Abstract

CATO, GAVIN RICHARD. Partitioning and Scheduling DSP Algorithms for Parallel Execution Using the Order Graph Method. *(Under the direction of Dr. Douglas S. Reeves and Dr. Winser E. Alexander.)*

Recent efforts to speed up signal and image processing algorithms have focused on executing them in parallel on multiprocessor systems. An important issue in achieving the maximum parallelism is how to automatically partition and schedule algorithms onto the individual processors. In this research, we present the Order Graph Method (OGM) as an effective technique for parallel task scheduling and implement OGM in a scheduling tool to highlight the method's effectiveness.

OGM is developed as a methodology to automatically partition and schedule hierarchical, fully specified flow graphs for parallel computing while considering interprocessor communication and network topology. Using OGM, Digital Signal Processing (DSP) algorithms may be scheduled to optimize the processing rate for a specified number of processors, to optimize the number of processors for a specified processing rate, or to optimize both the processing rate and the number of processors. The use of hierarchical flow graphs allows OGM to exploit fine to coarse-grained parallelism while minimizing the scheduling time by partitioning algorithms at the highest possible level in the hierarchy. The reduction of inter-processor communication by preprocessing the flow graphs minimizes the need for bi-directional communication. In turn, the lower level of bi-directional communication reduces the probability of con-

tention on the network topology. Validating schedules against hardware limitations within individual processors and on the network topology guarantees the effectiveness of the schedules during physical implementation. In addition, OGM guarantees a parallel computing schedule by supporting two potential schedules - parallel execution and software pipelining. For either schedule, OGM balances the workload of the individual processors.

We have implemented OGM in a scheduling program named "Calypso". Using Calypso, a variety of signal processing and matrix operations algorithms are scheduled onto a multi-processor system called the Block Data Parallel Architecture (BDPA). Results for scheduling a 2nd order IIR filter, a 15th order FIR filter, a 4th order lattice filter, a 2nd order 2-D IIR filter, LU Decomposition, and QR Factorization are presented. For each algorithm a variety of hierarchical specifications demonstrates OGM's ability to effectively handle hierarchy during the scheduling process. Insights into the effects of hierarchy on scheduling image processing algorithms are identified and incorporated into OGM. Runtime results and potential schedules for all of the algorithms are presented. In addition, some standard benchmarks are proposed for comparing methods of parallel task scheduling.

Partitioning and Scheduling DSP Algorithms for Parallel Execution Using the Order Graph Method

by
Gavin Richard Cato

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Department of
Electrical and Computer Engineering

Raleigh, NC

May 1996

Computer Engineering

Approved By:

Co-chair of Advisory Committee	Co-chair of Advisory Committee

Biography

Gavin Richard Cato was born in Palo Alto, California on April 7, 1968. He received his B.S. degree in Electrical Engineering from Duke University in 1989 and his M.S. degree in Electrical Engineering from Duke University in 1991.

In 1990 he was commissioned into the United States Army as a 2LT and was promoted to 1LT in 1993.

His main research interests are in parallel processing, digital signal processing, virtual reality, and multimedia. His hobbies include wrestling, scuba diving, snow skiing, and flying airplanes.

Acknowledgements

I would like to thank Dr. Douglas S. Reeves for his support throughout my research and the writing of my dissertation. His insightful comments and constructive suggestions have been useful and sincerely appreciated. I would also like to thank the other members of my committee Dr. Winser E. Alexander, Dr. Clay Gloster, Dr. Dharma Agrawal, and Dr. Rex Dwyer who have generously offered their advice and help.

I would also like to thank my sister Lauren E. Cato for her words of encouragement throughout my graduate career. She has always brought a ray of sunshine to my life.

Finally, I would like to thank my parents Mr. Richard W. and Emily D. Cato. Their unfailing love and constant support have helped me through the obstacles and challenges in my life. Their guidance and encouragement have made everything possible. Words are not enough to express my deepest appreciation. I dedicate this work to them.

Contents

List of Figures vi

List of Tables ix

1 Introduction 1

2 Previous Work on DSP Task Scheduling 6

2.1 Programming Languages, Compilers, and Loop Optimization 8

2.2 Mathematical Approaches 20

2.3 Heuristic Techniques 25

2.4 Graph-Based Methods 28

3 The Order Graph Method 38

3.1 Overview 38

3.2 Preprocessing Step - Handling Feedback 43

3.3 Partitioning Step - Generating Candidate Partitionings 46

3.4 Validating and Scheduling Steps - Incorporating Communication . . . 50

3.4.1 Communication Concerns 50

3.4.2 Modeling Contention Due To Communication 50

3.5 Contributions of the Order Graph Method 62

4 Hierarchical FSFGs 64

4.1 Motivation 64

4.2 Hierarchical Specification 65

4.3 Modifications for Scheduling 69

4.3.1 Preprocessing 69

4.3.2 Partitioning 74

4.3.3 Validating and Scheduling 76

4.4 Summary 76

5 Evaluation of CALYPSO **78**
5.1 Overview of the BDPA 79
5.2 Implementation and Experimental Conditions 81
5.3 Filters 83
5.3.1 2nd Order IIR Filter 83
5.3.2 15th Order FIR Filter 87
5.3.3 4th Order Lattice Filter 91
5.3.4 2-D Filtering 94
5.4 Matrix Operations 103
5.4.1 LU Decomposition 103
5.4.2 QR Factorization 109
5.5 Conclusions 116

6 Summary and Future Research **119**
6.1 Summary 119
6.2 Future Research 120

Bibliography **124**

List of Figures

1.1 Challenges presented by the U.S. government High Performance Computing Initiative in the 1990's 2

2.1 Flow graph for a 1-D 2nd order IIR filter 9
2.2 SISAL representation for a 1-D 2nd order IIR filter 10
2.3 Sample code with the component affinity graph 13
2.4 (a) Example loop and the corresponding iteration space representation (b) with skewing of one 17
2.5 Example loop transformation matrices 17
2.6 Wavefront transformation matrix 18
2.7 Wavefronts for Figure 5(b) 18
2.8 Coarse-grained execution for Figure 5(b) 19
2.9 Geometric representation of the canonical linear transform $y = A \cdot x$. 23
2.10 Geometric representation of the Leiserson and Kung transform 23
2.11 Representation of the Leiserson and Kung task schedule 24
2.12 Example (a) processor graph and (b) task graph for DFBN 26
2.13 FSFG for a 1-D 2nd order IIR filter 28
2.14 Example of a non-iterative FSFG with an early schedule and a late schedule 30
2.15 Example of combining an early schedule and a late schedule to form a scheduling-range chart 31
2.16 1-D 2nd order IIR filter for RGM 32
2.17 RGM scheduling steps for a 1-D 2nd order IIR filter 32
2.18 Data movement for an example partitioning of a 2nd order IIR filter . 34
2.19 Basic Example for 4 processor system executing FSFG of 1-D 2nd Order IIR FSFG: (a) Cycling vector (b) Period matrix (c) Six iteration periods for (a) and (b) 35
2.20 Schedule Transformations: (a) original period matrix and cycling vector (b) rows 3 and 4 of the period matrix are exchanged (c) columns of the period matrix are rotated left by one (d) rows 2 and 4 of the period matrix are spliced to rows 1 and 3 respectively (e) rows 1 and 2 of the cycling vector are exchanged 36

3.1 Flow Chart for the Order Graph Method 41
3.2 (a) Cycling vector for parallel execution (b) Period matrix 53
3.3 (a) Cycling vector for software pipelining (b) Period matrix 58

4.1 Hierarchical FSFG for a 2-D 2nd order IIR of a 256x256 image using each row as an input 66
4.2 Hierarchical FSFG for a 3rd order lattice filter 67
4.3 Example Hierarchical FSFG 68
4.4 Potential FSFG hierarchical specification for the 2nd order IIR filter (IIR2) 69
4.5 Sample input files for the hierarchical specification of a 2nd order IIR filter (IIR2) 70

5.1 Block Diagram of the BDPA 79
5.2 Flow graph for a 2nd order IIR filter (IIR1) 83
5.3 Potential FSFG hierarchical specification for the 2nd order 1-D IIR filter (IIR2) 84
5.4 Potential FSFG hierarchical specification for the 2nd order 1-D IIR filter (IIR3) 84
5.5 Potential FSFG hierarchical specification for the 2nd order 1-D IIR filter (IIR4) 85
5.6 FSFG for the 2nd-order 1-D IIR filter, with a software pipelining, rate and processor optimal schedule generated by Calypso using the Order Graph Method with $t_c = 1$ 88
5.7 FSFG for the 15th order FIR filter (FIR1) 88
5.8 Potential FSFG hierarchical specification for a 15th order FIR filter (FIR2) 89
5.9 FSFG for the 15th order FIR filter, with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 7$, $T_d = 5$, and $t_c = 1$ 90
5.10 FSFG for the 15th order FIR filter, with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 7$, $T_d = 5$, $t_c = 1$, and not splitting atomic nodes across processors 91
5.11 FSFG for a 4th order lattice filter (LAT1) 91
5.12 Potential FSFG hierarchical specification for a 4th order lattice filter (LAT2) 92
5.13 FSFG for a 4th order lattice filter, with a software pipelining, rate and processor optimal schedule generated by Calypso using the Order Graph Method with $t_c = 1$ 93
5.14 FSFG of the computational primitive equations for the 2nd order 2-D IIR Filter (2DIIR1) 95

5.15 FSFG for the 2nd order 2-D IIR filter's computational primitive equations, with a software pipelined schedule generated by Calypso using the Order Graph Method for $P_d = 7$, $T_d = 8$, and $t_c = 1$ 98
5.16 Potential FSFG hierarchical specification of the computational primitive for the 2nd order 2-D IIR Filter (3x1) (2DIIR2) 99
5.17 Potential FSFG hierarchical specification of the computational primitive for the 2nd order 2-D IIR Filter (3x3) (2DIIR3) 100
5.18 FSFG of the computational primitive for the 2nd order 2-D IIR Filter (3x3), with a software pipelining schedule generated by Calypso using the Order Graph Method for $P_d = 3$, $T_d = 150$, and $t_c = 1$ 101
5.19 FSFG of the computational primitive for the 2nd order 2-D IIR Filter (3x3), with a parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 3$, $T_d = 150$, and $t_c = 0.2$ 102
5.20 FSFG for LU Decomposition of a 4 X 4 matrix (LUD1) 106
5.21 Potential FSFG hierarchical specification for LU Decomposition of a 4 X 4 matrix (LUD2) 107
5.22 FSFG for LU Decomposition of a 4X4 matrix, with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 4$, $T_d = 34$, and $t_c = 1$ 108
5.23 FSFG for QR Factorization of a 3X3 matrix (QR1) 112
5.24 Potential FSFG hierarchical specification for QR Factorization of a 3X3 matrix (QR2) 113
5.25 FSFG for QR Factorization of a 3X3 matrix with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 8$, $T_d = 28$, and $t_c = 1$ 114

List of Tables

3.1 List of symbols used to detail the Order Graph Method 40
3.2 Comparison of RGM, CRM, and OGM 63

5.1 Summary of Calypso's results for scheduling hierarchical FSFGs of a 2nd order IIR filter with $t_c = 1$ 86
5.2 Summary of Calypso's results for scheduling hierarchical FSFGs of a 2nd order IIR filter for parallel execution with $t_c = 1$ 87
5.3 Summary of Calypso's results for scheduling hierarchical FSFGs of a 15th order FIR filter 89
5.4 Summary of Calypso's results for scheduling hierarchical FSFGs of a 4th order lattice filter 93
5.5 Summary of Calypso's results for scheduling the computational primitive equations' FSFG for a 2-D 2nd order IIR filter with $t_c = 1$ . . . 95
5.6 Summary of Calypso's results for scheduling the computational primitive equations' FSFG for a 2-D 2nd order IIR filter with $t_c = 1$, and no shifting in the partitioning strategy 96
5.7 Summary of Calypso's results for scheduling hierarchical FSFGs of a 2-D 2nd order IIR filter 100
5.8 Summary of Calypso's results for scheduling hierarchical FSFGs for LU Decomposition with $t_c = 1$ 105
5.9 Summary of Calypso's results for scheduling hierarchical FSFGs for QR Factorization with $t_c = 1$ 115

Chapter 1

Introduction

As computer technology develops, demands for high processing rates are increasing dramatically. Research and development of high-performance computing systems has continuously advanced processing rates for commercial systems. However, the U.S. government challenged researchers with several computationally intensive applications in its High Performance Computing Initiative in the 1990's. Applications requiring processing rates as high as 1 TFLOP are presented as goals for future implementation (See Figure 1.1, taken from [1]). Motivated by the high costs of hardware, early research efforts focused on the reduction of an algorithm's computational requirements in order to achieve effective real-time processing. Even with these efforts, single processor systems will not be able to meet the initiative's required processing rates in the near future.

During the past decade, the focus of research has changed as several trends have been established with advances in VLSI technology [2]:

- Decreased cost of hardware
- Increased processing rates
- Decreased size of individual processors
- Increased integration of multiple processors

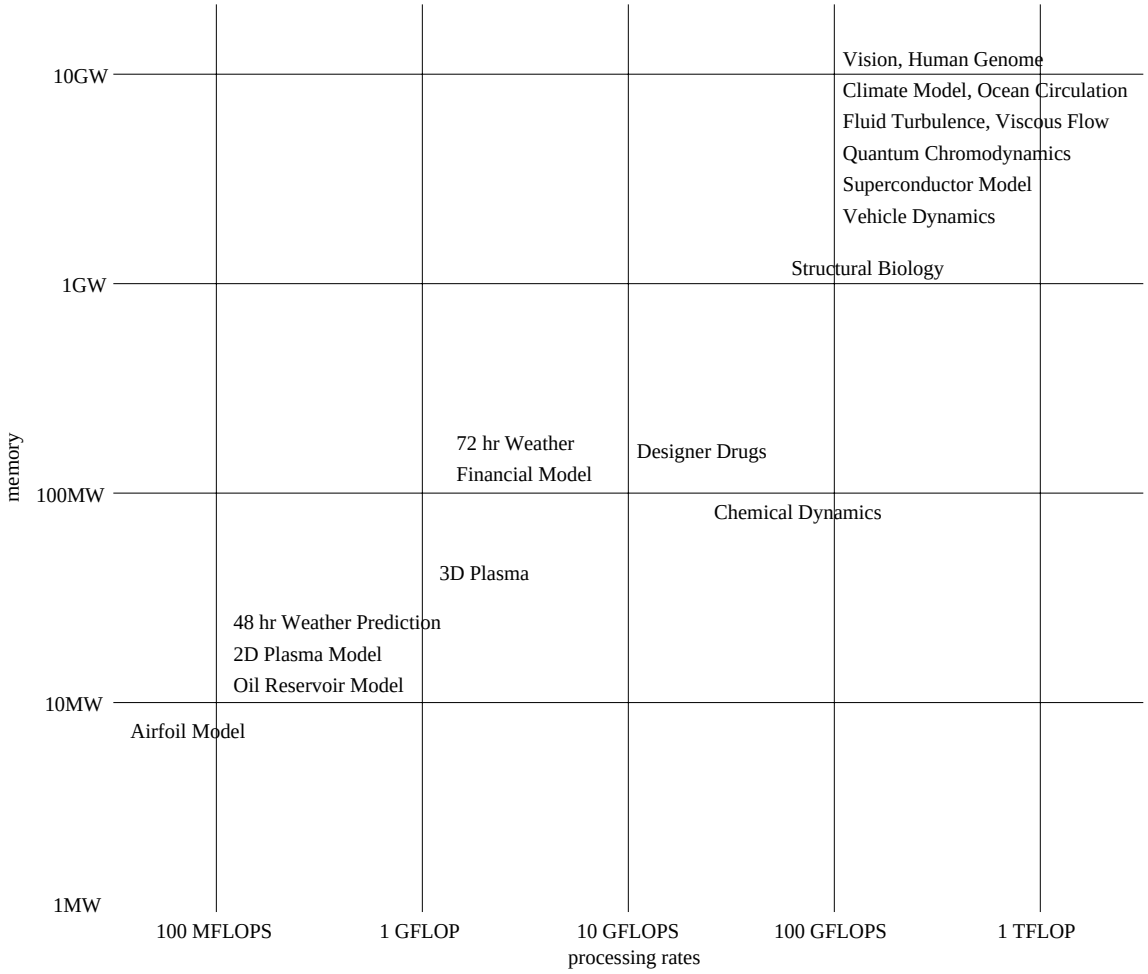


Figure 1.1: Challenges presented by the U.S. government High Performance Computing Initiative in the 1990's

- Increased inter-processor communication bandwidth

Taking advantage of these trends, researchers are increasingly using multiprocessing systems to provide real-time execution for computationally intensive applications.

Attempts to develop these multiprocessor systems has lead to a variety of architectures, some of which are now commercially available. Architectures are often designed around a particular set of applications as effective general purpose multiprocessor systems are increasingly difficult to achieve. In this research we target distributed-memory multiprocessor systems for implementation of high performance real-time applications. To fully utilize the processing capabilities of the system, an ef-

fective methodology for scheduling algorithms onto a multiprocessor system is needed. The task assignment problem requires the assignment of tasks from a user-defined algorithm onto multiple processors, in a manner which increases the throughput of the system. This is accomplished by balancing the computational load among the processors in the system while ensuring that the communication time requirements do not exceed the computational time. This creates the need to balance competing goals: increased distribution of tasks versus minimized interprocessor communication. Increasing the distribution of tasks results in an increase of interprocessor communication. Over distributing the tasks produces a multiprocessor schedule in which the time required for communication exceeds the computation time. In this case, the schedule becomes communication bound and does not achieve the maximum throughput rate. Task scheduling to satisfy these conflicting objectives is well known as an NP-complete problem [3]. For a good general overview of the task assignment problem refer to [4, 5, 6].

Years of research has proved that automatic identification of parallelism is difficult. Fortunately, digital signal processing algorithms are easier to parallelize due to their simplicity and regularity. DSP algorithms may be partitioned to exploit parallelism during the processing of a single sample (intra-iteration), as well as partitioned to exploit the parallelism in the processing of multiple input samples (inter-iteration). Techniques for partitioning and scheduling DSP algorithms onto multiprocessor systems can be divided into four categories: language and compiler methods, mathematical approaches, heuristic techniques, and graph-based methods. There are several factors which may limit the performance of parallel execution on applications. These challenges include:

- Uneven load balancing across processors
- The need to synchronize processors during parallel execution

- The time required for interprocessor communication
- Contention for shared hardware resources, such as communication links

Research at North Carolina State University has introduced the Block Data Flow Paradigm (BDFP) as a methodology to achieve high processing rates for DSP algorithms [2, 7, 8]. Using the Block Data Parallel Architecture (BDPA), the BDFP has been used to map and simulate a variety of DSP algorithms. Current goals for this research group include building the BDPA system, and developing techniques for mapping algorithms onto the system. The research presented in this dissertation focuses on the development and implementation of an effective task scheduling methodology for the BDPA system. To be effective for the BDPA, the method must be able to automatically partition algorithms to optimize the system's throughput and/or the number of processors. During this process, communication and contention must be considered. We wish to automate this process such that the user of the BDPA does not need to know the details of its architecture.

We will present the Order Graph Method (OGM) as a new methodology for automatically partitioning and scheduling digital processing algorithms for parallel computing. OGM accepts algorithms specified by hierarchical fully specified flow graphs (FSFGs) and generates schedules for parallel execution and software pipelining. Exploiting fine to coarse-grained parallelism, OGM schedules algorithms to meet user-defined goals for the processing rate or the number of processors. The effect of inter-processor communication and the network topology are considered during the scheduling process. In addition, we will introduce "Calypso" as a tool which implements OGM. Using Calypso, we will show the flexibility and effectiveness of the OGM methodology by scheduling a variety of hierarchical FSFGs.

This dissertation is organized as follows. In the next chapter, we discuss previous research for DSP task scheduling, focusing on the characteristics necessary for effective

implementation of DSP algorithms on the BDPA. Examples from each of the four categories of task scheduling methods are highlighted. Chapter 3 introduces the Order Graph Method and details the process used to partition and schedule algorithms with consideration for communication and network topology. Chapter 4 presents a method for algorithm specification using hierarchical fully specified flow graphs. The effects of hierarchy on task scheduling and the OGM process are described. In Chapter 5, an evaluation of Calypso, a tool which implements OGM for hierarchical flow graphs, is presented with results for scheduling various filtering algorithms and matrix operations. Chapter 6 concludes with a summary of the research and offers some goals for future efforts.

Chapter 2

Previous Work on DSP Task Scheduling

The methodologies for producing multiprocessor schedules from an algorithm specification vary widely. Methods based on programming languages include pure languages [9, 10, 11, 12, 13], extensions to existing languages [14], integrated environments, parallelizing compilers [15, 16, 17, 18], and loop optimization methods [1, 19, 20, 21, 22, 23]. Mathematical methods have generally been probabilistic techniques [24] or manipulations of recurrence equations for scheduling onto systolic arrays [25, 26, 27, 28, 29]. Heuristic methods have included both work-greedy and non-work-greedy assignment schemes, some of which consider communication costs [30, 31, 32, 33, 34]. Graph-based techniques include graph minimization through critical path identification [35], the range-graph method [36], and the cyclostatic realization method [37, 38, 39], as well as others [40, 41, 42]. Good recent reviews of many of theses methods may be found in [43, 44, 45, 46, 47]. While an analysis of all available methodologies is impractical, we will discuss important examples from each of the general categories, with particular attention to graph-based mapping techniques.

Considering the diversity of these methodologies and the complexity of the task scheduling problem (NP-complete), benchmarks should be used to compare different methodologies. Unfortunately, no clear benchmarks have been established which cover the diverse methods of task scheduling. In the absence of such benchmarks,

we propose the use of several criteria for measuring the effectiveness of partitioning/scheduling methods:

- Use of a familiar and convenient algorithm specification technique
- Requirements for the user to explicitly indicate potential parallelism
- Scalability up to large problem instances
- Level of parallelism supported - coarse, medium, and/or fine grained parallelism
- Intra-iteration (during processing of a single sample) and/or inter-iteration (during processing of multiple samples) parallelism exploited
- Consideration of communication overhead and the network topology
- Consideration of input/output communication bandwidth and distribution
- Optimization of the processing time and/or the number of processors
- Automated, semi-automatic, and/or interactive optimization
- Suitability for use on multiple parallel architectures
- Practicality of implementation
- Acceptable running time

Through the remainder of this chapter we identify key methodologies and discuss them with respect to these criteria. The representative methods are chosen based on their justification and their promise for application to the BDPA. We begin with a look at programming languages, compilers, and loop optimization techniques for multiprocessing systems.

2.1 Programming Languages, Compilers, and Loop Optimization

Over the last decade, parallel processing languages have received considerable attention. The research efforts have developed a wide array of languages which target parallel processing systems [13]. They include: SISAL, Linda, Occam, Id, Lucid, Silage, Lustre, Signal, Gabriel, Aachen, Ptolemy, Schedule, Hense, Crystal, C^*, Fortran D, Kali, and DINO (among others). Included in the list are pure parallel languages such as SISAL, Linda, and Occam as well as language extensions such as Schedule and C^*. Each of the numerous languages has a different basis for data input, a unique manner for specifying algorithms, and independent compilers for partitioning and scheduling of tasks. Most of these languages target single-program multiple-data systems where the same program is executed on each processor using different data samples. For many of the programming languages, there is no clear explanation of the mapping methodology used to partition and schedule algorithms for parallel execution. This is because research in the development of parallel programming languages tends to focus on the representation of input data streams and the algorithm specification method of individual languages. Details of the mapping process are left to compiler development for the individual languages. While not all languages can be discussed, we will focus on one of the more effective and widely accepted parallel programming languages - SISAL.

SISAL was developed by Lawrence Livermore National Laboratory for several multiple-program multiple-data architectures including the Encore Multimax, Sequent Balance, Sun, CRAY Y-MP, and the IBM RS-6000 [48]. In many parallel processing languages, elements of the same data type (including input data elements) are represented by streams. The SISAL project has developed a language with strict semantics for representation and execution on the streams. An important capability of SISAL is its ability to execute computations on a portion of a data stream

before the stream is completely input. This allows SISAL to execute algorithms on unbounded input streams. Streams are defined by the data type of the variable (*type int_stream = stream[int]*). To manipulate these streams, SISAL provides a set of operations. Some primitive operations include [48, 49]:

Create defines a new stream

Append adds a value to the end of the stream

Head/Tail selects the item at the front/back of the stream

Empty clears a stream

Concatenate adds one stream to the end of another stream

To pass data between functions, SISAL provides pipelines which connect the output of one stream operation to one or more input operations. For example, the 2nd order IIR filter in Figure 2.1 can be specified as shown in Figure 2.2. Once the algorithm

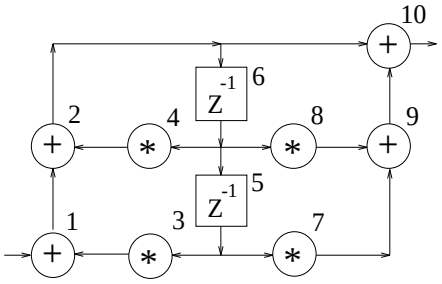


Figure 2.1: Flow graph for a 1-D 2nd order IIR filter

is specified, the Optimizing SISAL Compiler (OSC) partitions and schedules the algorithm for parallel execution. The code OSC produces is highly optimized with data streams implemented as circular buffers [48, 49]. During code generation, OSC allows users to set a group of parameters which effect optimization. The number of processors, the size of circular buffers between tasks, and the wake-up threshold

```
define IIR

type st = stream[real]

function multiply_stream (operand1 : real; operand2 : st returns st)
  for i in operand2
    returns stream of i*operand1
  end for
end function

function initialize_stream ( n : integer returns st)
  returns stream of n
end function

function delay_stream (not_delayed : st returns st) ...
function add_stream (operand1, operand2 : st returns st) ...

function IIR (a1, a2, b1, b2 : real; in_stream : st returns st)
  let
    s6 = add_stream(in_stream, multiply_stream(a2,s2));
    s5 = add_stream(multiply_stream(a1,s1), s6);
    s9 = add_stream(multiply_stream(b1,s1), multiply_stream(b2,s2));
    s2 = delay_stream(s1);
    s1 = delay_stream(s5);
  in
    add_streams(s5, s9)
  end let
end function
```

Figure 2.2: SISAL representation for a 1-D 2nd order IIR filter

(determines how full/empty the circular buffer will be when interleaving the buffer's consumer/producer) are all determined by the user. The efficiency of the code produced depends on the setting of these values. Experiments have shown that code performance consistently increased with the size of the buffer, that effective wake-up thresholds are about one-half the size of the buffer, and that buffer sizes of one result in demand-driven execution.

While SISAL permits the use of high-level parallel constructs, it is incompatible with accepted sequential programming languages and may be difficult to learn (particularly for non-programmers). Advanced capabilities of SISAL allow C programs and libraries to interface with SISAL code and permit conversion from C to SISAL. However, the C language was developed for sequential computer systems and does

not naturally support parallelism. Thus, a programmer must generate additional C code to incorporate synchronization, communication parameters, and consideration for race-conditions within the description of the algorithm. Extensions to existing languages (C, Pascal, Fortran, and LISP) offer an attractive alternative to experienced programmers; however, it usually remains up to the programmer to identify parallelism in algorithms. A tool for parallelizing SISAL programs has been developed for the iPSC/860 [50]; however, parallelism must be specified in terms of a directed acyclic graph (DAG). Thus, the tool will not be effective in exploiting fine-grained parallelism within iterative digital signal processing algorithms. Due to the nature of SISAL, scalability to algorithm size is not a problem. Portability is limited to the architectures supported by OSC (Encore Multimax, Sequent Balance, Sun, CRAY Y-MP, and the IBM RS-6000); however, there are no inherent reasons preventing SISAL from being ported to other architectures. Unlike most parallel languages, OSC recognizes and supports (to some degree) parallelism during processing of a single sample as well as parallelism during execution on multiple samples [49]. Detection of the fine and medium-grained parallelism is semi-automatic with users identifying potential parallelisms through stream declarations. In addition, the efficiency of the code depends on the user-defined parameters [48]. The network topology and required communication messages are considered by the OSC compiler. OSC has an acceptable running time, but there are currently some open issues with the OSC compiler concerning the appropriate uses for stream operations [48, 49].

With the increasing use of multiprocessor systems, parallelizing compilers have experienced an increase in attention from the research community. A significant amount of the research has focused on the automatic generation of communication messages between processors. In this case, the recognition of an algorithm's parallelism is still up to the programmer. Thus, the data partitioning and the allocation of tasks to processors are not handled by the compiler. The compiler simply uses the partition-

ing defined by the programmer to identify and generate the required communication. Although a difficult task, automatic parallelization of sequential code has become the goal of some research efforts in parallelizing compilers. Because the data partitioning problem is NP-complete, a common method for developing parallelizing compilers is to narrow the focus to a specific problem. Among the problems addressed are [16]:

- data partitioning for individual loops and strongly connected components [1]
- data partitioning for individual loops arranged in a sequential order
- tools for evaluating the performance of different data partitioning solutions
- communication between processors resulting from distributed arrays which cross reference one another
- data partitioning based on identification of loop constraints

Detection and scheduling of parallelism is restricted to loops; non-loop parallelism is not exploited.

One approach of particular interest is a constraint-based approach to automatic data partitioning [16]. This method was chosen for discussion based on favorable comparisons to other methodologies and a broad scope of application to data arrays and loops. Defined for the Intel iPSC/2 hypercube, the NCUBE and the WARP systolic machine using Parafrase-2, the concepts remain applicable to any programming languages similar to Fortran. The compiler analyzes every loop in a program and determines constraints on the data structures being referenced. The constraints on the data distribution represent requirements that should be met, but are not required

[1] A FSFG is a directed graph (digraph) $G = (V, E)$ where V is the set of all nodes and E is the set of all edges. A digraph G is strongly connected if for each pair of vertices $v, w \in V$ there is a directed path from v to w and a directed path from w to v as well. A strongly connected component (strong component) of a digraph is a maximal strongly connected subgraph (G) such that there is no pair of vertices $v \in V$ and $w \notin V$ with a directed path from v to w and a directed path from w to v.

to be met. There are essentially two types of constraints. Parallelization constraints are defined to execute loops with an even distribution across as many processors as possible. Communication constraints are identified such that input data elements for an operation reside on the same processor which holds the output data element (attempting to reduce interprocessor communication). In order to effectively identify data reference patterns and the data constraints, the compiler limits its focus to loops containing assignments to arrays. Constraints on data structures are assigned quality measures based on the structure's importance to the program's performance. Quality measures for parallelization constraints are determined by estimating the time for sequential execution and the potential speedup. For communication constraints, quality measures are determined from referencing a library of communication patterns with previously defined communication costs [16]. Using the quality measures, the compiler combines constraints if it reduces the execution time of the program.

The partitioning strategy makes decisions on data distribution using a component affinity graph (CAG). A CAG is a graphical representation of the dependencies. Nodes represent the dimensions of arrays and edges have weights equal to the quality measure of the constraint between the connected nodes (See Figure 2.3). Assuming a maximum array dimension D, the CAG may be partitioned into D dis-

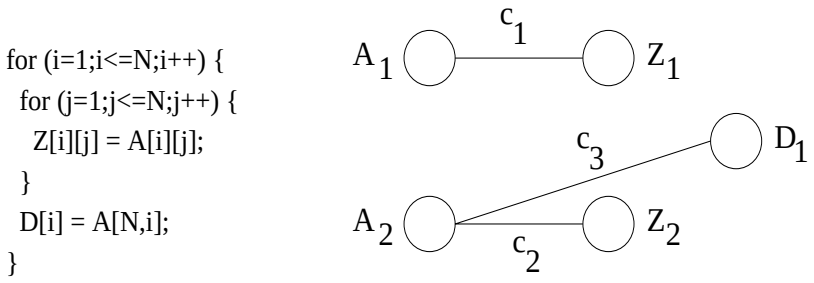


Figure 2.3: Sample code with the component affinity graph

joint sets [51]. Each set is assigned a weight (w_i) equal to the total computation time of the set (i). The CAG is partitioned such that the maximum weight across all sets

($\max w_i \ \ \forall\ i : 0 < i \leq D$) is minimized. For each of the D sets, array dimensions may or may not provide speedup when distributed on multiple processors [16]. Array dimensions which will not produce speedup are sequentialized to save communication costs without sacrificing potential speedup. Other dimensions are analyzed for contiguous or cyclic distribution using the constraints.

Consider the following loop which imposes constraints on the first dimension of A ($A1$) with the second dimension of B ($B2$).

```
for {i=1;i<=N;i++} {
  A[i][c_1]  = A[i][c_1] + m * B[c_2][i];
}
```

The constraints are based on the use of variable i in $A1$ and $B2$, and require $B[c_2][i]$ to be available when $A[i][c_1]$ is used. The loop also suggests sequentialization of $A2$ and $B1$ to allow software pipelining. Using software pipelining, an input sample is completely processed by a single processor. Subsequent input samples are processed on adjacent processors by skewing the time at which the subsequent input samples begin execution. For example, assume a processor P_i operates on a data item d_j beginning at time t_k. The next data item d_{j+1} would begin execution on processor P_{i+1} at time t_{k+c} where c is a constant greater than 1.

Now consider another loop which requires the cyclic distribution of A due to the inner loop's index j being fixed by the outer loop's index i.

```
for {i=1;i<=N;i++} {
  for {j=1;j<=i;j++} {
    sum = sum + A[i][j];
  }
}
```

This would allocate $A[i][j]\ \ \forall\, j : 1 \geq j \leq i$ to a processor i. The combination of constraints results in the cyclic distribution of A along dimension 1. Thus, the distribution attempts to minimize the interprocessor communication while maximizing speedup.

Conflicts in the determination of the distribution are resolved using the quality measures [16]. Scheduling maps nodes onto a processor grid of dimension D according to the decisions made on distribution. If the user wishes to restrict scheduling to a two dimensional network topology and $D > 2$, additional array dimensions may be sequentialized to reduce D to 2 [16]. In this manner, the method may be used to generate a suggested network topology or to map an algorithm onto a given instance of a processor array. The number of processors for each dimension of the processor array is determined according to the distribution decisions and the size of the array dimensions. Using this partitioning strategy there is no guarantee that the schedule produced is optimal [16].

Medium grained parallelism is exploited through the automatic recognition of parallelism in loops with data array assignments. The method may be adapted for use with familiar specification techniques and is practical for implementation on other systems. However, for parallelism to be exploited, the user must represent available parallelism in terms of data arrays manipulated within loops. Thus, parallelism between samples is exercised, but parallelism within processing of a single sample is not exploited. Communication is considered during partitioning via the communication constraints, but input/output bandwidth is not taken into account. Complete optimization of processing time and/or the number of processors is not guaranteed. The nature of loops allows varied problem sizes some of which have been used as benchmark programs - tred2, dgefa, trfd, mdg, and flo52 [16]. The benchmark programs have shown favorable results; however, the method assumes that the compiler knows

the probabilities of executing conditional branches [16]. Currently the user must supply this information. Running time for task scheduling has not been published.

Yet another area of extensive research is loop optimization. Existing parallelizing compilers make extensive use of loop optimization techniques to achieve a higher system throughput. Many of these compilers apply a set of transformations to loops through a series of steps. In each step, the compiler must determine that the transformation is legal and will improve the overall performance. However, there are several methods to determine the order of these transformations. Some methodologies even test all possible orderings and combinations of the transformations [52]. Another method is based on matrix transformations which models loop interchange, reversal, and skewing transformations as linear transformations [53]. A new approach by Wolf and Lam combines the advantages of these techniques to provide a unified methodology. The Wolf and Lam method was chosen based on the mathematical rigor and generality of its application.

Their method determines valid and desirable transformations which will maximize a desired level of parallelism (fine to coarse grained). While a brief explanation of the methodology follows, we refer the reader to [22] for the details not discussed. Developed for the CMU Warp and the Intel iWarp, Wolf and Lam's methodology represents a set of nested loops by a finite convex polyhedron. Each node in a polyhedron represents an iteration in one of the loops and is identified by an index vector $\vec{p} = [p_1, p_2, \cdots, p_n]^T$ where p_i is the i^{th} loop index beginning from the outermost loop (p_1) to the innermost loop (p_n). Edges represent dependencies between iterations and are defined by dependence vectors $\vec{d} = [d_1, d_2, \cdots, d_n]$ such that $\vec{p_2} = \vec{p_1} + \vec{d}$ if iteration $\vec{p_1}$ must be executed before $\vec{p_2}$. The set of all dependence vectors for a loop is defined as D. In nested sequential loops, index vectors are executed in lexicographic order, therefore, dependencies are represented by a set of lexicographically positive vectors [22]. A vector $\vec{d}$ is lexicographically positive if $\exists i : (d_i > 0$ and

$\forall\ j < i : d_j \geq 0$) [22]. An example of a loop and the corresponding iteration space representation are shown in Figure 2.4 (a) (taken from [22]).

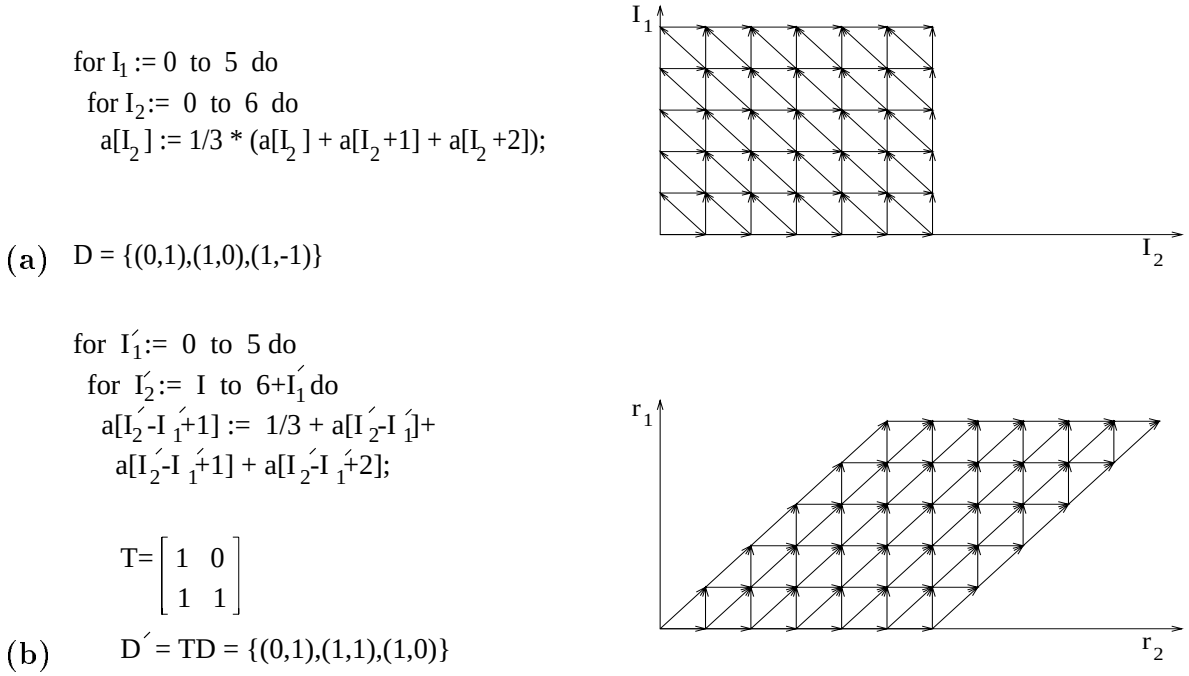


Figure 2.4: (a) Example loop and the corresponding iteration space representation (b) with skewing of one

Loops defined using an iteration space representation are transformed to maximize the degree of parallelism . For individual loops, matrix operations which represent loop transformations may be applied to dependence vectors to produce a new set of dependence vectors. Transformations are represented by matrices similar to those presented in Figure 2.5 for a $2x2$ matrix. Iterations of a loop may be executed in

$$\underset{(permutation)}{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}} \quad \underset{(reversal)}{\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}} \quad \underset{(skewing\ by\ one)}{\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}}$$

Figure 2.5: Example loop transformation matrices

parallel if there are no dependences within the loop (commonly know as a DOALL loop). Transformations are applied to an iteration space representation to minimize the dependencies which in turn maximizes the number of DOALL loops. Given a loop nest $(I_1, \cdots, I_n)$ with lexicographically positive dependences $\vec{d} \in D$, I_i is parallelizable if and only if $\forall\ \vec{d} \in D, (d_1, \cdots, d_{i-1}) \succ \vec{0}$ or $d_i = 0$ [22]. With n nested loops and no dependencies between iterations, there are n degrees of parallelism. However, if dependencies exist, transformations may be executed to achieve at least $n-1$ degrees of parallelism by skewing the innermost loop [22]. An example application of the skewing transformation is presented in Figure 2.4 (b). Using this transformation, Figure 2.4 (b) has obtained a greater degree of parallelism. This can be seen by applying a wavefront transform represented by the matrix shown in Figure 2.6. Application of the wavefront transform to Figure 2.4 (b) is illustrated in Figure 2.7

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

Figure 2.6: Wavefront transformation matrix

(taken from [22]) and indicates the execution sequence for computations.

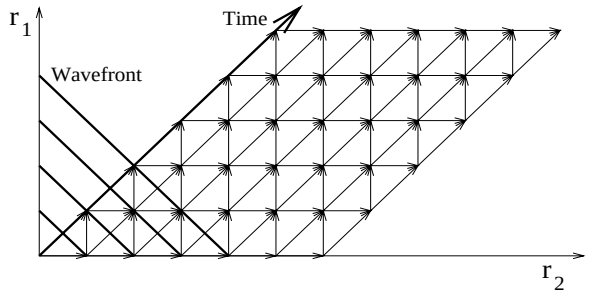


Figure 2.7: Wavefronts for Figure 5(b)

Combinations of the transformations are represented by the product of the elementary matrices. Assuming iterations in a sequential loop are executed in lexicographic order according to their index, a transformation which produces lexicographically positive dependence vectors is legal. Thus, maximizing parallelism is simply a matter of finding the transformation matrix which maximizes the available parallelism. To obtain fine-grained parallelism, the nested loops are skewed and then wavefronted which is always legal. To wavefront the nested loops, independent iterations of loops with similar dependencies are identified and assigned to a single wavefront. Thus, each operation along a wavefront is executed on a different processor during the same time period. Coarse-grained parallelism can be obtained by grouping together local operations across wavefronts as seen in Figure 2.8 (taken from [22]). In this case, each group of operations across a wavefront is executed on a different processor. In

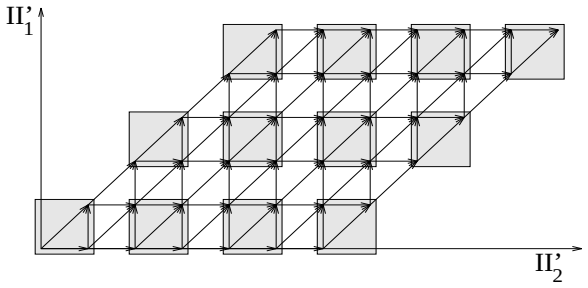


Figure 2.8: Coarse-grained execution for Figure 5(b)

either case, the resulting code is optimized and allows parallelism to be exploited in up to five levels of loop nesting.

Wolf and Lam's method, exploits fine to coarse-grained parallelism to optimize processing time and the number of processors for execution of nested loops. The specification technique is familiar and convenient but requires that available parallelism be defined in terms of nested loops. Parallelism between samples as well as parallelism during processing of a single sample is exploited. Due to the nature of loops, the

method is scalable to large problem instances. However, during the transformation process, potential communication costs are not considered including interprocessor communication and the input/output bandwidth. As a result, the compiler (which uses this loop optimization technique) will need to recursively call the loop optimizing code when interprocessor communication exceeds computation time. The complexity of the algorithm is $O(n^3d)$ where n is the number of nested loops and d is the number of dependencies. Thus, the running time should be acceptable. There are no inherent reasons which would limit the portability of this methodology to other systems.

In summary, the research into languages, compilers, and loop optimization is and continues to be extensive. However, techniques developed in individual research efforts must be incorporated into a single methodology. Languages are effective but usually require users to explicitly define available parallelism. Parallelizing compilers effectively partition their target code, but often fail to recognize all of an algorithm's available parallelism. When used in parallelizing compilers, loop optimization techniques provide excellent parallelism, but require users to write programs using loops and arrays within the scope of the methodology. Mathematical approaches offer alternative techniques to language and compiler task scheduling. We now describe some of theses mathematical techniques.

2.2 Mathematical Approaches

Mathematical approaches to task scheduling are based on recurrence equations or probabilistic modeling. Research into recurrence equations focuses on systolic arrays and provides methodologies which go beyond basic partitioning and scheduling. Many of the methodologies will also suggest how many processors are needed and what network topology should be used. Cappello and Steiglitz's method is one method of task scheduling using recurrence equations which has proven effective and has been

referenced frequently [54]. Their method provides a geometric representation for recurrence equations and generation of mapping solutions. Although we will not discuss probabilistic methods in this paper, we will note that Lin and Yang present a promising probabilistic method. Their probabilistic method provides effective load balancing by simultaneously reducing the standard deviation of each processor's execution time and increasing the correlation coefficient between every two processors' execution times [24].

Cappello and Steiglitz's method defines an algorithm in the form of a system of recurrence equations to be mapped onto a systolic array. From the recurrence equations, a canonical representation may be formed by adding an additional index to represent time. Each index in the canonical equations is representative of a dimension in geometric space. Points may be defined in space to represent the sets of indices where the equations are defined. The location of the points in space is determined by interpreting the indices on the left-hand side of the recurrence equations as coordinate locations. Each of the defined points is associated with a primitive computation (i.e. the equation on the right-hand side of the equation). The result is a geometric representation of the recurrence equations. Using the geometric representation, static data and communication requirements are identifiable. Individual nodes which require the same static data element are connected by a solid line. Therefore, each static data element is represented by a solid line and the points on the line represent locations which require the data element. Similarly, individual nodes which use the same variable data element are connected by a dashed line. Thus, dashed lines are used to identify communication paths for variable data elements. Movement of the data along the paths is determined by lexicographic order of the indices within time.

By applying matrix transformations representing geometrical transformations (such as those presented in Figure 2.5), a geometric representation can be transformed into a variety of potential mappings [54]. A geometrical projection in a single dimension

(other than the dimension which represents time) directly maps the primitive computations onto processors in a systolic array. Nodes which are projected onto the same point are assigned to the same processor with the number of projected points determining the size of the systolic array. The timing and data flow are determined from the geometric space representation by maintaining precedence relations. Thus, the geometrical representation enables users to easily visualize the systolic array during algorithm transformations and partitioning.

For example, $y = A \cdot x$ can be expressed $y_i = \sum_{j=1}^{n} a_{i,j} \cdot x_j$ where n is the size of vector x. Defining the initial values, the recurrence equations can be written

$$\begin{aligned} y_{i,0} &\equiv 0 \\ y_{i,j} &= y_{i,j-1} + a_{i,j} \cdot x_j \\ y_i &\equiv y_{i,n} \end{aligned}$$

A canonical representation is formed by adding a time index.

$$y_{i,j,t} = y_{i,j-1,t} + a_{i,j} \cdot x_j \qquad t = 0 \tag{2.1}$$

Figure 2.9 (taken from [54]) illustrates a geometric representation (Γ) of Equation 2.1 for $n = 3$. Using this figure, the equation $y_{1,1,0} = y_{1,0,0} + a_{1,1} \cdot x_1$ occurs at coordinates $(1, 1, 0)$. Thus, $a_{1,1}$ and x_1 must be defined at location $(1, 1)$ during time $t = 0$. Solid lines in the graph represent the required locations for x; while dashed lines identify communication paths used to pass $y_{i,j,t}$ for addition. Using the geometric representation, various mappings of individual algorithms can be investigated. For example, a transformation suggested by Leiserson and Kung to schedule $y = A \cdot x$ (shown in Figure 2.9) results in the potential mapping [55]

$$\Lambda = T \cdot R(\Gamma) \;\; where \;\; R = \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \;\; and \;\; T = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

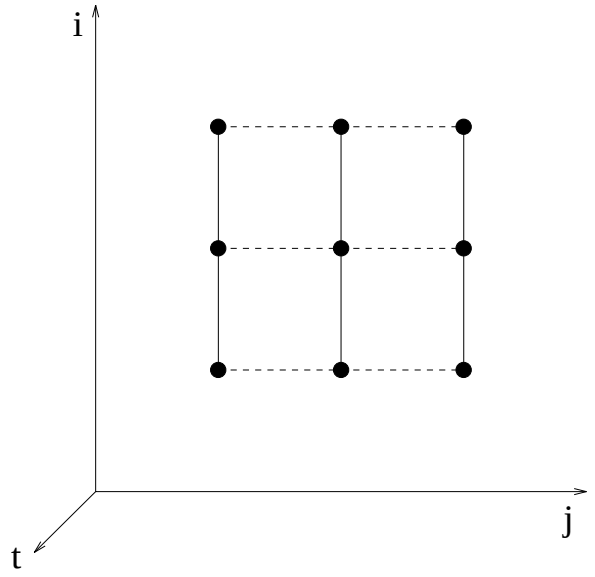


Figure 2.9: Geometric representation of the canonical linear transform $y = A \cdot x$

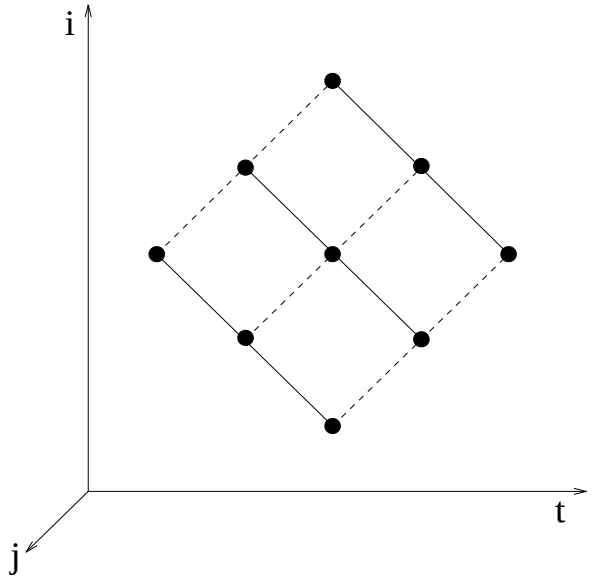


Figure 2.10: Geometric representation of the Leiserson and Kung transform

with Λ shown in Figure 2.10 (taken from [54]). Projecting Λ onto the t axis (note that with the transformation t no longer represents time) yields the task schedule shown in Figure 2.11 (taken from [54]).

Cappello and Steiglitz's methodology targets fine grained parallelism for systolic arrays. Available parallelism in algorithms must be specified in terms of recurrence

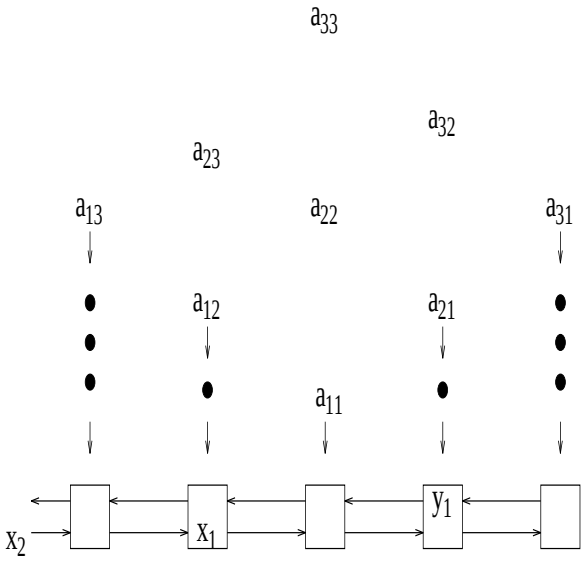


Figure 2.11: Representation of the Leiserson and Kung task schedule

equations which are familiar but not always convenient. The method results in a combination of parallelism during processing of a single sample and between samples; however, both types of parallelism cannot be fully exploited simultaneously. Communication patterns are accounted for during mapping to the systolic arrays, but communication costs and input/output bandwidth are not considered. Scalability is provided by the indices in the recurrence equations. The concepts are limited to systolic arrays which are special purpose architectures with rigid synchronization. In general, they are now out of favor except for low-level repetitive operations. Essentially, Cappello and Steiglitz have provided a visual framework for considering transformations, but do not identify a method to obtain the ideal transformation. The process of applying various transformations can be automated, but running time is potentially unbounded. We now move on to describe some heuristic techniques for task scheduling.

2.3 Heuristic Techniques

We refer to a paper by Manoharan and Topham for a good comparison of heuristic techniques which consider communication costs [46]. The most promising heuristic identified is a non-work-greedy assignment scheme Depth First Breadth Next (DFBN) [46, 56]. While work-greedy assignment methodologies attempt to keep all processors busy as long as there is a task capable of being executed, non-work-greedy schemes do not keep processors busy as a main goal. DFBN attempts to assign independent tasks across multiple processors while assigning dependent tasks onto a single processor. This is done in an effort to balance the goals of maximum parallelism and minimum interprocessor communication.

Unlike many task scheduling methodologies, DFBN allows the user to specify the target architecture with a processor graph. Nodes in the processor graph represent the processors with the edges identifying the communication paths. Values are associated with each edge to represent the link capacity. Algorithms are specified in the form of a task graph. In the task graph, nodes identify a task and edges represent dependencies between tasks. Tasks are defined to be a portion of an algorithm which must be executed sequentially. Values which define the sequential execution time are assigned to nodes. Edges are assigned values to represent the volume of interprocessor communication. The task graph is assumed to be acyclic.

Task graphs may be partitioned into chains of nodes using a combination of depth-first and breadth-first search algorithms [56]. The chains are developed such that the nodes in a chain are dependent. Thus, nodes in different chains will be independent. To determine which chains to schedule first, priorities are assigned to individual nodes. A critical factor is defined for each node (task) such that

$$CF_i = \max_{j=1,n} [LCT_j - ECT_j] - (LCT_i - ECT_i)$$

where ECT_i and LCT_i are the earliest and latest completion times for task T_i [46]. ECT and LCT are initially estimates; exact values are calculated when task assignments are done. The priorities are set through the equation

$$p_i = w_0(CF_i)^{\alpha} + w_1\tau + w_2 \sum_{T_j \in succ(T_i)} v_{i,j} + w_3 \sum_{T_j \in succ(T_i)} 1 + w_4 \sum_{T_j \in succ(T_i)} \tau_j + w_5 s_i$$

where w's are empirical weights, $\alpha \gg 1$ (to emphasize CF), τ_i is the execution time of T_i, $succ(T_i)$ identifies the successors of task T_i, $v_{i,j}$ is the volume of the communication between T_i and T_j, and s_i is the memory requirements of T_i [56]. The order of available processors is determined by choosing the processor with the best overall link capacity and then ordering the other processors according to their distance from the chosen processor. The distance between two connected processors P_i and P_j is defined as $\frac{1}{c_{i,j}}$ where $c_{i,j}$ is the link capacity between P_i and P_j. For processors not directly connected, the distance equals the sum of the distances along the shortest path (using Dijkstra's algorithm). Task chains are assigned in order of priority to the best available processor.

For the example processor graph and task graph in Figure 2.12 (taken from [46]), the resulting assignment is

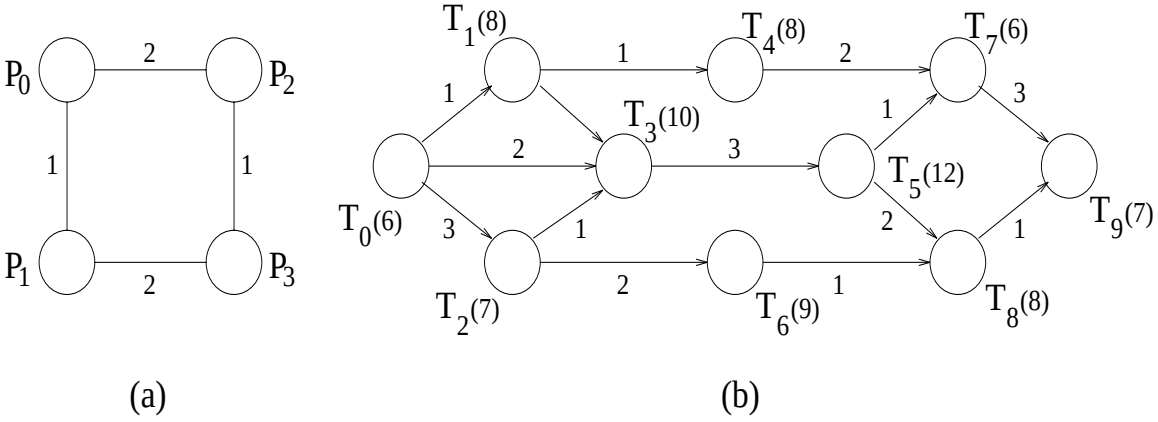


Figure 2.12: Example (a) processor graph and (b) task graph for DFBN

$P_0 : 0, 1, 3, 5, 8, 9$
$P_1 : 4, 7$
$P_2 : -$
$P_3 : 2, 6$

While the partitioning obtained by the DFBN assignment scheme is not guaranteed to be optimal, DFBN is shown to consistently produce partitionings close to optimal for acyclic task graphs [46]. Note that optimality for acyclic task graphs is determined by minimizing the throughput delay, the number of processors, and the interprocessor communication. The specification method is familiar and convenient using task graphs and processors graphs. The heuristic automatically identifies and exploits available fine-grained parallelism specification of the algorithm in the task graph) without requiring the user to specify available parallelism. However, the task graph is acyclic, allowing only parallelism during processing of a single sample to be exploited. This method will not work effectively for iterative algorithms. The cost of interprocessor communication is considered during the evaluation of priorities; however, final communication patterns are not validated and input/output bandwidth is not considered. Assuming that the task graph is able to contain non-atomic nodes, the methodology is scalable and a variety of architectures may be supported. Perhaps the biggest drawback to this methodology (for DSP algorithms) is its failure to support feedback loops for iterative algorithms. The algorithm's complexity is $O((n+m)\log m + e)$ where n is the number of tasks, m is the number of processors, and e is the number of edges in the task graph. Thus, the running time should be acceptable.

Graphical algorithm specification methods provide yet another challenge for research in parallel task scheduling. In the next section, we discuss three graph-based methodologies for scheduling DSP algorithms for parallel execution.

2.4 Graph-Based Methods

Graph-based task scheduling techniques begin with the user defining an algorithm in the form of a flow graph. Flow graphs can be in one of several forms. We begin by describing fully specified flow graphs (FSFGs) as a specification method which effectively represents both computation and communication properties of algorithms. A FSFG is a flow graph with computational nodes, communication edges and ideal delays. Figure 2.13 (taken from [37]) presents an FSFG for a 1-D 2nd order IIR filter. Nodes represent an atomic arithmetic or logic function taking place, with

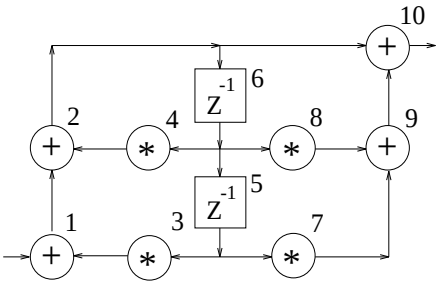


Figure 2.13: FSFG for a 1-D 2nd order IIR filter

a fixed computational delay. An edge indicates dependence of one node on data produced by another node. An ideal delay is represented by block labeled "Z^{-1}" and denotes a delay in time when a signal passes through the block (representative of hardware registers and memory). Using FSFGs, computational information is completely captured in the atomic nodes of the flow graph, while the communication is contained entirely in the directed edges. Once the order of these atomic nodes is defined, the FSFG has already bound the computation and communication pretty tightly to a particular model of execution. As a result, the initial definition of a FSFG tends to limit the possible mappings for a given algorithm.

Fundamental bounds to multiprocessor performance are defined to determine how well a FSFG might be executed [37, 38, 39]. The Iteration Period Bound (IPB) is

one performance bound of interest for FSFGs with feedback for iterative execution. The IPB is defined as

$$T_v = \max_{l \in L} \left\lceil \frac{D_l}{n_l} \right\rceil \tag{2.2}$$

where D_l is the computational delay of all nodes in loop $l \in L$ and n_l is the number of ideal delays in loop $l \in L$. The set of all computational loops (L) may be found by tracing directed edges which form a loop without re-using a directed edge when defining a single loop l. The IPB ensures that the period between inputs is long enough to process all the computational loops in the FSFG. A FSFG which is executed at an iteration period equal to T_v takes a new input every T_v time units and is said to be rate optimal. For the purposes of this research, we assume it takes one time unit for an addition and two time units for multiplication. Using these assumptions, $T_v = 3$ for the example FSFG shown in Figure 2.13.

Once an algorithm has been specified by a FSFG, the nodes of the FSFG may be scheduled for execution on a multiprocessor system. One basic technique for obtaining a multiprocessor schedule is through the identification of critical paths in the FSFG [35]. However, this method does not attempt to optimize the number of processors or to consider communication costs. Another graphical approach, the Range-Graph Method (RGM), partitions algorithms with the goal of finding an optimal position for scheduling individual nodes within a scheduling window [36]. This method is discussed on the basis of favorable comparisons with other graph-based scheduling methods. RGM defines

critical path path containing a set of nodes with a mobility of zero

scheduling window the length of the critical path

scheduling range the time interval an operation can start its execution within the scheduling window

mobility the length of the scheduling range

scheduling range chart display of the scheduling range and the mobility for all operations

An early schedule for a FSFG defines the schedule in which nodes are executed as soon as possible within the scheduling window. Similarly, a late schedule identifies a schedule in which nodes are executed as late as possible within the scheduling window. Combining the early and late schedules yields a scheduling range chart. Figure 2.14 (taken from [36]) presents an example of a non-iterative FSFG, an early schedule, and a late schedule. Figure 2.15 illustrates the combination of an early schedule and a late schedule to form a scheduling-range chart. RGM provides two types

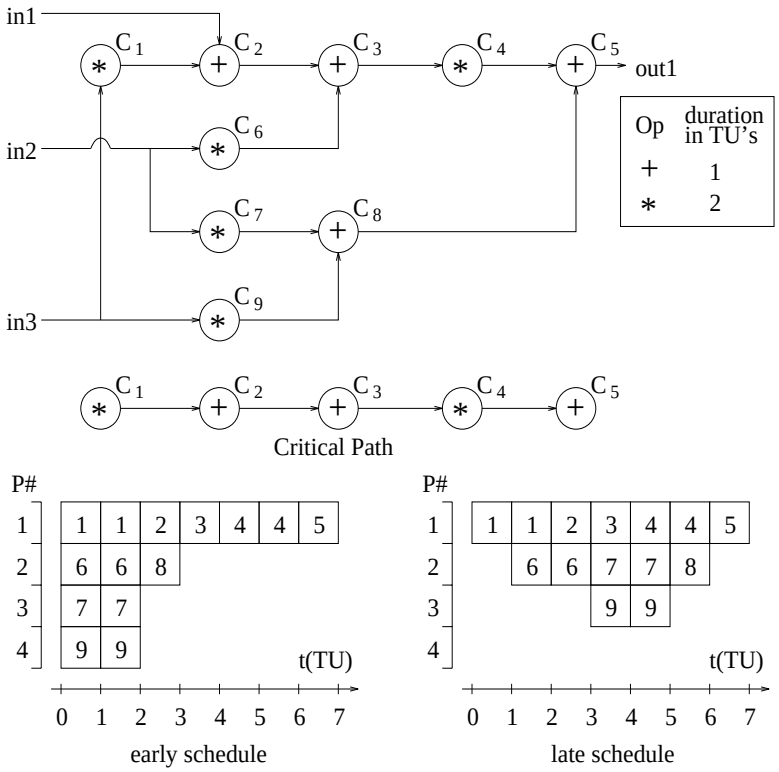


Figure 2.14: Example of a non-iterative FSFG with an early schedule and a late schedule

of scheduling: non-iterative and iterative. Non-iterative scheduling defines a finite

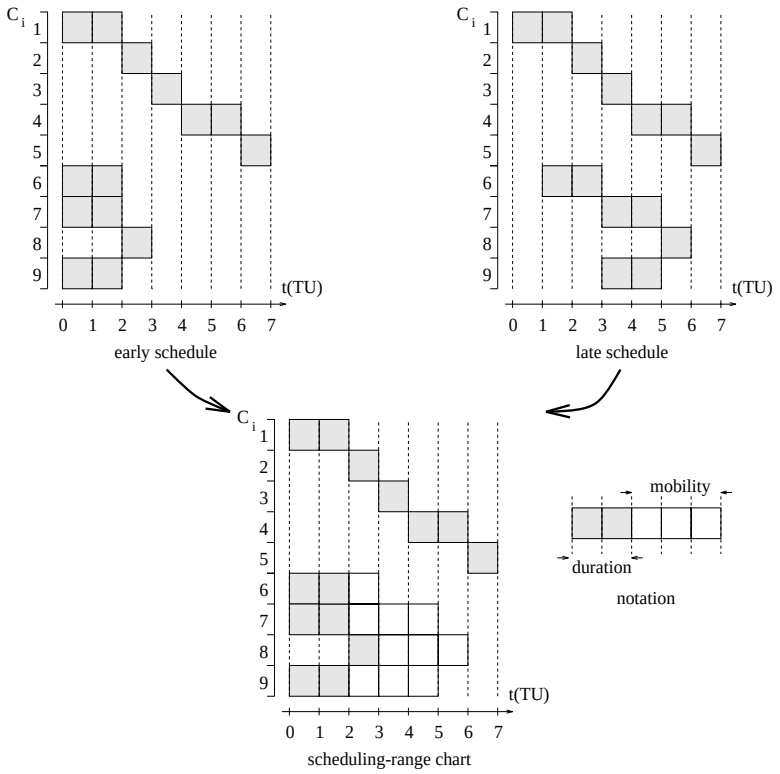


Figure 2.15: Example of combining an early schedule and a late schedule to form a scheduling-range chart

scheduling range determined by the iteration period bound (set by the critical path). For a non-iterative schedule, the critical path is scheduled and then other nodes are scheduled in order of increasing mobility. Heuristics are used to choose processors based on the availability of a time slot which matches the duration of the node to be scheduled (with consideration of mobility). Iterative scheduling requires an infinite scheduling range; therefore, a reference node is used to fix the schedule in time. The reference node is chosen from the nodes in the critical loop and is scheduled first. The iteration period bound is used to fix the schedule range around the reference node with the critical loop identifying the critical path. Other nodes in the FSFG are scheduled with respect to the reference node in order of increasing mobility [2].

[2] A threshold scheduling technique similar to RGM has been developed for use with SISAL [50]. Parallelism between operations is represented by a directed acyclic graph, and schedules are computed for each possible threshold between the early and late schedules.

For example, consider the 1-D 2nd order IIR filter presented in Figure 2.16 (taken from [36]), the critical loop can be identified as $c_4 \rightarrow c_2 \rightarrow d_1$. Choosing c_2 as the

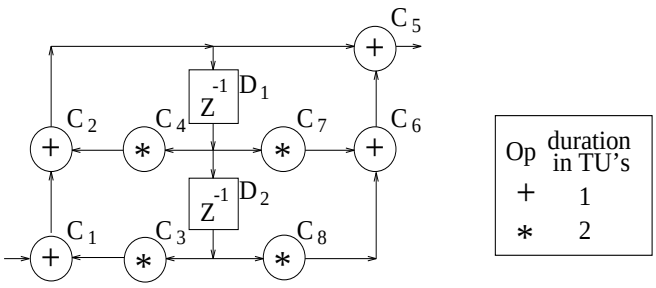


Figure 2.16: 1-D 2nd order IIR filter for RGM

reference node, the scheduling steps are illustrated in Figure 2.17. The scheduling range chart in Figure 2.17 (a) is developed by fixing node c_2 in time, obtaining early and late schedules, and combining the early and late schedules to form the range

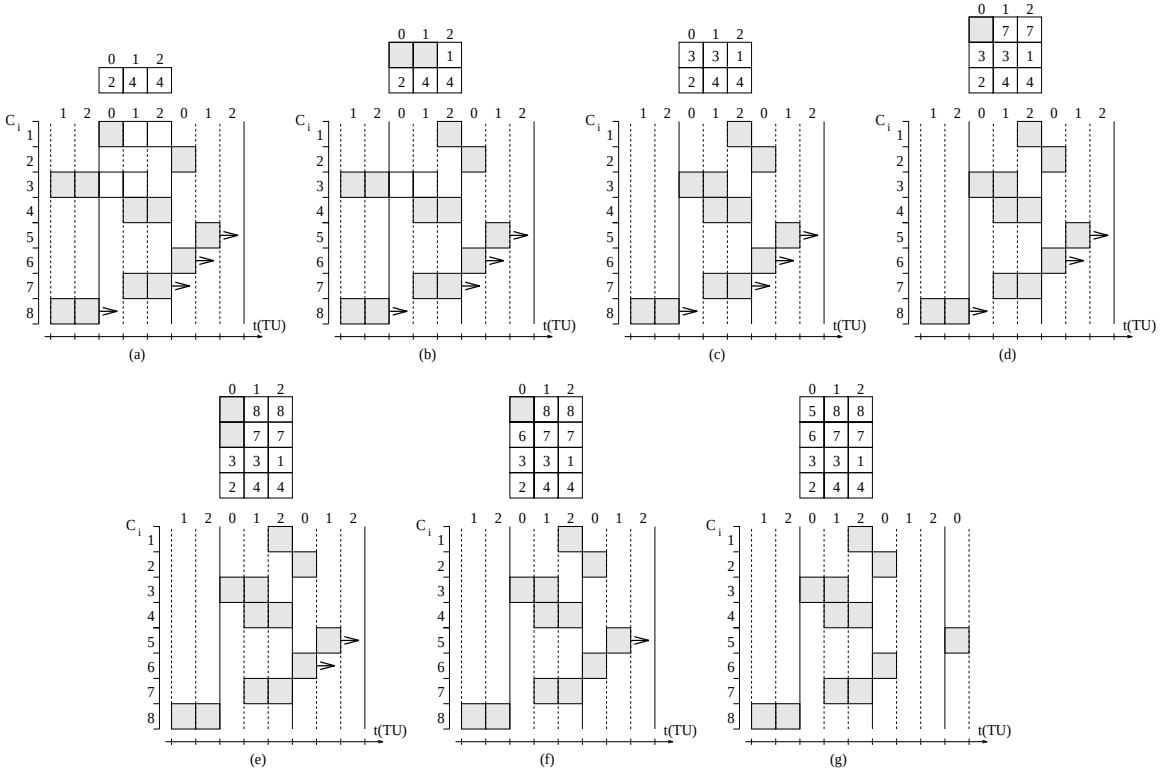


Figure 2.17: RGM scheduling steps for a 1-D 2nd order IIR filter

chart. Arrows pointing forward in time indicate an infinite scheduling range for the late schedule. In Figure 2.17 (a) node c_2 is scheduled as the reference node followed by the scheduling of node c_4 which has zero mobility. In Figure 2.17 (b) and (c) nodes c_1 and c_3 are scheduled in order of increasing mobility. Nodes c_7, c_8, c_6, and c_5 are scheduled into empty time slots to complete the multiprocessor schedule.

RGM provides automatic scheduling for fine-grained parallelism by exploiting parallelism during processing of a single sample and between samples. However, interprocessor communication time is assumed to be negligible with respect to computation, and may result in unacceptable schedules. In addition, the input/output communication bandwidth is not considered. Familiar and convenient FSFGs are used to specify algorithms, but scalability to larger problems may prove problematic with requirements for large FSFGs and large range charts. RGM is capable of optimizing processing time and/or the number of processors without requiring users to identify potential parallelism. RGM is suitable and practical for use on multiple parallel architectures (which do not use wormhole routing) with published running times of under one CPU second for FSFGs up to 44 nodes. Scheduled algorithms include FIR, IIR, lattice, Jaumann and elliptic filters.

Another graph-based approach of particular interest is the work being done on optimal, periodic multiprocessor scheduling for fully specified flow graphs, which we term the cyclostatic realization method (CRM) [37, 38, 39]. This method combines the familiarity of a flow graph specification technique with the concise representation of matrix theory to provide a flexible, generalized mapping methodology.

The cyclostatic realization method attempts to schedule the FSFG to meet given bounds and has been widely published and referenced. Barnwell et. al. developed a theory of algorithm mapping whose goal is to simplify the handling of the FSFG during the mapping process while obtaining the performance bounds. While a detailed description of CRM can be found in [37, 38, 39], we will provide a brief overview.

Using CRM, the problem is modeled by a period matrix and a cycling vector. The period matrix has one row for each processor used for computing the FSFG and one column for each time unit in the iteration period. The elements of the period matrix are computational time units in the FSFG. Each of the nodes in the FSFG is numbered and placed in the period matrix with a single node entry appearing in the period matrix once for each time unit it takes to execute that node. The cycling vector has a single column and one row for each row of the period matrix. The entries of the cycling vector are integers 1 to r where r is the number of rows of the period matrix. A cycling vector determines the movement of the rows of the period matrix between iterations. Superscripts in the period matrix represent which samples are being processed in parallel in different partitions. If some (arbitrary) partition is defined as processing the "current" input sample, a superscript of i denotes concurrent processing of the i^{th} sample after this (or before this, if the superscript is negative). An example partitioning for Figure 2.13 is illustrated in Figure 2.18. This partitioning is represented by the cycling vector and period matrix shown in Figure 2.19.

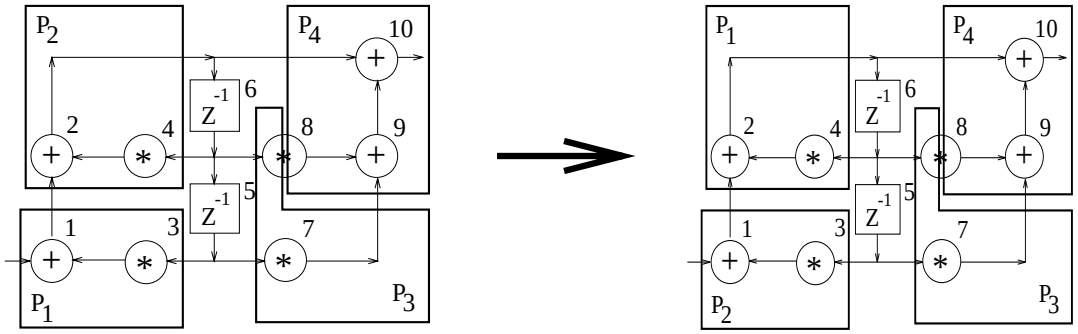


Figure 2.18: Data movement for an example partitioning of a 2nd order IIR filter

Specifying an algorithm in an intermediate form using matrices allows different multiprocessor schedules to be obtained by modifying the matrices. The period matrix and the cycling vector may be manipulated by exchanging rows and columns to yield different multiprocessor schedules. Manipulations of the period matrix include the

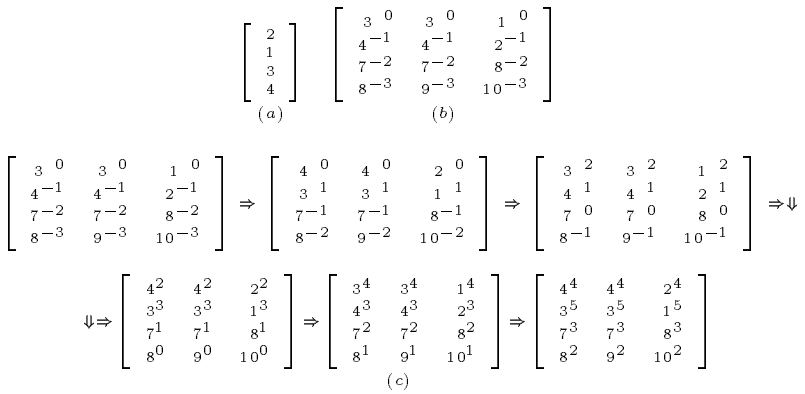


Figure 2.19: Basic Example for 4 processor system executing FSFG of 1-D 2nd Order IIR FSFG: (a) Cycling vector (b) Period matrix (c) Six iteration periods for (a) and (b)

exchange of rows, the rotations of the columns, and the slicing and splicing of rows. The cycling vector can be manipulated using an exchange of rows to obtain any ordering of the vector's elements. Resulting periodic schedules must be checked for viability against a set of admissibility conditions presented in [37, 38]. The precedence condition states that nodes in the FSFG must be calculated in the order that they occur from the input to the first delay, between delays, and from the last delay to the output. In other words, each node must have the necessary inputs in order to be scheduled. The nonpreemption condition states that once a node of the FSFG has been scheduled on the processor, that node can not be preempted by the scheduling of another node.

To find an optimal mapping, the methodology enumerates all possible sets of transformations of the period matrix and the cycling vector. (Figure 2.20 shows several possible manipulations of the period matrix and the cycling vector for Figure 2.13.) The resulting periodic schedules are checked against the admissibility criteria and the performance bounds. If the periodic schedule is both permissible and optimal, the periodic schedule is accepted as a valid mapping. Depending on the size and complexity of the period matrix, the process of finding all possible transformations

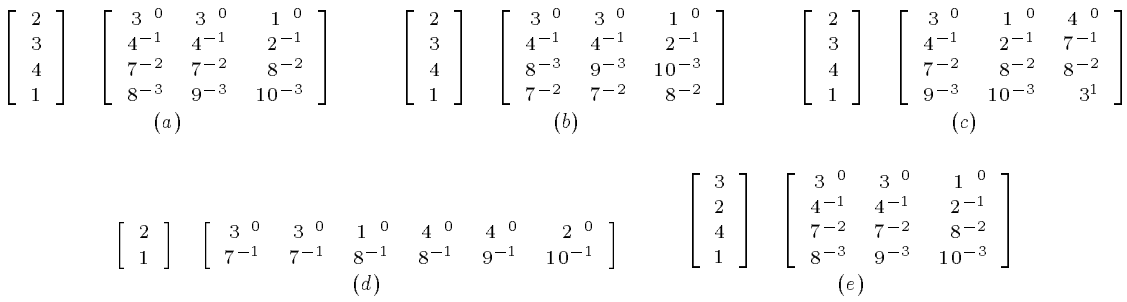


Figure 2.20: Schedule Transformations: (a) original period matrix and cycling vector (b) rows 3 and 4 of the period matrix are exchanged (c) columns of the period matrix are rotated left by one (d) rows 2 and 4 of the period matrix are spliced to rows 1 and 3 respectively (e) rows 1 and 2 of the cycling vector are exchanged

could take an amount of time which is exponential in the problem size.

CRM is a convenient, powerful method for partitioning and scheduling parallel computations. CRM automatically identifies parallelism within processing of a single sample and between samples to produce parallel processing schedules. CRM generates schedules to meet the optimal processing time and the optimal number of processors, and does not allow schedule generation for non-optimal solutions. It is worth noting that there are some FSFGs which can not be scheduled at the optimal iteration period on the optimal number of processors. Interprocessor communication costs and input/output bandwidth limitations are not explicitly considered (although a recent paper has mentioned that consideration of interprocessor communication costs is possible [38]). In general, CRM assumes an interconnection network which is a crossbar switch with unlimited bandwidth. This assumption is clearly unrealistic. Algorithms are specified in the form of FSFGs which limit scalability to the practical size for non-hierarchical FSFGs. CRM has been developed in C to permit a high degree of portability and is suitable for scheduling algorithms onto a variety of parallel architectures. Running times are unpublished, but are noted to be acceptable. Practicality for implementation should be determined by running a set of standard, realistic benchmarks.

All of the methods presented have characteristics beneficial to the task scheduling problem; however, none of the methods are totally effective. The focus of this research is to provide an effective methodology to schedule DSP algorithms onto the BDPA. Some of the major hurdles include continuous input data, high throughput requirements, feedback loops, communication costs, and a need for a user friendly system. Programming languages in combination with their compilers are effective when the target system fits in the mold of the language's development system. A compiler must exist for the target system and users must have experience with the language to achieve optimal parallelism when scheduling an algorithm. Mathematical approaches are geared toward systolic arrays and would be unable to take full advantage of the BDFP. Heuristic approaches have shown promise; however, even the best heuristic, DFBN, approaches the task scheduling problem assuming an acyclic FSFG. Graph-based methods are perhaps the friendliest for users of the BDPA. The RGM handles feedback loops well but does not consider communication costs in producing a parallel schedule. CRM provides the best representation for parallel processing schedules; however, the partitioning and scheduling algorithm does not handle the communication costs associated with the partitioning of feedback loops. In addition, CRM's generation of all possible cycling vectors and period matrices is computation time intensive and cannot guarantee a schedule.

In the following chapter we introduce the Order Graph Method (OGM) as an effective alternative to existing parallel task scheduling techniques. OGM provides a user-friendly task scheduling methodology for scheduling algorithms specified by hierarchical FSFGs. Feedback loops, communication costs, and the network topology are considered during the scheduling process to guarantee an effective, valid schedule.

Chapter 3

The Order Graph Method

3.1 Overview

The Order Graph Method begins with some basic assumptions about the multiprocessor system's architecture.

- **processors are non-pipelined general purpose processors**
- **a processor is capable of communicating (receiving and transmitting) no more than t_c units of data per clock cycle**
- **a link in the network topology is capable of transmitting no more than t_n units of data per clock cycle**
- **communications and computations are overlapped with the overall execution time determined by the larger of the two**
- **each processor is capable of communicating directly with system input/output devices**

These assumptions are based on the use of additional hardware to handle communication independently of computation [1].

[1] The Texas Instruments TMS320 C40 processor has separate hardware for communication and computation. Thus, communication and computation can be overlapped.

The goal of the Order Graph Method is to produce a multiprocessor schedule for a user-defined FSFG which achieves a desired iteration period (T_d) [57, 58]. This serves two purposes. There is no need to attempt to schedule the algorithm with a faster iteration period than necessary. Therefore, the execution time for the scheduling process is reduced as T_d becomes greater than the iteration period bound (T_v). In addition, by scheduling to T_d, OGM minimizes the cost of the system by using the minimum number of processors necessary to execute the FSFG. It should be noted that the OGM also permits scheduling to a specific number of processors (P_d) by using the total computational time in the FSFG (C_N) to identify T_d for the system. Thus, OGM requires two inputs - the FSFG to be scheduled and either T_d or P_d with the relationship.

$$T_d = \left\lceil \frac{C_N}{P_d} \right\rceil$$

The notation used to detail the Order Graph Method. is shown in Table 3.1.

In general, OGM has four major steps:

- Preprocess the FSFG to minimize interprocessor communication
- Generate candidate partitionings for the graph using heuristics to balance the workload across processors
- Generate schedules for the candidate partitionings
- Validate schedules against the interprocessor communication limitations, the link capacity, T_d, and P_d

Figure 3.1 presents an outline for the algorithm used in the OGM. OGM accepts hierarchical FSFGs as input with a user specified target for scheduling in the form of a target iteration period, a target number of processors, or the goal to optimize both the iteration period and the number of processors. OGM will output a schedule

Symbol	*Definition*
C_i	total amount of communications into and out of processor i during T_d clock cycles
C_N	total computational time for a FSFG
$d_i(n)$	member of the set $D_i(n)$
$D_i(n)$	set of all ideal delays for communications between node n on processor i to node n's outputs on other processors
D_l	total computational delay of nodes in loop l
E	set of all edges in a digraph
E'	set of all edges in a directed acyclic graph
f_i	subgraph of a directed acyclic graph consisting of the nodes to be executed on processor i
F_i	set of nodes in a FSFG to be executed on processor i
G	directed graph
G'	directed acyclic graph
I_i	indegree of a subgraph
l	particular loop in a graph
l_x	set of links required to support the communication due to the nodes in set x
L	set of all loops in a graph
L_i^j	number of communications over the link between processor i and processor j in a network topology
M_n	set of all output nodes for node n
n	number of nodes in a FSFG
n_l	number of ideal delays in loop l
$n_{n,m}$	number of ideal delays along a directed path between node n and m in a graph
$o_i(n)$	member of the set $O_i(n)$
O_i	outdegree of a subgraph
$O_i(n)$	set of all processors which contain the output nodes for node n on processor i
P	poset for a FSFG
P_d	desired number of processors
r	number of processors for a candidate partitioning
r_{min}	minimum number of processors for all validated candidate partitionings at the desired iteration period
t	number of linear extensions which satisfy a set of precedence conditions (the size of a poset)
t_c	maximum number of communications into and out of a processor during a clock cycle
t_n	maximum number of communications across a link in the network topology during a clock cycle
T_d	desired iteration period
T_v	iteration period bound
v	node in a digraph
v'	node in a directed acyclic graph
V	set of all nodes in a digraph
V'	set of all nodes in a directed acyclic graph
w	node in a digraph
w'	node in a directed acyclic graph
w_k	weight of node k
X_i	number of communications between processor i and input/output devices

Table 3.1: List of symbols used to detail the Order Graph Method

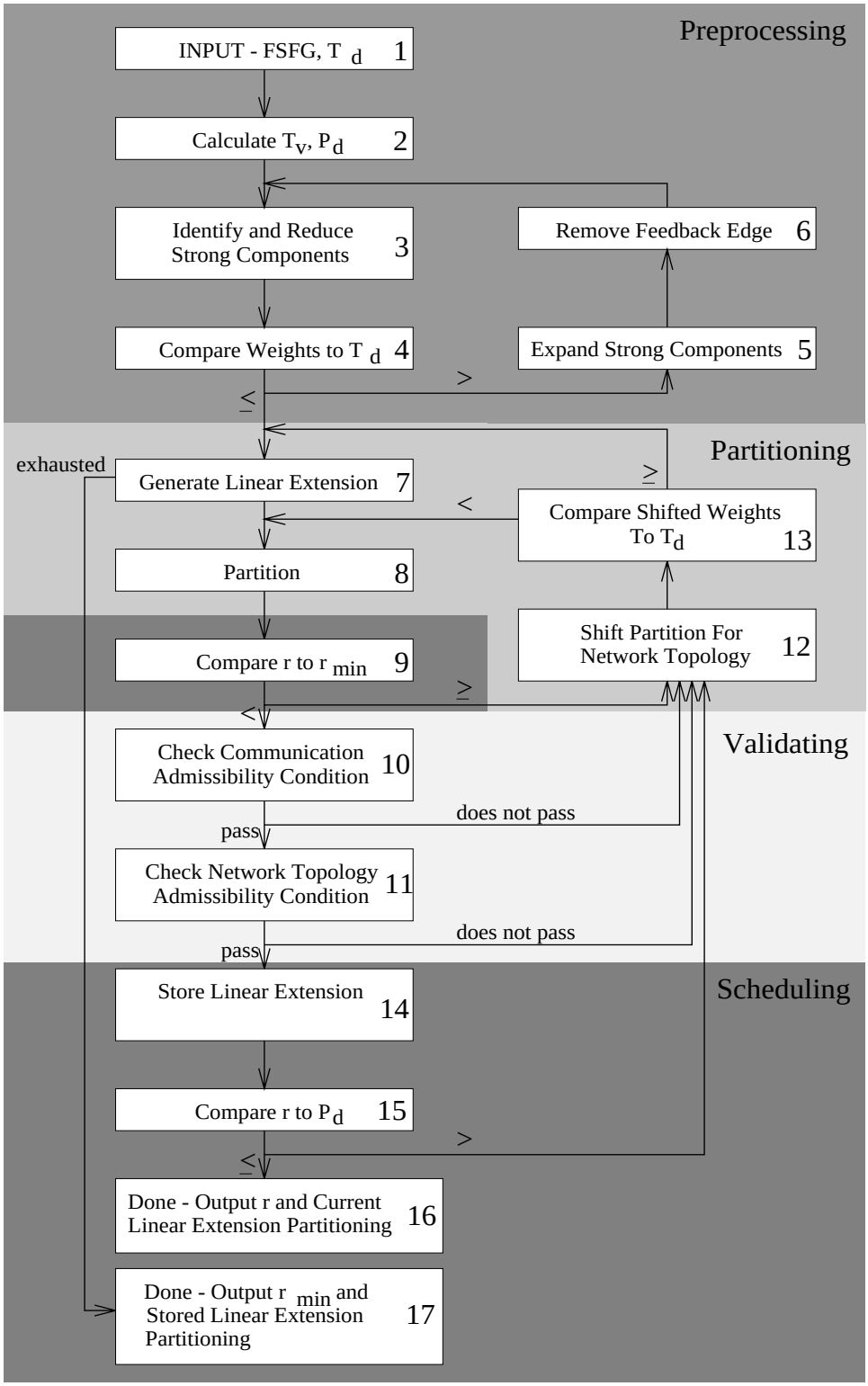


Figure 3.1: Flow Chart for the Order Graph Method

which assigns the nodes in the FSFG to processors and determines when nodes are executed on the processors. OGM begins by calculating T_v and P_d if the user inputs T_d as the scheduling goal. If P_d is presented as the goal, T_v and T_d are calculated by OGM. Feedback conditions are handled through the identification and manipulation of strong components. Nodes within a strong component are merged into a single node in an effort to represent each feedback loop as a single node. Removing the feedback loops (cycles), the FSFG is reduced to a directed acyclic graph (DAG). If $T_d < T_v$, interleaving is used to lower the iteration period below the IPB. Using the partially ordered set (poset) denoted by the DAG, linear extensions are generated to identify a specific ordering of the nodes in the poset. The linear extensions are then partitioned into sets with computational weights less than T_d. In this dissertation, an individual set is called a partition. Together these sets define a candidate partitioning which can be assigned to a set of processors to form a multiprocessor schedule. Thus, each linear extension is partitioned according to the desired iteration period (T_d) into r partitions with each partition (F_i) containing a set of nodes to be executed on processor i. The set of all these F_i partitions determines the candidate partitioning. The candidate partitioning is a potential schedule to be validated against a set of communications and network topology admissibility conditions. A valid candidate partitioning is one that has passed both the communication and network topology admissibility conditions. Through the enumeration of a FSFG's linear extensions and the linear extensions' candidate partitionings, a valid multiprocessor schedule is obtained.

In order to incorporate the communication and network topology admissibility conditions into multiprocessor scheduling, OGM necessarily iterates. At any time in the scheduling process, the lowest value of r for any validated candidate partitioning is held in r_{min}. Therefore, candidate partitionings with $r > r_{min}$ do not need to be validated against the admissibility conditions. A better valid schedule already has

been produced with r_{min}. When a linear extension can no longer produce unique candidate partitionings, it is eliminated from consideration and another linear extension is generated for partitioning and validation. Candidate partitionings which pass all admissibility conditions are checked for processor optimality. For a given T_d the optimal processor utilization occurs with $P_d = \lceil \frac{C_N}{T_d} \rceil$ where C_N is the total computational time for the FSFG. Once r_{min} achieves the target number of processors P_d, a processor optimal schedule has been found and the scheduling process stops. In the event that unique linear extensions are exhausted without being able to schedule the FSFG an iteration period of T_d on the optimal number of processors (P_d), the best valid schedule is presented as a schedule with an iteration period of T_d on r_{min} processors.

We now describe some of the important details in OGM - beginning with a description of the preprocessing stage which handles feedback in FSFGs.

3.2 Preprocessing Step - Handling Feedback

When determining scheduling for multiprocessor systems, feedback loops in the FSFG require particular attention. If feedback loops must be split across two separate processors, there becomes a need for bi-directional communication between processors. Thus, the communication time increases for any processors involved in the communication of the feedback loop. In addition, the potential for contention in the network topology increases. Therefore, OGM prevents unnecessary divisions of feedback loops on multiple processors by preprocessing the FSFG.

To prevent the unnecessary splitting of feedback loops, OGM identifies the feedback requirements and attempts to contain each feedback loop in a single node. A FSFG is a directed graph (digraph) $G = (V, E)$ where V is the set of all nodes and E is the set of all edges. A digraph G is strongly connected if for each pair of vertices

$v, w \in V$ there is a directed path from v to w and a directed path from w to v as well. A strongly connected component (strong component) of a digraph is a maximal strongly connected subgraph (G) such that there is no pair of vertices $v \in V$ and $w \notin V$ with a directed path from v to w and a directed path from w to v. Thus, a strong component will contain either a single node or the set of all the nodes involved in a feedback loop. The nodes in the strong component of a digraph can be merged into a single node producing a digraph, $G' = (V', E')$, with no cycles (directed acyclic graph - DAG). OGM merges the nodes in each strong component into a single node. The weight of the single node equals the sum of the computational times for the nodes in the strong component. During the process of identifying the strong components in the FSFG, the iteration period bound is found by analyzing each strong component for loops. For each loop, the computational delay and the number of ideal delays is used to find the maximum iteration period (T_d). OGM uses a strong component algorithm by Tarjan which requires $O(\mid V \mid\mid E \mid)$ time [59]. Johnson's algorithm is used to identify loops in the strong components and requires $O((\mid V \mid + \mid E \mid)(\mid L \mid +1))$ time where L is the number of loops in a strong component [60].

In order to effectively schedule the nodes of the resulting DAG, the weight of any node $v' \in V'$ must not be greater than T_d. If the weight of v' is greater than T_d, v' must be split to a set of nodes with weights less than T_d. This is accomplished by splitting v' back into the nodes of its strong component, removing a feedback edge, identifying the new strong components, and merging the nodes of the new strong components into composite nodes. To remove a feedback edge from the strong component, an edge adjacent to a delay node is removed from the cycle having the lowest iteration period. For each loop $l \in L$ of the strong component, the computational delay D_l and the number of ideal delays n_l have been identified during the calculation of T_v. Thus, the feedback edge of loop l is removed from the cycle where $\frac{D_l}{n_l}$ is the lowest. Once the correct cycle is found, additional checks ensure that the removed edge is

adjacent to a delay node and is shared by the fewest number of cycles in the strong component. These checks ensure that the minimum number of loops (but at least one) within the strong component are broken. In addition, this heuristic ensures that the FSFG's critical path is contained within a strong component. This is done in an effort to schedule the critical path on a single processor.

The strong components for the remaining digraph are represented by composite nodes, and the digraph is reduced to a DAG in the same manner as before. This process of merging and splitting nodes from digraphs to DAGs continues until all the weights in the DAG are less than T_d (See Figure 3.1). If strong components are split into more than one node, the iteration period will remain equal to the IPB because the feedback loop still exists across multiple processors. However, by interleaving independent streams of data, the average iteration period can be reduced [57, 58]. Although the DAG no longer captures the exact precedence constraints in the original FSFG, it can be used to generate linear extensions which minimize the probability that feedback loops will be split across multiple processors. If the exact dependencies are needed, the original FSFG is used.

For example, consider the FSFG for the 2nd order IIR filter shown in Figure 2.13. Preprocessing begins by identifying the nodes in the strong component - $\{1, 2, 3, 4, 5, 6\}$. These nodes are represented by a composite node (C_1) with a weight of 6. A comparison to $T_d = 3$, indicates that this composite node is too big to schedule in a single partition and must be split. Analyzing the loops for the strong component, loop (A) $2 - 4 - 6$ has an iteration period equal to 3 and loop (B) $1 - 3 - 5 - 6 - 2$ has an iteration period equal to 2. Edge $3 - 5$ is an edge from loop (B) which is adjacent to a delay node and is not shared with loop (A). This edge is removed. The remaining strong component $2 - 4 - 6$ is collapsed back into a composite node (C_1) with a weight of 3. This time, a comparison of the nodes in the DAG to T_d validates the DAG $\{C_1, 1, 3, 7, 8, 9, 10\}$ which is then forwarded to the partitioning stage.

The DAG produced by the preprocessing stage is used by the partitioning stage to generate candidate partitionings. This stage determines, to a great degree, the amount of time the scheduling process will require. We now describe the generation of these candidate partitionings.

3.3 Partitioning Step - Generating Candidate Partitionings

The partitioning stage generates a complete set of candidate partitionings for validation and scheduling. In order to schedule the DAG on a set of processors, an ordering for the execution of the nodes must be determined. For execution on a single processor, an order which preserves the precedence constraints may be obtained by listing all of the possible orderings (linear extensions) and picking the best one. Execution on multiple processors requires either parallel processing of the linear extension or pipelining of the linear extension (with each processor executing a portion of the linear extension). In either case, it is necessary to identify a valid linear extension. For the purposes of OGM, each unique linear extension can produce one or more candidate partitionings to be validated against a set of admissibility conditions. In many cases, the generation of these linear extensions begins with a partially ordered set (poset).

Essentially a poset (P) is a method of ordering nodes of a graph in a way which preserves the data dependencies of the individual nodes. Thus, the DAG is simply a representation for the poset. Using the precedence constraints identified from the DAG, we represent the poset in the form of a relations matrix. For a node w' in the DAG which requires an output from node v', the relations matrix contains a "TRUE" entry for v', w'. In addition, the poset contains an initial ordering for the nodes in the DAG which maintains precedence. This initial ordering can be found

using a depth-first search from each input node until all nodes in the graph have been covered. The generation of all linear extensions from a poset has been addressed in several research efforts - Pruesse and Ruskey [61, 62], Kalvin and Varol [63], Varol and Rotem [64]. Varol and Rotem accomplish the enumeration of the linear extensions with complexity $O(\mid V \mid t)$ where t is the number of linear extensions which satisfy the precedence constraints. The algorithm is very simple and has been efficiently used in practice. A faster (but more complicated) algorithm for generating linear extensions from a poset was developed by Gara Pruesse and Frank Ruskey [61]. Using this methodology, a complete set of linear extensions is generated in $O(t)$ time with $O(\mid V \mid^2)$ time for pre-processing. For example, a graph G' with $V' = \{1, 2, 3, 4, 5\}$ and $E' = \{1 \rightarrow 3, 2 \rightarrow 3, 2 \rightarrow 4, 3 \rightarrow 5, 4 \rightarrow 5\}$, the list of linear extensions includes $[1, 2, 3, 4, 5]$, $[2, 1, 3, 4, 5]$, $[1, 2, 4, 3, 5]$, $[2, 1, 4, 3, 5]$, and $[2, 4, 1, 3, 5]$. Using Varol and Rotem's algorithm, the linear extensions are generated from the poset (This occurs in the Generate Linear Extension block of Figure 3.1).

Each linear extension produced may be executed on a single processor; however, execution on multiprocessor systems requires that the linear extension be partitioned for scheduling. OGM greedily partitions a linear extension into r partitions such that the weight of any partition does not exceed T_d. The set of all r partitions is a candidate partitioning. Let us define the linear extension to be a sequence of entries v'_k $1 \leq k \leq n$. Each v'_k is associated with a weight w_k representing the computational time requirements. The linear extension is greedily partitioned into sets F_i (maintaining order) such that the sum of the weights in $F_i \leq T_d \ \forall\, i$. That is,

$$F_i = \{v'_j, \cdots, v'_k\} \mid w_j + \cdots + w_k \leq T_d \bigwedge \left[(w_j + \cdots + w_k + w_{k+1}) > T_d \bigvee k = n\right]$$

Assume that each node in the linear extension $[1, 2, 3, 4, 5]$ has a computational weight of 1 and $T_d = 2$, the greedy partitioning is $F_1 = \{1, 2\}$, $F_2 = \{3, 4\}$, and $F_3 = \{5\}$ The goal of this heuristic is to create the minimum number of partitions (F_i) with a

balanced computational delay across all F_i. Each F_i partition defines a set of nodes to be executed on the i^{th} processor. Thus, this goal results in the minimum number of processors required for parallel execution of a set of partitions and in a balanced workload for the processors.

Adding to the flexibility of this heuristic is the ability to consider network topologies with known point to point interconnections. Because the interconnections are known, the partitioning process can begin with *any* node in the linear extension. Partitioning of the linear extension moves from left to right using a wraparound from the end of the extension to the start of the extension. The first partitioning of a linear extension begins with the leftmost node and forms the first candidate partitioning for a linear extension. Subsequent candidate partitionings are generated from the same linear extension by shifting the starting point one node to the right. Duplicate candidate partitionings are formed once the starting point has been shifted to the right over nodes with a total computational weight greater than or equal to T_d. Thus, the maximum number of candidate partitions produced by a single linear extension is determined by the number of members in F_1 for the first candidate partitioning. By adding this flexibility, OGM provides a greater number of candidate partitionings to be considered for validation and scheduling.

OGM identifies the number of processors used for a particular candidate partitioning (r) and maintains the best valid candidate partitioning with the minimum r (referred to as r_{min}) at any point in time. Therefore, candidate partitionings are immediately rejected when they require more partitions than r_{min}. Linear extensions are exhaustively enumerated until an extension is partitioned and a candidate partitioning is validated with P_d sets or until there are no more unique linear extensions to be generated.

When a candidate partitioning needs to be validated, OGM represents the candidate partitioning in terms of a cycling vector and period matrix. The period matrix

is formed from the F_i partitions by assigning the elements of F_i to row i of the period matrix in order of precedence. Each element of F_i appears w_k times where w_k is the weight of the element. As in CRM, the cycling vector determines the movement of the rows of the period matrix between iteration periods. OGM supports two potential cycling vectors - one for parallel execution and one for software pipelining. The cycling vector for parallel execution is defined by $[1, \cdots, r]$ with the software pipelining cycling vector defined by $[r, 1, \cdots, r-1]^T$. A software pipelining cycling vector will result in a single processor executing all of a FSFG's nodes for a single set of inputs. Subsequent sets of inputs are processed by adjacent processors by skewing the start times for execution of the FSFG. Thus, all processors execute the same operations on different data sets. Parallel processing cycling vectors schedule a portion of the FSFG to be executed on a processor. Each processor executes a different portion of the FSFG and every processor will execute their portion of the FSFG for each data set.

OGM's partitioning stage refers the cycling vectors and the period matrix to the validation and scheduling stages of OGM to check for communication requirements and for processor optimality. The validation stage incorporates the communication and network topology admissibility conditions while the scheduling stage determines processor optimality. Successful completion of the validation and scheduling stages produces a valid multiprocessor schedule for the FSFG. Failure in either stage results in the partitioning stage generating another candidate partitioning. We now describe the validation and scheduling stages.

3.4 Validating and Scheduling Steps - Incorporating Communication

3.4.1 Communication Concerns

Successful scheduling of the period matrix may require more than simply meeting the precedence and non-preemption conditions. Frequently, communication processing and contention in the communication network may result in an iteration period where communication actually takes longer than computation. This results in the need to schedule an iteration period with enough time to allow communication, as well as computation, to be completed. In addition to the communication time, two other communication concerns need to be addressed. The first is the effect of the network topology. A fully connected topology provides the most flexibility but is expensive to build. The second concern is contention occurring in the links of the interconnection network and within the processors' memory. If two processors try to communicate with one processor, P_i, simultaneously, processor P_i may not have enough resources to allow memory access to both communicating processors. Additionally, if the two communication signals attempt to use the same physical link in the network at the same instant in time, a contention will result within the network. In order to absolutely deal with the effects of communication, a communication admissibility condition and a network topology admissibility condition are incorporated into OGM.

3.4.2 Modeling Contention Due To Communication

The communication admissibility condition is introduced using the same communications assumptions presented in Section 3.4.1. The goal is to ascertain whether there is enough time for each processor to perform the necessary communications within

an iteration period. Using the cycling vector and period matrix provided by the partitioning stage of OGM, the validation stage inspects the potential parallel execution and software pipelining schedules for communication requirements. This process can be complicated by the assignment of multiple partitions to a single processor (by using other permutations of the cycling vector) to form a combination of a parallel execution and software pipelining schedule. In this section, we will focus on the two potential schedules - a strictly parallel schedule and a fully pipelining schedule. Details for analyzing communication for schedules which combine the two can be found in [57].

Strictly Parallel Schedules

We begin by forming subgraphs f_i within G' which contain the nodes to be executed on processor i. The nodes for subgraph f_i are easily identified from either row i of the period matrix or from the set F_i. To represent system input/output devices, an additional node is added to G'. f_{r+1} is a subgraph containing only this node. Edges within G' which represent system inputs/outputs will originate/terminate to this node. An edge in graph G' from subgraph f_i to subgraph f_j indicates data which must be communicated from one processor to another if f_i and f_j are scheduled on different processors. For a strictly parallel schedule, the required communications are identified from the subgraphs f_i. To accomplish this, we temporarily (only while checking communications for parallel execution) remove duplicate edges which originate from a node n in f_i and terminate to the same subgraph f_j. The purpose of removing these edges is to prevent the validation process from considering communications due to fan-out from a node n to multiple nodes on the same processor. We denote the indegree of subgraph f_i as I_i, and the outdegree as O_i. If an atomic node is split between subgraphs, a communication occurs between the subgraphs, and I_i or O_i is incremented accordingly.

For example, a graph G' with $V' = \{1, 2, 3\}$ and $E' = \{1 \rightarrow 2, 2 \rightarrow 3\}$ has a linear extension $[1, 2, 3]$. If nodes 1 and 3 have a weight of 1 and node 2 has a weight of 2, a partition with $T_d = 2$ is $F_1 = \{1, 2\}$ and $F_2 = \{2, 3\}$. Assume that node 1 receives a system input and node 3 produces a system output. The indegree and outdegree of the subgraphs f_1 and f_2 are $I_1 = 1$, $O_1 = 1$, $I_2 = 1$, and $O_2 = 1$.

The total number of communications into and out of processor i is defined by

$$C_i = I_i + O_i \tag{3.1}$$

where C_i represent the number of inter-processor communications for processor i (including communications from/to input/output devices). This allows us to check whether the communication for processor i can take place in the time available (T_d). Denote t_c as the maximum number of communications for a processor during a single clock cycle. Any communication overhead associated with initiating the data transfer should be incorporated into the value of t_c. The admissibility condition for communication is then

$$\frac{C_i}{t_c} \leq T_d \qquad \forall\, i : 1 \leq i \leq r \tag{3.2}$$

If this admissibility condition holds, there is enough time during the iteration period T_d to complete the necessary communications. For a strictly parallel schedule, this applies for both synchronous and asynchronous systems.

Example 3.4.1 *Assuming $T_d = 3$, the partitioning stage of OGM may produce the following F_i partitions for Figure 2.13:*

$$F_1 = \{3, 1\} \quad F_2 = \{4, 2\} \quad F_3 = \{7, 8\} \quad F_4 = \{8, 9, 10\}$$

The validation stage of OGM receives these partitions in form of a period matrix and cycling vector. Figure 3.3 identifies the cycling vector and period matrix for parallel execution. For a parallel execution schedule, the indegree and outdegree of the

$$\underset{(a)}{\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}} \qquad \underset{(b)}{\begin{bmatrix} 3 & 3 & 1 \\ 4 & 4 & 2 \\ 7 & 7 & 8 \\ 8 & 9 & 10 \end{bmatrix}}$$

Figure 3.2: (a) Cycling vector for parallel execution (b) Period matrix

subgraphs f_i are

$$I_1 = 2 \quad I_2 = 1 \quad I_3 = 1 \quad I_4 = 3$$
$$O_1 = 1 \quad O_2 = 3 \quad O_3 = 2 \quad O_4 = 1$$

Thus, the total communication requirements for each processor is equal to

$$C_1 = 3 \quad C_2 = 4 \quad C_3 = 3 \quad C_4 = 4$$

Assuming $t_c = 1$ and $T_d = 3$,

$$T_d = 3 \geq \frac{C_1}{1} = \frac{3}{1} \quad T_d = 3 < \frac{C_2}{1} = \frac{4}{1}$$
$$T_d = 3 \geq \frac{C_3}{1} = \frac{3}{1} \quad T_d = 3 < \frac{C_4}{1} = \frac{4}{1}$$

the parallel execution schedule fails the communication admissibility condition for processors 2 and 4. For the assumption $t_c = 2$ with $T_d = 3$,

$$T_d = 3 \geq \frac{C_1}{2} = \frac{3}{2} \quad T_d = 3 \geq \frac{C_2}{2} = \frac{4}{2}$$
$$T_d = 3 \geq \frac{C_3}{2} = \frac{3}{2} \quad T_d = 3 \geq \frac{C_4}{2} = \frac{4}{2}$$

and the parallel execution schedule passes the communication admissibility condition.

The connection network has the potential to reduce or improve the effectiveness of communication between processors. For each communication path shared, the potential for contention on a particular link increases. In addition, communication between processors may require a number of hops (intermediate processor communications) through the connection network; thereby, increasing the time for processor to processor communication (as compared to a fully connected network). The goal

to understanding network topology is to identify communication requirements due to the physical network topology and to determine link utilization.

In order to consider the routing path used, the structure of the network, and the contention on the network, we focus our attention on topologies which contain defined point to point communications. For a communication from processor i to processor j, we assume that there one unique routing path, such that the set of processors involved in the communication is known. One such topology is the ring topology. Because there is a defined path between each pair of processors, we can compute exactly how much communication exists on a link and ensure that there is enough time for the communication to occur.

For parallel execution, we consider all the communication from a processor P_i for node n, and identify the links used to support this communication. We begin by placing the processors which contain the output nodes for node n in f_i into a set $O_i(n)$. The set $O_i(n)$ represents the required inter-processor communication for node n. We denote a member of $O_i(n)$ by a set $o_i(n)$ with a single member. For each $o_i(n)$, communication from P_i to $P_{o_i(n)}$ may involve several links in the network topology. We represent a link in the network topology from processor i to processor j as L_i^j, and identify the set of links used for communication from P_i to $P_{o_i(n)}$ by $l_{o_i(n)}$. The set of links used to support communication due to $O_i(n)$ equals

$$l_{O_i(n)} = \bigcup_{\forall o_i(n) \in O_i(n)} l_{o_i(n)} \tag{3.3}$$

Communications due to members of $O_i(n)$ for each node n in f_i and each processor (P_i) must be considered to identify all of the communication over the network topology. Thus, communications are determined for all L_i^j by adding one communication to each link in We now introduce a variable t_n (similar to t_c) to represent the number of communications a single link on the network topology can handle per time unit.

Overhead for data transfers in the network topology are incorporated into the value of t_n. Using t_n, the network topology admissibility condition is

$$\frac{L_i^j}{t_n} \leq T_d \qquad \forall\, i : 1 \leq i \leq r \tag{3.4}$$

If this condition holds, there is enough time during the iteration period T_d to allow the network topology to complete the necessary communications for parallel execution on both synchronous and asynchronous systems.

Changes in the amount of inter-processor communications, affect the inter-processor communication used in the communication admissibility condition. For parallel processing and software pipelining, the number of input inter-processor communications for P_i on both synchronous and asynchronous systems becomes

$$I_i = L_{i-1}^j \qquad \forall\, j \tag{3.5}$$

The number of inter-processor communications resulting from outputs for P_i is now

$$O_i = L_i^j \qquad \forall\, j \tag{3.6}$$

Thus, the total communications for P_i for parallel execution becomes

$$C_i = I_i + O_i + X_i \tag{3.7}$$

with the admissibility conditions remaining unchanged.

In addition to a network topology admissibility condition, the network topology requires modification of the linear extension partitioning strategy. Taking advantage of a network topology which allows system input/output to any processor and communications between processor P_1 and P_r, the partitioning of the linear extension may be shifted to allow processing of the first and last nodes in the linear extension on the same processor. Thus, consideration of the network topology will provide several different mappings for each linear extension and increase the number of potential

mappings from the FSFG to a multiprocessor schedule. Modifying the partitioning algorithm, the partitioning of each linear extension may begin with any v_j as long as

$$T_d < \sum_{i=1}^{j-1} w_j$$

This condition ensures that duplicate partitionings are not considered.

Example 3.4.2 *For the purposes of this example, we assume a unidirectional ring topology with links occurring from P_i to $P_{i+1 \bmod r}$. Incorporating this network topology into Example 3.4.1, the link utilization for a strictly parallel schedule is found beginning with the identification of communications due to node n.*

$$O_1(1) = \{2\} \quad O_2(2) = \{1,3,4\} \quad O_3(7) = \{4\} \quad O_3(8) = \{4\}$$

The links supporting communication for each $O_i(n)$ is defined by

$$l_{O_1(1)} = \{L_1^2\} \quad l_{O_2(2)} = \{L_2^3, L_3^4, L_4^1\} \quad l_{O_3(7)} = \{L_3^4\} \quad l_{O_3(8)} = \{L_3^4\}$$

Calculating the number of communications on each L_i^j link results in

$$L_1^2 = 1 \quad L_2^3 = 1 \quad L_3^4 = 3 \quad L_4^1 = 1$$

Assuming $t_n = 1$ with $T_d = 3$,

$$\begin{array}{cc} T_d = 3 \geq \frac{L_1^2}{1} = \frac{1}{1} & T_d = 3 \geq \frac{L_2^3}{1} = \frac{1}{1} \\ T_d = 3 \geq \frac{L_3^4}{1} = \frac{3}{1} & T_d = 3 \geq \frac{L_4^1}{1} = \frac{1}{1} \end{array}$$

Adjusting the inter-processor communication for network topology

$$\begin{array}{cccc} I_1 = 1 & I_2 = 1 & I_7 = 1 & I_3 = 3 \\ O_1 = 1 & O_2 = 1 & O_7 = 3 & O_3 = 1 \end{array}$$

With communications due to input/output devices remaining the same, the total required communication (with consideration for the network topology) is

$$C_1 = 3 \quad C_2 = 2 \quad C_3 = 4 \quad C_4 = 5$$

Assuming $t_c = 1$ *with* $T_d = 3$,

$$T_d = 3 \geq \frac{C_1}{1} = \frac{3}{1} \quad T_d = 3 \geq \frac{C_2}{1} = \frac{2}{1}$$
$$T_d = 3 < \frac{C_3}{1} = \frac{4}{1} \quad T_d = 3 < \frac{C_4}{1} = \frac{5}{1}$$

and the communication admissibility condition fails for processors 3 *and* 4. *Assuming* $t_c = 2$ *with* $T_d = 3$,

$$T_d = 3 \geq \frac{C_1}{2} = \frac{3}{2} \quad T_d = 3 \geq \frac{C_2}{2} = \frac{2}{2}$$
$$T_d = 3 \geq \frac{C_3}{2} = \frac{4}{2} \quad T_d = 3 \geq \frac{C_4}{2} = \frac{5}{2}$$

Therefore, the parallel execution schedule is not valid for $t_c = 1$, $t_n = 1$, *and* $T_d = 3$; *however, it is admissible if* $t_c = 2$, $t_n = 2$, *and* $T_d = 3$.

Software Pipelining Schedules

A completely software pipelined schedule requires additional consideration of delay nodes. For this type of schedule, delay nodes determine inter-processor communication. We identify m as an output node for node n. For each node n in f_i, we define $n_{n,m}$ as the number of ideal delays between node n and node m. The ideal delays which result in inter-processor communication for node n are placed in a set $D_i(n)$ such that

$$D_i(n) = \left\{ \bigcup \{ n_{n,m} \ \mathbf{mod}\ r \} \right\} - \{0\} \qquad \forall\, m : m \in M_n \tag{3.8}$$

where M_n is the set of all n's output nodes. In addition, we represent the number of communications between f_i and f_{r+1} by X_i. Thus, X_i contains the number of communications from P_i to input/output devices. For software pipelining, the total number of communications into and out of each processor can now be determined by

$$C_1 = 2 \sum_{i=1}^{r} \sum_{n \in f_i} |D_i(n)| + \sum_{i=i}^{r} X_i \tag{3.9}$$

with each processor having C_1 communications. For software pipelining, communications may occur over $T_d P_d$ clock cycles. Thus, we define a communication admissibility

condition for software pipelining as

$$\frac{C_1}{t_c} \leq T_d P_d \tag{3.10}$$

If this admissibility condition holds, there is enough time to complete the necessary communications on either a synchronous or an asynchronous systems.

Example 3.4.3 *Assuming $T_d = 3$, the partitioning stage of OGM may produce the following F_i partitions for Figure 2.13:*

$$F_1 = \{3, 1\} \quad F_2 = \{4, 2\} \quad F_3 = \{7, 8\} \quad F_4 = \{8, 9, 10\}$$

The validation stage of OGM receives these partitions in form of a period matrix and cycling vector. Figure 3.3 identifies the cycling vector and period matrix for parallel execution. For a software pipelining schedule, the number of ideal delays between

$$\underset{(a)}{\begin{bmatrix} 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}} \qquad \underset{(b)}{\begin{bmatrix} 3 & 3 & 1 \\ 4 & 4 & 2 \\ 7 & 7 & 8 \\ 8 & 9 & 10 \end{bmatrix}}$$

Figure 3.3: (a) Cycling vector for software pipelining (b) Period matrix

individual nodes is identified as

$$n_{2,3} = 2 \quad n_{2,4} = 1 \quad n_{2,7} = 2 \quad n_{2,8} = 1$$

with

$$M_2 = \{3, 4, 7, 8, 10\}$$

The delay nodes which have an effect on inter-processor communication are placed in a set

$$D_2(2) = \{1, 2\}$$

and the number of communications to input output devices is

$$X_1 = 1 \quad X_4 = 1$$

For software pipelining, the total number of communications into and out of each processor can now be determined as

$$C_1 = 2(2) + 2 = 6$$

Assuming $t_c = 1$ *with* $T_d = 3$, P_d *will be* 4 *and*

$$T_d P_d = 12 \geq \frac{C_1}{1} = \frac{6}{1}$$

Therefore, the software pipelining schedule passes the communication admissibility condition. If we assume $t_c = 2$ *with* $T_d = 3$,

$$T_d P_d = 12 \geq \frac{C_1}{2} = \frac{6}{2}$$

and the software pipelining schedule passes the communication admissibility condition.

For software pipelining, validating the link utilization in the network topology is a matter of evaluating the delays for r iteration periods. We denote a member of $D_i(n)$ by a set $d_i(n)$ with a single member. For each $d_i(n)$, communication occurs from P_i to $P_{i+d_i(n) \bmod r}$ and may involved several links in the network topology. We identify the set of links used for this communication from as $l_{d_i(n)}$. The set of links used to support communication due to $D_i(n)$ equals

$$l_{D_i(n)} = \bigcup_{\forall d_i(n) \in D_i(n)} l_{d_i(n)} \tag{3.11}$$

Communications due to members of $D_i(n)$ for each node n in f_i and each processor (P_i) must be considered for r iteration periods. Thus, communications are determined

for all L_i^j by adding one communication to each link in $l_{D_i(n)}$ $\forall\, n \in f_i$, $\forall\, i : 1 \leq i \leq r$ for r iteration periods. The network topology admissibility condition for software pipelining is

$$\frac{L_i^j}{t_n} \leq T_d P_d \qquad \forall\, i : 1 \leq i \leq r \tag{3.12}$$

If this condition holds, there is enough time to allow the network topology to complete the necessary communications for software pipelining on both synchronous and asynchronous systems.

Adjusting the communication admissibility condition for the changes in the inter-processor communication, the number of input inter-processor communications for P_i on both synchronous and asynchronous systems becomes

$$I_i = L_{i-1}^j \qquad \forall\, j \tag{3.13}$$

The number of output inter-processor communication for P_i is now

$$O_i = L_i^j \qquad \forall\, j \tag{3.14}$$

The total communications for P_i using software pipelining becomes

$$C_i = I_i + O_i + X_i \tag{3.15}$$

with the admissibility conditions remaining unchanged.

Example 3.4.4 *We incorporate the unidirectional ring topology into Example 3.4.3 by reversing the direction of communication to better support the pipelined schedule. The link utilization for a software pipelining schedule is found beginning with the identification of communications due to node n. For the first iteration period,*

$$l_{D_2(2)} = \{L_2^1, L_1^4\}$$

The second, third, and fourth iteration periods produce communications over the links in

$$l_{D_1(2)} = \{L_1^4, L_4^3\} \quad l_{D_4(2)} = \{L_4^3, L_3^2\} \quad l_{D_3(2)} = \{L_3^2, L_2^1\}$$

The number of communications on a link in the ring topology is defined by

$$L_1^2 = 2 \quad L_2^3 = 2 \quad L_3^4 = 2 \quad L_4^1 = 2$$

Assuming $t_n = 1$ *with* $T_d = 3$,

$$\begin{array}{llll} T_d = 3 \geq \frac{L_1^2}{1} = \frac{2}{1} & T_d = 3 \geq \frac{L_2^3}{1} = \frac{2}{1} \\ T_d = 3 \geq \frac{L_3^4}{1} = \frac{2}{1} & T_d = 3 \geq \frac{L_4^1}{1} = \frac{2}{1} \end{array}$$

Adjusting the inter-processor communication for software pipelining using the ring topology

$$\begin{array}{llll} I_1 = 2 & I_2 = 2 & I_3 = 2 & I_4 = 2 \\ O_1 = 2 & O_2 = 2 & O_3 = 2 & O_4 = 2 \end{array}$$

With communications due to input/output devices remaining the same, the total required communication (with consideration for the network topology) is

$$C_1 = 6 \quad C_2 = 6 \quad C_3 = 6 \quad C_4 = 6$$

Assuming $t_c = 1$ *with* $T_d = 3$,

$$\begin{array}{llll} T_d P_d = 12 \geq \frac{C_1}{1} = \frac{6}{1} & T_d P_d = 12 \geq \frac{C_2}{1} = \frac{6}{1} \\ T_d P_d = 12 \geq \frac{C_3}{1} = \frac{6}{1} & T_d P_d = 12 \geq \frac{C_4}{1} = \frac{6}{1} \end{array}$$

Assuming $t_c = 2$ *with* $T_d = 3$,

$$\begin{array}{llll} T_d P_d = 12 \geq \frac{C_1}{2} = \frac{6}{2} & T_d P_d = 12 \geq \frac{C_2}{2} = \frac{6}{2} \\ T_d P_d = 12 \geq \frac{C_3}{2} = \frac{6}{2} & T_d P_d = 12 \geq \frac{C_4}{2} = \frac{6}{2} \end{array}$$

Therefore, the software pipelining schedule is valid for $t_n = 1$, *and* $T_d = 3$ *with* $t_c = 1$ *or* $t_c = 2$.

In this section we have increased the flexibility and effectiveness of OGM by incorporating the communication admissibility condition, the network topology admissibility condition, and the effects of network topology on the partitioning strategy. We now summarize the characteristics of OGM which lead to an effective partitioning and scheduling process.

3.5 Contributions of the Order Graph Method

The order graph method currently supports the automatic partitioning of FSFGs for fine-grained parallelism exploiting parallelism during processing of a single sample and between samples. FSFGs are used as a convenient and familiar method for OGM algorithm specification without requiring users to explicitly define available parallelism. OGM considers interprocessor communication and input/output bandwidths during the scheduling process. In addition, support is provided for ring topologies such as the BDPA. OGM allows users to optimize processing time and/or the number of processors for a given FSFG. Similar to CRM and the RGM, scalability is limited by the acceptable size for a non-hierarchical FSFG. However, the introduction of hierarchy to OGM in Chapter 4 extends the scalability for OGM. Implemented in C, OGM is practical for implementation on multiple parallel architectures (although support for individual network topologies must be developed). Running time has been acceptable for FSFGs of up to 2624 nodes with schedules generated for both parallel and software pipelined implementations.

Considering the diversity of methodologies and the complexity of the task scheduling problem (NP-complete), benchmarks should be used to compare different methodologies. Unfortunately, no accepted standardized benchmarks have been established which cover the diverse methods. In the absence of such benchmarks, we have evaluated the two methods closest to OGM using several criteria. Table 3.2 presents a

comparison of OGM with respect to other key graph-based mapping methodologies highlighting the advantages of OGM. In the table, + indicates that there is some formal support for consideration of communication. In addition, $\sim$ indicates that there is the ability to support communication; however, the support has not been presented formally.

Characteristic	*RGM*	*CRM*	*OGM*
FSFG Specification	Y	Y	Y
Parallelism Not Explicitly Defined	Y	Y	Y
Scalable To Large Problems			Y
Level Of Parallelism	fine-grained	fine-grained	fine-grained to coarse-grained
Parallelism Processing Single Sample	Y	Y	Y
Parallelism Processing Multiple Samples	Y	Y	Y
Considers Interprocessor Communication	+	$\sim$	Y
Considers Network Topology			Y
Considers Input/Output Bandwidth			Y
Set T Optimize P	Y		Y
Set P Optimize T	Y		Y
Optimize T and P	Y	Y	Y
Automated Schedule Generation	Y	Y	Y
Suitable For BDPA	Y	Y	Y
Acceptable Running Time	Y (45 nodes)	unpublished [2]	Y (2624 nodes)
Software Pipelining Support		Y	Y

Table 3.2: Comparison of RGM, CRM, and OGM

We now extend the Order Graph Method to add support for hierarchical FSFGs. This permits OGM to be scalable to larger problems while reducing the running time for algorithms specified using hierarchy.

[2] Recent research efforts have developed a multiprocessor integer programming based optimal compiler (MULTIPROC). Developed from some basic concepts in CRM, MULTIPROC runs on an IBM RS/6000 Model 560 and has scheduled FSFGs with less than 48 nodes. Published run times average approximately 27 minutes; however, one runtime for a FSFG with 33 nodes required nearly 10 hours. [65]

Chapter 4

Hierarchical FSFGs

4.1 Motivation

This research presents the Order Graph Method as a new methodology for task scheduling on parallel processing systems. OGM has been implemented in a tool called "Calypso" to allow experimentation with parallel signal processing. To enhance OGM and Calypso, we have added support for hierarchical FSFGs. The need for hierarchical FSFGs is immediately apparent when attempting to specify the flow graph for a 2-D 2nd order IIR filter of a 512x512 image. Assuming that each row of the image serves as an input to the flow graph, the complete FSFG would contain a total of 20,990 nodes. If a single pixel is used as an input, the example flowgraph would contain 10,743,808 nodes! Using a flattened flow graph is not a realistic solution since the graph would have an excessive number of nodes. A flattened graph would make the FSFG impossible to visualize and result in unrealistic scheduling times.

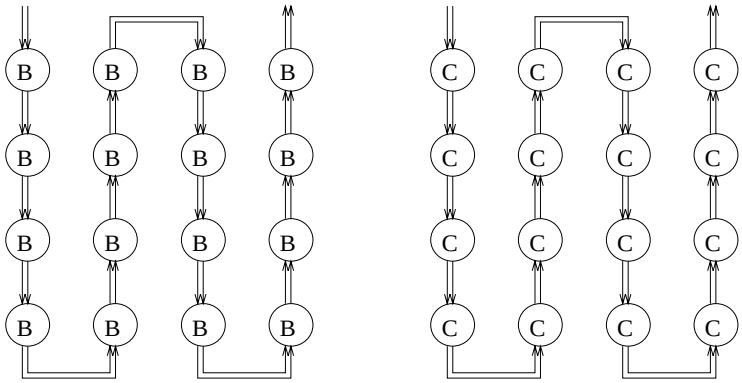
Practical experience has shown that a majority of applications use hierarchical constructs which can be translated directly to FSFGs. To support this translation, the definition of an algorithm's hierarchical FSFG must include nodes which can be used at any level of the hierarchy. Nodes may be simple nodes which identify the execution of a single operation such as multiplication. They may also be composite nodes which represent the execution of another level of the hierarchy (i.e. another

FSFG). Thus, a FSFG at any particular level l of the hierarchy may be limited to a reasonable number of nodes by introducing a series of composite nodes. These composite nodes are defined by another FSFG at level $l-1$ in the hierarchy. For example, the FSFG for a 2-D 2nd order IIR filter on a 256x256 image (assuming each row is an input) may be represented by the FSFGs in Figure 4.1. In FSFG - A, each of the nodes represents the execution of FSFG - B. Edges in FSFG - A represent the input and output communications for FSFG - B. Similarly, nodes in FSFG - B identify the execution of FSFG - C, and edges identify the communication due to FSFG - C.

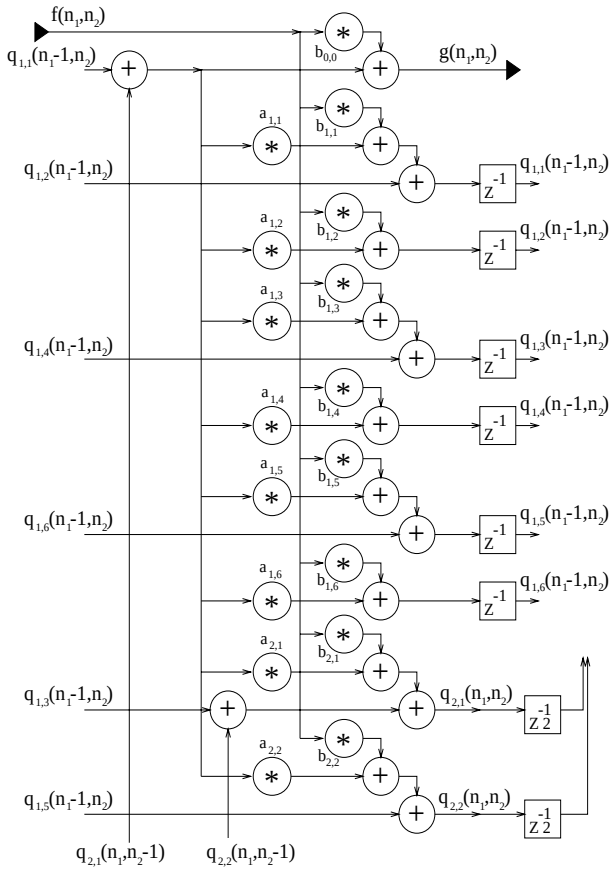
4.2 Hierarchical Specification

We begin the introduction of hierarchical FSFGs to OGM by defining a method for hierarchical specification of an algorithm. This technique must address several issues which determine the limitations for an algorithm schedule:

- An algorithm specification technique must include a method for defining composite nodes which identify the execution of another FSFG.
- When a composite node is used at level l in the hierarchy, the technique must include a method for defining a composite node's effect at level $l-1$ (i.e. what happens inside the composite node). Hierarchical specification must enable OGM to recognize feedback loops, input/output paths, and the total amount of computation within a composite node.
- The specification technique must enable OGM to identify input/output communications for a composite node at level l and $l-1$ in the hierarchy. At level l in the hierarchy, precedence conditions must be established using the communications patterns of composite nodes. In addition, inputs to a composite node at level l must be tied to the inputs of the FSFG which define the



FSFG - A **FSFG - B**



FSFG - C

Figure 4.1: Hierarchical FSFG for a 2-D 2nd order IIR of a 256x256 image using each row as an input

composite node at level $l-1$. Similarly, the outputs from a composite node at level l must be identified with the outputs from the composite node's defining FSFG at level $l-1$. This guarantees that potential feedback loops which span multiple levels of a hierarchical FSFG do not go unrecognized.

- Inputs and outputs between levels of the hierarchy must be distinguished from inputs and outputs to system I/O devices.

These issues are identified in the hierarchical FSFG presented in Figure 4.2. While

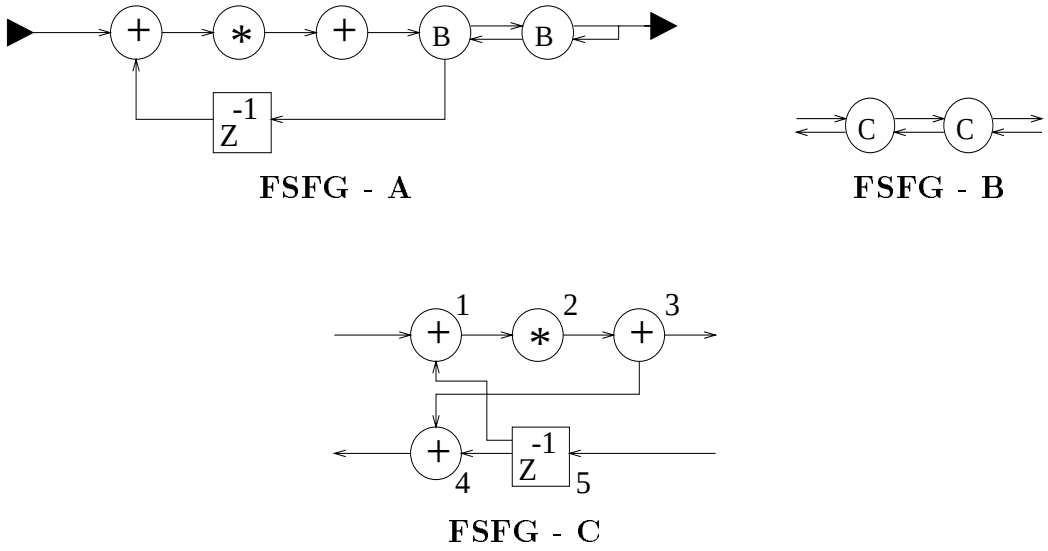


Figure 4.2: Hierarchical FSFG for a 3rd order lattice filter

FSFG - C does not have any internal feedback loops, the paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $5 \rightarrow 1 \rightarrow 2 \rightarrow 3$ combine to complete the feedback loop in FSFG - B. It should be noted that the reverse is not always true. An apparent loop at level l in the hierarchy does not necessarily mean that a feedback loop exists. Edges in level $l-1$ of the hierarchy may break the loop (See Figure 4.3). Thus, there is the potential for a composite node at level l to contain an input/output path at level $l-1$ which forms part of a feedback loop at level l. In this case, all potential input/output paths for a composite node's definition at level $l-1$ must be identified and tied to

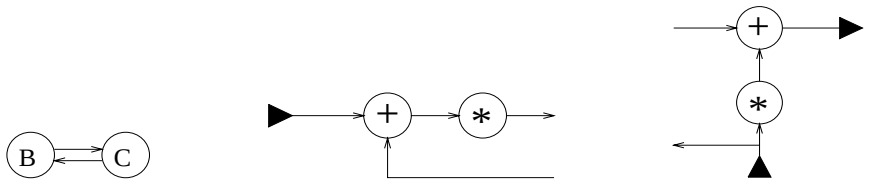


Figure 4.3: Example Hierarchical FSFG

the communication edges at level l where the composite node is used. Thus, the specification technique must relate the two input and two output communications in FSFG - C of Figure 4.2 to the composite node's communications in FSFG - B. In addition, consideration for communication to system I/O devices must distinguish the output $g(n_1, n_2)$ (a system output) from the output $q_{2,1}(n_1, n_2)$ (an output to the next level of the hierarchy) in FSFG - C of Figure 4.1.

The Order Graph Method incorporates a method for hierarchical algorithm specification which maintains the inherent flexibility of a hierarchy. We define OGM's hierarchical specification technique as follows:

FSFG	:=	edge
edge	:=	fromnode tonode data=int$_{opt}$
fromnode	:=	nodename instance | innode instance fsfgname instance
tonode	:=	nodename instance | outnode instance fsfgname instance
int	:=	**0**$\cdots$ **32,767**
nodename	:=	[**a**$\cdots$ **z**]* | **INPUT** | **OUTPUT**
innode	:=	**input**
outnode	:=	**output**
fsfgname	:=	[**a**$\cdots$ **z**]*
instance	:=	int

For example, the hierarchical 2nd order IIR filter shown in Figure 4.4 may be defined by the input files in Figure 4.5.

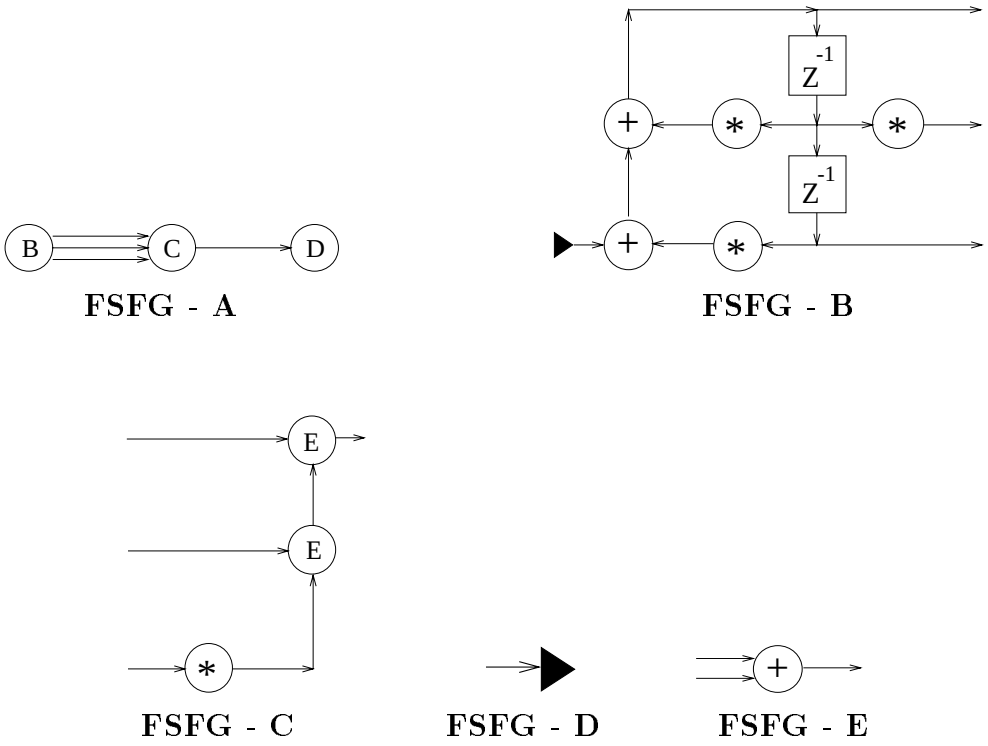


Figure 4.4: Potential FSFG hierarchical specification for the 2nd order IIR filter (IIR2)

4.3 Modifications for Scheduling

4.3.1 Preprocessing

We establish a methodology which details the handling of composite nodes to effectively accomplish the preprocessing stage of OGM. During preprocessing, OGM defines the iteration period bound (T_v), identifies feedback loops, and reduces the FSFG to a DAG with node weights (w_k) less than the desired iteration period (T_d). Incorporating composite nodes into these processing steps requires OGM to expand some composite nodes into their internal nodes (i.e. flatten a portion of the FSFG). Unnecessarily expanding composite nodes slows the preprocessing stage by increasing the number of nodes and edges considered by strong component, depth-first search, and loop finding algorithms. Therefore, conditions which determine when the composite nodes are expanded include consideration for the following:

```
filename:  hiir2
================================
output1 hiir2a.1 input3 hiir2b.1
output2 hiir2a.1 input1 hiir2b.1
output3 hiir2a.1 input2 hiir2b.1
output1 hiir2b.1 input1 hiir2c.1

filename:  hiir2a
================================
INPUT1 add1
add1 add2
add2 output1 delay1
multiply1 add1 data=2
multiply2 data=3 add2
delay1 multiply2 multiply3 data=1 delay2
delay2 data=1 output3 multiply1
multiply3 output2

filename:  hiir2b
================================
input3 input2 sum.2 data=5
input1 input2 sum.1
input2 multiply1
output1 sum.1 input1 sum.2
multiply1 input1 sum.1
output1 sum.2 output1

filename:  hiir2c
================================
input1 OUTPUT1

filename:  sum
================================
input1 add1
input2 add1
add1 output1
```

Figure 4.5: Sample input files for the hierarchical specification of a 2nd order IIR filter (IIR2)

- Is the computational delay of the composite node known? The computational delay for the node must be identified for comparison to the desired iteration period and the calculation of strong component weights.

- Is the composite node involved in a feedback loops? This must be determined at the level of the hierarchy which uses an instance of the composite node and at the level of the hierarchy which defines the composite node.

- Is the desired iteration period less than the computational time of the composite node? If so, the node must be expanded to permit partitioning of the DAG.

- Is the composite node a member of a strong component which must be expanded to reduce the strong component's weight less than the desired iteration period? When expanding a strong component, composite nodes within a strong component should be expanded prior to removing an edge in the strong component.

- Has the composite node previously been expanded? If so, the required information should already be available without requiring further analysis of the composite node. If the composite node is not expanded, data should be available for each instance of the composite node without repetitive analysis.

For example, FSFG - C in Figure 4.2 should only be expanded once when analyzing the potential feedback loop in FSFG - B. Under these conditions composite nodes will only be expanded when absolutely necessary allowing fine-grained parallelism to be exploited with a minimum amount of processing time.

OGM determines a composite node's total computational weight and associates the weight with each instance of the composite node. During the process of reading a hierarchical FSFG's nodes into Calypso, OGM determines the total computational time for each FSFG. Once all of the FSFG nodes have been input, OGM establishes the edges within each level of the hierarchy. While assigning edges to an instance of a composite node, the total computational time is also assigned. As long as the composite node remains unexpanded, OGM will reference this total weight. Once expanded, OGM will reference the weights of the composite node's internal nodes.

Composite nodes will effect the determination of feedback loops on two levels of the hierarchy: the level which uses an instance of the composite node and the level

which defines the composite node. We begin by discussing feedback loops which incorporate a composite node. Assume a level l of the hierarchy uses an instance of a composite node within a feedback loop. OGM uses the input/output paths within the composite node to determine if the feedback loop truly exists. For each composite node, all input/output paths to other levels of the hierarchy are identified by performing depth-first searches from intermediate input nodes. OGM executes these depth-first searches immediately after defining the FSFG's edges. It should be noted that the process of defining FSFG edges and input/output paths begins with the lowest level of the hierarchy and ends with the highest level of the hierarchy. By doing this, OGM is able to handle composite nodes which are used within a potential input/output path of another composite node (i.e. composite nodes within composite nodes). Using the composite node's input/output paths at level $l-1$, OGM checks for a connection between the input and output nodes at level l. If a path exists between the input and output nodes on level l of the hierarchy, the feedback loop is considered valid. If not, the potential feedback loop is considered to be invalid. For the hierarchical FSFG in Figure 4.2, the input/output paths for FSFG - B would validate the feedback loop in FSFG - A. While, the input/output paths for FSFG - B and FSFG - C in Figure 4.3 would invalidate the potential feedback loop in FSFG - A.

A second way composite nodes may be involved with feedback loops is if the iteration period bound must be identified to produce a rate optimal schedule. Allowing embedded delays in the hierarchy permits feedback loops to be embedded within composite nodes. In order to identify the iteration period bound for the digraph, any feedback loops which exist within the digraph's composite nodes must be identified. This will only occur once in the preprocessing stage and prior to the reduction of the digraph to a DAG. For a composite node which has not been expanded internal loops are identified using a modified version of Johnson's algorithm [60]. Because a composite node can occur within a composite node, this may involve analyzing composite

nodes on multiple levels of the hierarchy. As before, a composite node (y) within a composite node (x) is not expanded to evaluate potential loops within x. Instead, input/output paths for y are used to validate potential loops within x. In addition, OGM will evaluate a composite node's internal loops only once and will not perform a loop search for each instance of the composite node. Using the loops from both expanded and unexpanded composite nodes, the iteration period bound T_v may be found using Equation 2.2.

Using the feedback information, preprocessing identifies strong components and reduces the digraph to a DAG. With hierarchical specification, the DAG can consist of atomic nodes, composite nodes which identify a strong component, and composite nodes from the original digraph (which do not identify a strong component). In keeping with the goals of the preprocessing stage, composite nodes with a weight greater than the desired iteration period must be expanded. Expanding composite nodes which do not identify a strong component is handled by expanding the DAG to include the internal nodes of the composite node. Depending on the number of levels of the hierarchy within the composite node, this process may need to be repeated recursively.

Without hierarchical specification, a composite node which identifies a strong component is expanded by considering internal nodes, identifying loops, removing an edge from a loop, and collapsing the remaining strong component back into a composite node. To incorporate hierarchy, OGM must determine if a member of the strong component is a composite node. Before removing an edge from the strong component, any composite nodes within the strong component must be expanded. The purpose of this step is to ensure that edges of the graph are not unnecessarily removed. Composite nodes within a strong component may contain internal nodes which are not members of the strong component. Because the composite node is not expanded, these nodes add to the weight of the strong component. By expanding the

composite node, the strong component will contain the minimum number of nodes and will have the minimum weight. Therefore, if composite nodes are found within a strong component, one of the composite nodes is expanded and the remaining strong component is reduced back into a composite node. A strong component which does not contain composite nodes is still expanded by removing an edge from the loop with the lowest iteration period.

During preprocessing, OGM prevents the analysis of a composite node more than once by maintaining data for each composite node and recognizing multiple occurrences of the node. OGM evaluates an instance of a composite node by determining if the data is available from another instance of the composite node. The absence of the necessary data from other instances indicates that the composite node has not been evaluated. For example, identification of input/output paths for a composite node is done once and re-used for each instance of the composite node. In addition, OGM reduces the requirement for composite node analysis by determining a composite node's representative data during the loading of the FSFG. Details such as the computational delay for a composite node are determined during the loading of individual nodes within the composite node.

4.3.2 Partitioning

During the partitioning stage, OGM develops a set of candidate partitionings from the DAG provided by the preprocessing stage. This is accomplished through the use of a relations matrix to generate linear extensions which produce candidate partitionings. Entries in the relations matrix are representative of data dependencies between nodes in the DAG. Because nodes in the DAG can consist of composite nodes, embedded delays will play a key role in determination of data dependencies for the relations matrix. If a delay exists between node x and node y, then there is no data dependency in the relations matrix. The absence of a delay results in the entry of xRy (where x

is related to y) in the relations matrix. Thus, it is necessary to determine if outputs from a composite node originate at a delay or from some other element. If they originate from a delay, there is not a data dependency on the composite node. Similarly, input communications which terminate at a composite node are identified as terminating at a delay or some other element. Consider FSFG C in Figure 4.1, all output communications originate at delay elements and do not result in data dependencies across composite nodes in FSFG B.

Finally, OGM incorporates a methodology for dealing with the partitioning inherently defined by the hierarchy. Using a hierarchy, different levels of parallelism are available (fine to coarse-grained parallelism). OGM deals with the inherent partitioning by allowing the desired iteration period to determine the required level of parallelism. If the desired iteration period is greater than the composite nodes' computational time, coarse-grained parallelism is exploited and the composite nodes are not expanded. In this case, the initial definition of the composite nodes will essentially determine the operations which are performed on an individual processor. It is possible that the user has not developed the hierarchical FSFGs with an effective partitioning in mind. Thus, the coarse-grained implementation may not be efficient in reducing excessive communication. However, this is acceptable if the goal is to produce a valid parallel processing or software pipelining schedule which meets the admissibility conditions (including communication and network topology admissibility conditions). If the desired iteration period is less than the weights of the composite nodes, fine-grained parallelism will be exploited. With a lower iteration period, the composite nodes will be expanded to allow partitioning of the internal nodes. The partitioning of the internal nodes will result in a schedule which exploits parallelism between a smaller set of operations. These ideas are illustrated by examples in Chapter 5.

4.3.3 Validating and Scheduling

There are no hierarchical issues which effect the validation of communication and network topology or the scheduling of the cycling vector and period matrix.

4.4 Summary

In summary, the major issues handled by OGM during task scheduling of hierarchical FSFGs includes:

- Providing an effective specification technique for hierarchical FSFGs.
- Recognizing feedback loops within and across levels of the hierarchy, and preventing the unnecessary partitioning of a feedback loop.
- Avoiding unnecessary flattening the hierarchical FSFG during the scheduling process.
- Preserving the freedom to partition the hierarchical FSFG to achieve the desired iteration period bound and/or the number of processors.

Methodologies to handle these issues include:

- The flexibility of the hierarchy is maintained by using nodenames and instances to define atomic and composite nodes in the hierarchical FSFG.
- Key information about a composite node is stored to prevent repetitive analysis. Input/output paths, internal loops, and the total computational delay for composite nodes is maintained.
- Composite nodes are initially analyzed for input/output paths to enable the validation of feedback loops which incorporate composite nodes. This prevents a flattening of the hierarchy to determine feedback loops.

- Composite nodes are analyzed for internal feedback loops if the iteration period bound must be determined. If this must occur, the input/output paths are used to prevent a complete flattening of the hierarchy when composite nodes occur within composite nodes.

- Strong components are expanded by expanding any composite nodes within the strong component prior to removing a feedback edge. This ensures that the largest possible strong component will be retained to minimize the number of nodes generated in a candidate partitioning.

- After ensuring all strong components have a computational delay less than the desired iteration period, all composite nodes with a weight greater than the desired iteration period are expanded. This ensures that the partitioning process will be able to partition the linear extension using the desired iteration period without splitting composite nodes across multiple processors.

With these issues addressed, Calypso has been developed to accept hierarchical FSFGs. Incorporating the methodology to handle these FSFGs, Calypso produces optimized schedules for hierarchical FSFGs with basic consideration for interprocessor communication and ring topologies.

Chapter 5

Evaluation of CALYPSO

The results of the previous chapters suggest a methodology to increase processing rates for DSP multiprocessor systems. Research at North Carolina State University has developed a high performance DSP architecture, termed the Block Data Parallel Architecture (BDPA) [2, 8, 66], which may benefit from this scheduling theory. The current BDPA system is continually being developed to support a wide array of real-time signal processing problems. Several algorithms have been successfully mapped to the BDPA including a 2-D digital IIR filter, a 2-D digital FIR filter, LU Decomposition, and a 2-D discrete cosine transform [2, 7, 8, 66]. For each of these algorithms, an effective mapping was obtained based on insight into the problem and the architecture. The mapping theory presented in this dissertation will automate this process for the BDPA, as well as other multiprocessor architectures.

In this chapter, we demonstrate the effectiveness of our mapping technique on the the BDPA for a variety of applications. Due to the complexity of the task scheduling problem, we present performance data for several benchmark applications to establish the usability and flexibility of OGM for hierarchical FSFGs. Well known as an NP-complete problem, scheduling time determines to a great extent the usability of a mapping methodology [3]. Therefore, results for scheduling several benchmark algorithms are presented, including FIR filters, IIR filters, lattice filters, LU Decomposition and QR Factorization. For each of the algorithms, a variety of hierarchical

specifications have been used to determine the effects of the initial hierarchy on the resulting multiprocessor schedule and on the scheduling time. The next section briefly describes the BDPA.

5.1 Overview of the BDPA

The BDPA system consists of three major components: an Input Module (IM), a Processor Module Array (PMA), and an Output Module (OM) as depicted in Figure 5.1. In general, the IM acts as a buffer for incoming data from an input device

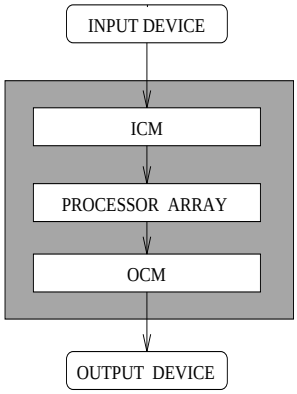


Figure 5.1: Block Diagram of the BDPA

and splits up the input stream into blocks of data. The blocks of data are then transferred to different processors in the PMA by the IM. Similarly, the OM serves as a buffer between the PMA and the output device. It receives data from the different processors in the PMA and routes the data to the output device. The PMA consists of a number of identical processors each executing the same program. In order to prevent communication contention, interprocessor communication is limited to local unidirectional communication only. This forms the basis of the BDPA system with the processors executing whenever all the required data elements for the next operation become available.

The block-data flow characteristic of the system emphasizes interaction of communication and computation. Data flow permits the elimination of lock-step operation synchronized by a global clock. The concept allows a processor to receive data into buffers and then execute as long as the data required for the next operation is available in the buffers. The block concept reduces the communication time within the system by eliminating the handshaking between processors for each data element. Using block-data flow, processors setup for communication and then pass a block of data elements before closing the communication channel. The characteristics of the BDPA system may be summarized as follows:

- block data processing and the block data flow paradigm
- globally asynchronous and locally synchronous data transmission protocol
- linear array topology and "skew" operations among processors
- local data transmission in only one direction
- overlap of data movement and interprocessor communication with data computations

One key feature of the BDPA is the ring topology. The interconnections among the BDPA processors impose a basic limitation on the scheduling of a FSFG. To handle this limitation, a scheduling algorithm should attempt to restrict a processor's dependence to data from processors which proceed it in the ring topology. Any cut in a feedback loop requires bi-directional communication. For a unidirectional ring topology, this translates to communication through each processor all the way around the ring. Thus, the effort to limit a processor's dependence helps avoid a parallel execution schedule which generates excessive communications to support a partitioned feedback loop. If a feedback loop has a total computational time greater than T_d, software pipelining may be used to reduce the required communications for execution at T_d by scheduling the feedback loop on a single processor. While the

unidirectional communication may be a disadvantage when dealing with feedback loops, it prevents contention from occurring on the ring topology and increases the effectiveness of "skew" operations. If data can be skewed to permit processing of several elements of a single data stream, dependent as well as independent data streams may often be interleaved [58].

Using OGM, the BDPA's features can be incorporated into the scheduling process for hierarchical flow graphs. Calypso has been used on a set of benchmark algorithms to produce parallel computing schedules for execution on the BDPA. The next section describes the experimental conditions and assumptions used to obtain the scheduling data for the benchmark algorithms.

5.2 Implementation and Experimental Conditions

Calypso is currently implemented in C using approximately 11,000 lines of code and consists of three executable files. The main program executes the OGM mapping methodology and calls the other two programs to verify communications and network topology admissibility. Therefore, Calypso can be implemented for another architecture by replacing the network topology executable with another architecture specific executable.

The results presented in this dissertation were obtained on a personal computer with an Intel 486DX2-50MZ processor and 16MB of memory. The target multiprocessor system was assumed to have generalized processors which perform a multiplication in 2 computational time units, an addition in 1 time unit, a divide in 10 time units, and a square root in 20 time units. The network topology is assumed to be a ring topology with each link transmitting no more than 1 unit of data per time unit.

Results for Calypso are presented in Tables 5.1 to 5.9. In the tables, column *HFSFG* identifies the initial specification of the algorithm. *Nodes* indicates the num-

ber of nodes in the initial specification, including system input/output nodes, intermediate input/output nodes and delay nodes. Column *Levels* is the number of levels in the initial specification's hierarchy. P_d and T_d identify the number of processors and the iteration period for the scheduled HFSFG. These values are the result of the scheduling process and may differ from the initial target values [1]. The average processor utilization for T_d and P_d is shown in column *Avg. Proc. Util.*. The number of linear extensions considered (during the partitioning stage of OGM) before obtaining a valid schedule is indicated by *Linear Extensions*. The running time for Calypso to obtain a schedule for the HFSFG is shown in *CPU time*. To ensure the accuracy of the CPU time, the measurements for running time are the average of 5 runs on Calypso. *SP* and *PE* indicate whether a valid schedule was found for software pipelining and parallel execution respectively. The amount of communication allowed per time unit is shown by t_c. The value of t_c is 1 for tables which do not have a t_c column. In the tables, bold entries for T_d indicate a rate optimal schedule for the value of t_c. Bold entries for an entire row in the tables indicates that the HFSFG is communication bound at the initial target value for T_d. Therefore, Calypso has identified the best possible schedule under the conditions imposed by t_c.

We now present the results for scheduling the benchmark algorithms with Calypso using the Order Graph Method. For each algorithm, we briefly describe the algorithm, illustrate non-hierarchical and hierarchical FSFGs, present Calypso's scheduling results, and illustrate a potential schedule for parallel computing. We begin with filter algorithms for digital signal processing.

[1] Calypso will attempt to meet a user specified target for T_d or P_d. During partitioning, Calypso may be able produce a schedule which improves on these values. In addition, there are some cases in which a schedule cannot be produced for the target values. In these cases, Calypso identifies and outputs the best possible schedule.

5.3 Filters

For edge enhancement and noise removal during image processing, a variety of filters may be used [67]. In this section, we present some of these filters and potential schedules produced by Calypso. For each of the filters, different hierarchical specifications are used to highlight Calypso's features and to provide insight into the effects of hierarchy. We begin with a look at the 2nd order IIR Filter.

5.3.1 2nd Order IIR Filter

The transfer function for a N^{th} order IIR filter is defined as

$$H(Z) = \frac{\sum_{n=0}^{N} b(n)Z^{-n}}{1 + \sum_{n=1}^{N} a(n)Z^{-n}}$$

with a 2nd order IIR equal to

$$H(Z) = \frac{b_0 + b_1 Z^{-1} + b_2 Z^{-2}}{1 + a_1 Z^{-1} + a_2 Z^{-2}}$$

Four different ways of specifying a 2nd order IIR are shown in Figures 5.2, 5.3, 5.4, and 5.5. We refer to these four specifications as IIR1, IIR2, IIR3, and IIR4 respectively. IIR1 is non-hierarchical while IIR2, IIR3, and IIR4 use hierarchy.

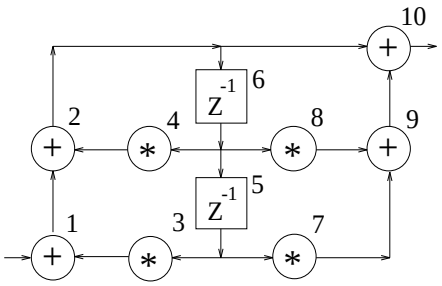


Figure 5.2: Flow graph for a 2nd order IIR filter (IIR1)

A badly specified hierarchy (IIR2) demonstrates Calypso's ability to manipulate a

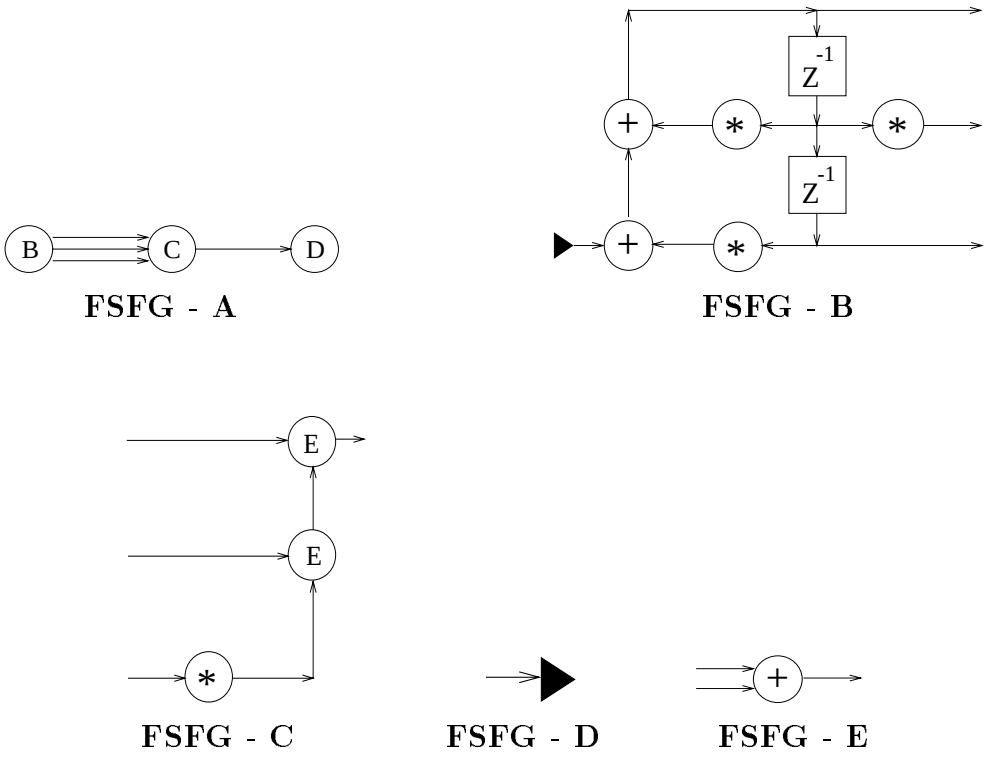


Figure 5.3: Potential FSFG hierarchical specification for the 2nd order 1-D IIR filter (IIR2)

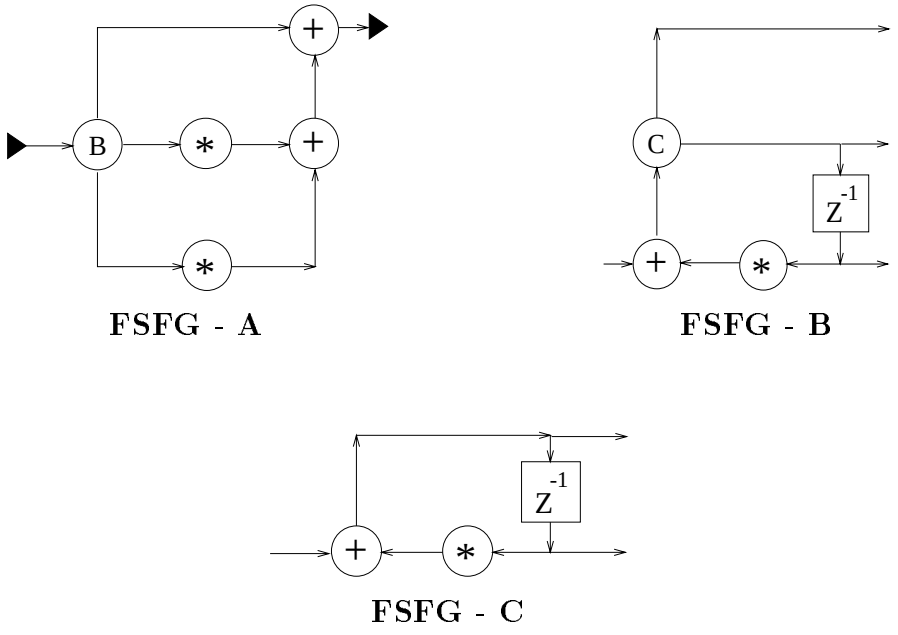


Figure 5.4: Potential FSFG hierarchical specification for the 2nd order 1-D IIR filter (IIR3)

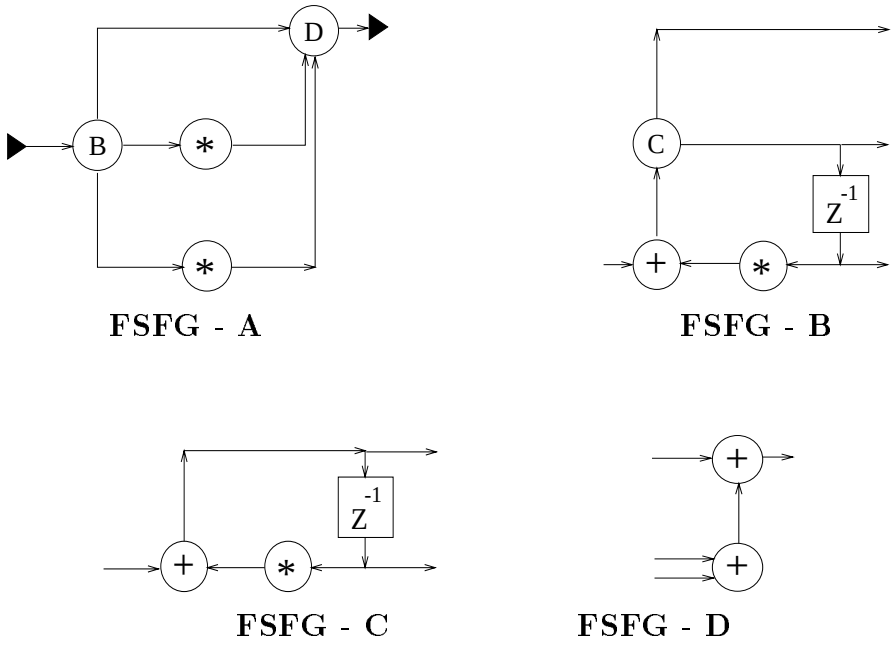


Figure 5.5: Potential FSFG hierarchical specification for the 2nd order 1-D IIR filter (IIR4)

specification's inherent partitioning to produce an optimal schedule. IIR3 identifies strong components in terms of composite nodes and will reduce the amount of time the preprocessing stage initially spends identifying strong components. Additionally, IIR4 groups nodes which are not members of a strong component into a composite node. This benefits the partitioning stage by reducing the number of nodes for consideration during the generation of linear extensions.

Results for scheduling these hierarchical FSFGs are shown in Table 5.1. With the relatively small size of the 2nd order IIR filter, the differences in the CPU times for scheduling the various specifications is negligible. However, these results illustrate the ability of OGM to effectively handle different hierarchical specifications. Even with smaller graphs, the effects of the hierarchy can be seen. For IIR1 and IIR3 with $T_d = 4$, the same number of linear extensions was generated, but the running time was greater for IIR3. This is due to the composite nodes within the linear extension. During the greedy partitioning of the linear extension for IIR3, the partitioning algorithm

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*
IIR1	10	1	1	12	1	100	0.94	Y	Y
			2	6	1	100	0.93	Y	Y
			3	4	1	100	1.15	Y	N
			4	**3**	1	100	1.20	Y	N
IIR2	10	3	1	12	1	100	0.99	Y	Y
			2	6	1	100	1.04	Y	Y
			3	4	1	100	1.20	Y	N
			4	**3**	1	100	0.99	Y	N
IIR3	10	3	1	12	1	100	1.10	Y	Y
			2	6	1	100	1.04	Y	Y
			3	4	1	100	1.04	Y	N
			4	**3**	1	100	1.54	Y	N
IIR4	10	3	1	12	1	100	0.99	Y	Y
			2	6	1	100	1.10	Y	Y
			3	4	1	100	1.04	Y	N
			4	**3**	1	100	1.10	Y	N

Table 5.1: Summary of Calypso's results for scheduling hierarchical FSFGs of a 2nd order IIR filter with $t_c = 1$

attempted to split IIR3 which resulted in more than P_d partitions. Taking advantage of the network topology, the starting point for the partitioning algorithm was shifted and another attempt was made to partition the same linear extension. For IIR3, the second attempt to partition the linear extension was successful while IIR1 was successful on the first attempt.

In general, Calypso will terminate execution as soon as a valid schedule is found. Because of communication costs, valid software pipelining schedules tend to be found prior to schedules for parallel execution. If we force Calypso to produce a schedule for parallel execution, additional linear extensions are considered until a potential candidate partitioning passes communication and network topology admissibility conditions for parallel execution. Results for the same 2nd order IIR filter FSFGs are shown in Table 5.2. A key result from forcing a parallel execution schedule with

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*
IIR1	10	1	1	12	1	100	1.01	Y	Y
			2	6	1	100	0.99	Y	Y
			3	4	42	100	7.25	Y	Y
			3	**4**	**253**	**100**	**34.99**	**Y**	**Y**
IIR2	10	3	1	12	1	100	1.04	Y	Y
			2	6	1	100	1.10	Y	Y
			3	4	6	100	3.29	Y	Y
			3	**4**	**253**	**100**	**37.08**	**Y**	**Y**

Table 5.2: Summary of Calypso's results for scheduling hierarchical FSFGs of a 2nd order IIR filter for parallel execution with $t_c = 1$

$t_c = 1$ is the realization that the rate optimal schedule for $P_d = 4$ and $T_d = T_v = 3$ cannot be accomplished. Parallel execution schedules at $T_d = 3$ become communication bound and cannot accomplish the required communications during the iteration period. Calypso recognizes that no schedules pass the communication and network topology admissibility conditions, increases the desired iteration period, and generates another set of linear extensions. The process of incrementing T_d continues until a valid parallel execution schedule is found. Thus, the difference between the number of linear extensions generated for the entries in the table under $P_d = 3$ and $T_d = 4$ is due to the initial scheduling goal. Table entries in bold resulted from attempting to schedule the HFSFGs with an iteration period of $T_d = 3$.

Figure 5.6 presents a rate and processor optimal software pipelining schedule for a 2nd order IIR filter.

5.3.2 15th Order FIR Filter

The transfer function for a N^{th} order FIR filter is defined as

$$H(Z) = \sum_{n=0}^{N} b(n) Z^{-n}$$

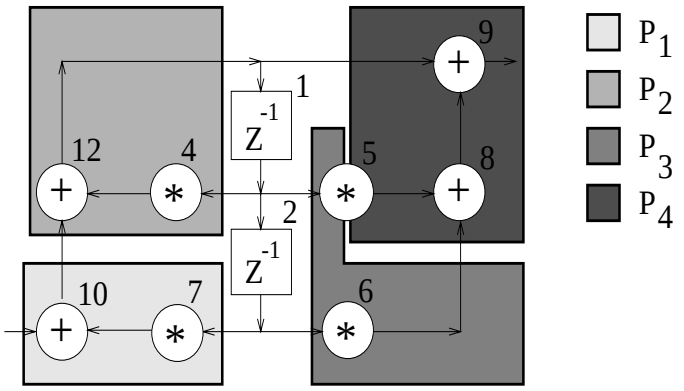


Figure 5.6: FSFG for the 2nd-order 1-D IIR filter, with a software pipelining, rate and processor optimal schedule generated by Calypso using the Order Graph Method with $t_c = 1$

with a 15th order FIR equal to

$$H(Z) = b_0 + b_1 Z^{-1} + b_2 Z^{-2} + \cdots + b_{15} Z^{-15}$$

If the coefficients are symmetric, the FSFG for a 15th order FIR is shown in Figure 5.7 (FIR1). A potential hierarchical specification which takes advantage of the FIR's repetitive nature of this flow graph is shown in Figure 5.8(FIR2). Results for

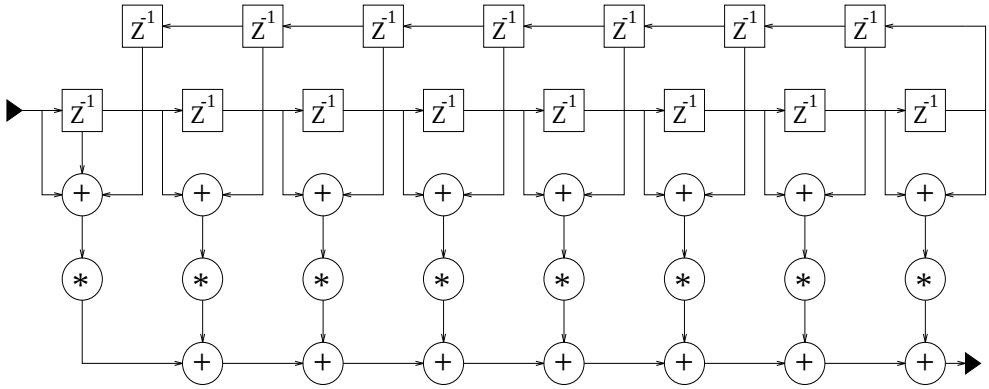
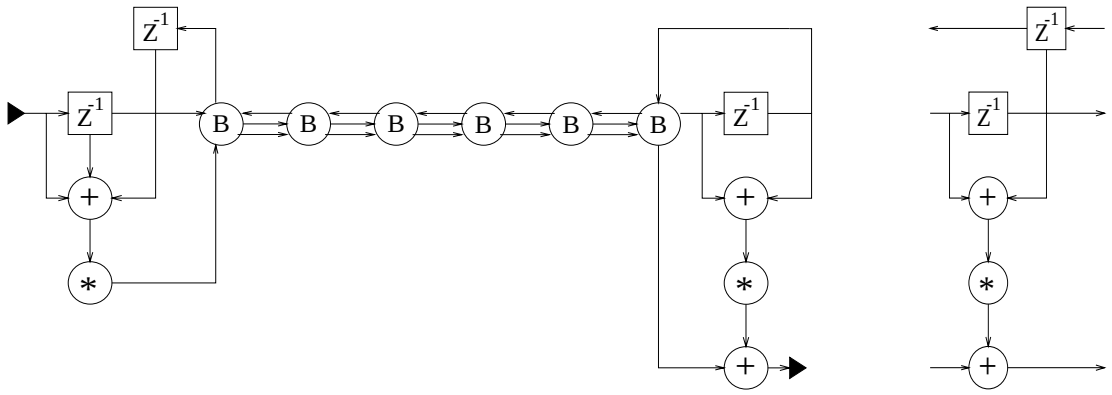


Figure 5.7: FSFG for the 15th order FIR filter (FIR1)

scheduling these FSFGs are shown in Table 5.3.

In the case of the FIR filter, there is not a feedback loop to restrict the iteration period to T_v. Therefore, the iteration period for a potential schedules becomes limited



FSFG - A **FSFG - B**

Figure 5.8: Potential FSFG hierarchical specification for a 15th order FIR filter (FIR2)

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*	t_c
FIR1	38	1	1	31	1	100	1.05	Y	Y	1
			2	16	1	96.9	1.10	Y	Y	1
			3	11	1	93.9	0.99	Y	Y	1
			4	8	1	96.9	1.05	Y	Y	1
			7	5	1	88.6	1.27	Y	Y	1
			8	4	1	96.9	1.04	Y	N	1
			11	3	1	93.9	1.05	Y	N	1
			16	**2**	1	96.9	1.15	Y	N	1
			8	**4**	1	96.9	1.60	Y	N	0.5
			4	**10**	1	77.5	1.32	Y	N	0.2
FIR2	38	1	1	31	1	100	1.26	Y	Y	1
			2	16	1	96.9	1.05	Y	Y	1
			3	11	1	93.9	1.04	Y	Y	1
			4	8	1	96.9	1.09	Y	Y	1
			7	5	1	88.6	1.16	Y	Y	1
			8	4	1	96.9	1.16	Y	N	1
			11	3	1	93.9	1.10	Y	N	1
			16	**2**	1	96.9	1.15	Y	N	1
			8	**4**	1	96.9	1.27	Y	N	0.5
			4	**10**	1	77.5	1.21	Y	N	0.2

Table 5.3: Summary of Calypso's results for scheduling hierarchical FSFGs of a 15th order FIR filter

by the communication requirements. For example, a schedule for parallel execution with one atomic node per processor requires a minimum of 2 communications. Thus, the preprocessing stage determines that the desired iteration period must be greater than $\frac{2}{t_c}$. If Calypso is forced to obtain a schedule for parallel execution, an attempt will be made to partition and schedule the algorithm with $T_d = \frac{2}{t_c}$. If a valid schedule does not exist, T_d will be increased by one and Calypso will begin the partitioning and scheduling process again. This process continues until a valid schedule is found.

For the 15th order FIR filter in Figure 5.7 with $t_c = 1$, Calypso obtains a parallel execution schedule with $T_d = 5$ and $P_d = 7$ and a rate optimal software pipelining schedule at $T_d = 2$ and $P_d = 16$. Changing the value for t_c changes the admissibility of communications for individual schedules and results in different optimal schedules. Figure 5.9 illustrates a potential partitioning generated by Calypso for an optimal parallel execution schedule allowing atomic nodes to be split across processors. Choosing not to split atomic nodes across processors, Calypso produces the schedule depicted in Figure 5.10.

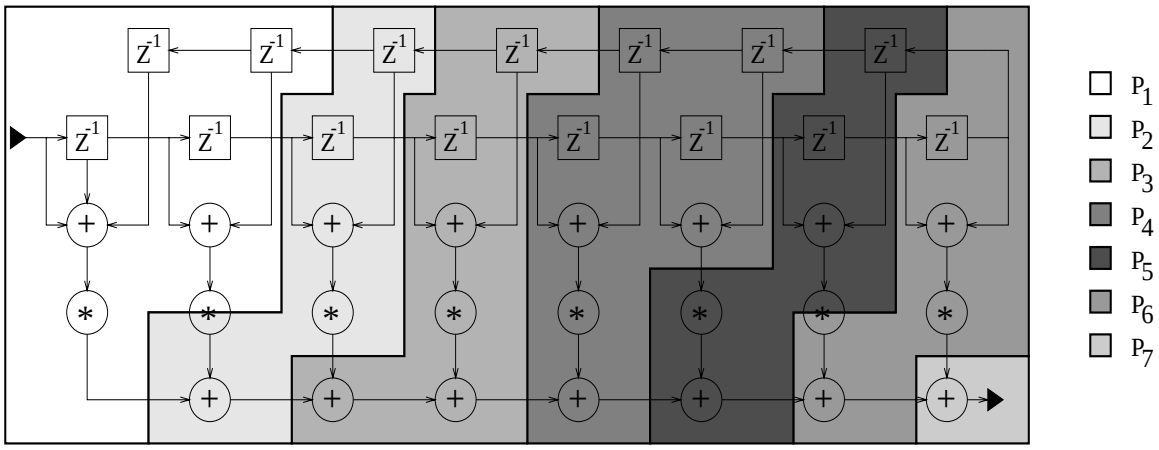


Figure 5.9: FSFG for the 15th order FIR filter, with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 7$, $T_d = 5$, and $t_c = 1$

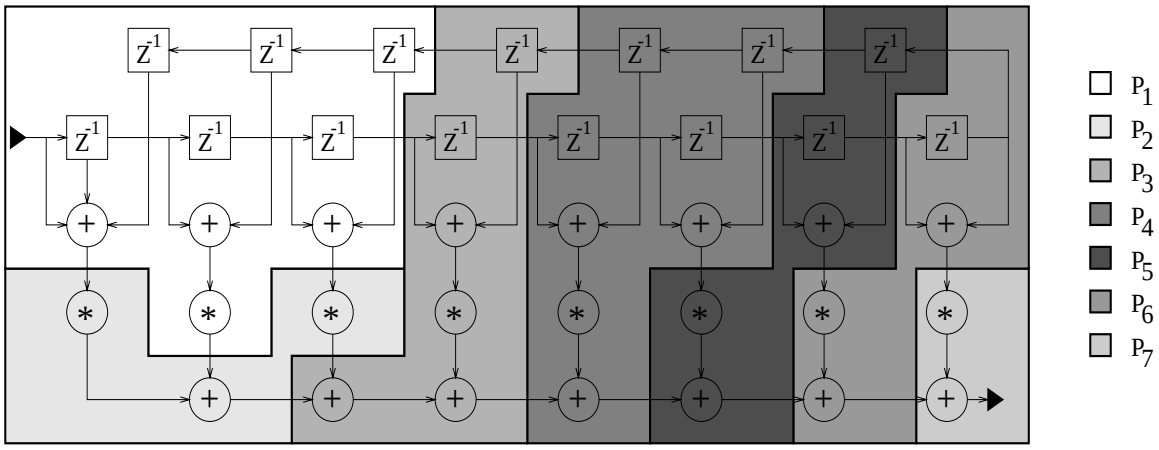


Figure 5.10: FSFG for the 15th order FIR filter, with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 7$, $T_d = 5$, $t_c = 1$, and not splitting atomic nodes across processors

5.3.3 4th Order Lattice Filter

In this section, we introduce another IIR filter structure called the lattice filter or the lattice realization. Used extensively in digital speech processing and in the implementation of adaptive filters, the lattice filter is derived from the difference equations for an IIR filter [68]. The FSFG for a 4th order lattice filter is shown in Figure 5.11 with a potential hierarchy (using the repetitive nature of the lattice filter's design)

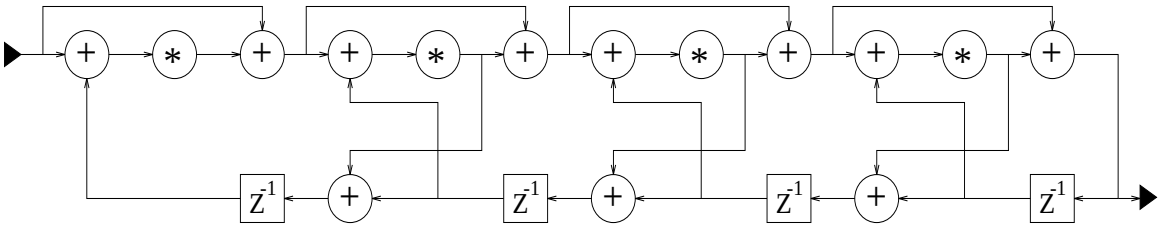
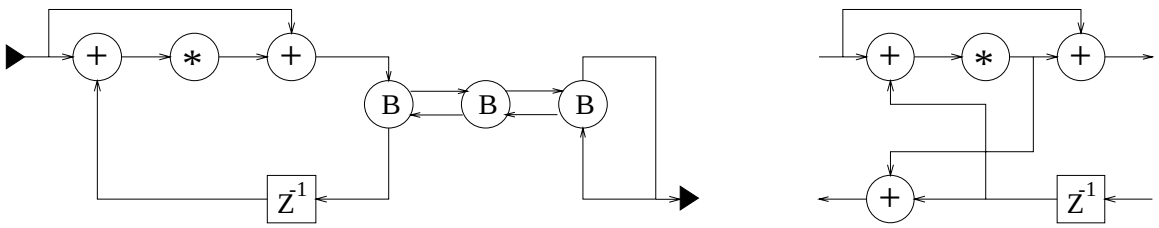


Figure 5.11: FSFG for a 4th order lattice filter (LAT1)

shown in Figure 5.12.

This filter highlights Calypso's ability to deal with multiple feedback loops across levels of the hierarchy during preprocessing and the identification of T_v. In the 4th order lattice filter, there are over 40 loops to evaluate in order to determine T_v for



FSFG - A **FSFG - B**

Figure 5.12: Potential FSFG hierarchical specification for a 4th order lattice filter (LAT2)

rate optimal schedules. The initial reduction of strong components into composite nodes will produce a graph with 3 nodes - an INPUT, a composite, and an OUTPUT. For the FSFG in Figure 5.12 the composite node will contain other composite nodes internally. During preprocessing, reducing the composite node's weight to less than T_d will require the composite node to be expanded and reduced multiple times. Therefore, the preprocessing stage of OGM will require more time than the 2nd order IIR and 15th order FIR.

Results for scheduling these hierarchical FSFGs are shown in Table 5.4. An important aspect of the results present in Table 5.4 is the slight increase in running time between LAT1 and LAT2. During the preprocessing stage of OGM, an attempt is made to contain a feedback loops within a composite node. For the case of the lattice filter, the composite node repeatedly exceeds the maximum computational delay and must be repeatedly expanded and reduced until the computation weight is less than T_d. In this case, composite nodes resulted in the process taking a longer amount of time. This is due to the requirement to expand composite nodes within the composite node prior to removing any edges from the graph.

Lowering the value of t_c for the lattice filter results in the parallel computing schedule becoming communication bound. Attempts to schedule the LAT1 and LAT2 with $T_d = 8$ and $t_c = 0.2$ fails the communication admissibility condition. As a result, OGM increases the value of T_d and attempts to produce a valid candidate partitioning

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*	t_c
LAT1	19	1	1	19	1	100	1.10	Y	Y	1
			2	10	1	95.0	1.87	Y	Y	1
			3	**8**	1	79.2	1.15	Y	N	1
			1	19	1	100	1.05	Y	Y	0.5
			2	10	1	95.0	2.15	Y	Y	0.5
			3	**8**	1	79.2	1.54	Y	N	0.5
			1	19	1	100	1.21	Y	Y	0.2
			2	13	406	73.1	283.01	Y	N	0.2
			2	**13**	**648**	**73.1**	**410.54**	**Y**	**N**	**0.2**
LAT2	19	1	1	19	1	100	1.11	Y	Y	1
			2	10	1	95.0	1.97	Y	Y	1
			3	**8**	1	79.2	1.27	Y	N	1
			1	19	1	100	1.32	Y	Y	0.5
			2	10	1	95.0	2.46	Y	Y	0.5
			3	**8**	1	79.2	1.43	Y	N	0.5
			1	19	1	100	1.31	Y	Y	0.2
			2	13	406	73.1	272.84	Y	N	0.2
			2	**13**	**648**	**73.1**	**431.80**	**Y**	**N**	**0.2**

Table 5.4: Summary of Calypso's results for scheduling hierarchical FSFGs of a 4th order lattice filter

from the same set of linear extensions. Similar to IIR1 and IIR2, this process continues until a valid candidate partitioning is found.

Figure 5.13 illustrates the rate and processor optimal software pipelining schedule obtained by Calypso for the 4th order lattice Filter.

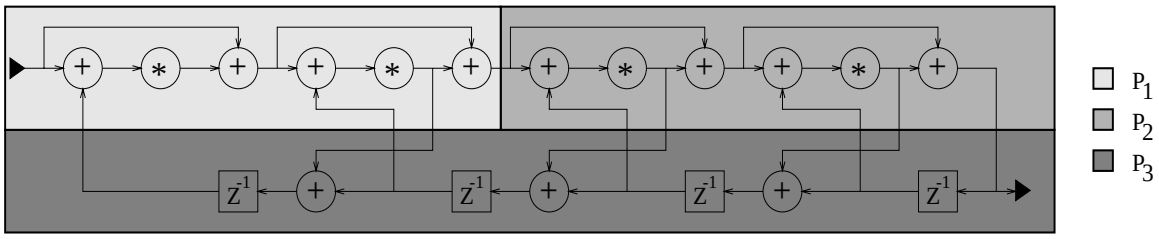


Figure 5.13: FSFG for a 4th order lattice filter, with a software pipelining, rate and processor optimal schedule generated by Calypso using the Order Graph Method with $t_c = 1$

5.3.4 2-D Filtering

The transfer function for the 2-D IIR filter is given by

$$H(Z_1 Z_2) = \frac{\sum_{n=0}^{N} \sum_{m=0}^{2} b(n,m) Z_1^{-n} Z_2^{-m}}{1 + \sum_{\substack{n=0 \\ n+m>0}}^{N} \sum_{m=0}^{2} a(n,m) Z_1^{-n} Z_2^{-m}}$$

The BDPA currently implements the 2nd order 2-D IIR filter with a 3 stage pipeline, computational primitive. The equations for the pipeline are derived from the state space representation for the 2-D system

$$\begin{aligned}
y(n_1,n_2) &= q_{1,1}(n_1-1,n_2) + q_{2,1}(n_1,n_2-1) \\
g(n_1,n_2) &= b_{0,0} f(n_1,n_2) + y(n_1,n_2) \\
q_{1,1}(n_1,n_2) &= b_{1,1} f(n_1,n_2) + a_{1,1} y(n_1,n_2) + q_{1,2}(n_1-1,n_2) \\
q_{1,2}(n_1,n_2) &= b_{1,2} f(n_1,n_2) + a_{1,2} y(n_1,n_2) \\
q_{1,3}(n_1,n_2) &= b_{1,3} f(n_1,n_2) + a_{1,3} y(n_1,n_2) + q_{1,4}(n_1-1,n_2) \\
q_{1,4}(n_1,n_2) &= b_{1,4} f(n_1,n_2) + a_{1,4} y(n_1,n_2) \\
q_{1,5}(n_1,n_2) &= b_{1,5} f(n_1,n_2) + a_{1,5} y(n_1,n_2) + q_{1,6}(n_1-1,n_2) \\
q_{1,6}(n_1,n_2) &= b_{1,6} f(n_1,n_2) + a_{1,6} y(n_1,n_2) \\
q_{2,1}(n_1,n_2) &= b_{2,1} f(n_1,n_2) + a_{2,1} y(n_1,n_2) + q_{1,3}(n_1-1,n_2) + q_{2,2}(n_1,n_2-1) \\
q_{2,2}(n_1,n_2) &= b_{2,2} f(n_1,n_2) + a_{2,2} y(n_1,n_2) + q_{1,5}(n_1-1,n_2)
\end{aligned}$$

where $q_{1,x}(n_1,n_2)$ are horizontal state variables, $q_{2,x}(n_1,n_2)$ are the vertical state variables, and $g(n_1,n_2)$ is the output (for details see [2, 7, 8]). These equations can be represented by the FSFG in Figure 5.14. Using Calypso, the 2nd order 2-D IIR filter's computational primitive equations can be mapped to an execution sequence requiring 5 cycles for a single iteration. However, a feedback loop will be split across multiple processors unless software pipelining is used. Results for scheduling the computational primitive equations' FSFG are shown in Table 5.5.

Results in Table 5.5 show a fairly consistent running time with a greater number of linear extensions generated by Calypso for T_d = 7, 6, and 5. The higher number of linear extensions for the lower values of the iteration period are attributed to Calypso's insistence that the communication due to feedback loops be minimized and

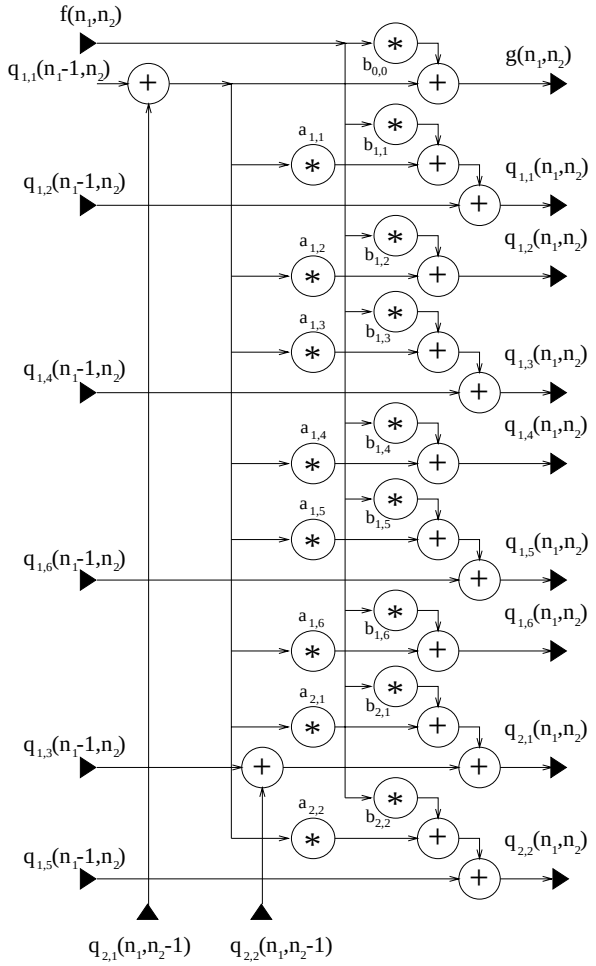


Figure 5.14: FSFG of the computational primitive equations for the 2nd order 2-D IIR Filter (2DIIR1)

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*
2DIIR1	41	1	1	50	1	100	1.05	Y	Y
			2	25	1	100	1.10	Y	Y
			3	17	1	98.0	1.15	Y	Y
			4	13	1	96.2	1.67	Y	Y
			5	10	1	100	1.70	Y	N
			7	8	1	89.3	1.15	Y	N
			8	7	37	89.3	1.30	Y	N
			9	6	37	92.6	1.26	Y	N
			10	**5**	42	100	1.86	Y	N

Table 5.5: Summary of Calypso's results for scheduling the computational primitive equations' FSFG for a 2-D 2nd order IIR filter with $t_c = 1$

that split feedback loops be scheduled on adjacent processors. In the case of the 2DIIR1 FSFG, the feedback loop has a computational weight of 8. Therefore, when the desired iteration period is 8, multiple linear extensions are generated in order to schedule the feedback loop on a single processor. With a desired iteration period less than 8, the feedback loop must be split into a composite node and two atomic nodes. Multiple linear extensions are produced before a candidate partitioning which keeps the three nodes (which comprise the feedback loop) on adjacent processors.

Another major result was produced using the 2DIIR1 FSFG by turning off Calypso's ability to shift the starting point for partitioning a linear extension. For each linear extension, multiple candidate partitionings can be produced by allowing the starting position for the partitioning algorithm to change. When this feature (shown in block 12 of Figure 3.1) is turned off, there is a one to one relationship between the linear extension and the candidate partitioning which can be produced. For the 2DIIR1, the results are shown in Table 5.6.

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*
2DIIR1	41	1	1	50	1	100	1.05	Y	Y
			2	25	1	100	1.10	Y	Y
			3	17	1	98.0	1.15	Y	Y
			4	13	4	96.2	1.60	Y	Y
			5	10	81,117	100	48.61	Y	N
			7	8	1	89.3	1.15	Y	N
			8	7	37	89.3	1.21	Y	N
			9	6	37	92.6	1.16	Y	N
			10	**5**	16,777,720	100	11,126.04	Y	N

Table 5.6: Summary of Calypso's results for scheduling the computational primitive equations' FSFG for a 2-D 2nd order IIR filter with $t_c = 1$, and no shifting in the partitioning strategy

The results in Table 5.6 are highlighted by the 3 hours of CPU time to schedule the primitive equations at $T_d = T_v = 5$. This can be explained with a detailed

look at the partitioning stage for the 2DIIR1 FSFG. The partitioning stage generates linear extensions based on the dependencies between nodes in the DAG. From a linear extension, the candidate partitionings are determined. For the 2DIIR1 FSFG, candidate partitionings are valid only if there are 10 partitions and the feedback loop is split across adjacent partitions. In addition, the partitioning algorithm requires that the composite node cannot be split across multiple partitions. Therefore, when an attempt is made to split the composite node, the number of partitions increases by one with the composite node being placed in the partition. This results in candidate partitionings which exceed the goal of 10 partitions and other candidate partitionings must be considered. With the shifting for the network topology turned off, a new candidate partitioning can only be generated from another new linear extension. In the case of the 2DIIR1 with $T_d = 5$ and $P_d = 10$, the feedback loop is comprised of a composite node with a weight of 5 and two atomic nodes. In the initial linear extension, the composite node happens to be positioned where OGM will continually attempt to split it. Therefore, the candidate partitionings are invalid and additional linear extensions must be produced.

Another factor in the the generation of the large number of linear extensions for the 2DIIR1 is the low mobility of the composite node within the linear extension. In this case, 66% of the nodes in the graph are constrained by the composite node. Subsequent linear extensions are more likely to swap two atomic nodes than to swap an atomic node with the composite node. Therefore, the position of the composite node in the linear extension changes infrequently and linear extensions are rejected for having more than 10 partitions. For the 2DIIR1 FSFG, the location of the composite node within the initial candidate partitioning and the reduced mobility of the other nodes, translates to approximately 0.0016% of the linear extensions being valid for the number of partitions. This percentage is then reduced even further by the requirement that the nodes which denote the feedback loop be scheduled on adja-

cent processors. As a result, numerous linear extensions are generated before a valid candidate partitioning can be found to begin checking admissibility conditions for communications and the network topology.

One potential software pipelining schedule for the computational primitive equations with $P_d = 7$ and $T_d = 8$ is shown in Figure 5.15.

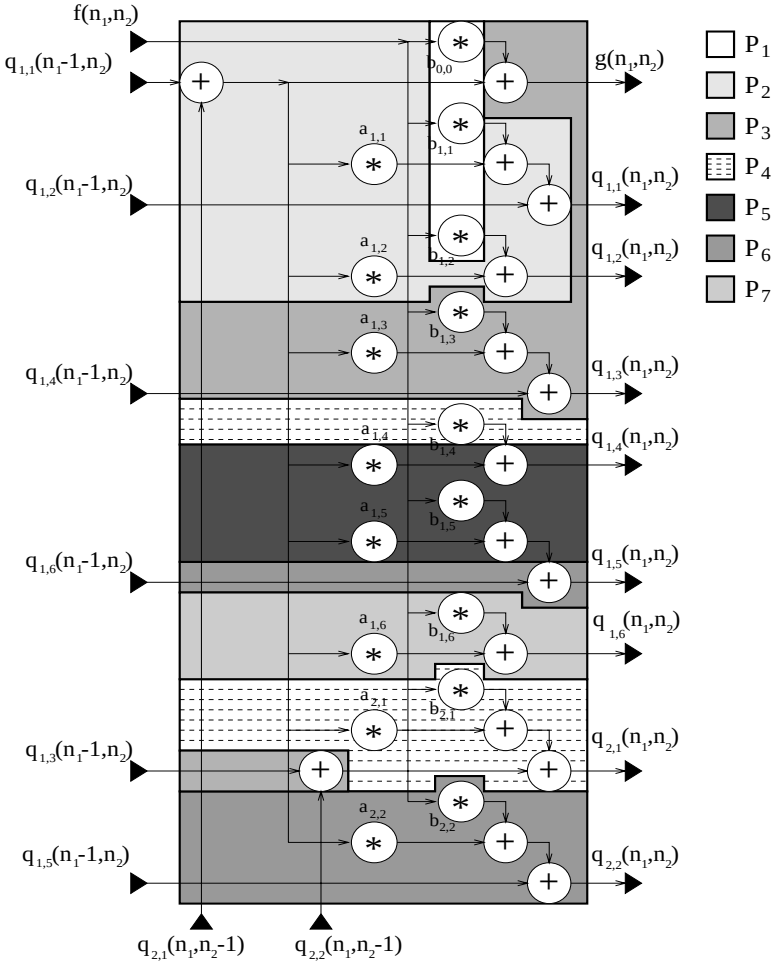
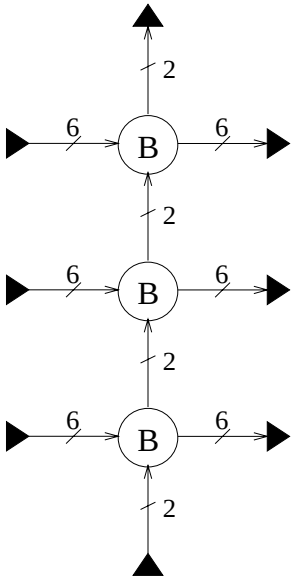


Figure 5.15: FSFG for the 2nd order 2-D IIR filter's computational primitive equations, with a software pipelined schedule generated by Calypso using the Order Graph Method for $P_d = 7$, $T_d = 8$, and $t_c = 1$

To achieve the 2-D aspect for this filter, the FSFG for the computational primitive equations is repeated three times with the values of $q_{2,1}(n_1, n_2 - 1)$ and $q_{2,2}(n_1, n_2 - 1)$ being passed between the instances of the FSFG. Using a row of data as input into the

FSFG, horizontal state variables are not passed between the instances of the FSFG. Figures 5.16 and 5.17 present potential hierarchical specifications for the 2-D aspect of this filter. Each composite node in Figure 5.16 takes a row of data from an image



FSFG - A

Figure 5.16: Potential FSFG hierarchical specification of the computational primitive for the 2nd order 2-D IIR Filter (3x1) (2DIIR2)

as input. In Figure 5.17, composite nodes take a single pixel as input. In this case, the horizontal state variables are passed between instances of the FSFG which process pixels from the same row. Therefore, FSFG - B for Figure 5.17 (shown in Figure 5.14) is modified to change the horizontal state values from system input/outputs to intermediate input/outputs. Results for scheduling these hierarchical FSFGs are shown in Table 5.7.

In general, Calypso is able to minimize the amount of scheduling time for hierarchical FSFGs. During the partitioning stage, Calypso generates linear extensions based on the composite nodes derived from the hierarchy. When all of the composite nodes have a weight equal to T_d, the linear extension is guaranteed to generate the optimal number of partition (P_d). Thus, scheduling simply requires the generation

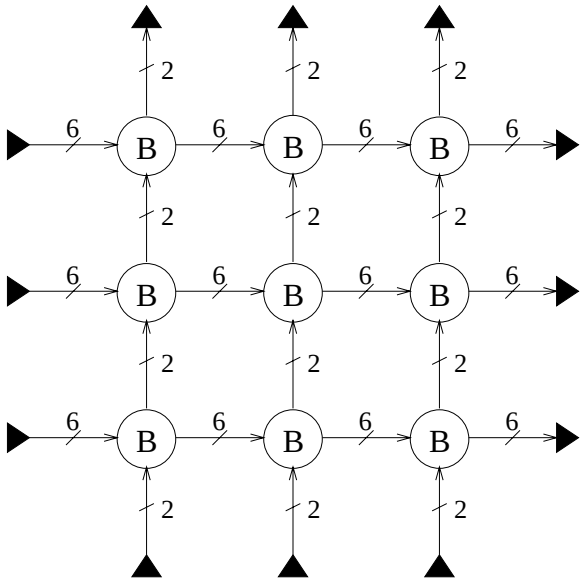


Figure 5.17: Potential FSFG hierarchical specification of the computational primitive for the 2nd order 2-D IIR Filter (3x3) (2DIIR3)

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*	t_c
2DIIR2	123	2	3	50	1	100	1.54	Y	Y	1
3x1			3	50	1	100	1.93	Y	Y	0.5
			3	50	1	100	1.92	Y	Y	0.36
2DIIR3	369	2	3	150	1	100	3.90	Y	Y	1
3x3			3	150	1	100	4.06	Y	Y	0.5
			3	150	1	100	4.01	Y	N	0.2
			3	150	5	100	20.38	Y	Y	0.2
2DIIR4	656	3	4	200	1	100	9.01	Y	Y	1
4x4			4	200	1	100	9.17	Y	Y	0.5
			4	200	1	100	9.17	Y	Y	0.25
			4	200	1	100	9.34	Y	N	0.2
2DIIR5	2624	4	8	400	1	100	40.89	Y	Y	1
8x8			8	400	1	100	46.37	Y	Y	0.5
			8	400	1	100	48.34	Y	Y	0.25
			8	400	1	100	47.89	Y	N	0.2

Table 5.7: Summary of Calypso's results for scheduling hierarchical FSFGs of a 2-D 2nd order IIR filter

of a candidate partitioning which passes the communication and network topology admissibility conditions. In addition, the time between the generation of successive linear extensions is minimized because the minimum number of nodes has been used to represent the FSFG. The efficiency of this representation within OGM is apparent in the running times for scheduling the 2DIIR2, 2DIIR3, and 2DIIR4 FSFGs. There is not a great deal of difference in the running times for the FSFGs despite the increase in the number of nodes scheduled. In this case, the increase in the running time can be attributed to the validation stage of OGM. With the increase in the number of nodes per processor and the number of processors, the number of communications between processors increased. Therefore, the validation stage required more time to check the communications and network topology admissibility conditions.

Figures 5.18 and 5.19 present schedules obtained for a 2DIIR3 with $P_d = 3$ and $t_c = 0.2$. Figure 5.18 illustrates the first valid schedule found by Calypso - a software

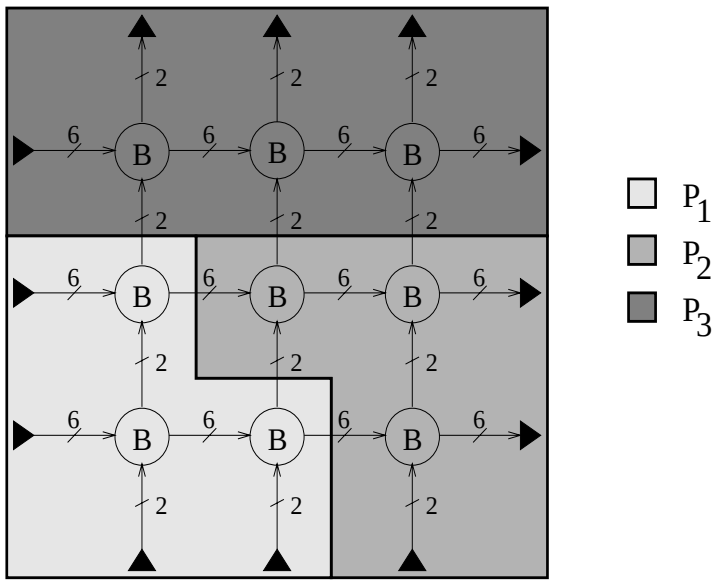


Figure 5.18: FSFG of the computational primitive for the 2nd order 2-D IIR Filter (3x3), with a software pipelining schedule generated by Calypso using the Order Graph Method for $P_d = 3$, $T_d = 150$, and $t_c = 1$

pipelining schedule. Figure 5.19 depicts the result for the same hierarchical FSFG if

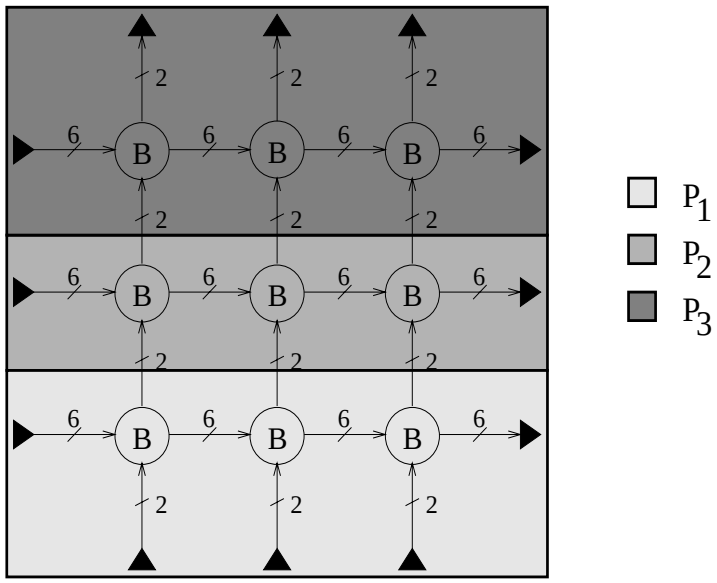


Figure 5.19: FSFG of the computational primitive for the 2nd order 2-D IIR Filter (3x3), with a parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 3$, $T_d = 150$, and $t_c = 0.2$

Calypso is forced to obtain a schedule for parallel execution. The difference between these schedules is due to the communication costs associated with parallel execution. For the schedule in Figure 5.18, software pipelining communication validates while parallel execution communication requirements do not pass the communication and network topology admissibility conditions. When a schedule (produced from a candidate partitioning) fails to pass communication or network topology admissibility conditions, Calypso continues to search for a candidate partitioning with lower communication costs. It is interesting to note that the parallel schedule produced by Calypso allows a row of data from an image to be processed by a single processor (as shown in Figure 5.16).

5.4 Matrix Operations

5.4.1 LU Decomposition

LU decomposition is a method widely used in solving sets of linear algebraic equations and in single value decomposition for digital filter design. A square matrix, A, is factored into two triangular matrices, L and U, where L is a lower triangular matrix with diagonal elements equal to one, and U is an upper triangular matrix.

$$A = LU$$

where A, L, and U are n by n matrices. Thus,

$$Ax = b$$

is equivalent to

$$LUx = b$$

Using forward substitution to solve for y in the following

$$Ly = b$$

and then backward substitution in the following

$$Ux = y$$

obtains the solution x.

For example, the following matrix A is factored into L and U matrices.

$$A = \begin{bmatrix} 3 & 1 & 3 & 2 \\ 6 & 6 & 8 & 7 \\ 9 & 23 & 22 & 22 \\ 12 & 16 & 24 & 23 \end{bmatrix}$$

Using row one and eliminating the values in column one of the following rows obtains:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 3 & 1 & 3 & 2 \\ 0 & 4 & 2 & 3 \\ 0 & 20 & 13 & 16 \\ 0 & 12 & 12 & 15 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 1 & 3 & 2 \\ 2 & 4 & 2 & 3 \\ 3 & 20 & 13 & 16 \\ 4 & 12 & 12 & 15 \end{bmatrix}$$

where the matrix to the right represents the storage of both L and U matrices as a single matrix (the diagonal of L is omitted because it will be all 1s). Using row two and eliminating the values in column two of the following rows obtains:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 5 & 0 & 0 \\ 4 & 3 & 0 & 0 \end{bmatrix} \begin{bmatrix} 3 & 1 & 3 & 2 \\ 0 & 4 & 2 & 3 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 6 & 6 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 1 & 3 & 2 \\ 2 & 4 & 2 & 3 \\ 3 & 5 & 3 & 1 \\ 4 & 3 & 6 & 6 \end{bmatrix}$$

Using row three and eliminating the values in column three of the following rows obtains:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 5 & 1 & 0 \\ 4 & 3 & 2 & 0 \end{bmatrix} \begin{bmatrix} 3 & 1 & 3 & 2 \\ 0 & 4 & 2 & 3 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 1 & 3 & 2 \\ 2 & 4 & 2 & 3 \\ 3 & 5 & 3 & 1 \\ 4 & 3 & 2 & 4 \end{bmatrix}$$

Using row four as the pivot row obtains:

$$LU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 5 & 1 & 0 \\ 4 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 & 3 & 2 \\ 0 & 4 & 2 & 3 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 3 & 2 \\ 6 & 6 & 8 & 7 \\ 9 & 23 & 22 & 22 \\ 12 & 16 & 24 & 23 \end{bmatrix} = A$$

A FSFG representation of the LU Decomposition algorithm for a 4 X 4 matrix is shown in Figure 5.20 with a potential hierarchical FSFG shown in Figure 5.21.

Similar to the FIR filter, the iteration period is limited by the communication requirements of any particular schedule. Results for scheduling the LU Decomposition FSFGs are shown in Table 5.8. Figure 5.22 shows the schedule for LUD2 for $P_d = 4$ and $T_d = 34$, and $t_c = 1$.

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*
LUD1	46	1	1	135	1	100	1.26	Y	Y
4X4			2	68	1	99.2	1.26	Y	Y
			3	45	1	100	1.26	Y	Y
			4	34	1	99.2	1.30	Y	Y
			8	17	1	99.2	1.26	Y	N
			15	9	1	100	1.37	Y	N
			23	6	1	97.8	1.32	Y	N
			27	5	1	100	1.32	Y	N
			34	4	1	99.2	1.37	Y	N
			45	3	1	100	1.43	Y	N
			68	**2**	1	99.2	1.59	Y	N
LUD2	46	2	1	135	1	100	1.19	Y	Y
4X4			2	68	1	99.2	1.20	Y	Y
			3	45	1	100	1.26	Y	Y
			4	34	1	99.2	1.27	Y	Y
			8	17	1	99.2	1.21	Y	N
			15	9	1	100	1.38	Y	N
			23	6	1	97.8	1.26	Y	N
			27	5	1	100	1.21	Y	N
			34	4	1	99.2	1.26	Y	N
			45	3	1	100	1.43	Y	N
			68	**2**	1	99.2	1.38	Y	N
LUD3	133	2	1	280	1	100	4.92	Y	Y
8X8			2	140	1	100	4.87	Y	Y
			3	94	1	99.2	5.05	Y	Y
			4	70	1	100	5.09	Y	Y
			8	35	1	100	5.03	Y	N
			16	18	1	97.2	5.21	Y	N
			32	9	1	97.2	5.17	Y	N
			56	5	1	100	5.11	Y	N
			70	4	1	100	5.09	Y	N
			94	3	1	99.2	5.28	Y	N
			140	**2**	1	100	5.11	Y	N

Table 5.8: Summary of Calypso's results for scheduling hierarchical FSFGs for LU Decomposition with $t_c = 1$

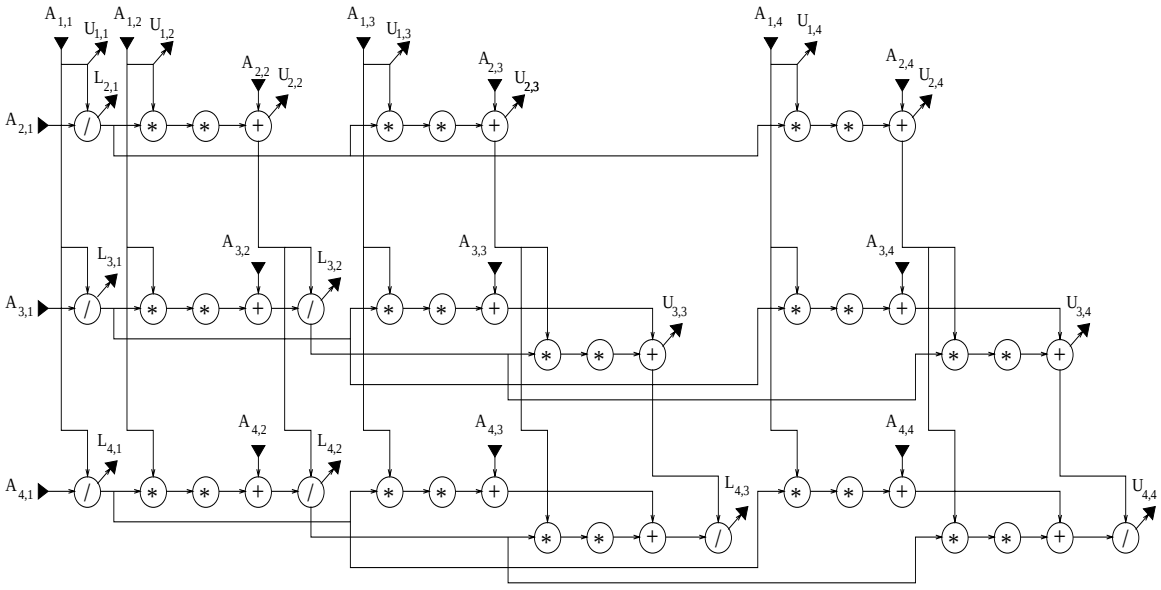
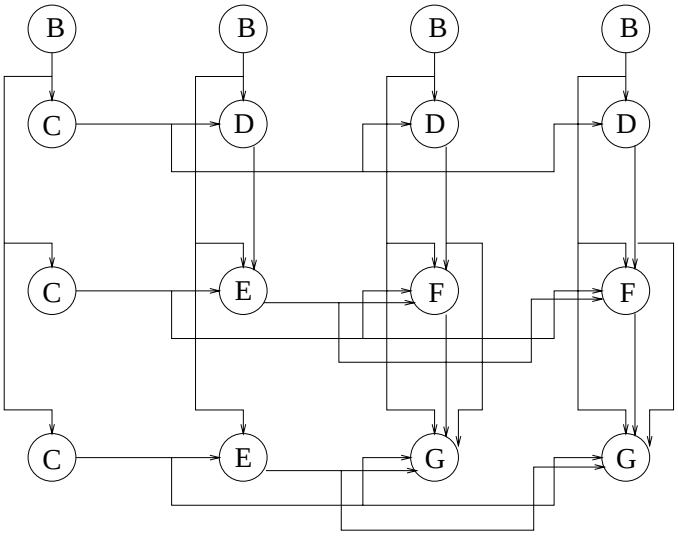
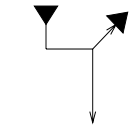


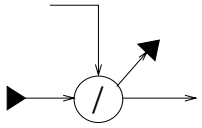
Figure 5.20: FSFG for LU Decomposition of a 4 X 4 matrix (LUD1)



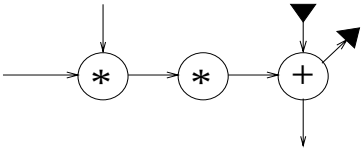
FSFG - A



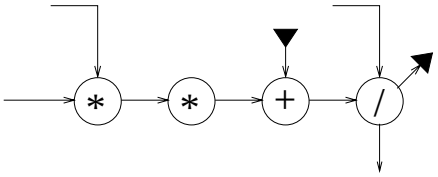
FSFG - B



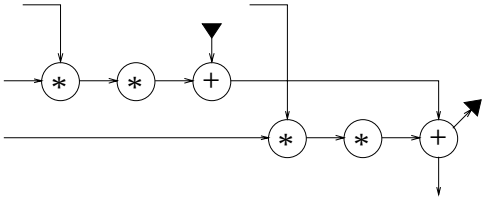
FSFG - C



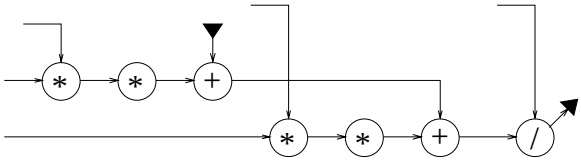
FSFG - D



FSFG - E



FSFG - F



FSFG - G

Figure 5.21: Potential FSFG hierarchical specification for LU Decomposition of a 4 X 4 matrix (LUD2)

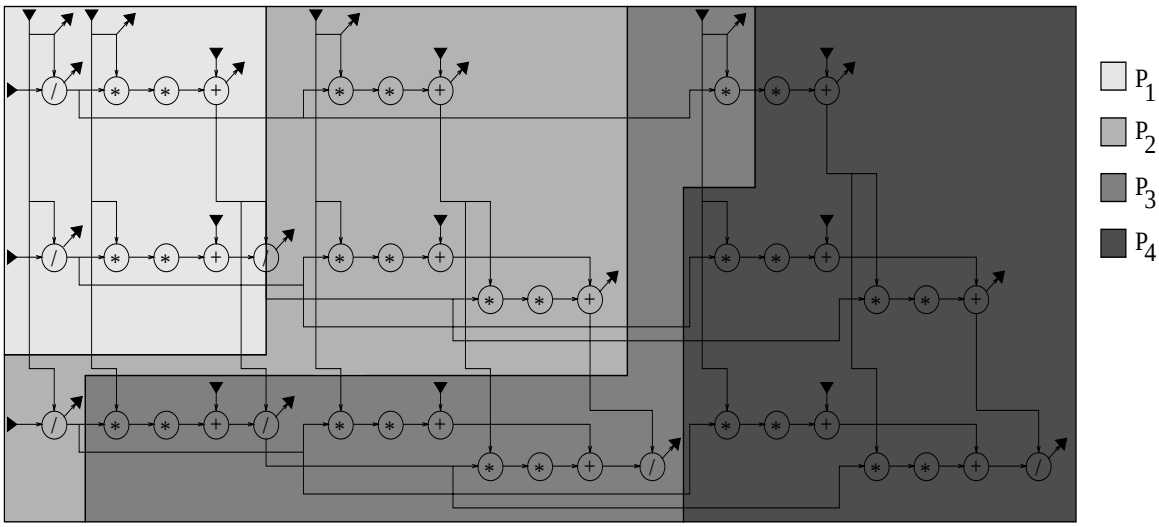


Figure 5.22: FSFG for LU Decomposition of a 4X4 matrix, with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 4$, $T_d = 34$, and $t_c = 1$

5.4.2 QR Factorization

QR Factorization is a method used in solving sets of linear algebraic equations, in finding eigenvalues and eigenvectors, and in least squares updating and downdating. In QR Factorization, a square matrix, A, is factored into an upper triangular matrix through a series of mathematical manipulations. For example, consider the following matrix:

$$A = \begin{bmatrix} 4 & 1 & 6 \\ -4 & 3 & 2 \\ 8\sqrt{2} & -4-2\sqrt{2} & \sqrt{2} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Using Givens Rotations, the equations

$$\begin{aligned} \theta &= \tan^{-1}\tfrac{r_{21}}{r_{11}} = \tan^{-1}\tfrac{-4}{4} = -\tfrac{\pi}{4} \\ r^1_{1x} &= r_{1x}\cos\theta + r_{2x}\sin\theta \\ r^1_{2x} &= -r_{1x}\sin\theta + r_{2x}\cos\theta \end{aligned}$$

produce G_{12}. The matrix can be reduced into

$$G_{12}A = \begin{bmatrix} 8\sqrt{2} & -2\sqrt{2} & 4\sqrt{2} \\ 0 & 4\sqrt{2} & 8\sqrt{2} \\ 8\sqrt{2} & -4-2\sqrt{2} & \sqrt{2} \end{bmatrix}$$

Similarly, the equations

$$\begin{aligned} \theta &= \tan^{-1}\tfrac{r_{31}}{r^1_{11}} = \tan^{-1}\tfrac{8\sqrt{2}}{8\sqrt{2}} = \tfrac{\pi}{4} \\ r^2_{1x} &= r^1_{1x}\cos\theta + r_{3x}\sin\theta \\ r^1_{3x} &= -r^1_{1x}\sin\theta + r_{3x}\cos\theta \end{aligned}$$

form G_{13} and reduce the matrix to

$$G_{13}G_{12}A = \begin{bmatrix} 32 & -6 & 10 \\ 0 & 4\sqrt{2} & 8\sqrt{2} \\ 0 & -4\sqrt{2} & -6 \end{bmatrix}$$

Using the new elements of the third column, G_{23} is found using the equations

$$\begin{aligned} \theta &= \tan^{-1}\tfrac{r^1_{32}}{r^1_{22}} = \tan^{-1}\tfrac{-4\sqrt{2}}{4\sqrt{2}} = -\tfrac{\pi}{4} \\ r^2_{2x} &= r^1_{2x}\cos\theta + r^1_{3x}\sin\theta \\ r^2_{3x} &= -r^1_{2x}\sin\theta + r^1_{3x}\cos\theta \end{aligned}$$

and the matrix is factored such that

$$Q^T A = G_{23}G_{13}G_{12}A = \begin{bmatrix} 32 & -6 & 10 \\ 0 & 16 & 16+6\sqrt{2} \\ 0 & 0 & 16-6\sqrt{2} \end{bmatrix} = R$$

The matrix can be factored without the use of trigonometry functions in the following manner.

$$A = \begin{bmatrix} 4 & 1 & 6 \\ -4 & 3 & 2 \\ 4\sqrt{2} & -\sqrt{2} & \sqrt{2} \end{bmatrix}$$

Using the modified equations

$$\begin{aligned} r^1_{1x} &= r_{1x}\frac{r_{11}}{\sqrt{r^2_{11}+r^2_{21}}} + r_{2x}\frac{r_{21}}{\sqrt{r^2_{11}+r^2_{21}}} \\ r^1_{2x} &= -r_{1x}\frac{r_{21}}{\sqrt{r^2_{11}+r^2_{21}}} + r_{2x}\frac{r_{11}}{\sqrt{r^2_{11}+r^2_{21}}} \end{aligned}$$

to define G_{12}, the matrix can be reduced to

$$G_{12}A = \begin{bmatrix} 4\sqrt{2} & -\sqrt{2} & 2\sqrt{2} \\ 0 & 2\sqrt{2} & 4\sqrt{2} \\ 4\sqrt{2} & -\sqrt{2} & \sqrt{2} \end{bmatrix}$$

Likewise, the equations

$$\begin{aligned} r^2_{1x} &= r^1_{1x}\frac{r^1_{11}}{\sqrt{(r^1_{11})^2+r^2_{31}}} + r_{3x}\frac{r_{31}}{\sqrt{(r^1_{11})^2+r^2_{31}}} \\ r^1_{3x} &= -r^1_{1x}\frac{r_{31}}{\sqrt{(r^1_{11})^2+r^2_{31}}} + r_{3x}\frac{r^1_{11}}{\sqrt{(r^1_{11})^2+r^2_{31}}} \end{aligned}$$

form G_{13} and reduce the matrix to

$$G_{13}G_{12}A = \begin{bmatrix} 8 & -2 & 3 \\ 0 & 2\sqrt{2} & 4\sqrt{2} \\ 0 & 0 & -1 \end{bmatrix}$$

Once again, using the new elements of the third column in the equations

$$\begin{aligned} r^2_{2x} &= r^1_{2x}\frac{r^1_{22}}{\sqrt{(r^1_{22})^2+(r^1_{32})^2}} + r^1_{3x}\frac{r^1_{32}}{\sqrt{(r^1_{22})^2+(r^1_{32})^2}} \\ r^2_{3x} &= -r^1_{2x}\frac{r^1_{32}}{\sqrt{(r^1_{22})^2+(r^1_{32})^2}} + r^1_{3x}\frac{r^1_{22}}{\sqrt{(r^1_{22})^2+(r^1_{32})^2}} \end{aligned}$$

defines G_{23} and the matrix is factored into

$$Q^T A = G_{23} G_{13} G_{12} A = \begin{bmatrix} 8 & -2 & 3 \\ 0 & 2\sqrt{2} & 4\sqrt{2} \\ 0 & 0 & -1 \end{bmatrix} = R$$

From this example, the dependencies in the algorithm become apparent. A FSFG representation for obtaining R in QR Factorization of a 4 X 4 matrix is is shown in Figure 5.23 with a potential hierarchy for the QR Factorization FSFG is shown in Figure 5.24.

Once again, there are no feedback loops to restrict the iteration period and schedules become bound by communication as t_c is lowered. Results for scheduling the QR Factorization FSFGs are shown in Table 5.9 for $t_c = 1$. Figure 5.25 illustrates the schedule produced for QR2 setting $P_d = 8$ and $T_d = 28$.

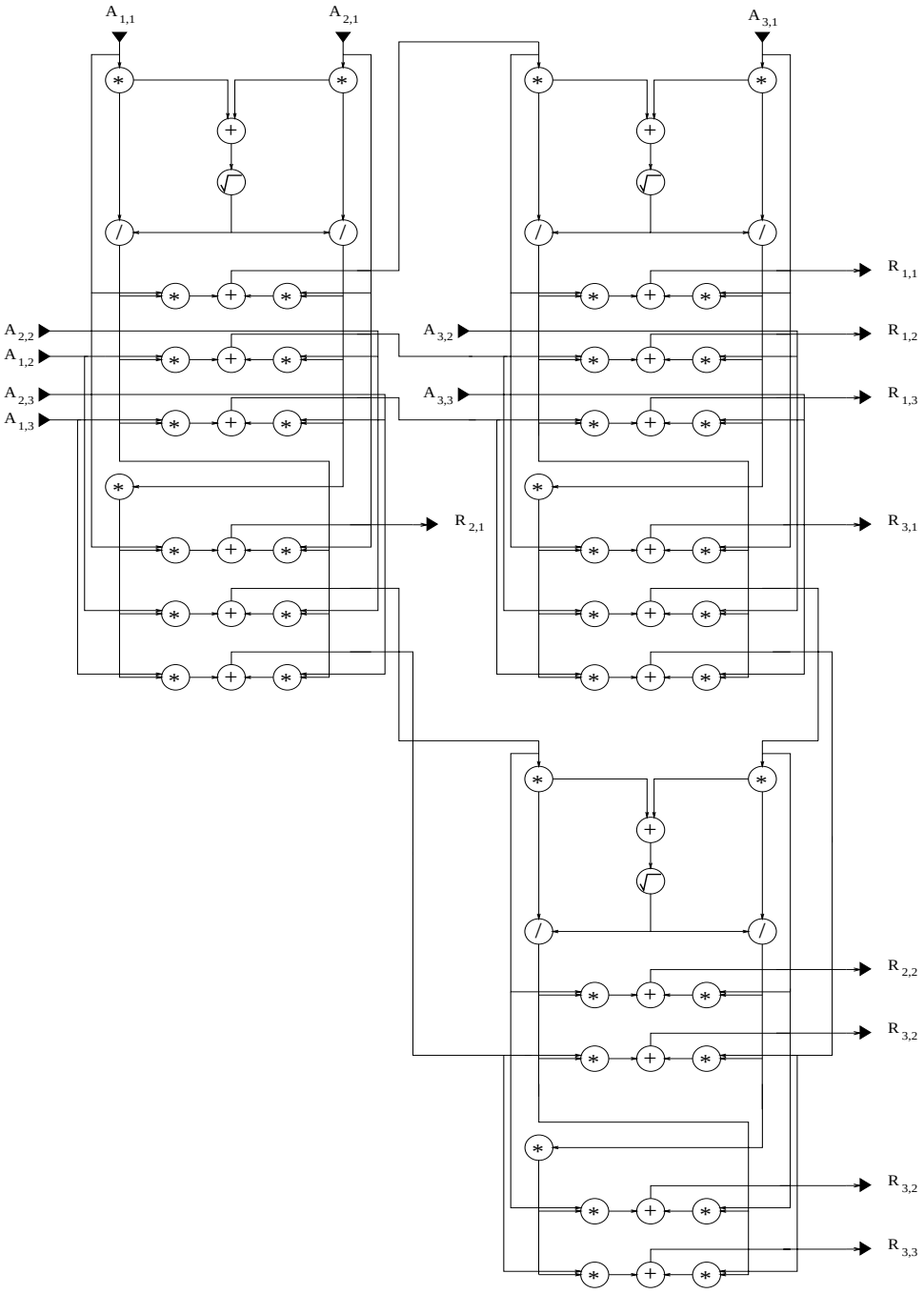


Figure 5.23: FSFG for QR Factorization of a 3X3 matrix (QR1)

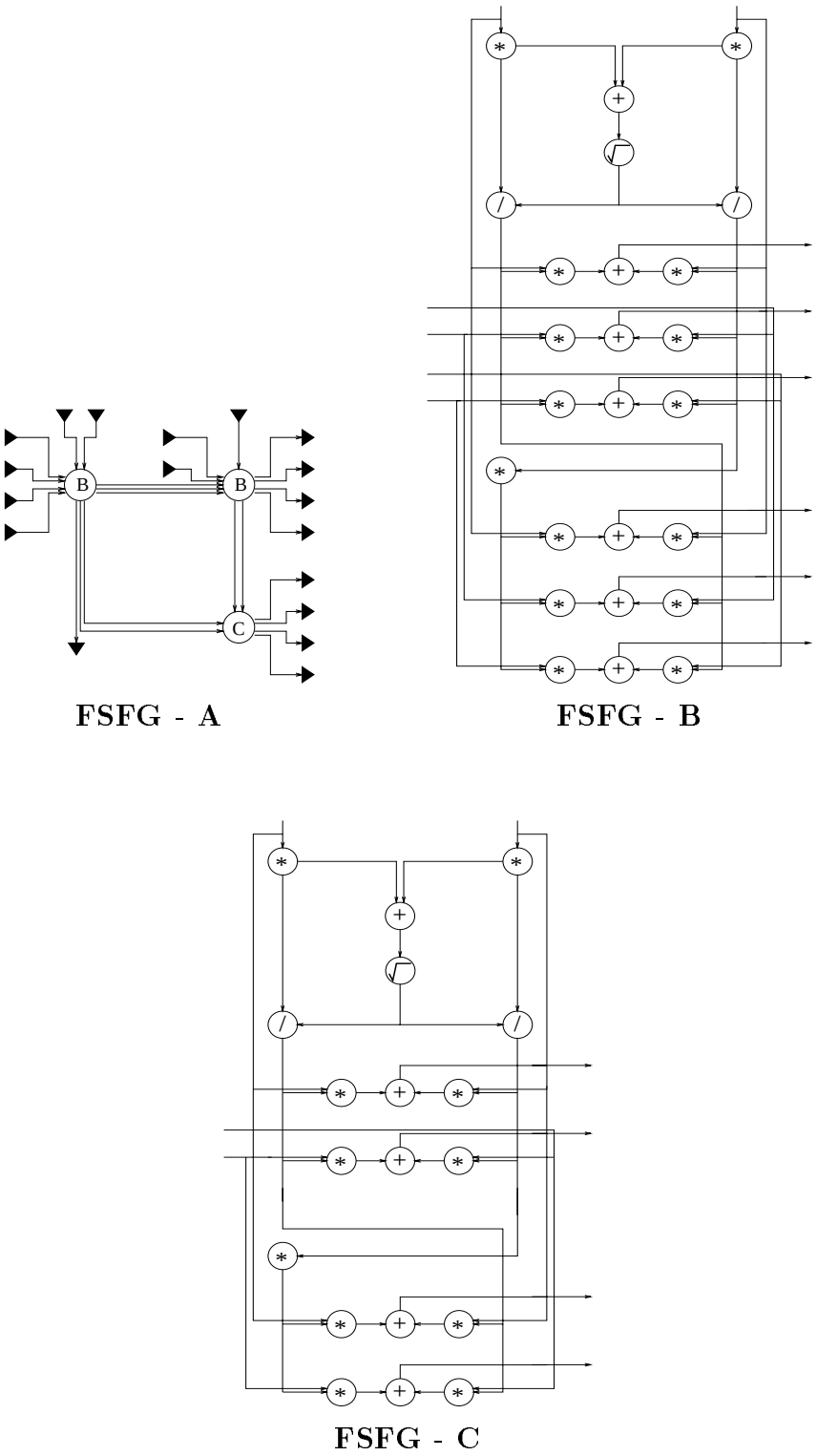


Figure 5.24: Potential FSFG hierarchical specification for QR Factorization of a 3X3 matrix (QR2)

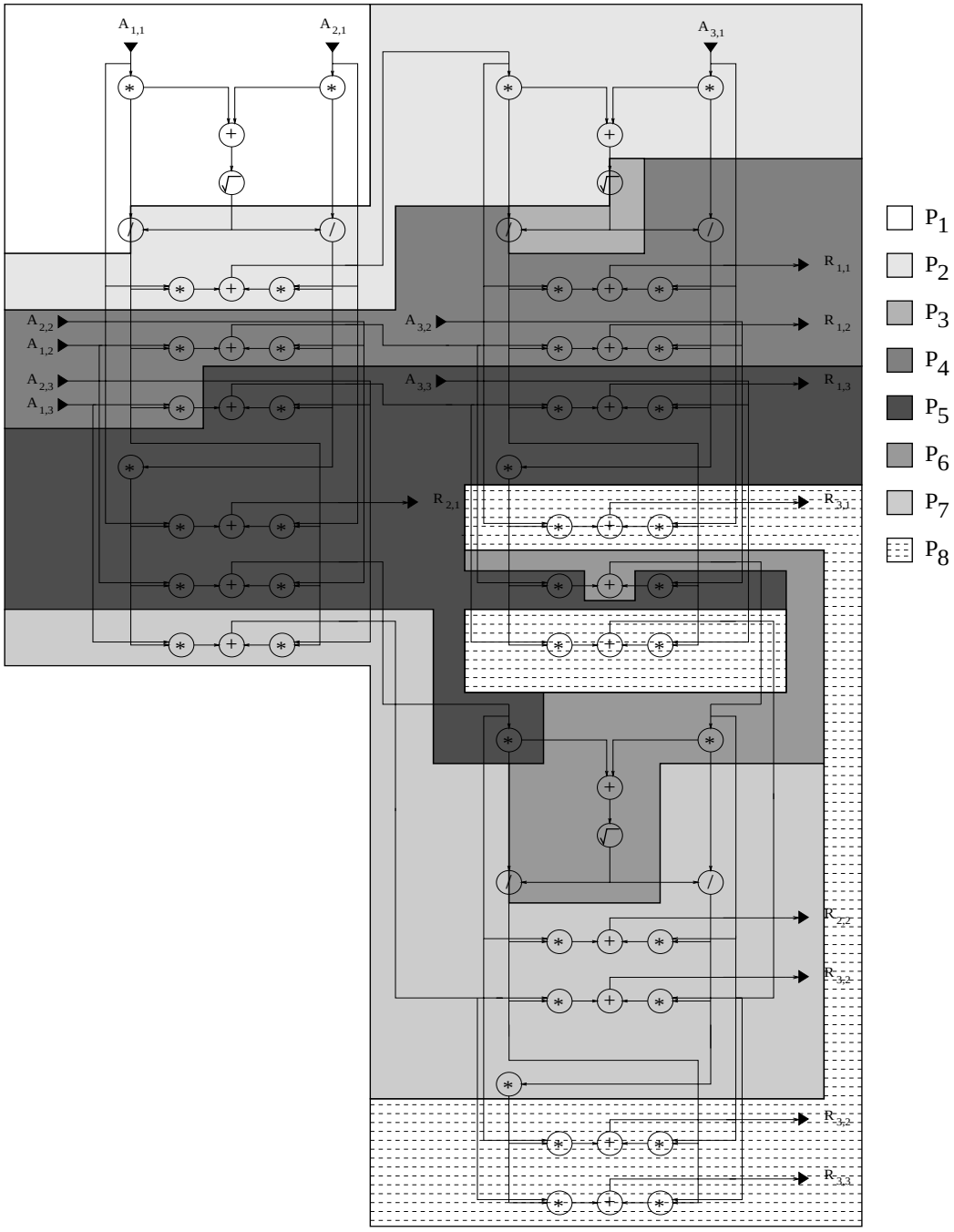


Figure 5.25: FSFG for QR Factorization of a 3X3 matrix with a software pipelining and parallel execution schedule generated by Calypso using the Order Graph Method for $P_d = 8$, $T_d = 28$, and $t_c = 1$

HFSFG	*Nodes*	*Levels*	P_d	T_d	*Linear Extensions*	*Avg. Proc. Util. (%)*	*CPU time* (sec.)	*SP*	*PE*
QR1	69	1	1	221	1	100	1.32	Y	Y
			2	111	1	99.5	1.26	Y	Y
			3	74	1	99.5	1.32	Y	Y
			4	56	1	98.7	1.26	Y	Y
			8	28	1	98.7	1.26	Y	Y
			16	14	1	98.7	1.38	Y	N
			32	7	1	98.7	1.37	Y	N
			45	5	1	98.2	1.42	Y	N
			56	4	1	98.7	1.61	Y	N
			74	3	1	99.5	1.54	Y	N
			111	**2**	1	99.5	1.60	Y	N
QR2	69	2	1	221	1	100	1.31	Y	Y
			2	111	1	99.5	1.31	Y	Y
			3	74	1	99.5	1.32	Y	Y
			4	56	1	98.7	1.32	Y	Y
			8	28	1	98.7	1.37	Y	Y
			16	14	1	98.7	1.42	Y	N
			32	7	1	98.7	1.43	Y	N
			45	5	1	98.2	1.48	Y	N
			56	4	1	98.7	1.54	Y	N
			74	3	1	99.5	1.59	Y	N
			111	**2**	1	99.5	1.65	Y	N

Table 5.9: Summary of Calypso's results for scheduling hierarchical FSFGs for QR Factorization with $t_c = 1$

5.5 Conclusions

Results from the previous sections suggest that specification of an algorithm in the form of a hierarchy has an effect on the preprocessing and partitioning stages of the OGM. The preprocessing stage does most of the work to handle hierarchical specification in dealing with composite node representations of hierarchical levels. Partitioning becomes affected during the initial identification of the relations matrix, the generation of linear extensions, and the generation of valid candidate partitionings. As shown in Table 5.7, many of the hierarchy's effects are beneficial in reducing the amount of processing time needed to obtain a multiprocessor schedule. In the case of the 2DIIR2, 2DIIR3, and 2DIIR4, the partitioning stage did not deal with the atomic nodes individually and the scheduling time was minimized.

An increase of composite nodes in the partitioning stage can potentially increase the amount of time it takes to identify a valid candidate partitioning. In the case of the 2DIIR1 FSFG for $T_d = 5$, turning the shifting for the network topology off demonstrated that a composite node may result in a series of linear extensions which can not be effectively partitioned. Because OGM does not distinguish between composite nodes which are formed from hierarchical levels and from strong components, a user-defined hierarchy has the potential to produce a composite node with the same effect. OGM could recognize user-defined composite nodes and inspect the internal nodes to determine if the composite node can be split. However, this increases the complexity of the partitioning stage and defeats the concept of composite nodes.

Partitioning is also affected by hierarchical specification in terms of limiting the mobility of nodes within a linear extension. A composite node with multiple outputs often result in multiple nodes depending on the execution of the composite node. This results in a decrease of the composite node's mobility within linear extensions. In the case of the 2nd order 2-D IIR computational primitive equations with out

shifting for network topology, a decrease of nodes' mobility due to a composite node added to the problems which prevented a valid candidate partitioning from being identified quickly. Allowing for shifting, the composite node was easily positioned into a partition which allowed a valid candidate partitioning to be produced. Hierarchical specification only increases the probability that composite nodes will hinder valid candidate partitionings from being found quickly. However, as demonstrated by the results for the hierarchical FSFG specifications in this section, the flexibility of OGM during the partitioning stage helps keep this probability low.

Judging from the results presented in the previous sections, there are some guidelines for defining a hierarchical specification. Failure to follow these guidelines will not prevent OGM from finding a parallel computing schedule; however, the number of linear extensions (and thus the execution time) may be effected. When defining a hierarchical specification for OGM, users should attempt to meet the following conditions:

- The computational weight of the nodes in the lowest level in the hierarchy should not be greater than the desired iteration period. This minimizes the number of nodes within linear extensions and the time to generate successive linear extensions during the partitioning stage of OGM.

- The hierarchy should be defined to minimize the number of communications into or out of each hierarchical level. This increases the mobility of composite nodes within linear extensions and increases the number of linear extensions which can be considered for a poset.

- A hierarchical level (l) should be defined to minimize the number of edges (the distance) between any node $v \in l$ and its nearest neighbor $w \in l$. If the hierarchical level l is reduced to a composite node (x) during the partitioning

stage, a greater distance between nodes increases the probability that other nodes will rely on x's execution. If more nodes rely on x's execution, the mobility of nodes within the linear extension will be reduced. In turn, this decreases the number of linear extensions which can be considered for a poset.

In addition to providing insight into hierarchical specification, the results in Sections 5.3 and 5.4 demonstrate the usability and effectiveness of OGM and Calypso. The average running time for OGM (with all it features on) is under 5 seconds with 97% of the schedules produced in under 2 seconds. There are no restrictions on the hierarchical specification which allows the maximum freedom to define algorithms for execution on Calypso using OGM. The results illustrate OGM's ability to specify a desired iteration period and optimize the number of processors, specify a desired number of processors and optimize the iteration period, or optimize the iteration period and the number of processors. For any target iteration period or number of processors, the results show OGM's ability to maximize the efficiency of the system (often times improving on the specified target). Variables for identifying the amount of data which processors and links in the network topology can communicate per unit of time increase the usability of OGM. Additionally, the flexibility of OGM and Calypso was exercised by preventing the splitting of atomic nodes and forcing the generation of parallel execution schedules. Together, the experimental results demonstrate OGM and Calypso's ability to schedule hierarchical FSFGs for parallel computing using software pipelining and parallel execution schedules.

We now conclude with a summary of the research presented in this dissertation and some ideas for future research.

Chapter 6

Summary and Future Research

6.1 Summary

In this research, we present the Order Graph Method (OGM) as a new methodology for automatically partitioning and scheduling digital signal processing algorithms for parallel computing. OGM accepts algorithms specified in the form of hierarchical Fully Specified Flow Graphs (FSFGs) and produces balanced schedules for parallel execution and software pipelining which exploit fine to coarse-grained parallelism. OGM schedules algorithms for rate and processor optimal execution or to meet user-specified bounds on the rate or the number of processors. In addition, OGM considers the effect of inter-processor communication and the network topology. OGM attempts to minimize inter-processor communication and validates schedules against restrictions placed on inter-processor communication due to memory limitations and contention on the network topology. We have evaluated the performance of OGM with other graph-based methodologies and have shown that OGM provides better support for scheduling issues. Scalability to larger problems, consideration of communication and network topology, support for software pipelining, and user-specified iteration period and processor bounds are some of OGM's advantages.

We implemented OGM in a tool called "Calypso" to allow experimentation with digital signal processing algorithms. Using Calypso, we demonstrated that OGM is

effective and flexible in producing schedules of DSP algorithms for parallel execution. Algorithms scheduled include a 2nd order IIR filter, a 15th order FIR filter, a 4th order lattice filter, a 2nd order 2-D IIR filter, LU Decomposition, and QR Factorization. For each of these algorithms a variety of hierarchical specifications highlights OGM's ability to effectively deal with the hierarchy and produce optimized schedules with a high average processor utilization. Runtime results for each of the algorithms shows that OGM performs well with varying algorithm sizes. In addition, the hierarchy is shown to increase OGM's effectiveness if the iteration period is greater than the computational weight of the lower levels in the hierarchy.

6.2 Future Research

While this research has contributed to our understanding of multiprocessor implementations for signal processing algorithms, application of the parallel computing schedules produced by Calypso onto the BDPA would provide additional insight into scheduling issues and the BDPA. Due to the flexibility of OGM's scheduling methodology, additional stages may be added to refine the parallel computing schedules as new insights are obtained. Some additional issues to consider include:

- The generation of linear extensions could be refined to recognize why particular candidate partitionings are not valid. There may be a way to identify composite nodes which are positioned where the partitioning algorithm will continually attempt to split the composite node. By identifying a problematic composite node, the mobility of other nodes could be targeted toward moving the problem node into a new position within the candidate partitioning. Subsequent linear extensions could then be intelligently generated to overcome the reasons previous candidate partitionings were rejected.

- Combinations of parallel execution and software pipelined schedules could be considered. OGM's communication model has been developed to support this combination; however, an investigation is needed to determine when combining the two scheduling techniques is beneficial.

- Additional support could be added for other network topologies (such as the hypercube). If the network topology has a basic interconnection network, replacing the program which checks the network topology admissibility condition may be all that is needed. However, a more complex topology with variations in the interconnection network will require modifying the partitioning and scheduling stages of OGM to consider how a partitioning should be assigned on the network topology. Some potential considerations, include how the distance and communication bandwidth between processors effects the scheduling of a partition with unbalanced communication.

- Different architectures could be investigated and incorporated into OGM and Calypso. A closer look at how pipelined processors, loosely coupled processors, and non-overlapped communication and computation effects OGM is needed. It may be possible to assign communications a weight using t_c and change the partitioning strategy to consider the communication weight during the formation of the f_i sets.

- A closer look is needed to determine the memory requirements for OGM and Calypso. It is possible that larger FSFGs will present memory problems for Calypso if composite nodes need to be expanded during the preprocessing stage. There may be a way to store a level of the hierarchy in memory and just reference the same memory location whenever an instance of the level is partitioned.

- Parameterization of FSFGs is another feature which may benefit OGM. A FSFG may be able to have parameters assigned to indicate multiple occurrences of the same graph. This would increase the usability of OGM by making it easier to define recursive algorithms. The preprocessing and partitioning stages could be rewritten to recognize the parameters and identify the precedence conditions necessary to obtain an initial ordering for the linear extension.

- A front end tool could be developed to optimize the FSFG before being scheduled by OGM. Potential issues include the elimination of duplicate calculations, retiming to eliminate node dependencies, reducing broadcast communications, and manipulating the hierarchical specification to reduce inter-level communications. Integration with a CAD tool which supports hierarchical algorithm specification could enhance the usability of Calypso by making the interface user-friendly.

- Additional work can be done to use Calypso's output schedules to generate code which can be compiled and run on multiprocessor systems. If the communication is synchronous, the communications program would need to be modified to schedule communications on processors in a specific sequence. It may be possible to use the candidate partitionings to determine the required sequence. However, when the network topology is incorporated, the communications schedule may need to compensate for data which must propagate through multiple processors. Asynchronous communications may not require this type of schedule. If communications are buffered, OGM's schedules could be used to generate code which communicates an output in the first available time slot and begins execution whenever data arrives at a processor. Because the network topology admissibility condition currently checks for the ring topology, the BDPA would be a prime candidate.

- Support for conditional branching should be investigated for integration into OGM and Calypso. For algorithms such as LU Decomposition with pivoting and LU Decomposition of a sparse matrix, conditional branching may significantly simplify the algorithm specification. In addition, this would allow OGM to be used for a more generic set of algorithms beyond digital signal processing.

- Additional communication and processing constraints may be incorporated into OGM and Calypso. For example, bounds on latency, communication buffers, system input/output connections, and broadcasting limits could be considered.

For future research, the next logical step is to use Calypso to map a variety of signal processing and matrix operations algorithms onto the BDPA to demonstrate the system's performance characteristics.

Bibliography

[1] K. Hwang. *Advanced Computer Architecture With Parallel Programming.* McGraw-Hill, San Francisco, CA, 1993.

[2] W.E. Alexander, S.T. Alexandre, and D.S. Reeves. Parallel execution of multidimensional DSP algorithms using the block data flow architecture. Internal Research Paper N.C. State University, 1993.

[3] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco, CA, 1979.

[4] S.Y. Kung. *VLSI Array Processors.* Prentice-Hall International, Englewood Cliffs, CA, 1988.

[5] E.W. Mayr. Basic parallel algorithms in graph theory. *Computer Suppl.*, 7:69–91, 1990.

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

[7] S.M. Park, W.E. Alexander, J.H. Kim, W.E. Batchelor, and W.T. Krakow. A novel VLSI architecture for the real-time implementation of 2-D signal processing systems. In *Proceedings of IEEE International Conference Computer Design: VLSI in Computers and Processors*, October 1988.

[8] Hongyu Xu. *BDFA - A Block Data Flow Architecture for Real-Time Signal Processing and Matrix Operations.* PhD thesis, North Carolina State University, Dept of Electrical and Computer Engineering, 1991.

[9] E. Barros and A. Sampaio. Towards probably correct hardware/software partitioning using OCCAM. In *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pages 210–217, 1994.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Development of a load balancing tool for the GRAPE rapid prototyping environment. In *Proceedings of the 4th International Workshop on Rapid System Prototyping*, pages 2–16, 1993.

[11] J.T. Buck. Dynamic dataflow model suitable for efficient mixed hardware and software implementations of DSP applications. In *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pages 165–172, 1994.

[12] D.D. Hills. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3:69–101, 1993.

[13] E.A. Lee. Multidimensional streams rooted in dataflow. In *Proceedings of the Workshow on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 295–306, 1993.

[14] P. Moore and P.G. O'Donoghue. Developing transputer-based systems using hood and parallel c. *Information and Software Technology*, 36(6):353–360, June 1994.

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

[16] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[17] P.H. Hartel, H. Glaser, and J.M. Wild. Compilation of functional languages using flow graph analysis. *Software-Practice and Experience*, 24(2):127– 173, February 1994.

[18] J. Ramanujam. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[19] F. Gasperoni and U. Schwiegelshohn. Scheduling loops on parallel processors: A simple algorithm with close to optimum performance. *Lecture Notes In Computer Science*, (634):625–636, 1992.

[20] K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40:178–195, February 1991.

[21] C.M. Wang and S.D. Wang. Efficient processor assignment algorithms and loop transformations for executing nested parallel loops on multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):71–82, January 1992.

[22] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[23] M. Wolfe. Automatic vectorization, data dependence, and optimizations for parallel computers. *Parallel Processing For Super-Computers and Artificial Intelligence*, pages 409–440, 1989.

[24] W-M Lin and B. Yang. Load balancing technique for parallel search with statistical model. In *Proceedings of the 1995 IEEE 14th Annual International Pheonix Conference on Computers and Communications*, pages 54–60, March 1995.

[25] M.C Chen and C.E. Mead. Concurrent algorithm as space-time recursion equations. In *Proceedings of the USC Workshop VLSI Modern Signal Processing*, pages 31–52, November 1982.

[26] V. Van Dongen and P. Quinton. The mapping of linear recurrence equations on regular arrays. *Journal VLSI Signal Processing*, 1:95–113, 1989.

[27] L. Johnsson and D. Cohen. A mathematical approach to modelling the flow of data and control in computational networks. In *CMU Conference on VLSI Systems and Computers*, pages 226–234, October 1981.

[28] W.L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32:93–114, 1984.

[29] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrence equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 208–214, 1984.

[30] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 478–491, New York, NY, 1986.

[31] P. Eades, M. Hickey, and R.C. Read. Some hamilton paths and a minimal change algorithm. *JACM*, 31:19–29, 1984.

[32] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *IEEE Computer*, pages 50–56, June 1982.

[33] S.H. Huang and J.M. Rabaey. Maximizing the throughput of high performance DSP applications using behavioral transformations. In *Proceedings of the European Design and Test Conference*, pages 25–30, 1994.

[34] W. Shen and D. Sweeting. Heuristic algorithms for task assignment and scheduling in a processor network. *Parallel Computing*, 20(1):1–14, January 1994.

[35] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under speed constraints. *IEEE Transactions on Circuits and Systems*, 28:196–202, 1981.

[36] Sonia M. Heemstra de Groot, Sabih H. Gerez, and Otto E. Herrmann. Range-chart-guided iterative data-flow graph scheduling. *IEEE Transactions on Circuits and Systems*, 39(5):351–364, May 1992.

[37] T.P. Barnwell III, V.K. Madisetti, and S.J.A. McGrath. The Georgia Tech digital signal multiprocessor. *IEEE Transactions on Signal Processing*, 41(7):2471–2487, July 1993.

[38] Bryce A. Curtis and Vijay K. Madisetti. Rapid prototyping on the Georgia Tech digital signal multiprocessor. *IEEE Transactions on Signal Processing*, 42(3):649–662, March 1994.

[39] P.R. Gelabert and T.P. Barnwell III. Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs. *IEEE Transactions on Signal Processing*, 41(2):858–888, February 1993.

[40] P. Evripidou and J.-L. Gaudiot. Block scheduling of iterative algorithms and graph-level priority scheduling in a simulated data-flow multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):398–413, April 1993.

[41] K. Hwang, Y.H. Cheng, F.D. Angers, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication. *SIAM Journal of Computing*, 18:244–257, 1989.

[42] M. Potkonjak and J.M. Rabaey. Scheduling algorithms for hierarchical data control flow graphs. *International Journal of Circuit Theory and Applications*, 20:217–233, 1992.

[43] S. Dandamudi. A comparison of task scheduling strategies for multiprocessor systems. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 423–426, 1991.

[44] K. Konstantinides, R.T. Kaneshiro, and J.R. Tani. Task allocation and scheduling models for multiprocessor digital signal processing. *IEEE Transactions on Signal Processing*, 38(12), December 1990.

[45] C.S.R. Krishnan, D.A.L. Piriyakumar, and C.S.R. Murthy. Task allocation and scheduling models for multiprocessor digital signal processing. *IEEE Transactions on Signal Processing*, 43(3):802–805, March 1995.

[46] S. Manoharan and N.P. Topham. An assessment of assignment schemes for dependency graphs. *Parallel Computing*, 21(1):85–107, January 1995.

[47] C.L. McCreary, A.A. Khan, J.J. Thompson, and M.E. McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 446–451, 1994.

[48] A.L. Wendelborn and H. Garsden. Exploring the stream data type in SISAL and other languages. In *Proceedings of the IFIP WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 283–294, 1993.

[49] J.T. Feo, D.C. Cann, and R.R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, December 1990.

[50] S. Pande, D.P. Agrawal, and J. Mauney. Scalable scheduling scheme for functional parallelism on distributed memory multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):388–399, April 1995.

[51] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: 3rd Symposium of Frontiers Massively Parallel Computation*, pages 424–433, 1990.

[52] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions On Programming Languages And Systems*, 9(4):491–542, April 1987.

[53] J.A.B. Fortes and D.I. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2:277–301, 1985.

[54] P.R. Cappello. Unifying VLSI array designs with geometric transformations. In *Proceedings of the 1983 IEEE International Conference on Parallel Processing*, pages 448–457, 1983.

[55] C.E. Leiserson and H.T. Kung. *Algorithms for VLSI Processor Arrays*. Addison-Wesley, Reading, PA, 1980.

[56] S. Manoharan and P. Thanisch. Assigning dependency graphs onto processor networks. *Parallel Computing*, 17(1):63–73, April 1991.

[57] Gavin R. Cato and Douglas S. Reeves. Parallel task scheduling using the order graph method. In *Proceedings of the 1995 IEEE 14th Annual International Pheonix Conference on Computers and Communications*, pages 69–75, March 1995.

[58] Gavin R. Cato and Douglas S. Reeves. Parallel task scheduling for the block data flow architecture. In *Proceedings of Southeastcon*, March 1995.

[59] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[60] D.B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal of Computing*, 4(1):77–84, 1975.

[61] Gara Pruesse and Frank Ruskey. Generating the linear extensions of certain posets by transpositions. *SIAM Journal of Discrete Mathematics*, August 1991.

[62] Gara Pruesse and Frank Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, April 1994.

[63] A.D. Kalvin and Y.L. Varol. On the generation of all topological sortings. *Journal of Algorithms*, 4:150–162, 1983.

[64] Y.L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *Computer Journal*, 24:83–84, 1981.

[65] V.K. Madisetti and B.A. Curtis. A quantitative methodology for rapid prototyping and high-level synthesis of signal processing algorithms. *IEEE Transactions on Signal Processing*, 42(11):3188–3208, November 1994.

[66] W.E. Alexander, D.S. Reeves, and C.S. Gloster. Parallel image processing with the block data parallel architecture. Accepted for publication in Proceedings of IEEE, 1996.

[67] D.E. Dudgeon and R.M. Mersereau. *Multidimensional Digital Signal Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[68] John G. Proakis and Dimitris G. Manolakis. *Introduction to Digital Signal Processing.* Macmillan Publishing Company, New York, NY, 1988.