objective algorithm can be accordingly synthesized.

Chapter 6

The ALIAS Design Aid System

The ALIAS design system based on the methodology that is described in the previous chapters has been implemented in programming language C on a Sun SparcStation. The synthesized designs are displayed in TGIF which is an Xlib-based drawing facility.

The system consists of three main modules: Matrix Algorithm Parser (MAP), Descriptor Manipulator (DM), and Display Interface (DIP). They are written in C. MAP takes input algorithms in the single-assignment form and by symbolic execution produces a linked list which correspond to the descriptors of the algorithms. After the parsing of the input is done, the DIP module displays the 3-D multi-mesh graph on the screen, along with a menu that allows the user to select the desired operations. After the user chooses the menu on the screen, the DM module applies transformers to the descriptor and thus different array implementations can be obtained. The array corresponding to the user selection will be displayed on the screen together with its scheduling parameters. The synthesis steps may be repeated, i.e., the user can quit displaying one array implementation and continue with other options on the menu.

In this chapter, we will see how ALIAS works and the implementation details of this system.

6.1 The MAP Module

The MAP module of ALIAS takes the input file with extension *.saa* and analyzes each token. It rejects the input and display diagnosis of error location and type if input format does not conform to the specification. If no errors are found, MAP proceeds in two stages: 1) it parses the algorithm and creates the data structure for the algorithm; 2) it symbolically executes the data structure and creates a linked list that is equivalent to the MGD of the mesh graph to be displayed.

The input to the design system is the single assignment algorithms (SAA) of matrix computations whose grammar in BNF form is specified in Table 6.1. Single-assignment algorithms conformed to this grammar can be found in Chapter 5.

<table>
<tr><td>SAA</td><td>::=</td><td>input output forclause stmtclause</td></tr>
<tr><td>input</td><td>::=</td><td>“input:” lst_mtrx</td></tr>
<tr><td>output</td><td>::=</td><td>“output:” lst_mtrx</td></tr>
<tr><td>lst_mtrx</td><td>::=</td><td>alph <“,” lst_mtrx | “;”></td></tr>
<tr><td>forclause</td><td>::=</td><td>for_st for_st <for | nil></td></tr>
<tr><td>stmtclause</td><td>::=</td><td>stmt stmt <stmt | nil></td></tr>
<tr><td>for_st</td><td>::=</td><td>“for” ind “=” <const | ind> “to” <const | ind> “do”</td></tr>
<tr><td>ind</td><td>::=</td><td><“i” | “j” | “k”></td></tr>
<tr><td>const</td><td>::=</td><td><“1” | “2” | “3” | ....></td></tr>
<tr><td>stmt</td><td>::=</td><td>array_el “=” expr</td></tr>
<tr><td>array_el</td><td>::=</td><td>alph “(” ind_exp “,” ind_exp “,” ind_exp “)”</td></tr>
<tr><td>alph</td><td>::=</td><td><“a” | “b” | “c” | ....></td></tr>
<tr><td>expr</td><td>::=</td><td>array_el << “+” | “−” | “∗” | “/”> expr | nil>
<“if” ind rel ind < “;” expr | nil> | nil></td></tr>
<tr><td>rel</td><td>::=</td><td><“==” | “ =” | “<” | “>” | “>=” | “<=”></td></tr>
<tr><td>ind_exp</td><td>::=</td><td>ind | ind <“+” | “−”> const</td></tr>
</table>

Table 6.1: Grammar of SAAs.

In the first stage, the MAP module determines the direction of flow for the array variables on the 3-D mesh graph. The MMG cannot be generated and properly displayed on the screen without assigning the coordinates to the index space. Each of the X, Y, asnd Z axes is labeled with either one of the indices. Having finished parsing, MAP creates, among other things, a table IND for indices and a table STAT for the statements. Each of the entries in STAT contains SUBC which is the count of the substatements, a table SUBSTAT for substatements, variable XYZ indicating axis direction that the target array variable points to, and a table TARGET for the target array variable along with its indices. Each of the entries in IND consists of variables XYZ to which the index points, LABEL which is the index, START which is iteration starting value, and END which is the iteration end value.

In the second stage, MAP first creates a table of iteration space for loop variables. Then it iterates over the maximal original index space to construct the linked list as a representation for the mesh graph descriptors. Each index point of the maximal original index space needs to be iterated since the computation nodes of the algorithm may form an irregular polyhedron and dummy nodes need to be created to hold the computation nodes in a 3-D cube.

The linked list constructed by MAP after symbolically executing the algorithm is pointed by MGD_HEAD. Through this pointer the entire mesh graph for the algorithm can be accessed. The linked list can be also viewed as a *complete n-ary* tree [HS78], in which n is the dimension of the input algorithm[1]. There are two types of nodes in the linked list: computation nodes (leaf) and composer nodes (non-leaf). A composer node contains the following fields:

- COMP: shows the axis of coordinates it points to.
- CHILDCNT: contains the number of its immediate children nodes.
- SIBLING: points to the next composer node at the same tree level, i.e., bound by the same constructor.
- CNCHD: points to the child composer node at a lower level.
- MGNCHD: points to the child computation node at a lower level.

A computation node contains the following fields:

[1] For algorithms with nonuniform index space, the number of the nodes at each level of the tree may differ.

- ITER: a table that shows the iteration space for the computation associated with the node.
- Xtarget: contains the array element assigned to the output along X axis.
- Xeval: contains the evaluation expression to be assigned to the X output.
- Ytarget: contains the array element assigned to the output along Y axis.
- Yeval: contains the evaluation expression to be assigned to the Y output.
- Ztarget: contains the array element assigned to the output along Z axis.
- Zeval: contains the evaluation expression to be assigned to the Z output.
- SIBLING: points to the peer computation node.
- PRISM: indicates which prism the node is assigned to.
- EXEC: indicates the execution time step for this node.

Figure 6.1 shows a typical representation for the linked list along with fields contained in both the composer node and the computation node.

Tree-like linked list structure is used in ALIAS to represent the descriptors. Another alternative considered is the use of text strings as the representation of descriptors. Each literal then can be associated with a pointer that points to a cell containing the necessary information (execution time step, computation information, etc.). A major disadvantage of this technique is the difficulty of applying transformers to the text strings. Another potential disadvantage is that the nested constructor structure incurred by transformers application is hard to be verified and debugged.

$$((a_1 \downarrow a_2) \rightarrow (a_3 \downarrow a_4)) \nearrow ((a_5 \downarrow a_6) \rightarrow (a_7 \downarrow a_8))$$

(a)

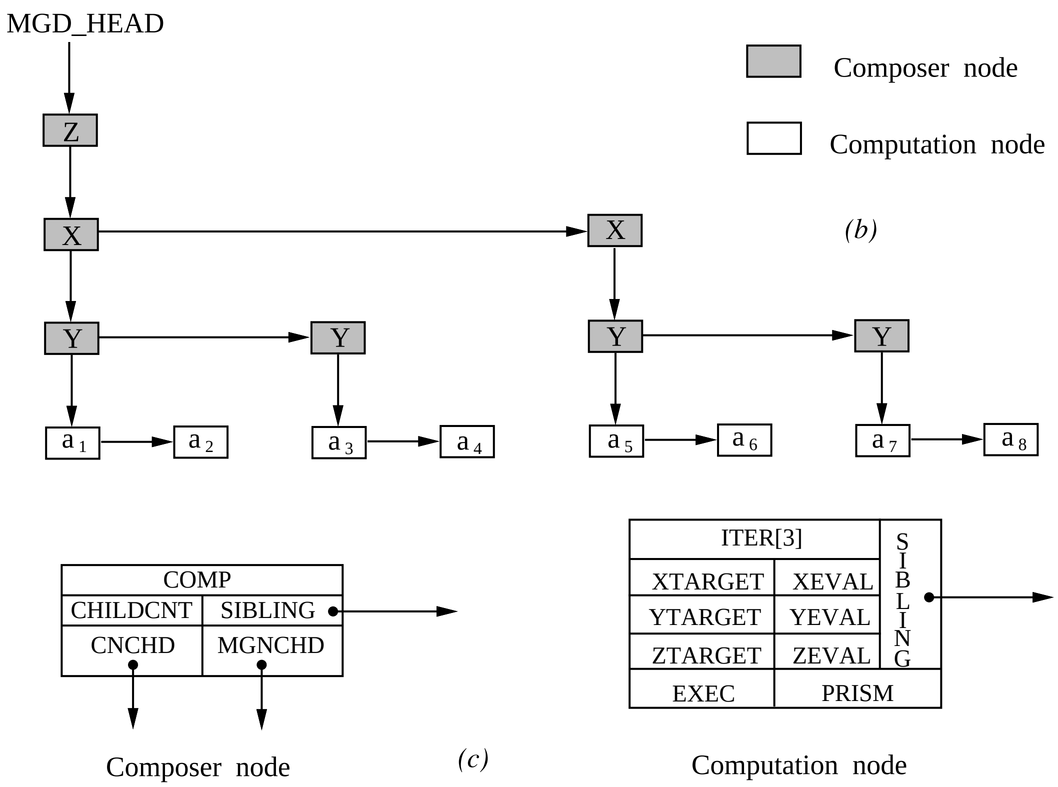

Figure 6.1: An example of an MGD and the corresponding data structure: (a) the descriptor, (b) the linked list representation in ALIAS system, (c) fields in composer node and computation node.

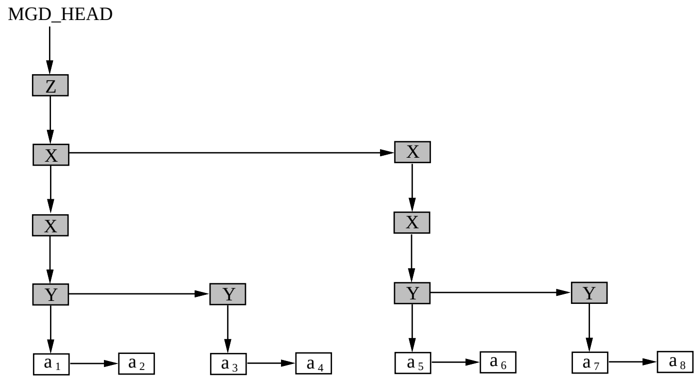

Figure 6.2: The effect of applying *split* on the MGD list

6.2 The DM Module

Upon leaving MAP, two files are created for the Descriptor Manipulator module (DM). They are "node.sch" which contains the scheduling and prism (grouping) information, and "node.lst" which contains the linked list. DM module performs *transformer application* and *node scheduling*. The four necessary transformations we use are *pair*, *split*, *transpose*, and *group*. Each call to the transformation procedures has the following parameters:

transformer_x(mgd_head, level),

where the variable *level* is the indication of how far down the tree structure of the linked list the transformation should be applied and the variable *mgd_head* is a pointer that points to the head of the linked list.

When *pair* is called, after dropping *level* levels of the tree, new composernodes are inserted. The level below new nodes is denoted by α. β refers to the level where new composernodes just created. Then α is broken into sets of two, and one set is attached to each of the composernodes of β. Figure 6.2 shows the change to part (b) of Figure 6.1 after the transformer split has been applied.

When *split* is called, after dropping *level* number of levels of the tree, two new composernodes are inserted. Then α is broken into two equal sets, and one set is attached to each of the composernodes of β. In the example here, the effect of applying split is equivalent to applying pair.

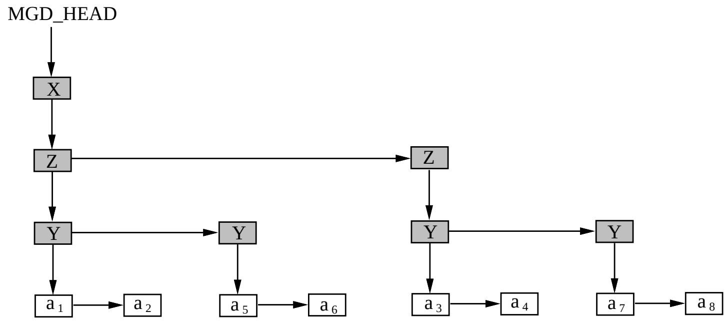

Figure 6.3: The effect of applying *transpose* on the MGD list

The transformer *transpose* unlike *split* and *pair*, does not add any new composernodes, rather, it simply transposes the two directions from parameter level passed to the procedure. Figure 6.3 shows the change to part (b) of Figure 6.1 after transpose has been applied. Note that sequence of composer nodes Z and X has been altered, so is the sequence of the computation nodes.

The scheduling transformation *group* contains the following header:

*(*mgd_head, trav, level)*

The two integers passed, represent the level at which the prisms are to be defined (*trav*) and the total number of levels present in the tree structure (*level*). While the linked list is a perfect data structure for applying transformations, it does not allow the maneuverability required for scheduling. Scheduling is comprised of two tasks: the first is assigning a set of nodes to a particular prism, and the second is to number the nodes both with respect to the total architecture and to the specific prism within which the node resides. Numbering is done sequentially, therefore it is essential that movement from one node to another be easy, and uninhibited. The tree structure provides easy movement in one specific direction only, i.e., the direction of the tree. To remedy the need for maneuverability, a 3-D array of *holdertype* was created.

holdertype is a structure containing the following fields:

- EXEC: the scheduled execution time of the node.
- PRISM: the # assigned to the prism that this node belongs to.

- MG: this variable is a pointer to the mgdnode within the tree structure.
- XT,YT,ZT: a boolean integer flag that signals whether data from a particular direction is transmittent or not. (0 = true, 1 = false)

When the function *group* is called, division of the tree into prism has already been accomplished by the action transformations. Now the linked list is traversed to the node that begins a particular prism, and that prism is than stored in the 3-D array. In assigning the scheduling, there are 6 possible patterns for numbering to take place. This number is arrived via knowledge that there are 3 possible directions to traverse, and thus the number of paths is $\{$*number of directions*$\}! = 6$. In order to ensure that we have found the optimal scheduling, all 6 directions are examined. The pattern which yields the lowest schedule time will be chosen and displayed on the screen. The following rules are followed, despite the sequential numbering.

1. Transmittent data paths require only 1 processing time step despite the number of nodes across the prism.
2. Null nodes are assigned an EXEC value of -1.

6.3 The DIP Module

After the MAP module parses the algorithm, the 3-D MMG can be displayed. After viewing the MMG of the input algorithm, the user may choose to look at the drivers screen. The driver screen shows several options for the user to choose to project the MMG. A new implementation of the algorithm can be obtained by each different option on the menu. The user is allowed to specify the synthesized parallel array of any dimension by choosing the arbitrary option on the menu.

The additional driver screen can be superimposed to the 3-D MMG screen. This is made possible by the "launch_to" command in TGIF, which allows the execution of UNIX commands from within the TGIF window. From the drivers screen, the user may click the architecture he wishes to implement. The user also may see the scheduling of the nodes by clicking on a particular node. This also makes use of the "launch_to" command. The final parallel array processor design can be displayed on the screen together with the input procession pattern and the optimal scheduling time steps to execute the given algorithm.

6.4 Conclusion

In this chapter the implementation details of the three modules of ALIAS, namely, MAP, DM, and DIP, are described. The grammar of input algorithms that can be accepted by MAP is defined in BNF form. The data structure created by MAP is implemented as a linked list which corresponds to the computation of the input algorithm. Examples are given on the application of the transformers in the DM module to the linked list. The function calls in the C code to implement the transformers are depicted. We use the APP problem as the input to our system and show a few X-window screen examples, which are the snapshots of the design system.

Chapter 7

Comparisons, Summary and Future Research

7.1 Comparisons

The design methodology has been presented in previous sections. It is of our interests to compare ALIAS with other design systems in the literature.

One of the common characteristics found in most synthesis tools is that transformational approaches are adopted in deriving the array designs that realize the goal algorithms. It is desirable that the transformations in the methods are systematic and regular so that a wide selection of designs can be obtained. Deriving more designs does not guarantee an optimal solution, however, it is more likely in this case that a suboptimal (if not optimal) can be found. Some design systems transform the given algorithmic descriptions into arrays in a rather ad hoc manner. These systems usually cannot provide more than one solution to the goal algorithm and the single solution is known beforehand. The application of the transformations becomes the technical descriptions of the route that leads the algorithm to the unique solution. Our belief is that a good synthesis system should be more comprehensive, exhaustive, and universal.

Some of the synthesis systems require user's prior knowledge about the design characteristics during the use of the system. For instance, the algorithmic descriptions to be fed into the software need to include the desired design styles and interconnection parameters. The systems do not hide the design details from the users of the synthesis software. Some systems are highly interactive. Users are required to be involved in every decision points in the synthesis process.

Very few of the proposed synthesis tools can provide both the algorithm-size-dependent and algorithm-size-independent array designs. Many tools only provide the systolic arrays that realize the given algorithm. Furthermore, analysis of the performance parameters like untilization rates and throughput is not easily found in many systems.

We survey eleven synthesis systems: SYS [LM85], DIASTOL [FGQ86], ADVIS [Mol87], SAGE [AP87], SYSTAR [Omt88], HIFI [AD88], DECOMPOSER [HOI88], SYSTOL [CP88] , SYS^3 [HL89], MSSM [HH90], ARREST [BJ92], and compare them with ALIAS . The comparison is listed in Table 7.1.

	Systematic transformation	*Users pre-knowledge*	*Designs provided*	*Performance provided*
ALIAS	**yes**	**no**	**alg-size-dept. & alg-size-indept.**	**yes**
SYS	heuristic	required	very limited	no
DIASTOL	ad hoc	no	alg-size-dept.	no
ADVIS	yes	no	alg-size-dept. & alg-size-indept.	no
SAGE	yes	no	alg-size-dept.	no
SYSTAR	yes	no	alg-size-dept.	yes
HIFI	yes	required	alg-size-dept.	no
DECOMPOSER	no	required	alg-size-dept.	no
SYSTOL	yes	no	alg-size-dept.	no
SYS^3	yes	required	alg-size-dept.	no
MSSM	heuristic	required	alg-size-dept.	yes
ARREST	no	required	alg-size-dept.	no

Table 7.1: Compare ALIAS with other systems

7.2 Summary

In this dissertation, we propose a methodological framework that allows automatic mapping from matrix algorithmic specifications to application specific arrays. The framework identifies the major components in a design environment by the name of ALIAS as well as some details as to the implementation issues. The proposed method is good for automation for it adopts symbolic formalism as the ground for transforming and synthesizing the designs, in contrast to *index transformation approaches* which are based on complicated mathematical approach [Rao85], [Mol87]. Although these approaches are also formal, the difference is that they are of greater complexity with respect to our method. We have demonstrated that our approach is capable of the synthesis level of index transformation approach with the merits of ease of use, maintenance, automation and expansion. We are convinced that a good methodology should be intuitive, pure, and simple, similar to the characteristics and spirit found in the design of systolic arrays. Only through a wide and thorough understanding to the research public can a good methodology be accepted and adopted. It is never intended to be conclusive in our research, but rather a quest for a new direction in the area of high-level synthesis.

A notation language tool called MGDL (Mesh Graph Descriptor Language) for characterizing MMG graphs is introduced. The advantages of adopting MGD in our design environment ALIAS are as follows.

- MGD is a formalized tool as to the various operations on the mesh graphs.
- MGD nicely captures the structure and multi-level abstraction property of mesh graphs. It can be viewed as the functional domain of the corresponding array designs.
- MGD provides justification and reasoning for investigating the classification (regularity) and expandability of mesh graphs.
- From MGD MMG can be generated in a coordinate-free manner. Unlike conventional node assignment procedure in which Cartesian coordinates are indispensable, definition and application of MGD descriptors opens up a new plausible methodology.

- MGD is defined as concise and clear as possible so as to increase maintainability and usability.

In a nutshell, the goals of our research are

- To create a practical and powerful design environment for design automation so that users' intervention and knowledge can be minimized.
- To show what characteristics of derived designs can be obtained, such as implementation cost, latency, and interconnectivity.
- To demonstrate the designs synthesized by the system are invariably correct as far as the original algorithm description is concerned. This is a result of using formal transformations as the method of design.
- To illustrate the usefulness of a specifically designed graphical language MGD and how efficiently, accurately, and systematically designs can be synthesized by the language's self-contained syntax and transformations.
- To show that a design aid system based on the proposed methodology can be actually implemented which provides the end user with full spectrum of the design possibilities that meet the given algorithmic specifications.

7.3 Future Research

The work involved in this study consists of two parts, namely, the methodology development and software implementation. Though both parts have been accomplished at this point, there is still some other work worth being looked into.

1. Identify system requirements and delineate the input and output functions of ALIAS . That is, define the class of algorithms that can be recognized by ALIAS . Determine the user input to the system and what is expected to be the outcome of the system.

2. Construct algorithms that can, by examining the MGD descriptors, determine the implementation requirements as well as the performance measures such as total latency, throughput, etc. This can only be done along with proper augmentation of the MGD syntax.

3. Modify ALIAS design system so that it can also provide not only the interconnectivity of parallel arrays, but also the cell structure. The refined cell structure can include the functional units, feed back loops, queues, and I/O ports to its neighbors.

4. Elevate the functional view of the system by investigating the communication module between the synthesized parallel arrays with the potential host to which the arrays can be linked. Considerations can be placed on the preparation of the data inputs to the system and the data collection after the computation, since I/O has usually been a major issue for a massively parallel system.

5. The interactive design system now only provides feedbacks for a small class of algorithms. The class size can be easily expanded since most matrix algorithms can be translated into single-assignment form which is acceptable by the design tool.

6. The design tradeoff can be further explored. The loop size of the input algorithms to ALIAS is under the order of 10. The complexity increases as the size of the input algorithms increases as well. How to obtain a set of optimal designs under certain user-given constraints become an attractive arena to embark upon.

Appendix A

The Use of ALIAS Design System

This appendix briefly describes the procedures to use the ALIAS interactive design system. The user need to specify the matrix algorithms in the single assignment form as defined by the grammar as discussed in Chapter 6.

An example of LU decomposition is as follows:

```
for k = 1 to 4 do
   for j = k to 4 do    (for statement)
      for i = k to 4 do
      S1:  u(i,j,k)=a(i,j,k-1) if i == k ;
                               (substatement 1 of statement 1)
                    u(i-1,j,k) if i ~= k
                               (substatement 2 of statement 1)
      S2:  l(i,j,k)=a(i,j,k-1) / u(i,j,k) if j == k ;
                    l(i,j-1,k) if j ~= k
      S3:  a(i,j,k)=a(i,j,k-1) - l(i,j,k) * u(i,j,k)
```

The input to the system should be edited and saved in a file with *.saa* extension. Once the file has been specified, ALIAS can be called to compile this program, and implement the mesh graph. This is done by invoking –

```
> Alias -c foo.saa foo.obj
```

After the above command creates the linked list which is the internal forms for MGDs, the user may view the structure and choose an architecture for implementation by invoking –

```
> tgif foo.obj
```

From the window created by the above command, the user will see the main screen with the mesh graph for the input algorithm and two boxes labeled with "View Architecture" and "Apply Drivers to Architecture". The user is allowed to get more information on each of the node. This probing operation can be done by clicking on the interested node with the middle mouse button while holding down the $< shift >$ key. Another screen will pop up at this point which contains the information such as the iteration space for the node, the computations for X, Y, and Z axes, computation time, and prism assignment. The computation time and prism assignment will display 0 at this point since no transformation in terms of driver application has been specified.

The user may proceed to click the "Apply Drivers to Architecture" box. Another window that shows a typical 2×2 mesh graph along with a menu of a line of transformations will be displayed. The user can choose from the menu which transformation he wishes to perform by clicking right on the interested box. To quit this driver screen, the user has to hit <*cntl*> Q key. Now we are back to the main screen. The array processor implementation for that particular transformation can be visualized by hitting the "View Architecture" box on the main screen. At this point, the array with interconnection and input pattern will be displayed; the execution time in terms of number of steps will be shown too.

The user can quit the architecture screen by hitting <*cntl*> Q and he is allowed to choose from many other transformations by repeating the aforementioned steps.

Appendix B

The Design and Implementation of ALIAS

In Chapter 6, the design of each module in ALIAS has been described. However, some of the low level details are not covered. In this appendix, we discuss each part of ALIAS design aid system.

This thesis can be viewed as two parts: 1) the *design methodology* which consists of the construction of mesh graph descriptors and transformations, 2) the *making* of the design synthesis system based on the proposed methodology.

To start out designing the interactive tool we need to specify the system requirements, i.e., the input description, the output format, the functionalities of the system. As discussed before, the input to the system is the single-assignment forms of the matrix algorithms. The grammar is defined stringently in BNF form in Chapter 6. The output is the layout of the synthesized arrays along with their performance parameters. The functionalities of the synthesis system mainly lie in the application of the drivers (serialized transformers) on the mesh graph descriptors to obtain various structures of the original data dependence graph. Some of those intermediate structures are amenable for the projection and mapping of the input algorithm onto the parallel arrays.

Since it is going to be an interactive and graph-oriented software tool on a workstation, it is to our advantage to select a well-established environment upon which applications can be built. Bill Chen (william@cs.ucla.edu) developed *TGIF* which in an interactive drawing tool that allows the user to maneuver objects in the X Window system. TGIF provides a mechanism for users to launch applications as well. If the middle mouse button is clicked on an object while the ¡shift¿ key being held, TGIF looks for an attribute named *launch* of that object. If such an attribute is found, the value part of the attribute is interpreted as a sh(1) command to execute.

We chose C language to implement the synthesis system because of its high portability and efficiency of running it under UNIX. A good design system should be responsive and fast, i.e., the turnaround time should be minimized.

The following are the C program modules in our design system:

`dp.c`	`drivers.c`	`getnode.c`	`se.c`
`drive.c`	`final.c`	`mp.c`	

We will discuss each of them with their functionalities in this appendix. For more details, the documented programs can be traced to have a better understanding of the implementation.

For instance, `mp.c` which is the main routine of Alias can take the following input of 2×2 matrix multiplication algorithm and produce a data file which can be accepted by TGIF to display the graphs. The parentheses are used to delimit comments. INPUT and OUTPUT sections are needed since distinguishing arrays of input or output is not obvious from the input algorithm.

```
{ ALGORITHM: Matrix multiplication }
{ INDEX: i, j, k }
{ INITIALIZATION:
    a(i, 1, k) = A(i, k)
    b(1, j, k) = B(k, j)
    c(i, j, 1) = 0 }

{ input and output matrices }
INPUT: a, b, c;
OUTPUT: c;

for i = 1 to 2 do
```

```
for j = 1 to 2 do
   for k = 1 to 2 do
      a(i,j+1,k) = a(i,j,k)
      b(i+1,j,k) = b(i,j,k)
      c(i,j,k+1) = c(i,j,k) + a(i,j,k) * b(i,j,k)
```

The program `mp.c` lexically parses this algorithm, whose tokens are stored in tables. The program `se.c` then symbolically executes the data structure in the tables and creates a tree-like linked list which is the representation of MGD at the implementation level. To produce the mesh graph of this 2×2 algorithm, the linked list is traversed and *mm.obj* file is produced. The file mm.obj is readable by TGIF and the corresponding design environment for the input algorithm can be displayed on the X Window by the program `dp.c`. The mesh graph of 8 nodes are plotted on the display by duplicating the *seed node* 8 times. A seed node for matrix multiplication is shown in Figure B.1.

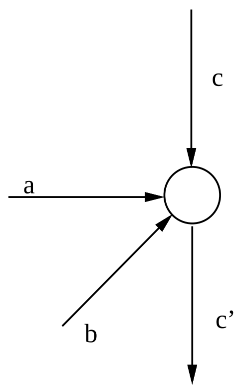

Figure B.1: A seed node for the matrix multiplication algorithm.

The following text is a segment of mm.obj file which corresponds to a seed node.

```
state(0,27,60,0,0,1,16,1,9,1,1,5,0,1,2,1,0,0,0,4,0,0,0,10,3,0,
0,1,0,16,0).
```

```
%%% node declaration %
oval('blue',316,393,348,425,2,5,1,44,0,0,0,[
attr("launch=", "getnode 1; tgif node", 0, 1, 0,
text('black',32,32,0,0,4,1,0,0,2,186,21,301,0,17,4,2,3,0,0,[
"launch=getnode 1; tgif node"]))
]).
poly('orange red',2,[
220,409,316,409],1,5,1,45,0,2,0,0,18,7,0,[
]).
text('black',168,399,0,0,2,1,0,0,1,49,19,300,1,15,4,2,3,0,0,[
"a(1,1,1)"]).
poly('orange red',2,[
332,297,332,393],1,5,1,46,0,2,0,0,18,7,0,[
]).
text('black',305,283,0,0,2,1,0,0,1,49,19,300,1,15,4,2,3,0,0,[
"c(1,1,1)"]).
poly('orange red',2,[
332,425,332,649],1,5,1,47,0,2,0,0,18,7,0,[
]).
poly('orange red',2,[
276,506,324,423],1,5,1,48,0,2,0,0,18,7,0,[
]).
text('black',234,509,0,0,2,1,0,0,1,49,19,300,1,15,4,2,3,0,0,[
"b(1,1,1)"]).

%%% node declaration %
oval('blue',316,649,348,681,2,5,1,49,0,0,0,[
attr("launch=", "getnode 2; tgif node", 0, 1, 0,
text('black',32,32,0,0,4,1,0,0,2,186,21,301,0,17,4,2,3,0,0,[
"launch=getnode 2; tgif node"]))
]).
```

After the main screen that shows the mesh graph of the algorithm has been displayed, the user may click the "Apply Drivers to Architecture" box on the screen. Then the program `drive.c` is activated. The purpose of this module is to allow an interface to occur between the linked list created by

Alias and the graphical representation as well as graphical manipulations that will occur with scheduling. Two procedures are defined in this module of code, namely, *savelist* and *sl*. *savelist* is called by the main routine `mp.c` of the Alias program. It works as a formatting and initializing routine. *sl* is called from the procedure savelist. It is written recursive in nature to properly traverse the linked list in defined by Alias. *sl* will output 2 files, the first file is *node.lst* which is the linked list stored with two elements: 1) ¡level¿: an integer value which is the depth of the linked list, 2) ¡indexinf¿: a struct indextype as specified in def.h formatted for the linked list (the composer and mgd nodes in tree structure). The second file is *node.sch* which is a listing of the nodes with their execution times set to 0. Later on, `getnode.c` will search this file to find the scheduled execution time value, and `drivers.c` will rewrite this file changing the execution times from 0 to some non-zero value.

The `drivers.c` module of Alias contains all the MGD manipulation functions as well as all the scheduling functions. `drivers.c` will open the file *node.lst* and perform manipulations depending on the architecture chosen from the tgif GUI. All the functions that deal with manipulations are similar in nature. They are recursive and take in as their inputs, the header file, and a depth to which to proceed before actually implementing the change to the linked list they were defined to do. The function *group* makes calls to many other functions. Those functions deal with scheduling. They will first place the manipulated linked list into a 3-D array to allow easy scheduling assignments. There are 6 possible numbering approaches, each depending on the direction of travel through the 3-d structure. This code will try all 6 approaches and look at the last of the nodes and determine the shortest path. This path will be saved and performed one last time. The manipulated link list will be written out to *node1.lst*. *node.sch* will be modified to contain the appropriate scheduling.

The purpose of `final.c` is to create the final tgif screen, or the final 2-D processor display. The input to `final.c` is *node1.lst*. Note that *node.lst* is the linked list, and *node1.lst* is the scheduled and manipulated linked list. The module `final.c` takes *node1.lst* in and traverses it to determine the number of processors that are used in this implementation, Once the number of processors is decided, it traverses the outer edges (or directions which are not being coalesced) and examines the inputs to those outer nodes. Since only the first 6 inputs are displayed on this screen, only the inputs of those nodes whose execution time is 6 or less are stored. Note that there are several nested statements because the direction of coalescence and the input directions will determine whether the Xtarget, Ytarget, or Ztarget will be used in labeling.

Bibliography

[AD88] J. Annevelink and P. Dewilde. "HIFI: A functional design system for VLSI processing arrays." In *International Conference on Systolic Arrays*, 1988.

[AP87] K.R. Allen and R.P. Pargas. "On the automatic development of systolic arrays." In *Second International Conference on Supercomputing*, 1987.

[Bac78] J. Backus. "Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs." *Comm. ACM*, pp. 613–641, August 1978.

[BJ92] W. Burleson and B. Jung. "ARREST: An interactive graphic analysis tool for VLSI Arrays." In *International Conference on Application Specific Array Processors*, 1992.

[Bor88] S. Borkar et al. "iWarp: An Integrated Solution to High-Speed Parallel Computing." In *Supercomputing Conference*, 1988.

[BR90] A. Benaini and Y. Robert. "Spacetime-minimal systolic architectures for Gaussian elimination and the Algebraic Path Problem." In *IEEE International Conference on Application Specific Array Processors*, 1990.

[Cap92] P. Cappello. "A Processor-Time-Minimal Systolic Array for Cubical Mesh Algorithms." *IEEE Transactions on Computers*, **3**(1):4–13, January 1992.

[Che85] M.C. Chen. "Synthesizing systolic designs." In *Proc. Int. Symp. on VLSI Technology, Systems, and Applications*, May 1985.

[CP88] P. Clauss and G.R. Perrin. "Synthesis of processor arrays." In C.R. Jesshope, editor, *CONPAR 88*. Cambridge University Press, 1988.

[CS83] P.R. Cappello and K. Steiglitz. "Unifying VLSI array design with geometric transformation." In *ICPP*, August 1983.

[FFW88] J.A.B. Fortes, K.S. Fu, and B.W. Wah. "Systematic Design Approaches to the design of algorithmically specified systolic arrays." In *Computer Architecture*, pp. 454–494. North-Holland, 1988.

[FGL90] P. Frison, E. Gautrin, D. Lavenier, and J.L. Scharbarg. "Designing specific systolic arrays with the API15C chip." In *International Conference on Systolic Arrays*, pp. 505–517. IEEE, September 1990.

[FGQ86] P. Frison, P. Gachet, and P. Quinton. *VLSI Signal Processing II*, chapter 9. IEEE, 1986.

[Gaj88] D.D. Gajski. *Silicon Compilation.* Addison-Wesley, Inc, 1988.

[HH90] Y.H. Hu and Y.T. Hwang. "MSSM - A design aid for multi-stage systolic mapping." In *IEEE Workshop on VLSI Signal Processing*, 1990.

[HL89] R.W. Hartenstein and K. Lemmert. "A CHDL-based CAD system for the synthesis of systolic architectures." In *International Conference on Systolic Arrays*. Prentice Hall, 1989.

[HOI88] P.P. Hou, R.M. Owens, and M.J. Irwin. "DECOMPOSER: A synthesizer for systolic systems." In *ACM/IEEE Design Automation Conference*, 1988.

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*, chapter 2. Computer Science Press, 1978.

[JHM89] S.L. Johnsson, T. Harris, and K. Mathur. "Matrix Multiplication on the Connection Machine." In *Proceedings of Supercomputing*, 1989.

[JK89] H.V. Jagadish and T. Kailath. "A Family of New Efficient Arrays for Matrix Multiplication." *IEEE Transactions on Computers*, **38**(1):149–155, January 1989.

[JRK87] H.V. Jagadish, S.K. Rao, and T. Kailath. "Array Architectures for Iterative Algorithms." *Proceedings of the IEEE*, **75**(9):1304–1321, September 1987.

[KL78] H.T. Kung and C.E. Leiserson. "Systolic Arrays (for VLSI)." In *Sparse Matrix Proceedings*, pp. 256–282, 1978.

[KL84] H.T. Kung and M.S. Lam. "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays." *Journal of Parallel and Distributed Computing*, **1**(1):32–63, August 1984.

[Kun88] S. Y. Kung. *VLSI Array Processors*. Prentice-Hall, 1988.

[Le92] D. Le. *"MAMACG: A Tool for Automatic Mapping of Matrix Algorithms into Mesh Array Computational Graphs."*. Master's thesis, Computer Science Department, University of California, Los Angeles, March 1992.

[LE93a] J.J. Liu and M.D. Ercegovac. "ALIAS Environment: A Design Tool for Application Specific Arrays." In *Fifth IEEE Symposium on Parallel and Distributed Processing*, December 1993.

[LE93b] J.J. Liu and M.D. Ercegovac. "Symbolic Synthesis of Parallel Processing Systems." In *7th International Parallel Processing Symposium*, April 1993.

[Lin89] R.A. Lincoln. *Dependence Graph Mapping Systolic Array Design for Signal Processing Applications*. PhD thesis, Dept. of Electrical Engineering, University of California, Los Angeles, 1989.

[LJ88a] W. Luk and G. Jones. "The Derivation of Regular Synchronous Circuits." In *Proceeding of International Conference on Systolic Arrays*, May 1988.

[LJ88b] W. Luk and G. Jones. "Parametrised Retiming of Regular Computation Arrays." In *Proceeding International Conference on Parallel Processing for Computer Vision and Display*, January 1988.

[LM85] M.S. Lam and J. Mostow. "A transformational model of VLSI systolic design." *IEEE Computer*, pp. 42–52, February 1985.

[LS83] C.E. Leiserson and J.B. Saxe. "Optimizing Synchronous Systems." *J. VLSI and Computer Systems*, **1**(1):41–68, 1983.

[Luk92] W.W.C. Luk. "Transformation Techniques for Serial Array Design." In *Proceedings of International Conference on Application Specific Array Processors*, pp. 574–588, August 1992.

[LW85] G.J. Li and B.W. Wah. "The Design of Optimal Systolic Arrays." *IEEE Transactions on Computers*, **c-34**(1):66–77, January 1985.

[MF86] D.I. Moldovan and J.A.B. Fortes. "Partitioning and mapping algorithms into fixed size systolic arrays." *IEEE Transactions on Computers*, **c-35**(1):1–12, January 1986.

[Mol82] D.I. Moldovan. "On the Analysis and Synthesis of VLSI Algorithms." *IEEE Transactions on Computers*, pp. 1121–1126, November 1982.

[Mol83] D.I. Moldovan. "On the design of algorithms for VLSI systolic arrays." *Proceeding of the IEEE*, **71**(1):113–120, January 1983.

[Mol87] D.I. Moldovan. "ADVIS: A software package for the design of systolic arrays." *IEEE Transactions on Computer-Aided Design*, **CAD-6**(1):33–40, January 1987.

[Mor89] J.H. Moreno. *Matrix Computation on Mesh Arrays*. PhD thesis, Dept. of Computer Science, University of California, Los Angeles, 1989.

[Mor90] J.H. Moreno. "Matrix computations on systolic type meshes: An introduction to the multimesh graph (MMG) method." *IEEE Computer*, **23**(4):32–51, April 1990.

[MW84] W.I. Miranker and A. Winkler. "Spacetime Representation of Computational Strutures." In *Computing*. Springer-Verlag, 1984.

[NLV87] J.J. Navarro, J.M. Llaberia, and M. Valero. "Partitioning: An Essential Step in Mapping Algorithms into Systolic Array Processors." *IEEE Computer*, pp. 77–88, July 1987.

[Omt88] E.T.L. Omtzigt. "SYSTARS: A CAD tool for the synthesis and analysis of VLSI systolic/wavefront arrays." In *International Conference on Systolic Arrays*, pp. 383–391. IEEE, 1988.

[Pat90] D. Patel. *An Applicative Layout Framework for Hardware Synthesis*. PhD thesis, Dept. of Computer Science, University of California, Los Angeles, 1990.

[PSE85] D. Patel, M. Schlag, and M. Ercegovac. "νFP: an Environment for the Multi-level Specification , Analysis, and Synthesis of Hardware Algorithms." In *Functional Programming Languages and its Applications*, pp. 238–255. Springer-Verlag, 1985.

[Qui84] P. Quinton. "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations." In *11th Annual Symposium on Computer Architecture*, pp. 208–214, 1984.

[Rao85] S.K. Rao. *Regular iterative algorithms and their implementations on processor arrays*. PhD thesis, Information Systems Lab. Stanford University, October 1985.

[Rib90] H.B. Ribas. *Automatic Generation of Systolic Programs From Nested Loops*. PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[RK88] S.K. Rao and T. Kailath. "Regular iterative algorithms and their implementation on processor arrays." *Proceedings of the IEEE*, **76**(3):259–269, March 1988.

[SC91] J.P. Sheu and C.Y. Chang. "Synthesizing nested loop algorithms using nonlinear transformation method." *IEEE Transactions on Parallel and Distributed Systems*, **2**(3):304–317, July 1991.

[Sch86] M.D.F. Schlag. *Layout from a topological description*. PhD thesis, Dept. of Computer Science, University of California, Los Angeles, 1986.

[She84] M. Sheeran. "muFP, a Language for VLSI Design." In *1984 ACM Conference on Lisp and Functional Programming*, pp. 104–112, August 1984.

[She85] M. Sheeran. "Designing Regular Array Architectures Using Higher Order Functions." In *Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1985.

[Tse90] P.S. Tseng. "A systolic array programming language." In *IEEE International Conference on Application Specific Array Processors*, pp. 104–112, Princeton, 1990.

[VR86] P.J. Varman and I.V. Ramakrishman. "Synthesis of an Optimal Family of Matrix Multiplication Algorithms on Linear Arrays." *IEEE Transactions on Computers*, **c-35**(11):989–996, November 1986.

[WD92] Y. Wong and J.-M. DeLosme. "Transformation of broadcasts into propagations in systolic algorithms." *Journal of Parallel and Distributed Computing*, **14**(2):121–145, February 1992.