This is to certify that the

dissertation entitled

ON PARTITIONING OF ALGORITHMS
FOR PARALLEL EXECUTION
ON VLSI CIRCUIT ARCHITECTURES

presented by

Michael A. Driscoll

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Electrical Engineering

David Fisher
Major professor

Date 9/7/88

ON PARTITIONING OF ALGORITHMS FOR PARALLEL EXECUTION ON VLSI CIRCUIT ARCHITECTURES

By

Michael A. Driscoll

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1988

ABSTRACT

ON PARTITIONING OF ALGORITHMS FOR PARALLEL EXECUTION ON VLSI CIRCUIT ARCHITECTURES

By

Michael A. Driscoll

Exploiting the potential of multiprocessor architectures and VLSI processor arrays requires increased understanding of the partitioning of algorithms for parallel execution. The Data Flow Scheduling (DFS) algorithm partitions acyclic data flow graphs for execution on message-based multiprocessor architectures to improve execution time and provides a vehicle for exploring the nature of the partitioning problem. The generality of the models used for graphs and multiprocessors makes the DFS applicable to a wide range of algorithms and architectures. The DFS makes use of fine-grained parallelism in the graph being partitioned and allows parallel execution within individual processors as well as among separate processors. A heuristic approach and a divide-and-conquer strategy allow large data flow graphs to be partitioned in reasonable time.

The evaluation of the DFS compared simulated execution times of graphs partitioned using the DFS algorithm with simulated execution times of random partitioning and with uniprocessor execution of the same graphs. In total, 600 simulations were conducted, varying key parameters of the graphs and multiprocessors such as the number of processors and functional units, the speed of computation relative to communication, and the topology, function, and number of nodes in the graphs. The simulation results show that graphs partitioned by the DFS execute faster than the same graphs execute on

uniprocessors. For some of the simulations, the DFS achieves near optimal partitions. In almost all cases, graphs partitioned by the DFS also execute quicker than random partitionings of the same graphs. The simulation results also suggest that the computational complexity of the DFS is $O(n^2)$, where n is the number of nodes in the graph being partioned.

Analytic models for the partitioning strategies were developed from the simulation results. The models show possible areas of improvement for the DFS algorithm and, in some cases, allow graphs to be partitioned without the use of computationally expensive simulations. The models show that interprocessor communication dominates the execution of randomly partitioned graphs, making this strategy unsuitable for architectures with powerful processors or expensive communication. The models provide a framework for partitioning some graph topologies without using the DFS and suggest several topics for further research.

To Katherine Jean Driscoll and beginnings.

ACKNOWLEDGEMENTS

Many people have provided support on the long journey to this end. Thanks to my parents and sister, David, Bette, and Karyn Driscoll for shaping my life. Thanks to my wife, Rebecca, for her unfailing love and confidence in me. Thanks to my daughter, Katie, for arriving after the completion of the dissertation. Thanks to Brian D. Musson for help in innumerable ways. Thanks to many people who pushed me in the right direction when I needed to be pushed: Will Gerard, Art Grumm, Vince Bralts, and James Resh. For their guidance and for making research fun, thanks to my guidance committee: P. David Fisher, Erik Goodman, Lionel Ni, James Resh, and Jacob Plotkin. Finally, thanks to P. David Fisher, who is always there when you need him, never there when you don't, and only criticizes things that need to be fixed.

TABLE OF CONTENTS

LIST OF TABLES ..... *viii*
LIST OF FIGURES ..... *ix*
LIST OF SYMBOLS ..... *x*
Chapter 1: Introduction 1
1.1 Problem Statement 1
1.2 Research Goal 2
1.3 Objectives 2
1.4 Thesis Overview 3
Chapter 2: VLSI Circuit Design Methodologies 4
2.1 Overview 4
2.2 The VLSI Circuit Design Process 8
2.3 The Algorithmic Design Methodology 18
2.4 Key Research Issues 20
2.5 Research Focus 21
Chapter 3: The Automatic Scheduling of Data Flow Graphs 23
3.1 Overview of the Scheduling Problem 23
3.2 Parallel Algorithm Model 24
3.3 Multiprocessor Architecture Model 26
3.4 The Data Flow Scheduling Algorithm 27
3.4.1 Overview 27
3.4.2 Initial Cluster Generation 28
3.4.3 Global Partitioning 30
3.4.4 Auxiliary Partitioning 31
3.4.5 Cluster Scheduler 32
3.4.6 Properties of the Data Flow Scheduling Algorithm 32
Chapter 4: Evaluating the DFS Algorithm 34
4.1 Approach 34
4.2 Evaluation Procedure and Criteria 35
4.3 Simulation Results 44
4.3.1 Binary Merge Graphs 44
4.3.2 FFT Graphs 50
4.3.3 Random Graphs 54
4.4 Conclusions 65
Chapter 5: Modeling Multiprocessor Execution Times 66
5.1 Overview 66
5.2 General Model 67
5.3 Modeling Execution Times for Binary Merge Graphs 72

5.3.1 Uniprocessor Execution 72
5.3.2 Random Partitioning 76
5.3.3 DFS Partitioning 79
5.4 Modeling Execution Times for FFT Graphs 88
5.4.1 Uniprocessor Execution 88
5.4.2 Random Partitioning 91
5.4.3 DFS Partitioning 92
5.5 Modeling Execution Times for Random Graphs 99
5.5.1 Uniprocessor Execution 99
5.5.2 Random Partitioning 102
5.5.3 DFS Partitioning 105
5.6 General Comments on Modeling Execution Times 105
Chapter 6: Summary and Conclusions 107
LIST OF REFERENCES 110

LIST OF TABLES

Table 1. Parameters for Fast and Slow Functional Units 43
Table 2. Multiprocessor Architectures Used in Simulations 43
Table 3. Uniprocessor Execution of Binary Merge Graphs 45
Table 4. Multiprocessor Execution of Binary Merge Graphs 46
Table 5. DFS Execution Time for Binary Merge Graphs 48
Table 6. Uniprocessor Execution of FFT Graphs 51
Table 7. Multiprocessor Execution of FFT Graphs 52
Table 8. DFS Execution Time for FFT Graphs 53
Table 9a. Uniprocessor Execution of First Set of Random Graphs 55
Table 9b. Uniprocessor Execution of Second Set of Random Graphs 55
Table 9c. Uniprocessor Execution of Third Set of Random Graphs 56
Table 9d. Uniprocessor Execution of Fourth Set of Random Graphs 56
Table 10a. Multiprocessor Execution of First Set of Random Graphs 57
Table 10b. Multiprocessor Execution of Second Set of Random Graphs 58
Table 10c. Multiprocessor Execution of Third Set of Random Graphs 59
Table 10d. Multiprocessor Execution of Fourth Set of Random Graphs 60
Table 11a. DFS Execution Time for First Set of Random Graphs 61
Table 11b. DFS Execution Time for Second Set of Random Graphs 62
Table 11c. DFS Execution Time for Third Set of Random Graphs 63
Table 11d. DFS Execution Time for Fourth Set of Random Graphs 64
Table 12. Model for Binary Merge Graphs on Uniprocessors 75
Table 13. Model for Random Partitioning of Binary Merge Graphs 78
Table 14. Model for DFS Partitioning of Binary Merge Graphs 86
Table 15. Model for FFT Graphs on Uniprocessors 90
Table 16. Model for Random Partitioning of FFT Graphs 92
Table 17a. Model First Set of Random Graphs on Uniprocessors 100
Table 17b. Model for Second Set of Random Graphs on Uniprocessors 100
Table 17c. Model for Third Set of Random Graphs on Uniprocessors 101
Table 17d. Model for Fourth Set of Random Graphs on Uniprocessors 101
Table 18a. Model for Random Partitioning of First Set of Random Graphs 103
Table 18b. Model for Random Partitioning of Second Set of Random Graphs 103
Table 18c. Model for Random Partitioning of Third Set of Random Graphs 104
Table 18d. Model for Random Partitioning of Fourth Set of Random Graphs 104

LIST OF FIGURES

Figure 1. The Y-Chart 5
Figure 2. Design of a Low Pass Filter 7
Figure 3. Macro-Cell Methodology 9
Figure 4. Systolic Methodology 11
Figure 5. Silicon Compilation Methodology 13
Figure 6. The Algorithmic Design Methodology 19
Figure 7. The Data Flow Scheduler 29
Figure 8. Fully Parallel Graph 37
Figure 9. Fully Sequential Graph 37
Figure 10. Binary Merge Graph for Eight Inputs 42
Figure 11. FFT Graph for Four Inputs 42
Figure 12. Example of Model Usage 70
Figure 13. Example Augmented Graph 70
Figure 14. Gantt Chart for Example Graph 71
Figure 15. Example Binary Merge Graph 73
Figure 16. Gantt Chart for Single Functional Units 74
Figure 17. Gantt Chart for Double Functional Units 74
Figure 18. DFS Two Processor Partitioning of Binary Merge Graph 80
Figure 19. DFS Four Processor Partitioning of Binary Merge Graph 81
Figure 20. Gantt Chart for Two Processors 82
Figure 21. Gantt Chart for Four Processors 83
Figure 22. Illustration of DFS Merging Heuristic 84
Figure 23. Example FFT Graph 89
Figure 24. Gantt Chart for Uniprocessor Execution 90
Figure 25. DFS Partitioning of 8-Input FFT Graph 95
Figure 26. DFS Partitioning of 4-Input FFT Graph 95
Figure 27. Numbered 8-Input FFT Graph 98

LIST OF SYMBOLS

O	set of recognized scalar operations.
$s_O(O_i)$	operand size for operation O_i.
t_{O_i}	computation time for operation O_i.
G	a data flow graph.
T_G	the set of nodes in data flow graph G.
A_G	the set of arcs in the data flow graph G.
O_T	a vector giving the funcion of each node in T_G.
$s_A(A_i)$	arc size for arc A_i.
n_P	number of processors in a multiprocessor architecture.
F_O	a vector giving the number of functional units of each type at a single processor.
t_{gc}	global communication time.
w_{gc}	global communication data block width.
t_{lc}	local communication time.
X	a vector giving the execution time for each type of functional unit.
τ	execution time for a given partitioning of a graph on a given multiprocessor architecture.
τ^0	reference execution time.
S	a measure of speedup.
S_P	processor speedup.
S_B	base speedup.
n_F	the number of functional units of each type at a single processor.
α_{gc}	the fraction of nonoverlapped global communications.
n_{gc}	the number of global communication arcs for a given graph partitioning.
α_{O_i}	the fraction of nonoverlapped execution of nodes performing operation O_i.
n_{O_i}	the number of nodes performing operation O_i in a graph.
α_{lc}	the fraction of nonoverlapped local communications.
n_{lc}	the number of local communication arcs for a given graph partitioning.
τ_l	a lower bound on graph execution time.
τ_u	an upper bound on graph execution time.

Chapter 1: Introduction

1.1. Problem Statement

The technology used to fabricate Very Large Scale Integrated (VLSI) circuits has improved dramatically in the last ten years. Current fabrication technology provides the potential for the creation of remarkably complex and powerful digital circuitry contained within a single integrated circuit chip. The availability of megabit random access memories and microprocessors made up of hundreds of thousands of transistors is but one example of the results of this technological progress.

The potential power of VLSI circuits offers great promise for the creation of Application Specific Integrated Circuits (ASIC's). In the past, the high cost of fabrication has limited integrated circuit production to general-purpose, programmable designs, which can be sold in large quantities. ASIC's, however, are designed to perform a single function and can be highly optimized for space, power, and speed. This specialization limits the potential market for ASIC's, and thus low non-recurring engineering (NRE) and fabrication costs are a prerequisite for their creation.

The reduction of fabrication costs has increased the importance of limiting NRE costs, including the high cost of human designers, which are beginning to dominate the total cost of producing VLSI circuits. As designs have become more and more complicated, the amount of human design time needed to create the designs has risen dramatically. This increase in complexity increases the likelihood of design errors, which also increases NRE costs. The successful creation of an integrated circuit requires knowledge in many areas, from theoretical computer science to solid-state physics. No single person is likely to possess this knowledge, so teams of designers are often used. Design teams can be expensive as miscommunication between group members can greatly increase the chance of faulty designs.

Design methodologies can help human designers in managing the complexity of the

VLSI circuit design process. Such methodologies prescribe a rigid sequence of steps to be followed in the creation of an integrated circuit. Following these steps can reduce wasted and duplicated effort, reducing design time dramatically. Design methodologies can also help reduce design errors by verifying that, at each step in the design process, the design meets specifications.

Even with the use of design methodologies, currently human designers are not fully exploiting the capabilities offered by VLSI circuit fabrication technology. There is a very clear need for design methodologies that can be automated. Systems based on such design methodologies can manage the details of the design, leaving the designer free to concentrate on high level design issues.

1.2. Research Goal

The overall goal of this research is to develop and evaluate a VLSI circuit design methodology suited for automation. This methodology would allow the designer to specify design behavior at a high level of abstraction and would produce a fixed (i.e., non-programmable) VLSI architecture suited for implementation as an ASIC. The evaluation of the design methodology and its suitability for automation requires the investigation of several difficult problems in computer engineering.

1.3. Objectives

Because of the great complexity inherent in any VLSI circuit design methodology, the research to be presented in this dissertation will focus on the following research objectives:

1. Define a design methodology as the basis for an automated VLSI circuit design system.
2. Identify the basic issues crucial to the automation of the design methodology.
3. Examine one of these basic issues in detail; specifically, develop an algorithm to efficiently partition parallel algorithms for execution on message-based

multiprocessor systems (and hence for implementation as ASIC's).

4. Evaluate the performance of the partitioning algorithm.
5. Model the results of applying the partitioning algorithm. These models will help improve the algorithm, reduce the need for computationally expensive simulations, and provide further insight into the problem.

1.4. Thesis Overview

Chapter 2 begins with an examination of some current design methodologies, highlighting some of their shortcomings. Then a review of research in areas important to the VLSI circuit design process is presented. This leads to the definition of a new VLSI circuit design methodology and the identification of key issues that must be investigated if the methodology is to be automated. The automatic partitioning of parallel algorithms for multiprocessor execution is selected as the focus of the dissertation.

Chapter 3 presents a model for message-based multiprocessor architectures and presents an algorithm for automatically partitioning parallel algorithms, expressed as data flow graphs, for execution on these architectures. Chapter 4 evaluates the performance of the partitioning algorithm by applying it to a number of test graphs and multiprocessor architectures and simulating the execution of the resulting partitions. Chapter 5 develops models for the simulation results presented in Chapter 4. These models allow for the accurate prediction of the execution times of some of the algorithms partitioned by the partitioning algorithm. Finally, Chapter 6 presents conclusions, reviews the usefulness of the research presented herein, and points out directions for future research.

Chapter 2: VLSI Circuit Design Methodologies

2.1. Overview

The design of an integrated circuit generally begins with a specification of the circuit's desired behavior. This specification gives a functional description of the design, along with requirements for speed, chip area, power, testability, etc. A designer or team of designers translate the specification into a physical implementation. Many paths are possible in this translation. Each such path is a design methodology, i.e., a set of steps used to transform a behavioral specification into a physical device.

Gajski [Gajs83] first presented the Y-chart as a method for defining different design methodologies. A Y-chart consists of three intersecting axes, as illustrated in Figure 1. Each axis gives a different type of representation for a design in progress. Each type of representation emphasizes certain facets of the design, while de-emphasizing other aspects. Points closer to the origin of the axes give less abstract representations of the design. The finished product can be seen as occupying the origin, at the lowest level of abstraction, and containing all facets of the design. Note that all possible representations need not be shown on an axis. Often, only the representations used by the set of methodologies under consideration will be shown.

The behavioral axis depicts representations of a design's specification, e.g., the design's function, power requirements, interface protocols, etc. These representations give no information about how the specifications will be met by the actual device. Some examples of behavioral descriptions include algorithms, state diagrams, and Boolean equations. The structural axis shows representations that define how a design achieves the design specification, depicting the flow of data and control through ideal components. Structural representations may be used to develop preliminary estimates of the design's ability to achieve the design specification. Examples of these representations include schematics and block diagrams. The geometrical axis gives representations that are concerned solely with the physical nature of the design. No information is included

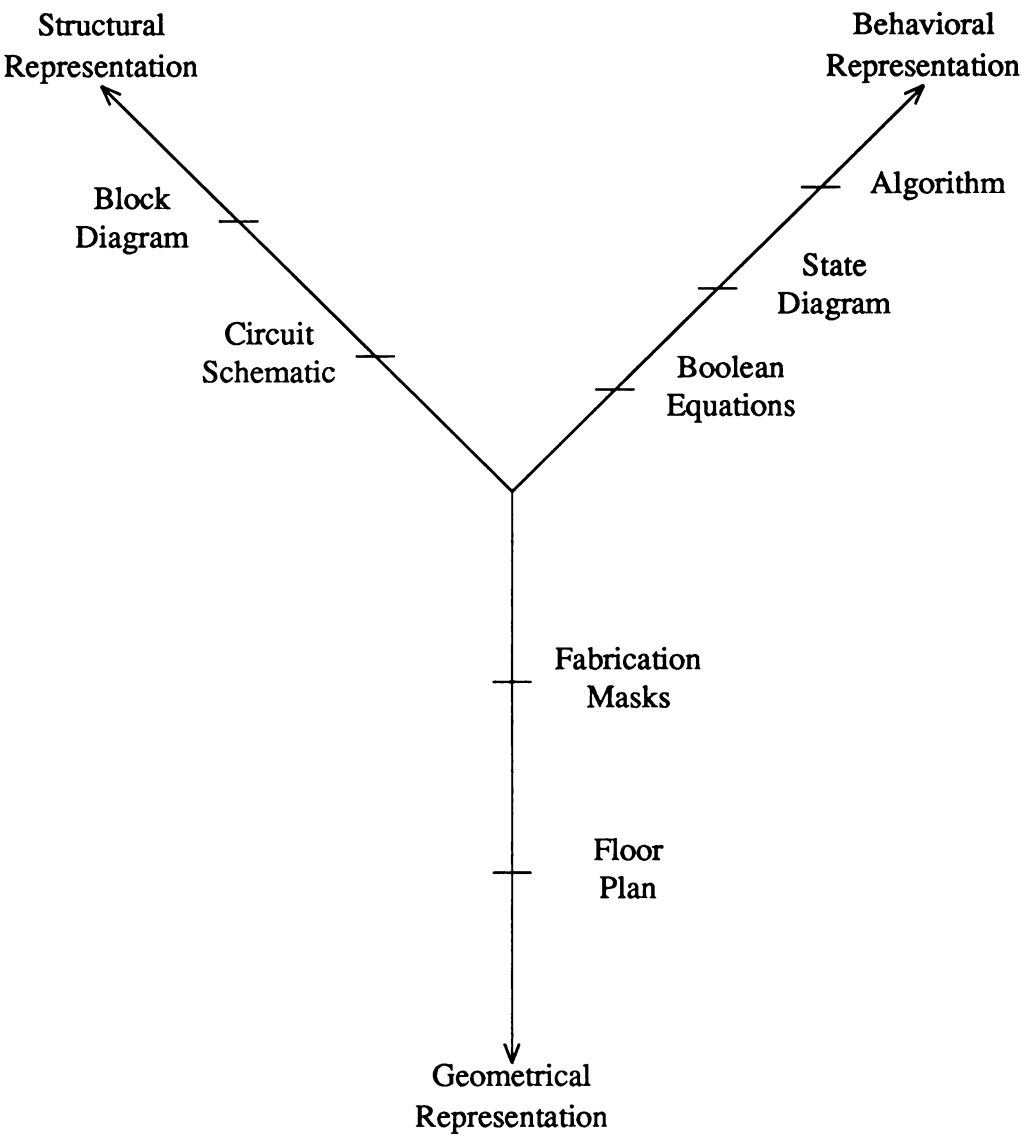

Figure 1. The Y-Chart

describing the design's behavior. Chip floor plans and layout masks are examples of geometrical representations.

The path taken by a design created using a specific design methodology is shown by placing three types of arcs on a Y-chart. Arcs that move from higher to lower levels of abstraction (on the same or different axes) indicate a refinement of the design as it moves towards final implementation at the chart's center. These arcs map a more abstract representation to a more concrete representation. Arcs that move from lower to higher levels of abstraction show a verification that the lower level representation accurately corresponds to the more abstract representation, i.e., that the mapping from the higher to lower level preserved the design specification. Finally, arcs that loop at a single representation show simulation used to determine the characteristics of a design in that representation.

The Y-chart can be useful in describing design processes other than those used for VLSI circuit design. Consider the design of a simple low-pass filter, for example, to illustrate the general form of a design methodology displayed on a Y-chart. The behavior of the low-pass filter may first be given as a set of parameters, such as low frequency gain and cutoff frequency. Figure 2 shows this as a point on the behavioral axis. The next step in the design may be to develop a transfer function for the filter that produces the desired behavior. In the figure, the development is shown as an arc from the parameter representation to the transfer function, which is still a behavioral representation. The transfer function is then used to develop a circuit schematic, consisting of ideal circuit elements such as operational amplifiers and capacitors. This representation describes how a design achieves its behavior, but gives no details of the physical implementation of the design. At this level some simulation may take place, as shown by the looped arc. The simulation results may be compared with the transfer function to verify the correctness of the schematic. The development of a geometrical representation might then proceed by selecting actual circuit components and developing a plan for placing the

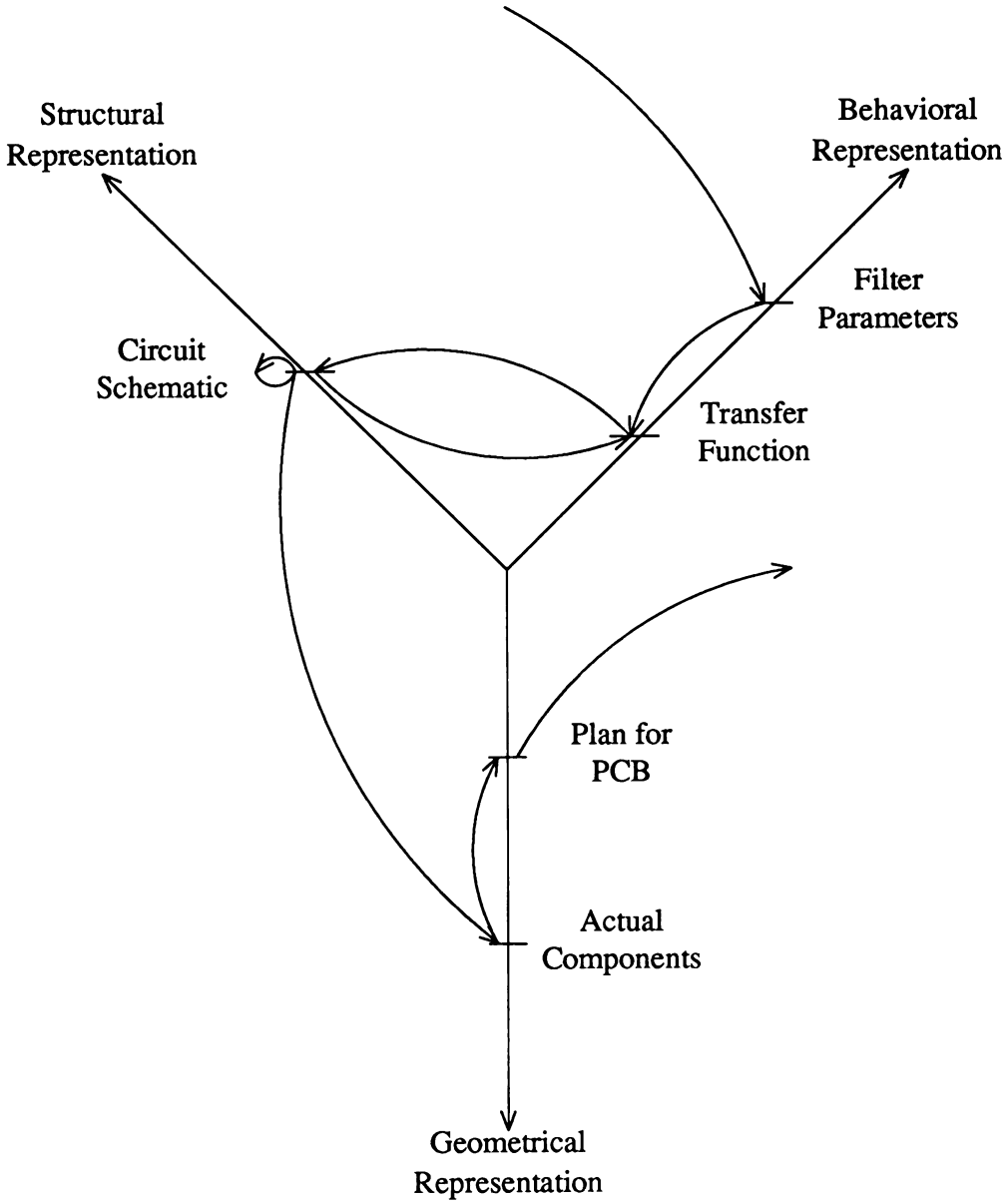

Figure 2. Design of a Low Pass Filter

components and routing their connections on a printed circuit board, resulting in a physical implementation evincing the desired behavior.

The development of a VLSI circuit design methodology requires the specification of steps that translate a design from a behavioral description to a physical device. Section 2 of this chapter examines some current design methodologies, highlights some of their shortcomings, and discusses previous work in mapping parallel algorithms to parallel systems, which will be the focus of this dissertation. Section 3 defines a new VLSI circuit design methodology in light of the information in Section 2. Section 4 identifies key issues that must be investigated in the implementation of the new design methodology. Finally, Section 5 identifies one of these issues as the focus of this dissertation.

2.2. The VLSI Circuit Design Process

Several VLSI circuit design methodologies are currently used. The macro- (or standard-) cell methodology is illustrated in Figure 3. This methodology is used in many commercial products, including those from LSI Logic and VLSI Technologies [Andr88]. In this methodology, a library of macro-cells is available to the designer. The cells can be used as basic building blocks for a design and include register-transfer level cells, random access memories, read-only memories, simple microprocessors, and cells providing functionality equivalent to that found in standard TTL integrated circuits. Many macro-cell based systems also include cell generators for creating some basic blocks, such as multipliers of various sizes. The designer can view these generators as another set of building blocks available in the cell library.

Given the macro-cell library, the designer proceeds to create a block diagram of a circuit that implements the desired behavior. The blocks in the diagram correspond to cells available in the macro-cell library. The designer creates the block diagram using standard digital design techniques. The block diagram can be simulated and it can be verified that the diagram implements the desired function. The block diagram is then translated (perhaps automatically) into a floor plan for an integrated circuit, which gives

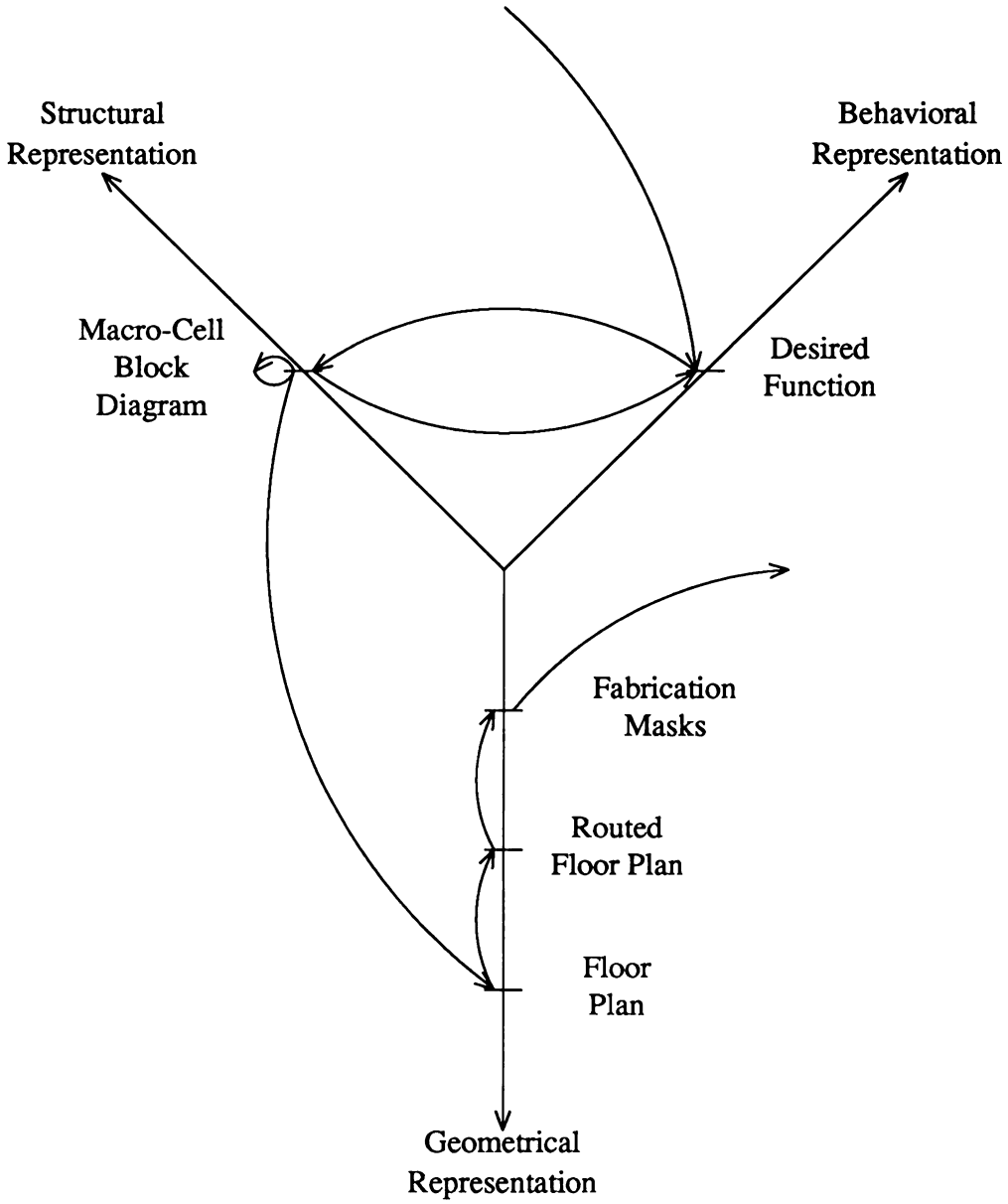

Figure 3. Macro-Cell Methodology

the placement of the macro-cells present in the diagram. Interblock connections are then routed. (Again, this may be done automatically.) Finally, fabrication masks for the integrated circuit are created and a chip is fabricated.

Benefits of the macro-cell methodology include the familiarity to designers of the block diagram approach and the success that has been achieved in automating the translation of the block diagram into a physical device. A problem with this method is the difficulty in automating the development of a block diagram.

Another popular design methodology is the systolic methodology, which is illustrated in Figure 4. The basic concepts of this methodology originated with Kung [Kung82]. In this methodology, behavior is initially specified as an algorithm. The algorithm is translated to systolic form by parallelizing and pipelining transformations. Note that not all algorithms may be translatable. A structural representation is created by mapping the systolic algorithm to a systolic architecture. A systolic architecture consists of an array of processors which are connected only with their nearest neighbors. A systolic architecture contains only a few types of processors (ideally, only a single type). The systolic architecture can be used to develop a physical design in a number of ways, including using the macro-cell design methodology outlined above.

The systolic design methodology has been used successfully in the creation of many application specific integrated circuits. Moldovan and Varma [Mold83a,Mold83b] developed a framework for describing systolic algorithms and their mapping to systolic architectures. In their approach, the algorithm is viewed as embedded in a multidimensional space with each point representing a single instance of the computation. Each point in this space can be viewed as being labeled with an n-tuple of loop indices, where n is the number of nested loops in the algorithm. Data dependencies are represented as directed arcs between the points in the space. The dependencies of the algorithm may be manipulated to place the algorithm into systolic form.

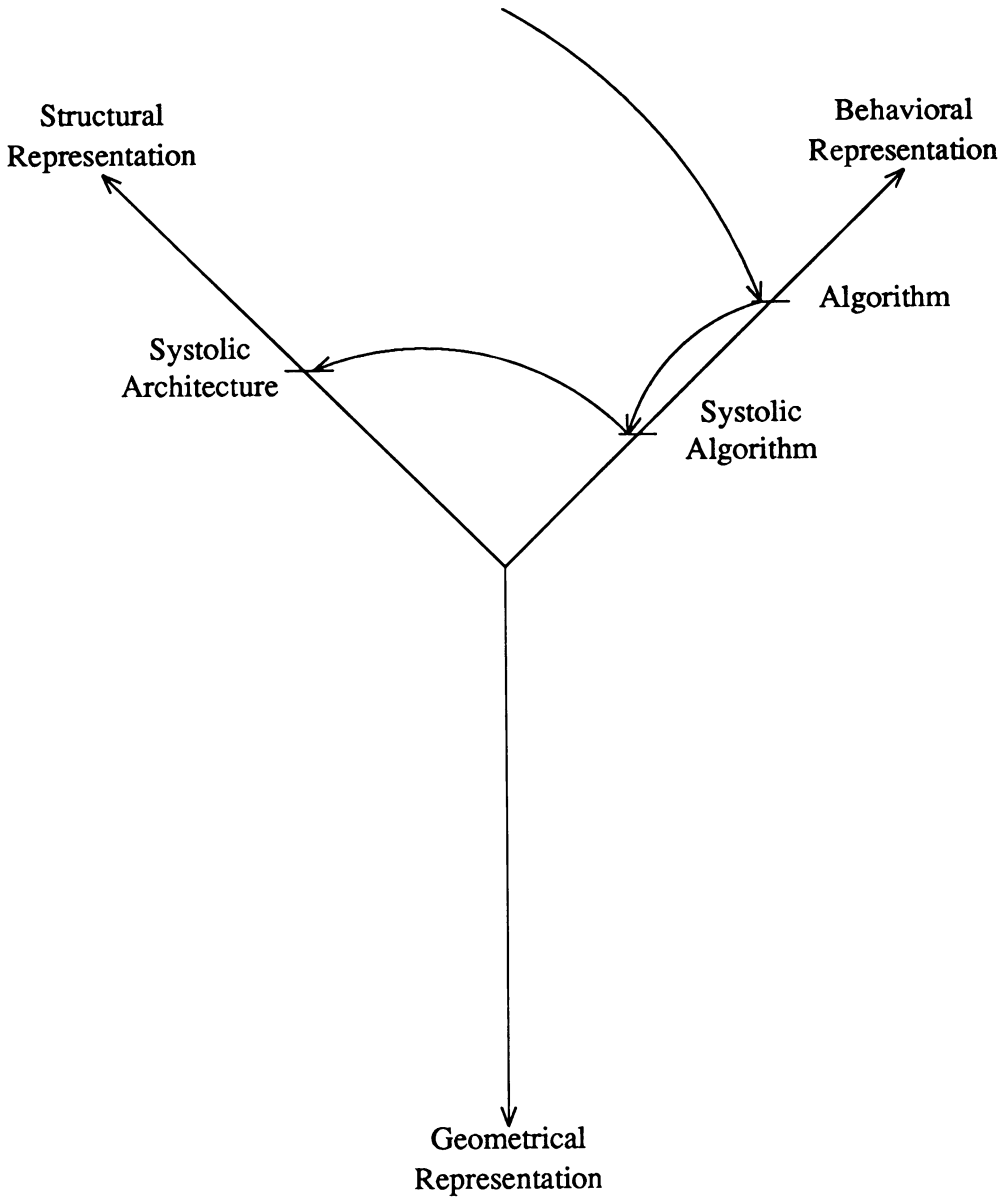

Figure 4. Systolic Methodology

The algorithm and its data dependencies, as represented in the multidimensional index space, are then mapped to three (or four) dimensional space, i.e., two (or three) spatial dimensions and time. This mapping gives a high level plan for the layout of an integrated circuit. Some examples of this methodology can be found in Quinton[Quin84],which deals with algorithms expressed as uniform recurrent equations, and in Chen [Chen86a,Chen86b], which show the use of the method for the design of a dynamic programming solver and a design to perform LU matrix decomposition.

The work of Faroughi and Shanblatt [Faro87a,Faro87b] automates the mapping of a systolic algorithm from the algorithmic index space to real space and time. In their approach, the designer must first transform the algorithm to a standard form, in which primitive computations are grouped to balance the processing load. A set of projection vectors are then calculated. Each such vector results in a systolic architecture that implements the algorithm. The performance of these architectures can be evaluated to select an architecture that best meets the design requirements. This design procedure solves a major part of the automated mapping problem for a limited but important class of algorithms.

Silicon compilation is a final example of a VLSI design methodology. Module generators, similar to the cell generators mentioned in the description of the macro-cell methodology above, can provide the basis for this methodology, which is illustrated in Figure 5. The original goal of silicon compilation was, as the name implies, to allow the designer to specify functionality at a high level of abstraction and produce a working integrated circuit. Some first attempts at this difficult problem include the bristle blocks system [Joha79] and the Carnegie-Mellon University design system [Park79]. When the enormity of the problem became apparent, the problem was limited to automatically producing designs for a limited class of behaviors. Currently, silicon compilers are limited to generating specific components such as read-only memories, programmable logic arrays, etc. These "silicon compilers" are currently available in many commercial

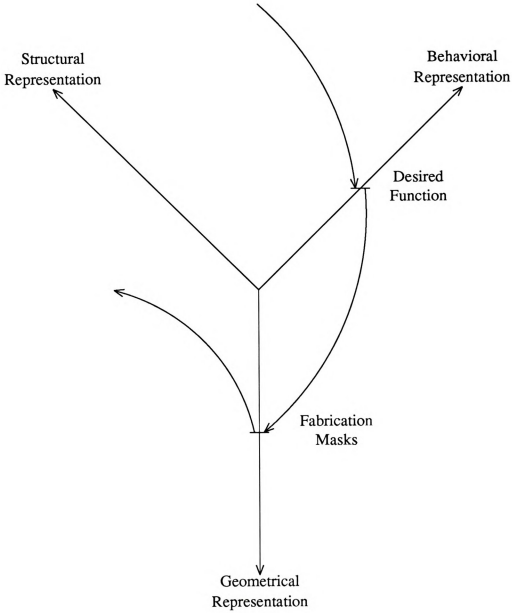

Figure 5. Silicon Compilation Methodology

design systems including those from LSI Logic, VLSI Technologies, and Silicon Compilers, Inc. [Andr88]. Peg, a system for automatically generating programmable logic array based finite-state machines, is available as part of the well-known University of California at Berkeley VLSI design tools [Scot85].

Current design methodologies have helped reduce the amount of costly human design time needed to develop an integrated circuit. However, much work remains to be done. The example methodologies discussed above illustrate several important points concerning design methodologies. First, notice that a design methodology need not deal with the entire design process to be successful. The systolic methodology and the macro-cell methodology, for example, deal with different portions of the design process and can be used together in creating a finished design. Second, the ability to estimate a design's performance at a high level can help speed the design process. If the chip area and processing speed of a systolic architecture can be estimated without generating a physical layout for the circuit, many possible architectures can be evaluated without incurring the expense of creating layouts for each possibility. Third, design methodologies currently in use are limited to certain classes of design, e.g., systolic architectures. Finally, some steps of current design methodologies are not easily automated, e.g., the creation of a block diagram for the macro-cell methodology. Each of these points will be discussed in more detail in the following paragraphs.

The first point made above suggests limiting the focus of any new design methodology to those aspects of the design process that are not well understood. The selection of an appropriate structural representation, for example, will allow the use of the already developed macro-cell methodology, thus reducing the portion of the design task with which a new methodology must deal. This representation should be similar to the systolic architecture, which has already been successfully used in designs. For generality, it should remove the limits on processor types and interprocessor connections that the systolic architecture imposes. A good example of this is the architecture of the CHiP com-

puter, as proposed by Snyder [Snyd82,Snyd84]. The CHiP computer is made up of a rectangular grid of processing elements, with adjacent rows and columns separated by rows and columns of programmable switches. An algorithm to be executed on the CHiP computer is modeled by five components: A graph whose edges give the communication of the algorithm and whose vertices represent processors; a set of process types which defines the types of computation performed by the algorithm; an assignment of process types to processors; a definition of the type of synchronization between the processors; and a description of the expected input/output data types for each processor.

The second point above mentions the desirability of estimating performance from a structural representation. Several VLSI circuit models have been proposed and used to develop theoretical limits on the performance of circuits implementing a specific algorithm. The best example of this work was presented by Thompson [Thom79,Thom80], who derived good upper and lower bounds on the product of chip area and the square of execution time. The low-level (i.e. physical) nature of these models makes them unsuitable for estimating performance from structural level models.

The third point made above concerns the lack of generality in current methodologies. Miranker and Winkler [Mira84] propose representing algorithms as general graphs in an effort to overcome the limits imposed by the systolic methodology. When using a graphical algorithmic model and a graphical structural model, such as an abstraction of the CHiP computer, the development of a structural representation from a behavioral representation become a problem of embedding one graph into another.

The problem of embedding of one graph (the source) into another graph (the host) has been studied extensively in recent years, both in a theoretical context and with a view towards applications in VLSI circuitry. Ellis [Elli84] and Simonson [Simo86] treat the topic theoretically. Since most graph embedding problems are NP-hard, these works attempt to develop good heuristic algorithms for limited classes of source graphs, host graphs, or both. Aleliunas and Rosenberg [Alel82] develop several techniques for

embedding rectangular grids in square grids. Ellis improves some of these algorithms. This work can be used to square up highly eccentric VLSI circuit layouts. This allows the area and delay of rectangular layouts to be evaluated to within a constant factor of a corresponding square layout.

Leiserson [Leis80] and Valiant [Vali81] develop a technique for embedding into grids any class of graphs with an acceptable separator theorem. A separator theorem defines the minimum number of edges that must be removed from a graph to form two disjoint subgraphs belonging to the same graph family as the parent graph (e.g., two subtrees of a parent tree). The technique uses a recursive divide-and-conquer approach followed by a rejoining of the subproblems to achieve efficient embeddings.

The final point made above calls for new methodologies that are more easily automated than those currently available. Unfortunately, the high complexity of most graph embedding problems does not help to achieve this goal. To simplify the task of automating the methodology, while still retaining the generality of the graph-based approach, it can be helpful to split the embedding task into multiple steps. Mendelson and Silberman [Mend87] and Koren and Silberman [Kore83] first partition a data flow graph into layers, map each layer to a separate row of their hexagonal array, ordering the nodes in the layer in the process, and finally route interprocessor connections. The overall usefulness of their method is limited by their processor model, a hexagonal array of processing elements in which communications are routed through processing elements, making them unusable for computation.

The problem of partitioning an algorithm for parallel execution is obviously related to the problem of scheduling tasks for execution in parallel and distributed multiprocessor systems. Efe [Efe82] summarizes the conflicting requirements such a scheduler must meet to achieve good performance. First, the system must minimize interprocessor communication, which is generally more time consuming than intraprocessor communication. This goal can be satisfied in the extreme by assigning all tasks to a single processor.

Second, the system must try to balance the processing load among all the processors in the system. Third, the tasks should be scheduled to take advantage of the parallelism existing in the set of tasks.

Processor scheduling has a long history of study. Some key points relative to the current problem are mentioned here. First, most scheduling algorithms require a priori knowledge of the execution times of each task to be executed, as well as the intertask communication requirements. Chou and Abraham [Chou82], Efe [Efe82], Ravi, et al. [Ravi87], and Ho and Irani [Ho83] all require full knowledge of task execution times. This is usually not practical, given the high granularity of most tasks being scheduled. However, for VLSI circuit design, tasks can be described at a low level of granularity, allowing detailed knowledge of execution time requirements.

A second important fact about scheduling is that most scheduling problems are NP-hard. Thus most researchers have developed heuristic algorithms for scheduling. A classic example is the critical path scheduling algorithm, first described by Hu [Hu61] and later evaluated by Adam, et al. [Adam74] and Kohler [Kohl75]. This algorithm is examined by Granski, et al. [Gran87] in determining its efficiency when implemented in the hardware of a data flow processor. The use of the critical path scheduling heuristic for scheduling in a data flow processor is examine by Ho and Irani [Ho83] and was found to be the most promising of the heuristics tested.

Finally, very little work has been done to quantify the influence of the many variables present in multiprocessor systems on the schedules that result from various algorithms. This lack of information makes it very difficult to select appropriate algorithms for specific situations. Indeed, in some cases randomly scheduling tasks to processors may achieve better results than computationally intensive heuristic algorithms [Dris88].

The work discussed above points out several deficiencies in current VLSI circuit design methodologies. The nature of high level VLSI circuit design methodologies is closely related to that of executing algorithms on multiprocessor systems. Key problems

in both areas include the parallelization of serial algorithms and the mapping of algorithms to multiprocessor systems (or VLSI processor arrays). A problem specific to VLSI circuits is estimating circuit performance prior to creating a low-level physical model for the circuit. The design methodology presented in the next section follows from some of the lessons learned in the study of multiprocessor systems and corrects some of the shortcomings in current VLSI circuit design methodologies.

2.3. The Algorithmic Design Methodology

The Algorithmic Design Methodology (ADM) is a new VLSI circuit design methodology which places few restrictions on the type of designs that can be developed and shows promise for complete automation. The ADM begins with an algorithm in the form of a sequential program in a high level programming language. A VLSI circuit architecture is developed from this algorithm via three transformations. Figure 6 illustrates the process. The first transformation parallelizes the sequential algorithm to create a scalar data flow graph. A scalar data flow graph is a data flow graph in which nodes perform simple operations, taking scalars as inputs and producing scalar results.

The second transformation partitions the scalar data flow graph into a number of clusters. The partitioning takes place in a manner that attempts to take advantage of parallelism inherent in the algorithm, balance the computational load among the clusters, and minimize intercluster communication (which translates into costly interprocessor communication). The clustered graph is a data flow graph itself, with the clusters becoming nodes and the intercluster arcs becoming arcs in the new data flow graph.

The final transformation embeds the clustered data flow graph into a graphical, structural model of a VLSI circuit architecture. The model is a VLSI circuit processor array. One or more clusters of the data flow graph are mapped to each processor in the array. The model gives the relative placement of the processors (shown as nodes), the function of each processor (expressed as data flow graphs), and the interprocessor connections (shown as edges). It gives no information about the implementation of each

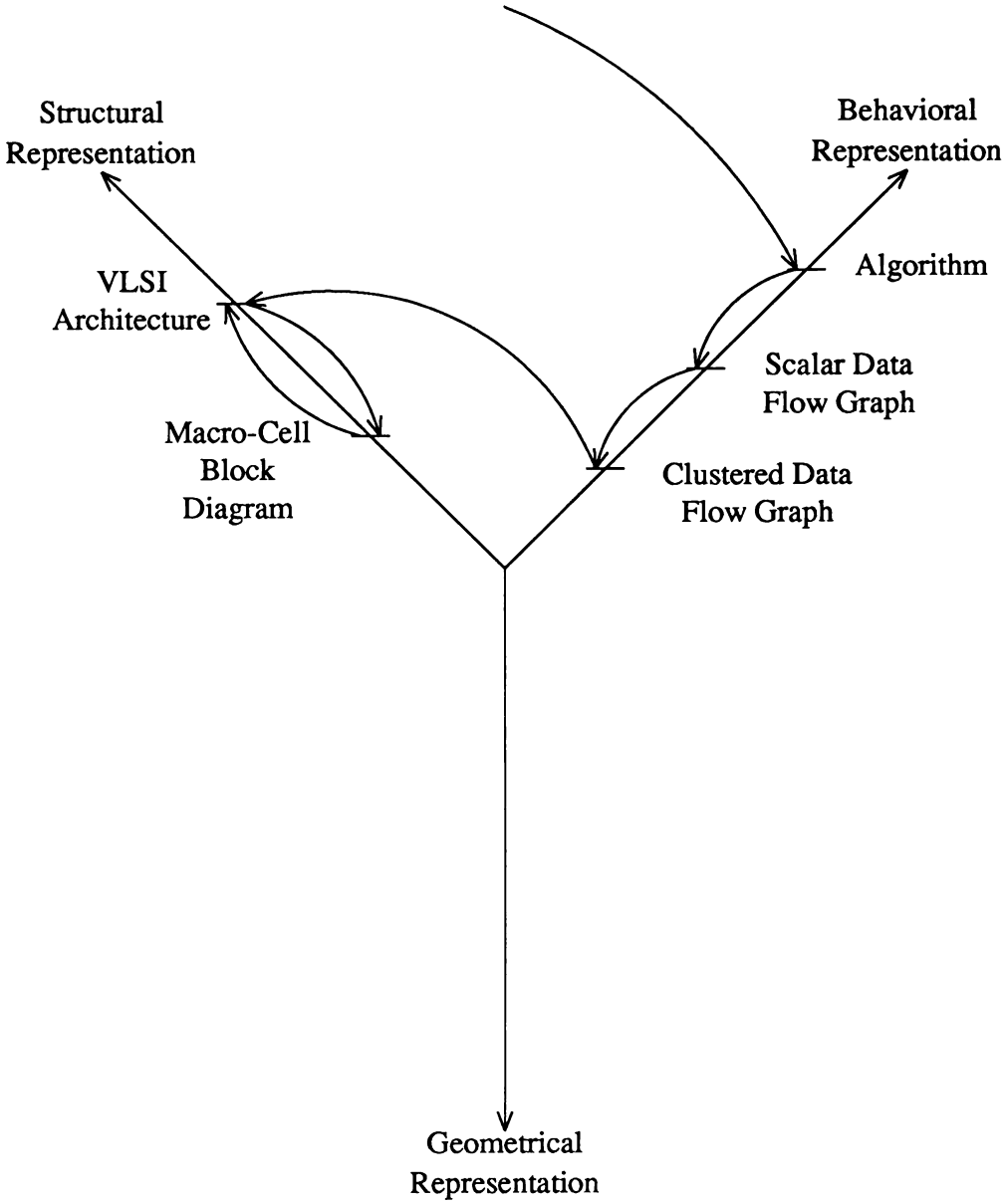

Figure 6. The Algorithmic Design Methodology

processor or the routing of interprocessor connections.

After the development of an architecture, its performance is evaluated by developing macro-cell level block diagrams for each of the processors. These diagrams are simulated to estimate the processing time and area for each processor. The area used to route the interprocessor connections is then estimated, along with the signal delay caused by interprocessor communication. These two estimates are then combined to give a full estimate of the proposed architecture's processing speed and area.

The automation of the new design methodology holds the promise for several improvements over current methodologies. First, it deals with the translation of a behavioral specification into a structural specification. Current methodologies do not handle this transformation well in general. The structural specification can make use of the successful macro-cell methodology for the creation of a physical layout. Second, the designer enters the methodology at a familiar, intuitive level, but, unlike the macro-cell methodology, need not spend time manually creating structural representations for the design. Third, the use of a graphical model for the algorithm and the VLSI circuit architecture removes limits on the types of circuits that can be designed. Fourth, the divide-and-conquer nature of the multiple transformations of the methodology help reduce the computational complexity that must be dealt with at each step. Finally, the hierarchical approach, as used in the partitioning of the scalar data flow graph, hides lower level details at each step in the methodology.

2.4. Key Research Issues

While the automation of the ADM would result in an improvement over current methodologies, several important issues must be investigated before the automation can occur. Since many of the ADM's transformation are, in general, NP-hard, the automation of the methodology will require the development of heuristic procedures to perform them. Specifically, the following issues are crucial to the success of the ADM:

1. The automatic parallelization of serial algorithms to produce scalar data flow graphs. While progress has been made in the development of optimizing and parallelizing compilers, most techniques are limited to certain types of parallelism or work with computational units larger than scalar operations [Aho86,Babb85].

2. The automatic partitioning of scalar data flow graphs to take advantage of parallelism inherent in the graph, balance the computational load among the partitions, and minimize interpartition communication. Previous research in this area has dealt with data flow graphs with few nodes [Ho83] or with limited classes of algorithms and architectures [Girk88].

3. The automatic embedding of this partitioned scalar data flow graph into the VLSI circuit architecture model in a manner that produces good designs. While some procedures do exist for mapping algorithms to VLSI circuit processor arrays, the mappings are inefficient in their use of processors [Mend87,Kore83].

4. The estimation from structural level models of the speed and area characteristics of VLSI circuits. Current estimation techniques use physical models of VLSI circuits [Thom79,Thom80].

These issues are crucial not only to the success of the design methodology, but also to many other areas of computer engineering.

2.5. Research Focus

The investigation of the issues defined in the previous section will require many man-years of research. The work to be presented in this dissertation will be limited to the investigation of a generalization of one of these issues. Specifically, a heuristic procedure will be developed to efficiently partition scalar data flow graphs for execution on message-based multiprocessor architectures in a manner that reduces execution time.

The procedure will be presented in the next chapter. Chapter 4 will evaluate the procedure via simulation. Chapter 5 will model the execution times resulting from the procedure's application. The results show that it is possible to effectively and automatically partition scalar data flow graphs for multiprocessor execution, and hence for implementation as ASIC's.

Chapter 3: The Automatic Scheduling of Data Flow Graphs

3.1. Overview of the Scheduling Problem

Scheduling tasks for execution on computer systems requires the mapping of the tasks to the computer system hardware in a manner that improves some aspect of system performance. Early research on the scheduling problem concentrated on scheduling tasks to uniprocessor systems using multitasking operating systems. Early work in multiprocessor scheduling includes work by Sethi [Seth76] and Fernandez and Lang [Fern75]. The general problem of scheduling has been divided into a number of classes. Gonzalez [Gonz77], for example, classifies scheduling problems by such things as the types of precedence among the tasks, the number of processors available in the system, and the type of processors in the system.

The algorithms used to solve various scheduling problems have also been classified. Coffman [Coff76] uses five different criteria for the classification of algorithms. First, a *static* algorithm runs only once, before a set of tasks is executed. A *dynamic* algorithm, on the other hand, may run during the execution of the task set. Second, if the scheduling algorithm has full knowledge of task execution times and communication requirements, the algorithm is termed *deterministic*. If an algorithm uses probabilistic models of task execution times and communication requirements, the algorithm is *stochastic*. Third, a *preemptive* algorithm can suspend a partially executed task and restart that task from the point of interruption at some later time. Once a *nonpreemptive* algorithm has scheduled a task, the task will run to completion. Fourth, scheduling algorithms can be classified by the system performance measure they attempt to improve. Some of these measures include response time, number or cost of processors needed, and resource utilization. Finally, algorithms are classified by the granularity of tasks which they schedule.

Several properties are desirable in algorithms that schedule tasks to multiprocessor architectures. First, algorithms should be applicable to a reasonably wide range of tasks and architectures. This can be accomplished by using abstract models for both the tasks

being scheduled and the architectures on which they execute. If the models accurately capture the important aspects of actual systems, then implementations of the algorithm can be tailored to a specific set of tasks and to specific architectures. Second, accurate estimates of task execution times are crucial in producing good schedules. The execution times of tasks at a low level of granularity can be estimated much more accurately than execution times for higher level tasks. Finally, it is important that the scheduling algorithm executes reasonably quickly and that its execution time increases relatively slowly as the size of the task set being scheduled increases. It makes little sense to use a scheduling algorithm requiring hours of execution time to schedule a job that only takes minutes to execute with a poor schedule. But, for some high performance applications which must execute many times, the increase in NRE costs incurred by the use of expensive schedulers may be justified. Note that the use of low-level granularity is at odds with quick scheduler execution time, so care must be taken to balance the trade-off between fast scheduler execution and accurate estimates of task execution times.

The next section describes the model used for parallel algorithms. This is followed by the definition of the multiprocessor architecture model. Then, a static, deterministic, nonpreemptive Data Flow Scheduling (DFS) algorithm, which schedules low-level granularity tasks, is presented. Finally, some comments are made about the properties of the DFS algorithm. A more detailed presentation of these topics can be found in Driscoll, et al. [Dris88].

3.2. Parallel Algorithm Model

To accurately represent parallel algorithms, a model should include several pieces of information. First, the model must be able to represent any low-level parallelism present in the algorithm. If this parallelism is not represented, it cannot be exploited during scheduling. Second, the model should include the data dependencies of the algorithm, i.e., constraints on the order of task execution. Third, the model should be general, while still hiding enough detail to allow schedulers to have reasonable execution times.

Finally, the model should be extendible so that new details can be added when required by specific situations.

The parallel algorithms scheduled by the DFS algorithm are described by an acyclic data flow graph with nodes performing scalar operations (a scalar data flow graph). These operations are members of the set of recognized scalar operations, $O = \{O_0, O_1, \cdots, O_{m-1}\}$. Operations in O may have multiple inputs, but are limited to producing a single output. Associated with each scalar operation is an operation size, $S_O(O_i)$, and an execution time, t_{O_i}. A scalar data flow graph, G, is defined by a set of nodes, $T_G = \{T_0, T_1, \cdots, T_k\}$, and a set of arcs, A_G. The function of each node in T_G is given by a vector, $O_T = (O_{T_1}, O_{T_2}, \cdots, O_{T_{m-1}})$. The amount of information passing through an arc in A_G is given by its arc size, $S_A(A_i)$.

By representing tasks at the level of primitive operations, this model explicitly represents low-level parallelism, allowing it to be exploited. Nodes in a data flow graph execute only when all of their predecessors have executed, thus enforcing an algorithm's dependency constraints. In allowing schedulers to operate quickly, the model limits the types of algorithms that can be represented. These limits can be removed by extending the model in several areas. Operations that produce multiple outputs can be modeled as a collection of nodes, each of which has the same inputs and produces one of the operation's outputs. Operations with different size inputs and outputs can be included by extending the definition of operation size to include multiple sizes for each operation. The most serious limitation imposed by the model is that of restricting the data flow graphs to be acyclic. For any algorithm in which the number of iterations of a loop is known prior to execution, the loops can be unrolled to produce an acyclic graph [Ullm84]. To include algorithms whose loops cannot be unrolled, the model must be extended to include conditional execution nodes. The addition of such nodes makes it very difficult to schedule graphs statically, i.e., prior to their execution. The model as presented allows a large number of useful algorithms to be scheduled even within its lim-

its, and is thus acceptable.

3.3. Multiprocessor Architecture Model

Multiprocessor architectures can be modeled at many levels of abstraction. In the work presented here, it is important that the model be general enough to include many real-world architectures. Another key element is the ability to model the internal structure of each processor in the architecture so that local parallelism as well as global parallelism can be exploited. As was the case with the model for parallel algorithms, some generality must be lost to allow the DFS algorithm to operate with reasonable execution time.

The architecture to which the DFS schedules parallel algorithms is similar to the data flow machine described by Arvind, et al. [Arvi80]. It consists of a fixed number of homogeneous processors. The processors communicate via an interconnection network, allowing each processor to send a fixed amount of data to any other processor in a constant time. Each processor in the architecture consists of a collection of non-pipelined, heterogeneous functional units. These units are connected to a store/matching unit which stores intermediate results and provides input data to the functional units. Each processor also has a communication processor which is connected to the interconnection network described above. Computations can proceed in parallel at a processor subject only to data dependency constraints and the availability of functional units. Computation and communication can proceed in parallel, but a single processor can only be involved in one communication at any time.

Formally, the architecture is defined by a number of parameters. The number of processors is represented by n_P. The number of functional units of each type at a single processor is given by the vector $F_O = (f_{O_0} f_{O_1}, \cdots f_{O_{m-1}})$. The time required to send a fixed-sized block of data from one processor to another via the interconnection network is the global communication time, t_{gc}. The amount of data that can be sent in one t_{gc} is

the global communication data block width, w_{gc}. The time required for a result to pass from one functional unit, through the store/matching unit, and to another functional unit within the same processor is the local communication time, t_{lc}. Finally, the execution times for each type of functional unit available at a single processor are given in the vector $X = (t_{O_0}, t_{O_1}, \cdots, t_{O_{m-1}})$.

The architecture model places two important limits on the types of architectures that can be represented. First, architectures with heterogeneous processors cannot be modeled. Second, architectures such as the hypercube cannot be modeled, because the communication time between two processors varies for different processors. These limits allow the DFS algorithm to ignore actual processor assignment while partitioning an algorithm. Removing these limitations from the model would greatly impede the working of the DFS. These restrictions may be relaxed by developing a separate algorithm to map the partitions created by the DFS to specific processors.

The model represents the complexities of each processor very well. Execution can proceed in parallel at all of a processor's functional units, subject to the dependency constraints of the algorithm. Also note that the architecture of an individual processor could actually be implemented as a data flow processor or a processor with a local memory and control unit to sequence operations. The model can easily be extended to include functional units which can perform more than one type of operation (e.g., arithmetic-logic units). The model as presented is a compromise between the details of actual architectures and the abstraction needed to efficiently schedule computation.

3.4. The Data Flow Scheduling Algorithm

3.4.1. Overview

The DFS algorithm schedules parallel algorithms to reduce the execution time of the algorithms. The algorithms are expressed as scalar data flow graphs and the architectures are represented using the previously defined multiprocessor architecture model. In

view of the properties mentioned in the first section of this chapter as desirable for scheduling algorithms, the DFS algorithm must have several characteristics. First, it must take advantage of the low-level parallelism inherent in the scalar data flow graph. Second, to exploit this parallelism, the DFS algorithm must make use of the structure of individual processors (e.g., multiple functional units). Finally, the execution time of the DFS algorithm must be reasonable and increase relatively slowly as the number of nodes in the scalar data flow graph increases. At the lowest level of detail, the DFS algorithm is quite complex. The reader is referred to Driscoll, et al. [Dris88] for a detailed presentation of the algorithm, including a formal description of the algorithm and proofs of complexity results.

The DFS algorithm operates in three phases. Figure 7 illustrates the DFS algorithm's flow. First, the DFS partitions the graph into a set of initial clusters, based on some basic properties of the graph. Second, one or more iterations of the global partitioning algorithm occur. For each iteration a heuristic is used to select a pair of clusters as candidates for merging. The cluster pair is merged into a single cluster if the single cluster's execution time would be less than a heuristic estimate of the time needed to execute the clusters on separate processors and pass results between them. Finally, after the completion of the global partitioning phase of the DFS, the auxiliary partitioning phase merges clusters as needed to insure that there are no more clusters than there are processors in the architecture. The cluster scheduler shown in Figure 7 calculates the time needed to execute a single cluster or a pair of clusters on a single processor, using a critical path scheduling algorithm.

3.4.2. Initial Cluster Generation

The initial cluster generation phase of the DFS algorithm partitions the scalar data flow graph into clusters that are chains of sequential nodes. This is reasonable because a sequential chain contains no parallelism and thus gains no benefit from being split among several processors. Each chain begins with a source node, i.e., a node which is the tail of

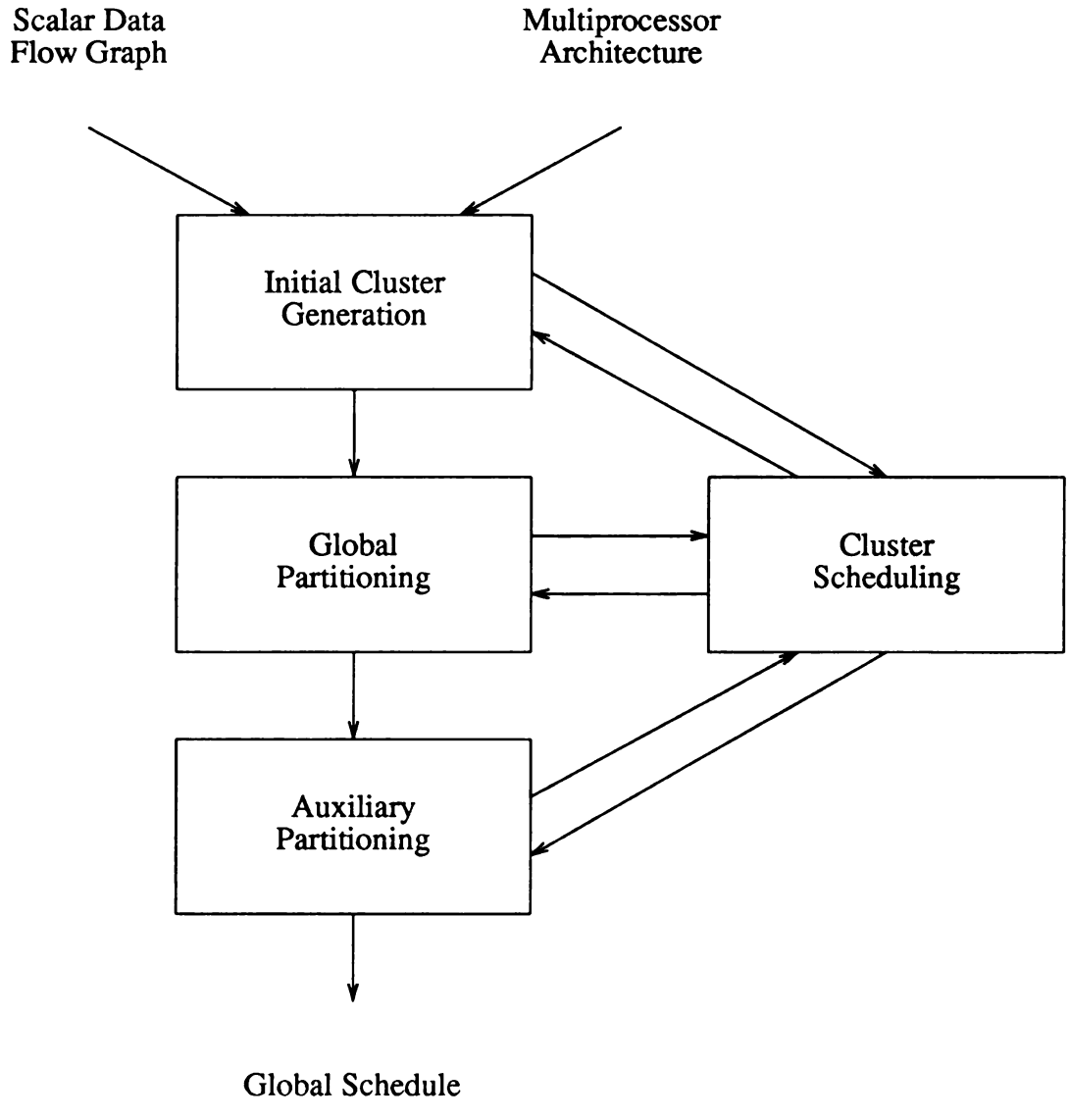

Figure 7. The Data Flow Scheduler

no unused arcs. Once a source node has been selected for a cluster, the cluster grows as nodes which are immediate successors (via an unused arc) to the last added node are included in the cluster. When a node is added to a cluster, all arcs to that node are marked as used. Nodes are added to a cluster until there are no unused arcs emanating from the last added node or the last added node is the head of more than one unused arc (i.e., the computation branches at the last added node). If the latter case ends the formation of a cluster, all arcs with the last added node as their head are marked as used. When the graph's supply of source nodes has been exhausted, all of the nodes have been placed in exactly one partition and the initial cluster generation phase finishes. This scheme of initial cluster generation is of time complexity $O(n^4)$, where n is the number of nodes in the scalar data flow graph [Dris88].

3.4.3. Global Partitioning

The global partitioning phase of the DFS algorithm iterates over the set of clusters, merging pairs of clusters when its heuristics indicate that doing so reduces response time. Initially, global partitioning makes use of the set of clusters generated in the initial clustering phase of the DFS. A cluster pair is selected via the global partitioning phase's first heuristic. The pair consists of the cluster under consideration in the current iteration of the phase and the cluster that is responsible for the highest percentage of the current cluster's intercluster communication.

Once a cluster pair has been selected, the cluster scheduler calculates the time needed to execute both clusters on a single processor. The time to execute the clusters on separate processors and to communicate results between the processors is estimated as the maximum of the individual clusters' execution times plus the total time required to complete all communication between the two clusters. This method of estimating separate execution time is the global partitioning phase's second heuristic. It assumes that the shorter of the two execution times completely overlaps the longer execution time. This heuristic does not always accurately model the parallel execution of the two

clusters, but calculating the estimate takes very little time. The use of other heuristics in forming this estimate is a subject for further research.

A pair of clusters is merged if its uniprocessor execution time is smaller than the estimate of the time required to execute each cluster of the pair on a separate processor. After the global partitioning phase finishes with one cluster pair, it moves to the next cluster in the set and repeats the process. When a complete iteration through the set of clusters does not merge any pairs, the global partitioning phase halts and the DFS algorithm moves to the auxiliary partitioning phase.

The global partitioning phase of the DFS was shown to be of time complexity $O(n^5)$, where n is the number of nodes in the scalar data flow graph, in [Dris88]. The derivation of this result did not make use of the divide-and-conquer nature of the DFS algorithm. Experimental results indicate that the overall time complexity for the DFS algorithm is $O(n^2)$.

3.4.4. Auxiliary Partitioning

After the completion of the global partitioning phase of the DFS algorithm, there may be more clusters than processors in the multiprocessor architecture. The auxiliary partitioning phase reduces the number of clusters so that there are no more clusters than there are processors. First, the auxiliary partitioning phase repeatedly merges the smallest cluster with the cluster that is responsible for the highest percentage of the smallest cluster's intercluster communication. (This uses the same heuristic as the global partitioning phase uses to select cluster pairs.) If this process still leaves more clusters than processors (which can occur if the graph is reduced to a number of disconnected clusters), the auxiliary partitioning phase repeatedly merges the two smallest clusters until there are no more clusters than processors. The time complexity of the auxiliary partitioning phase of the DFS is $O(n^4)$, where n is again the number of nodes in the scalar data flow graph [Dris88].

3.4.5. Cluster Scheduler

The cluster scheduler calculates the execution time for one or two clusters executing on a single processor in the multiprocessor architecture. The critical path list scheduling algorithm is used to assign nodes to functional units within the processor. A critical path ordered list is formed for each functional unit type. The cluster scheduler then assigns nodes to available functional units, choosing the first ready node from the appropriate list, removing nodes from the cluster after they have completed execution. The time at which the last node finishes executing is the execution time for the cluster (or pair of clusters). The time complexity of the cluster scheduler is $O(m^3)$, where m is the number of nodes in the cluster or pair of clusters being executed [Dris88]. Note that the cluster scheduler executes once for each iteration of the global partitioning phase of the DFS algorithm and is responsible for a large percentage of the DFS algorithm's execution time. The use of other, less time consuming, algorithms for the cluster scheduler is a subject for future research.

3.4.6. Properties of the Data Flow Scheduling Algorithm

The DFS algorithm has many properties that are desirable for multiprocessor scheduling algorithms, including applicability over a wide range of tasks and architectures (i.e., generality), the use of accurate estimates of task execution times, and reasonable speed in generating schedules. The models used for parallel algorithms and multiprocessor architectures are reasonably general without compromising DFS algorithm execution speed, allowing many real-world algorithms and architectures to be modeled. The low-level granularity of the operations in the data flow graph allows accurate estimates of their execution times to be used, improving the quality of the resulting schedules. This low-level granularity also allows parallelism at each processor in the architecture to be fully exploited. The divide-and-conquer nature of the DFS algorithm reduces the algorithm's time complexity from what would be expected for scalar data flow graphs, which contain large numbers of nodes. The cluster scheduler portion of the

DFS limits its attention to the nodes in one or two clusters, while ignoring all other nodes in the graph. The global partitioning phase of the algorithm, on the other hand, deals only with clusters and their interconnections, ignoring their internal structure. The simulation results to be presented in the next chapter show that the DFS algorithm achieves good improvement in parallel algorithm execution times with reasonable time complexity.

Chapter 4: Evaluating the DFS Algorithm

4.1. Approach

Evaluating heuristic solutions to NP-hard problems is generally difficult due to the time needed to generate optimal solutions for comparison. A common practice is to compare the algorithm being evaluated with other heuristic algorithms. Granski, et al. [Gran87], compare selecting a ready data flow graph node for execution on a data flow processor randomly or by using critical path information. Ravi, et al. [Ravi87], compare their algorithm with algorithms ignoring some of the information used by their algorithm.

The evaluation of the DFS algorithm uses this type of strategy, making the generation of optimal partitions unnecessary. The partitions resulting from application of the DFS algorithm will be compared with uniprocessor execution of the same graphs and with multiprocessor execution of random partitionings of the graphs. The comparison with uniprocessor execution evaluates the DFS algorithm's ability to make effective use of the computational power available in multiprocessor architectures. Random partitioning divides the graph being partitioned into as many clusters as there are processors in the architecture by randomly assigning each node in the graph to a cluster. Random partitioning completely ignores the structure of the graph and the details of the multiprocessor architecture in scheduling the graph. Thus, comparing the DFS algorithm with random partitioning evaluates the usefulness of the information employed by the DFS algorithm's heuristics.

The DFS algorithm's main purpose is to reduce the execution times of the algorithms being partitioned. Since this execution takes place on an abstract model of a multiprocessor architecture, execution times are obtained using a simulator. This simulator was implemented specifically for the parallel algorithm and multiprocessor architecture models used in this work. The simulator is written in the C programming language and currently executes on a VAX 8600 under the Ultrix 2.2 operating system. It schedules nodes for execution at each processor based on the critical path for each node. The

simulator is more realistic than the heuristics used by the DFS in that input data items for a cluster may arrive at different times and it is possible for the execution of two clusters to partially overlap.

This evaluation of the DFS algorithm concentrates on three aspects of the algorithm's behavior. The first is the ability of the DFS to provide reasonable improvement in parallel algorithm execution time as compared with uniprocessor execution and random partitioning. Second, the time complexity of the DFS algorithm must be low enough to make its use reasonable. Finally, the ease with which the execution times of algorithms partitioned using the DFS can be modeled must be evaluated. A number of different scalar data flow graphs and multiprocessor architectures have been developed for use in this evaluation. Each of the graphs are scheduled to each of the architectures using each of the three possible partitioning algorithms (uniprocessor, random, and DFS). The execution time of each schedule is obtained via simulation. The execution times of schedules created by the different algorithms are compared to evaluate the improvement in execution time obtained by use of the DFS algorithm. The time used by the DFS algorithm to create each schedule is used to determine the observed time complexity of the DFS as a function of the number of nodes in the test graphs. Finally, the execution times of the parallel algorithms are examined for patterns as a prelude to modeling the behavior of the three partitioning algorithms. The results of this evaluation will decide the usefulness of the DFS algorithm and provide the basis for the models (to be presented in the next chapter) of the three partitioning algorithms.

4.2. Evaluation Procedure and Criteria

Many factors influence the behavior of scheduling algorithms. For the parallel algorithm model, any change in the types of computation performed, the number of nodes in the graph, or the graph's interconnection topology alters the nature of the scheduling task. Scheduling a graph with no communication between nodes, for example, is very different from scheduling a graph with a great deal of communication. In the

multiprocessor architecture model, any change in the number of processors, the speed and number of functional units at each processor, or the amount of time needed to communicate results within and among processors changes the target of the scheduling algorithm. It may be relatively easy, for example, to schedule a graph to an architecture with more processors than there are nodes in the graph, while it is difficult to schedule the same graph to an architecture with few processors.

Formally, execution time is a function of several variables:

$$\tau=\tau(\Pi, n_P, F_O, X, t_{gc}, t_{lc}, M(G)),$$

where

τ is the execution time for a partitioning (developed using algorithm Π) of the graph, G, on the given architecture;

Π is the algorithm used to schedule the graph, G, to the given architecture;

n_P is the number of processors;

F_O is a vector giving the number of functional units of each function type;

X is a vector of execution times for each function type;

t_{gc} is the global communication time;

t_{lc} is the local communication time; and

$M(G)$ is a vector of functions, $(M_0(G), M_1(G), \cdots, M_q(G))$, describing characteristics of the scalar data flow graph G.

In justifying the dependence of execution time on the listed parameters, two simple graphs will be used as examples. Figure 8 shows a graph consisting of a number of completely disconnected nodes. The nodes in this graph can be executed in parallel and require no communication. Figure 9 shows a graph consisting of strictly sequential nodes. None of the nodes is this graph can be executed in parallel and a single communication is required for each computation.

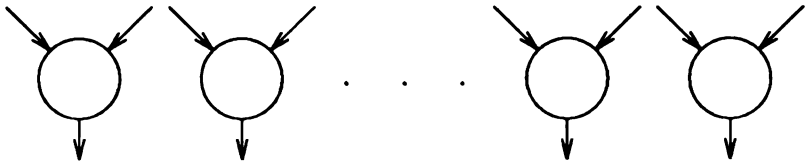

Figure 8. Fully Parallel Graph

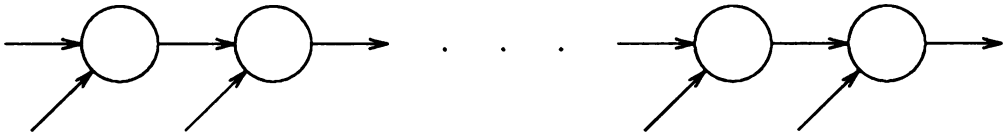

Figure 9. Fully Sequential Graph

The dependence of execution time on the number of processors, n_P, is illustrated by the fully parallel graph. For this graph, execution time decreases as the number of processors increases, provided there are more nodes than processors. Similarly, as entries in F_O increase, the execution time for the fully parallel graph will decrease, provided there are enough nodes to make use of the additional functional units.

The effects of functional unit execution times, X, global communication time, t_{gc}, and local communication time, t_{lc}, on execution time are illustrated by the fully sequential graph. The time required to execute this graph on a uniprocessor is the sum of the execution times and local communication times for each node. For a multiprocessor system, the execution time is a sum of the execution times, a number of local communication times, and a number of global communication times. Thus, if any of X, t_{gc}, or t_{lc} increases, the execution time also increases.

While specific measures of graph characteristics have not been defined, two factors do influence execution time. First, the effect of graph topology can be seen in the difference between the fully parallel and fully sequential graphs. For the fully parallel graph, the execution time decreases when the number of processors increases. For the fully sequential graph, the execution time can actually increase with the number of processors. Second, the functions performed by the graph's nodes also affect execution time. This can be seen from the discussion above showing dependence on X. There are other measures that may have an influence on execution time. The effect of particular graph characteristics seems to be determined by the partitioning strategy used.

Speedup is a useful measure of improvement in the execution time, especially when trying to isolate the effect of changing a single aspect of the problem, such as the number of processors in the architecture. Speedup is defined as the ratio of a reference execution time to another execution time that is under examination. Equivalently, speedup can be defined as the ratio of a throughput (i.e., the reciprocal of an execution time) to some reference throughput. Different measures of speedup can be defined by altering the

definition of the reference execution time. This reference execution time is denoted by τ^0 and parameters that are held constant in the reference will also be superscripted with 0 (e.g., t_{gc}^0). A speedup measure can then be denoted by:

$$S(\Pi, n_P, F_O, X, t_{gc}, t_{lc}, G) = \frac{\tau^0}{\tau(\Pi, n_P, F_O, X, t_{gc}, t_{lc}, G)}$$

To usefully measure speedup, care must be used in the selection of parameters which remain fixed in the reference. Driscoll, et al. [Dris88], fix the reference at one processor and vary all other parameters except the partitioning algorithm. This reference measures the effectiveness of random and DFS partitioning as processors of a given type are added to an architecture. Here, this measure will be called *processor speedup* and is given by:

$$S_P = \frac{\tau^0(\Pi, n_P^0=1, F_O, X, t_{gc}, t_{lc}, G)}{\tau(\Pi, n_P, F_O, X, t_{gc}, t_{lc}, G)}$$

A more detailed measure of speedup uses the performance of the least powerful architecture available as a reference. This measure will be called *base speedup* and is given by:

$$S_B = \frac{\tau^0(\Pi, n_P^0=1, F_O^0=(1,1,\cdots,1), X^0, t_{gc}^0, t_{lc}^0, G)}{\tau(\Pi, n_P, F_O, X, t_{gc}, t_{lc}, G)},$$

where X^0, t_{gc}^0, and t_{lc}^0 are the maximum execution and communication times possible within a family of architectures.

Base speedup results will be included with the simulated execution times to be presented in the next section.

In evaluating the DFS algorithm the effects of many of the factors mentioned above are examined. For the algorithms being scheduled, varying the operations performed by the scalar data flow graph tests the DFS algorithm's ability to effectively utilize functional units of different types and speeds. Varying the number of nodes in the scalar data flow graph tests the DFS algorithm's ability to make use of a limited number of processors and functional units as the amount of computation increases. Finally, varying the

interconnection topology of the graph tests the DFS algorithm's ability to handle a variety of algorithms.

For the multiprocessor architectures, varying the number of processors tests the DFS algorithm's ability to handle situations with too many or too few processors for algorithms being scheduled. Varying the number of functional units tests the DFS algorithm's ability to use more or less powerful processors. Varying the speed of functional units alters the cost of computation relative to communication and tests the DFS algorithm's ability to operate with different relative costs. The time for local communication is assumed to be very small relative to the time for global communication. Thus any reasonable variation in the local communication time is unlikely to greatly affect scheduler performance and this parameter will not be varied in this evaluation of the DFS algorithm. Also, global communication time will not be varied, since varying the speed of functional units already changes the relative costs of computation and communication.

With the above comments in mind, three different types of scalar data flow graph topologies are used in the evaluation procedure. First are the binary merge graphs, which compute vector inner products. Graphs with 127, 255, 511, 1023, and 4095 nodes are used. Figure 10 illustrates a binary merge graph with 7 nodes. Second are graphs with the FFT topology, which often occurs in signal processing operations. Graphs with 1024, 2304, and 5120 nodes are used. Figure 11 illustrates an FFT graph with 12 nodes. Finally, a number of scalar data flow graphs were randomly generated. The generator for these graphs first chooses the function of each node using a uniform distribution. Then each input item at each node is chosen as either coming from another node in the graph or as coming from outside the graph (i.e., the input item is an input for the algorithm), again using a uniform distribution. Four graphs each of sizes 500, 1000, 2000, and 5000 nodes are used, for a total of 16 graphs.

Two different sets of functional unit speeds were used in developing multiprocessor architectures for use in the evaluation procedure. The functions modeled are 32-bit addi-

tion, multiplication, division, and two's complement. The execution times are derived from VLSI circuit designs of blocks performing these functions. The *slow* set of execution times can be thought of as having small chip area and slow execution, while the *fast* set has larger area and faster execution. The execution times of the fast and slow sets, along with the local and global communication times, are shown in Table 1, scaled so that local communication time is one. Using these two sets of functional units, eight multiprocessor architectures are defined by varying the number of processors, the number of functional units of each type, and the speed of the functional units. To simplify the evaluation procedure, the number of functional units is identical for each function type (i.e., $F_O=(n_F,n_F,\cdots,n_F)$). Table 2 shows the 8 architectures that are used in the evaluation procedure.

In the evaluation, each of the scalar data flow graphs defined above is scheduled to each of the eight architectures using the three partitioning algorithms. The resulting partitions are then simulated to obtain execution times. The time required for the DFS algorithm to partition each graph for each architecture is also measured. These results provide the information used to evaluate the improvement in execution time obtained by using the DFS algorithm, the observed time complexity of the DFS algorithm, and the basic data used in developing models to describe the behavior of the three partitioning algorithms (uniprocessor, random, and DFS).

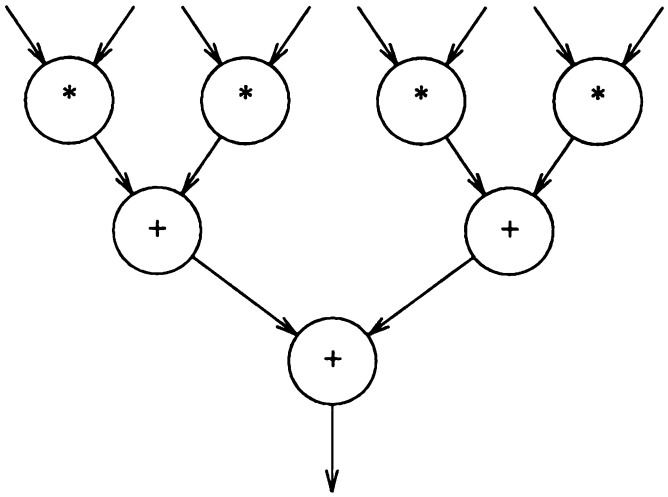

Figure 10. Binary Merge Graph for Eight Inputs

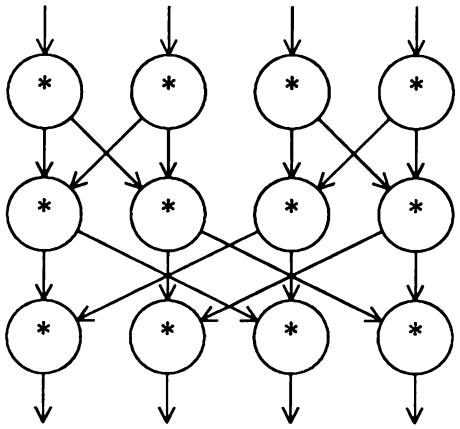

Figure 11. FFT Graph for Four Inputs

Table 1. Parameters for Fast and Slow Functional Units

	Local Communication t_{lc}	Global Communication t_{gc}	32-bit Integer Addition	32-bit Integer Multiply	32-bit Integer Division	32-bit Integer 2's Complement
Slow	1	40	4	37	154	3
Fast	1	40	1	6	15	3

Table 2. Multiprocessor Architectures Used in Simulations

	Functional Unit Speed	Number of Functional Units of Each Type	Number of Processors
A1	Slow	1	4
A2	Fast	1	4
A3	Slow	3	4
A4	Fast	3	4
A5	Slow	1	16
A6	Fast	1	16
A7	Slow	3	16
A8	Fast	3	16

4.3. Simulation Results

4.3.1. Binary Merge Graphs

Table 3 shows the results for uniprocessor execution. The reduction in execution time obtained by adding more functional units indicates that the binary merge graphs contain a large amount of potential parallelism. Notice that base speedup improves as the number of nodes in the graphs increases, indicating that parallelism in the graphs increases with the number of nodes. Table 4 shows the simulation results for the execution of binary merge graphs partitioned using the random and DFS algorithms. For random partitioning, base speedup is relatively insensitive to changes in the speed and number of functional units. Base speedup increases relatively slowly as the size of the graph being partitioned increases, indicating that parallelism is not being fully exploited. The DFS algorithm consistently outperforms random partitioning, and achieves base speedups near theoretical maximums as the graphs grow in size.

Table 5 present the time required for the DFS algorithm to partition binary merge graphs. For these simulations, the growth rate of this execution time is greater than n and less than $n^{1.5}$, where n is the number of nodes in the graph being scheduled.

Table 3. Uniprocessor Execution of Binary Merge Graphs

Number of Nodes	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time	Base Speedup
127	1	Slow	2398	1.00
127	1	Fast	396	6.06
127	3	Slow	844	2.84
127	3	Fast	144	16.65
255	1	Slow	4771	1.00
255	1	Fast	782	6.10
255	3	Slow	1626	2.93
255	3	Fast	272	17.54
511	1	Slow	9512	1.00
511	1	Fast	1552	6.13
511	3	Slow	3222	2.95
511	3	Fast	532	17.88
1023	1	Slow	18989	1.00
1023	1	Fast	3090	6.15
1023	3	Slow	6372	2.98
1023	3	Fast	1044	18.19
2047	1	Slow	37938	1.00
2047	1	Fast	6164	6.15
2047	3	Slow	12704	2.99
2047	3	Fast	2072	18.31
4095	1	Slow	75831	1.00
4095	1	Fast	12310	6.16
4095	3	Slow	25326	2.99
4095	3	Fast	4120	18.41

Table 4. Multiprocessor Execution of Binary Merge Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Random		DFS	
				Execution Time	Base Speedup	Execution Time	Base Speedup
127	4	1	Slow	2491	0.96	700	3.43
127	4	1	Fast	2207	1.09	186	12.89
127	4	3	Slow	2201	1.09	330	7.27
127	4	3	Fast	2329	1.03	117	20.50
127	16	1	Slow	1337	1.79	334	7.18
127	16	1	Fast	1410	1.70	177	13.55
127	16	3	Slow	1125	2.13	258	9.29
127	16	3	Fast	1092	2.20	117	20.50
255	4	1	Slow	4611	1.03	1297	3.68
255	4	1	Fast	4288	1.11	284	16.80
255	4	3	Slow	4601	1.04	520	9.18
255	4	3	Fast	4210	1.13	158	30.20
255	16	1	Slow	2031	2.35	487	9.80
255	16	1	Fast	1890	2.52	218	21.89
255	16	3	Slow	1893	2.52	302	15.80
255	16	3	Fast	1770	2.70	158	30.20
511	4	1	Slow	9249	1.03	2486	3.83
511	4	1	Fast	7968	1.19	478	19.90
511	4	3	Slow	7890	1.21	932	10.21
511	4	3	Fast	9009	1.05	226	42.09
511	16	1	Slow	3506	2.71	788	12.07
511	16	1	Fast	2409	3.95	268	35.49
511	16	3	Slow	3885	2.45	418	22.76
511	16	3	Fast	3331	2.86	199	47.80
1023	4	1	Slow	16618	1.14	4859	3.91
1023	4	1	Fast	16727	1.14	864	21.98
1023	4	3	Slow	17006	1.12	1714	11.08
1023	4	3	Fast	16129	1.18	354	53.64
1023	16	1	Slow	6065	3.13	1385	13.71
1023	16	1	Fast	5811	3.27	366	51.88
1023	16	3	Slow	6145	3.09	608	31.23
1023	16	3	Fast	5930	3.20	240	79.12

Table 4. (cont'd.)

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Random		DFS	
				Execution Time	Base Speedup	Execution Time	Base Speedup
2047	4	1	Slow	33380	1.14	9600	3.95
2047	4	1	Fast	32528	1.17	1634	23.22
2047	4	3	Slow	32565	1.16	3310	11.46
2047	4	3	Fast	32849	1.15	614	61.79
2047	16	1	Slow	11529	3.29	2574	14.74
2047	16	1	Fast	11370	3.34	560	67.75
2047	16	3	Slow	11653	3.26	1020	37.19
2047	16	3	Fast	11690	3.25	308	123.18
4095	4	1	Slow	65290	1.16	19077	3.97
4095	4	1	Fast	61528	1.23	3172	23.91
4095	4	3	Slow	63889	1.19	6460	11.74
4095	4	3	Fast	62567	1.21	1126	67.35
4095	16	1	Slow	23449	3.23	4947	15.33
4095	16	1	Fast	21771	3.48	946	80.16
4095	16	3	Slow	21653	3.50	1802	42.08
4095	16	3	Fast	21410	3.54	436	173.92

Table 5. DFS Execution Time for Binary Merge Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time (VAX 8600) (CPU Seconds)
127	4	1	Slow	32.36
127	4	1	Fast	27.71
127	4	3	Slow	27.62
127	4	3	Fast	27.39
127	16	1	Slow	29.70
127	16	1	Fast	26.91
127	16	3	Slow	26.88
127	16	3	Fast	27.74
255	4	1	Slow	68.65
255	4	1	Fast	58.40
255	4	3	Slow	58.17
255	4	3	Fast	56.32
255	16	1	Slow	64.54
255	16	1	Fast	54.97
255	16	3	Slow	53.92
255	16	3	Fast	56.41
511	4	1	Slow	143.40
511	4	1	Fast	123.21
511	4	3	Slow	120.84
511	4	3	Fast	118.24
511	16	1	Slow	135.80
511	16	1	Fast	115.89
511	16	3	Slow	115.25
511	16	3	Fast	114.50
1023	4	1	Slow	313.16
1023	4	1	Fast	268.97
1023	4	3	Slow	262.33
1023	4	3	Fast	255.10
1023	16	1	Slow	294.15
1023	16	1	Fast	249.36
1023	16	3	Slow	248.38
1023	16	3	Fast	242.24

Table 5. (cont'd.)

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time (VAX 8600) (CPU Seconds)
2047	4	1	Slow	740.17
2047	4	1	Fast	636.32
2047	4	3	Slow	606.14
2047	4	3	Fast	583.25
2047	16	1	Slow	675.95
2047	16	1	Fast	567.88
2047	16	3	Slow	561.48
2047	16	3	Fast	546.50
4095	4	1	Slow	1937.88
4095	4	1	Fast	1675.30
4095	4	3	Slow	1537.35
4095	4	3	Fast	1466.36
4095	16	1	Slow	1649.28
4095	16	1	Fast	1385.78
4095	16	3	Slow	1357.10
4095	16	3	Fast	1318.49

4.3.2. FFT Graphs

Table 6 gives uniprocessor execution times for the FFT graphs. As for the binary merge graphs, these results indicate that the FFT graphs are highly parallel in nature. However, the amount of parallelism in the graphs stays relatively constant as the number of nodes in the graph increases. Table 7 gives multiprocessor execution times for the FFT graphs when partitioned using the random and DFS algorithms. The random algorithm results are similar to those for binary merge graphs. Base speedup is relatively insensitive to changes in the power of individual processors in the architecture. The DFS algorithm is very sensitive to changes in processor characteristics, providing increased base speedup as processors become more powerful. The DFS algorithm performs poorly relative to random partitioning for architectures with 4 processors and a single, slow functional unit of each type. Also, the base speedup is surprisingly low for the 2304 node graph partitioned for execution on the architecture with 16 processors, each having 3 slow functional units of each type.

Table 8 gives the execution time required by the DFS algorithm to partition the FFT graphs. For these simulations, the growth rate of this execution time is greater than $n^{1.5}$ and less than n^2.

Table 6. Uniprocessor Execution of FFT Graphs

Number of Nodes	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time	Base Speedup
1024	1	Slow	37888	1.00
1024	1	Fast	6144	6.17
1024	3	Slow	12654	2.99
1024	3	Fast	2052	18.46
2304	1	Slow	85248	1.00
2304	1	Fast	13824	6.17
2304	3	Slow	28416	3.00
2304	3	Fast	4608	18.50
5120	1	Slow	189440	1.00
5120	1	Fast	30720	6.17
5120	3	Slow	63159	3.00
5120	3	Fast	10242	18.50

Table 7. Multiprocessor Execution of FFT Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Random		DFS	
				Execution Time	Base Speedup	Execution Time	Base Speedup
1024	4	1	Slow	28234	1.34	30784	1.23
1024	4	1	Fast	28172	1.34	4992	7.59
1024	4	3	Slow	28714	1.32	10286	3.68
1024	4	3	Fast	28292	1.34	3215	11.78
1024	16	1	Slow	9874	3.84	3081	12.30
1024	16	1	Fast	10664	3.55	1641	23.09
1024	16	3	Slow	10634	3.56	2038	18.59
1024	16	3	Fast	9972	3.80	1653	22.92
2304	4	1	Slow	65674	1.30	69264	1.23
2304	4	1	Fast	66052	1.29	11232	7.59
2304	4	3	Slow	63194	1.35	21312	4.00
2304	4	3	Fast	62892	1.36	6692	12.74
2304	16	1	Slow	21674	3.93	6180	13.79
2304	16	1	Fast	21212	4.02	3081	27.67
2304	16	3	Slow	23114	3.69	8832	9.65
2304	16	3	Fast	22332	3.82	3056	27.90
5120	4	1	Slow	141754	1.34	153920	1.23
5120	4	1	Fast	141612	1.34	24960	7.59
5120	4	3	Slow	142674	1.33	51319	3.69
5120	4	3	Fast	140172	1.35	13859	13.67
5120	16	1	Slow	48234	3.93	13244	14.30
5120	16	1	Fast	47892	3.96	6001	31.57
5120	16	3	Slow	46354	4.09	7469	25.36
5120	16	3	Fast	45932	4.12	5813	32.59

Table 8. DFS Execution Time for FFT Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time (VAX 8600) (CPU Seconds)
1024	4	1	Slow	792.74
1024	4	1	Fast	743.24
1024	4	3	Slow	707.13
1024	4	3	Fast	653.15
1024	16	1	Slow	647.29
1024	16	1	Fast	602.35
1024	16	3	Slow	646.88
1024	16	3	Fast	592.68
2304	4	1	Slow	4966.77
2304	4	1	Fast	2640.10
2304	4	3	Slow	2559.40
2304	4	3	Fast	2086.56
2304	16	1	Slow	4217.32
2304	16	1	Fast	1892.68
2304	16	3	Slow	2266.71
2304	16	3	Fast	1807.22
5120	4	1	Slow	10678.43
5120	4	1	Fast	10646.77
5120	4	3	Slow	8348.69
5120	4	3	Fast	7857.97
5120	16	1	Slow	7155.11
5120	16	1	Fast	7108.45
5120	16	3	Slow	7102.35
5120	16	3	Fast	6599.98

4.3.3. Random Graphs

Tables 9a through 9d show the uniprocessor execution times for the random graphs. Note that the graphs are grouped into four sets only for convenience in presenting these results. There is no special relationship between the graphs in each set. As was true for the other types of graphs the uniprocessor results indicate that the random graphs contain a large amount of parallelism. The amount of parallelism stays relatively constant as the size of the graph increases. Tables 10a through 10d give the multiprocessor execution times of the random graphs as partitioned by the random and DFS algorithms. Once again, the execution time of randomly partitioned graphs is insensitive to changes in individual processors. The results for the DFS algorithm seem to indicate that only four processors are used even when 16 are available. However, the partitionings are actually using all available processors in each case. This indicates that some portion of the graph determines execution time, and that this portion of the graph does not adequately use all available processors under DFS partitioning. The DFS algorithm does not perform well with respect to random partitioning when the architecture contains slow functional units. Unlike random partitioning, the DFS algorithm is sensitive to changes in processor characteristics.

Tables 11a through 11d give the execution time for the DFS algorithm when partitioning random graphs. The growth rate of this execution time for the graphs used in the simulations is greater than $n^{1.5}$ and less than n^2.

Table 9a. Uniprocessor Execution of First Set of Random Graphs

Number of Nodes	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time	Base Speedup
500	1	Slow	18942	1.00
500	1	Fast	1845	10.27
500	3	Slow	6314	3.00
500	3	Fast	615	30.80
1000	1	Slow	37730	1.00
1000	1	Fast	3675	10.27
1000	3	Slow	12628	2.99
1000	3	Fast	1230	30.67
2000	1	Slow	72380	1.00
2000	1	Fast	7050	10.27
2000	3	Slow	24178	2.99
2000	3	Fast	2355	30.73
5000	1	Slow	191422	1.00
5000	1	Fast	18645	10.27
5000	3	Slow	63910	3.00
5000	3	Fast	6225	30.75

Table 9b. Uniprocessor Execution of Second Set of Random Graphs

Number of Nodes	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time	Base Speedup
500	1	Slow	21560	1.00
500	1	Fast	2100	10.27
500	3	Slow	7238	2.98
500	3	Fast	705	30.58
1000	1	Slow	39578	1.00
1000	1	Fast	3855	10.27
1000	3	Slow	13244	2.99
1000	3	Fast	1290	30.68
2000	1	Slow	76076	1.00
2000	1	Fast	7410	10.27
2000	3	Slow	25410	2.99
2000	3	Fast	2475	30.74
5000	1	Slow	194348	1.00
5000	1	Fast	18930	10.27
5000	3	Slow	64834	3.00
5000	3	Fast	6315	30.78

Table 9c. Uniprocessor Execution of Third Set of Random Graphs

Number of Nodes	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time	Base Speedup
500	1	Slow	19712	1.00
500	1	Fast	1920	10.27
500	3	Slow	6622	2.98
500	3	Fast	645	30.56
1000	1	Slow	40194	1.00
1000	1	Fast	3915	10.27
1000	3	Slow	13398	3.00
1000	3	Fast	1305	30.80
2000	1	Slow	80542	1.00
2000	1	Fast	7845	10.27
2000	3	Slow	26950	2.99
2000	3	Fast	2625	30.68
5000	1	Slow	191884	1.00
5000	1	Fast	18690	10.27
5000	3	Slow	64064	3.00
5000	3	Fast	6240	30.75

Table 9d. Uniprocessor Execution of Fourth Set of Random Graphs

Number of Nodes	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time	Base Speedup
500	1	Slow	19712	1.00
500	1	Fast	1920	10.27
500	3	Slow	6622	2.98
500	3	Fast	645	30.56
1000	1	Slow	38808	1.00
1000	1	Fast	3780	10.27
1000	3	Slow	12936	3.00
1000	3	Fast	1260	30.80
2000	1	Slow	77770	1.00
2000	1	Fast	7575	10.27
2000	3	Slow	26026	2.99
2000	3	Fast	2535	30.68
5000	1	Slow	197120	1.00
5000	1	Fast	19200	10.27
5000	3	Slow	65758	3.00
5000	3	Fast	6405	30.78

Table 10a. Multiprocessor Execution of First Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Random		DFS	
				Execution Time	Base Speedup	Execution Time	Base Speedup
500	4	1	Slow	7436	2.55	12936	1.46
500	4	1	Fast	7705	2.46	1260	15.03
500	4	3	Slow	7392	2.56	4312	4.39
500	4	3	Fast	7284	2.60	420	45.10
500	16	1	Slow	3630	5.22	12936	1.46
500	16	1	Fast	3062	6.19	1260	15.03
500	16	3	Slow	3146	6.02	4312	4.39
500	16	3	Fast	3056	6.20	420	45.10
1000	4	1	Slow	14348	2.63	28490	1.32
1000	4	1	Fast	14327	2.63	2775	13.60
1000	4	3	Slow	14909	2.53	9548	3.95
1000	4	3	Fast	14442	2.61	930	40.57
1000	16	1	Slow	5725	6.59	28490	1.32
1000	16	1	Fast	6114	6.17	2775	13.60
1000	16	3	Slow	6742	5.60	9548	3.95
1000	16	3	Fast	6182	6.10	930	40.57
2000	4	1	Slow	26748	2.71	52360	1.38
2000	4	1	Fast	26981	2.68	5100	14.19
2000	4	3	Slow	27229	2.66	17556	4.12
2000	4	3	Fast	26367	2.75	1710	42.33
2000	16	1	Slow	20199	3.58	52360	1.38
2000	16	1	Fast	9958	7.27	5100	14.91
2000	16	3	Slow	10030	7.22	17556	4.12
2000	16	3	Fast	10013	7.23	1710	42.33
5000	4	1	Slow	66848	2.86	135674	1.45
5000	4	1	Fast	66727	2.87	13215	14.71
5000	4	3	Slow	67836	2.82	45276	4.35
5000	4	3	Fast	68767	2.78	4410	44.07
5000	16	1	Slow	23271	8.23	135674	1.45
5000	16	1	Fast	23014	8.32	13215	14.71
5000	16	3	Slow	24081	7.95	45276	4.35
5000	16	3	Fast	22662	8.45	4410	44.07

Table 10b. Multiprocessor Execution of Second Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Random		DFS	
				Execution Time	Base Speedup	Execution Time	Base Speedup
500	4	1	Slow	7752	2.78	15246	1.41
500	4	1	Fast	7407	2.91	1485	14.52
500	4	3	Slow	7115	3.03	6622	3.26
500	4	3	Fast	7562	2.85	495	43.56
500	16	1	Slow	3059	7.05	15246	1.41
500	16	1	Fast	2717	7.94	1485	14.52
500	16	3	Slow	3070	7.02	5082	4.24
500	16	3	Fast	2977	7.24	495	43.56
1000	4	1	Slow	14087	2.81	28644	1.38
1000	4	1	Fast	16136	2.45	2790	14.19
1000	4	3	Slow	15070	2.63	9548	4.15
1000	4	3	Fast	13802	2.87	930	42.56
1000	16	1	Slow	6236	6.35	28644	1.38
1000	16	1	Fast	5541	7.14	2790	14.19
1000	16	3	Slow	6738	5.87	9548	4.15
1000	16	3	Fast	5984	6.61	930	42.56
2000	4	1	Slow	27946	2.72	53130	1.43
2000	4	1	Fast	27656	2.75	5175	14.70
2000	4	3	Slow	27719	2.74	17710	4.30
2000	4	3	Fast	27567	2.76	1725	44.10
2000	16	1	Slow	10620	7.16	53130	1.43
2000	16	1	Fast	9902	7.68	5175	14.70
2000	16	3	Slow	12358	6.16	17710	4.30
2000	16	3	Fast	10207	7.45	1725	44.10
5000	4	1	Slow	97790	1.99	136598	1.42
5000	4	1	Fast	65776	2.95	13305	14.61
5000	4	3	Slow	65918	2.95	45584	4.26
5000	4	3	Fast	67082	2.90	4440	43.77
5000	16	1	Slow	24479	7.94	136598	1.42
5000	16	1	Fast	23370	8.32	13305	14.61
5000	16	3	Slow	22478	8.65	45584	4.26
5000	16	3	Fast	23974	8.11	4440	43.77

Table 10c. Multiprocessor Execution of Third Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Random		DFS	
				Execution Time	Base Speedup	Execution Time	Base Speedup
500	4	1	Slow	7558	2.61	14476	1.36
500	4	1	Fast	7453	2.64	1410	13.98
500	4	3	Slow	7518	2.62	4928	4.00
500	4	3	Fast	7007	2.81	480	41.07
500	16	1	Slow	6315	3.12	14476	1.36
500	16	1	Fast	2807	7.02	1410	13.98
500	16	3	Slow	3356	5.87	4928	4.00
500	16	3	Fast	3173	6.21	480	41.07
1000	4	1	Slow	14762	2.72	27412	1.47
1000	4	1	Fast	13256	3.03	2670	15.05
1000	4	3	Slow	15518	2.59	9240	4.35
1000	4	3	Fast	14456	2.78	900	44.66
1000	16	1	Slow	5504	7.30	27412	1.47
1000	16	1	Fast	5604	7.17	2670	15.05
1000	16	3	Slow	6100	6.59	9240	4.35
1000	16	3	Fast	6297	6.38	900	44.66
2000	4	1	Slow	27100	2.97	55594	1.45
2000	4	1	Fast	27376	2.94	5415	14.87
2000	4	3	Slow	28792	2.80	18634	4.32
2000	4	3	Fast	26696	3.02	1815	44.38
2000	16	1	Slow	10935	7.37	55594	1.45
2000	16	1	Fast	10242	7.86	5415	14.87
2000	16	3	Slow	15518	5.19	18634	4.32
2000	16	3	Fast	14456	5.57	1815	44.38
5000	4	1	Slow	67346	2.85	136906	1.40
5000	4	1	Fast	67256	2.85	13335	14.39
5000	4	3	Slow	65048	2.95	45738	4.20
5000	4	3	Fast	65042	2.95	4455	43.07
5000	16	1	Slow	24168	7.94	136906	1.40
5000	16	1	Fast	23820	8.06	13335	14.39
5000	16	3	Slow	23829	8.05	45738	4.20
5000	16	3	Fast	25327	7.58	4455	43.07

Table 10d. Multiprocessor Execution of Fourth Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Random		DFS	
				Execution Time	Base Speedup	Execution Time	Base Speedup
500	4	1	Slow	8830	2.23	12320	1.60
500	4	1	Fast	7496	2.63	1560	12.64
500	4	3	Slow	7152	2.76	4158	4.74
500	4	3	Fast	7362	2.68	405	48.67
500	16	1	Slow	3355	5.88	12320	1.60
500	16	1	Fast	3556	5.54	1200	16.43
500	16	3	Slow	3272	6.02	4158	4.74
500	16	3	Fast	3016	6.54	405	48.67
1000	4	1	Slow	14878	2.61	28798	1.35
1000	4	1	Fast	16093	2.41	2805	13.84
1000	4	3	Slow	14248	2.72	9702	4.00
1000	4	3	Fast	14808	2.62	945	41.07
1000	16	1	Slow	11088	3.50	28798	1.35
1000	16	1	Fast	5662	6.85	2805	13.84
1000	16	3	Slow	5632	6.89	9702	4.00
1000	16	3	Fast	5808	6.68	945	41.07
2000	4	1	Slow	29275	2.66	57596	1.35
2000	4	1	Fast	29168	2.67	5610	13.86
2000	4	3	Slow	31759	2.45	19250	4.04
2000	4	3	Fast	28842	2.70	1875	41.48
2000	16	1	Slow	10719	7.26	57596	1.35
2000	16	1	Fast	10197	7.63	5610	13.86
2000	16	3	Slow	10684	7.28	19250	4.04
2000	16	3	Fast	10922	7.12	1875	41.48
5000	4	1	Slow	67158	2.94	140756	1.40
5000	4	1	Fast	67093	2.94	13710	14.38
5000	4	3	Slow	68353	2.88	46970	4.20
5000	4	3	Fast	66847	2.95	5025	39.23
5000	16	1	Slow	23460	8.40	140756	1.40
5000	16	1	Fast	22882	8.61	13710	14.38
5000	16	3	Slow	24598	8.01	46970	4.20
5000	16	3	Fast	23336	8.45	4575	43.09

Table 11a. DFS Execution Time for First Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time (VAX 8600) (CPU Seconds)
500	4	1	Slow	277.15
500	4	1	Fast	194.45
500	4	3	Slow	245.87
500	4	3	Fast	181.63
500	16	1	Slow	275.10
500	16	1	Fast	191.11
500	16	3	Slow	241.89
500	16	3	Fast	178.53
1000	4	1	Slow	975.88
1000	4	1	Fast	759.34
1000	4	3	Slow	769.78
1000	4	3	Fast	447.75
1000	16	1	Slow	972.41
1000	16	1	Fast	758.40
1000	16	3	Slow	763.67
1000	16	3	Fast	447.70
2000	4	1	Slow	2953.53
2000	4	1	Fast	1485.37
2000	4	3	Slow	2336.14
2000	4	3	Fast	1278.36
2000	16	1	Slow	2877.93
2000	16	1	Fast	1471.37
2000	16	3	Slow	2324.50
2000	16	3	Fast	1283.85
5000	4	1	Slow	31730.40
5000	4	1	Fast	10065.54
5000	4	3	Slow	14476.42
5000	4	3	Fast	8729.60
5000	16	1	Slow	31572.41
5000	16	1	Fast	10100.74
5000	16	3	Slow	14403.45
5000	16	3	Fast	8584.48

Table 11b. DFS Execution Time for Second Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time (VAX 8600) (CPU Seconds)
500	4	1	Slow	344.32
500	4	1	Fast	199.45
500	4	3	Slow	296.75
500	4	3	Fast	186.65
500	16	1	Slow	342.10
500	16	1	Fast	195.20
500	16	3	Slow	270.91
500	16	3	Fast	182.79
1000	4	1	Slow	873.13
1000	4	1	Fast	478.83
1000	4	3	Slow	639.30
1000	4	3	Fast	412.80
1000	16	1	Slow	870.39
1000	16	1	Fast	474.96
1000	16	3	Slow	637.32
1000	16	3	Fast	410.71
2000	4	1	Slow	2767.33
2000	4	1	Fast	1551.52
2000	4	3	Slow	1962.73
2000	4	3	Fast	1320.21
2000	16	1	Slow	2771.63
2000	16	1	Fast	1539.90
2000	16	3	Slow	1944.41
2000	16	3	Fast	1273.62
5000	4	1	Slow	26731.74
5000	4	1	Fast	10749.80
5000	4	3	Slow	15252.81
5000	4	3	Fast	9049.96
5000	16	1	Slow	26782.43
5000	16	1	Fast	10501.85
5000	16	3	Slow	15208.50
5000	16	3	Fast	9072.57

Table 11c. DFS Execution Time for Third Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time (VAX 8600) (CPU Seconds)
500	4	1	Slow	266.60
500	4	1	Fast	219.95
500	4	3	Slow	246.26
500	4	3	Fast	175.62
500	16	1	Slow	263.53
500	16	1	Fast	217.50
500	16	3	Slow	242.10
500	16	3	Fast	173.42
1000	4	1	Slow	844.72
1000	4	1	Fast	520.89
1000	4	3	Slow	702.78
1000	4	3	Fast	441.53
1000	16	1	Slow	845.61
1000	16	1	Fast	508.42
1000	16	3	Slow	701.18
1000	16	3	Fast	443.48
2000	4	1	Slow	2749.98
2000	4	1	Fast	2501.39
2000	4	3	Slow	2171.80
2000	4	3	Fast	1365.53
2000	16	1	Slow	2777.63
2000	16	1	Fast	2484.89
2000	16	3	Slow	2213.31
2000	16	3	Fast	1370.68
5000	4	1	Slow	32575.77
5000	4	1	Fast	11157.41
5000	4	3	Slow	16670.36
5000	4	3	Fast	9805.20
5000	16	1	Slow	32170.55
5000	16	1	Fast	11183.61
5000	16	3	Slow	16694.78
5000	16	3	Fast	9676.86

Table 11d. DFS Execution Time for Fourth Set of Random Graphs

Number of Nodes	Number of Processors	Number of Functional Units of Each Type	Functional Unit Speed	Execution Time (VAX 8600) (CPU Seconds)
500	4	1	Slow	304.99
500	4	1	Fast	201.41
500	4	3	Slow	245.96
500	4	3	Fast	179.58
500	16	1	Slow	302.71
500	16	1	Fast	185.68
500	16	3	Slow	243.28
500	16	3	Fast	176.89
1000	4	1	Slow	955.11
1000	4	1	Fast	686.28
1000	4	3	Slow	642.94
1000	4	3	Fast	424.14
1000	16	1	Slow	946.32
1000	16	1	Fast	688.54
1000	16	3	Slow	637.21
1000	16	3	Fast	422.87
2000	4	1	Slow	3284.10
2000	4	1	Fast	1448.37
2000	4	3	Slow	2123.33
2000	4	3	Fast	1242.76
2000	16	1	Slow	3310.79
2000	16	1	Fast	1449.36
2000	16	3	Slow	2098.75
2000	16	3	Fast	1235.31
5000	4	1	Slow	24941.39
5000	4	1	Fast	10591.12
5000	4	3	Slow	17630.94
5000	4	3	Fast	9935.20
5000	16	1	Slow	24307.43
5000	16	1	Fast	10549.28
5000	16	3	Slow	17421.24
5000	16	3	Fast	9483.44

4.4. Conclusions

The simulation results presented in this chapter show that the DFS algorithm provides significant improvement over uniprocessor execution of algorithms. The DFS algorithm also outperforms random partitioning for a number of cases. Random partitioning is not affected by changes in the power of individual processors in a multiprocessor architecture. Thus little improvement in execution time can be expected when processors are improved if random partitioning is used to schedule algorithms. Random partitioning of random graphs is more effective than DFS partitioning of the same graphs when the slow functional units are used. Modeling the simulation results will be explored in the next chapter. The execution time of the DFS algorithm grows at a rate less than n^2, where n is the number of nodes in the graph being scheduled, for all the simulations. This result is significantly less than the theoretical result of n^5, which was cited in the previous chapter.

All of the graphs used in these simulations were shown to contain relatively large amounts of parallelism. Further simulations could explore graphs containing less parallelism. Examining the performance of the scheduling algorithms on graphs coming from working computer programs would also be helpful in further evaluation of the DFS algorithm.

Chapter 5: Modeling Multiprocessor Execution Times

5.1. Overview

The work presented in the previous chapters deals with an aspect of the problem of automatically mapping algorithms to VLSI architectures, specifically the automatic partitioning of algorithms for execution on parallel architectures. The simulation results discussed in the last chapter show that, for a wide range of cases, significant speedup over uniprocessor execution can be obtained using the DFS partitioning algorithm. Random partitioning was shown to produce good speedup in some test cases, occasionally outperforming the DFS. This chapter presents heuristic, analytic models describing the execution times of graphs partitioned using the DFS and random partitioning algorithms. These models are based on the simulation results.

Such models are helpful for several reasons. First, the performance of the DFS and random strategies varies widely depending on the data flow graphs and architectures used. Good models allow the selection of the strategy most suited to a given situation prior to the execution of the partitioning algorithm. Second, evaluation of a possible partitioning currently requires the use of a computationally expensive simulator (which uses as much as 50000 seconds of VAX 8600 CPU time for large graphs). With very little computational cost, the models predict the performance of a specific partitioning. Finally, and most importantly, the models provide further insight into the partitioning problem, which will allow the expansion of the results obtained in this research to other, related areas.

To be useful, the models must meet several criteria. First, they must accurately describe the simulation results. Second, they must be explainable in intuitive terms. It is certainly possible to develop a set of equations that fit the simulation results, but their validity outside the bounds of the specific tests performed here requires that they fit the situation being modeled. Third, the models must behave at various boundaries of the problem space, e.g., for architectures with one processor or with infinitely fast functional

units. Finally, since the models are heuristic in nature, it is not claimed that they are the only models describing the simulation results, but rather that they are useful and helpful in describing the results and in expanding the bounds of the research presented here.

5.2. General Model

The simulation results suggest that the details of characterization will differ for different partitioning strategies and graph topologies. However, these specific characterizations are special cases of a more general model which is presented and justified in this section. Recall that execution time, τ, is modeled as a function of several variables:

$$\tau=\tau(\Pi,n_P,n_F,X,t_{gc},t_{lc},M(G))$$

Several properties of the execution time function, f, can be observed without defining a more detailed model. First, when the number of processors, n_P, is 1, execution times obtained using any partitioning strategy must be identical to those obtained for uniprocessor execution of the same graph. Second, as the number of processors and the number of functional units of each type, n_F, grow, τ should be primarily determined by graph characteristics, e.g., for some graphs some processors and functional units may be under-utilized because of data dependencies. Third, if the function execution times, X, approach 0, uniprocessor execution time will approach 0. Fourth, if global communication time, t_{gc}, approaches 0 while the entries in X remain non-zero, execution time will be primarily determined by graph characteristics, n_P, and n_F. In the binary merge graphs used in the simulations, for example, if t_{gc}=0, execution time depends on the number of processors and functional units, up to the limit of parallel operations available in the graph. Finally, if both t_{gc} and the entries in X go to 0, both uniprocessor and multiprocessor execution times go to 0.

The problem being considered by this research is completely defined by the above parameters and the models to be presented below are given in terms of these parameters only. In addition, the nature of the simulations whose results are being modeled allow

two simplifying assumptions. First, each processor in the architecture is assumed to have the same number of functional units of each type, n_F, i.e., $F_O=(n_F, n_F, \cdots, n_F)$. Second, the global communication data block width, w_{gc}, is assumed to be wide enough to communicate any result in a single global communication time, t_{gc}. In the simulations, the sizes of all operations and arcs were set equal to w_{gc}. This assumption could easily be relaxed by replacing references to t_{gc} with $\left\lceil \frac{S_O(O_i)}{w_{gc}} \right\rceil \cdot t_{gc}$ for operation O_i.

Given the above assumptions and criteria, the execution time of a graph on a multiprocessor architecture can be defined as:

$$\tau(\Pi, n_P, n_F, X, t_{gc}, t_{lc}, M(G)) = \alpha_{gc} n_{gc} t_{gc} + \sum_{i=0}^{m-1} \alpha_{O_i} n_{O_i} x_{O_i} + \alpha_{lc} n_{lc} t_{lc}$$

where,

τ is the execution time for a given partitioning of the graph.

α_{gc} gives the fraction of the global communication arcs that do not overlap other operations and communications during graph execution.

n_{gc} is the the total number of global communication arcs for a given partitioning of the graph.

α_{O_i} gives the fraction of nodes performing operation O_i that do not overlap other operations and communications during graph execution.

n_{O_i} is the number of nodes in the graph performing operation O_i.

α_{lc} gives the fraction of the local communications arcs that do not overlap other operations and communications during graph execution.

n_{lc} is the number of local communications arcs for a given partitioning of the graph.

The above model treats communication identically with operations performed at nodes in the graph. Given a partitioning, the model works with an augmented graph, with each arc, A, replaced by a node and two new arcs, one from the head of A to the new node, and one from the new node to the tail of A. The new node performs an

operation that either requires t_{lc} (for a local communication) or t_{gc} (for a global communication) to complete. No time is required for data to traverse the new arcs. The only difference between communications and operations is that the number of global and local communication nodes is determined by the partitioning of the graph, while the number of nodes performing a given operation is fixed for a given graph, i.e., n_{gc} and n_{lc} are functions of the graph partitioning, while the n_{O_i}'s are not. In light of this equivalence, the following discussion will use operations to refer to both communication and node operations.

The concept of overlap needs to be more precisely defined. At any instant during the execution of the graph, several operations may be proceeding simultaneously, i.e., they are overlapped. In estimating the execution time of the graph, only one of the operations needs to be included in calculating the α for that operation type. Several possibilities are available for choosing which operation to include. As will be shown in the following sections, different choices may be required for different graphs and partitioning algorithms. If an included operation partially overlaps another operation, then some fraction of the second operation must be included in the α for that operation type as well. The α's are functions of the graph, the partitioning, n_P, n_F, t_{gc}, t_{lc}, and X.

Figure 12 shows a graph that will be used as a specific example of applying the model. The graph has been partitioned into two clusters, as indicated by the dashed boxes. Each cluster will execute on a separate processor. Figure 13 illustrates the graph after augmentation. The nodes have been numbered for reference. Note that the global communication (node 3) has not been placed in either cluster, as it makes use of the communication processor at both clusters. Figure 14 shows a Gantt chart for the execution of the graph in Figure 13. The Gantt chart assumes that local communication requires 1 time step, addition requires 4 time steps, multiplication, 37 time steps, and global communication, 40 time steps.

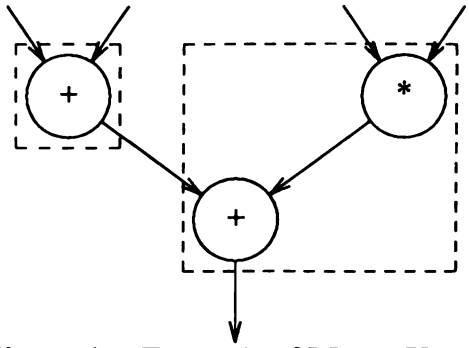

Figure 12. Example of Model Usage

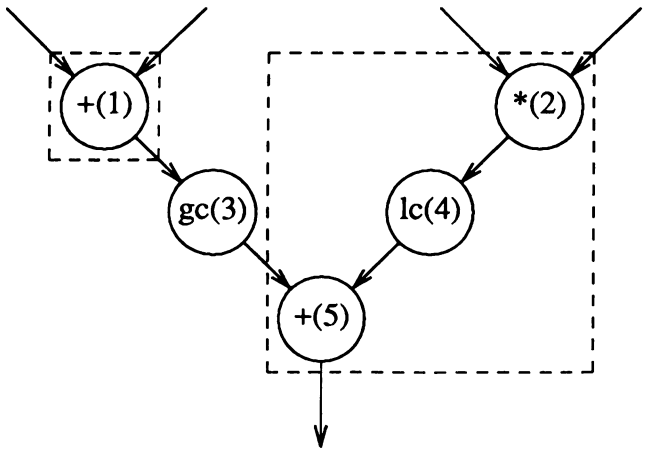

Figure 13. Example Augmented Graph

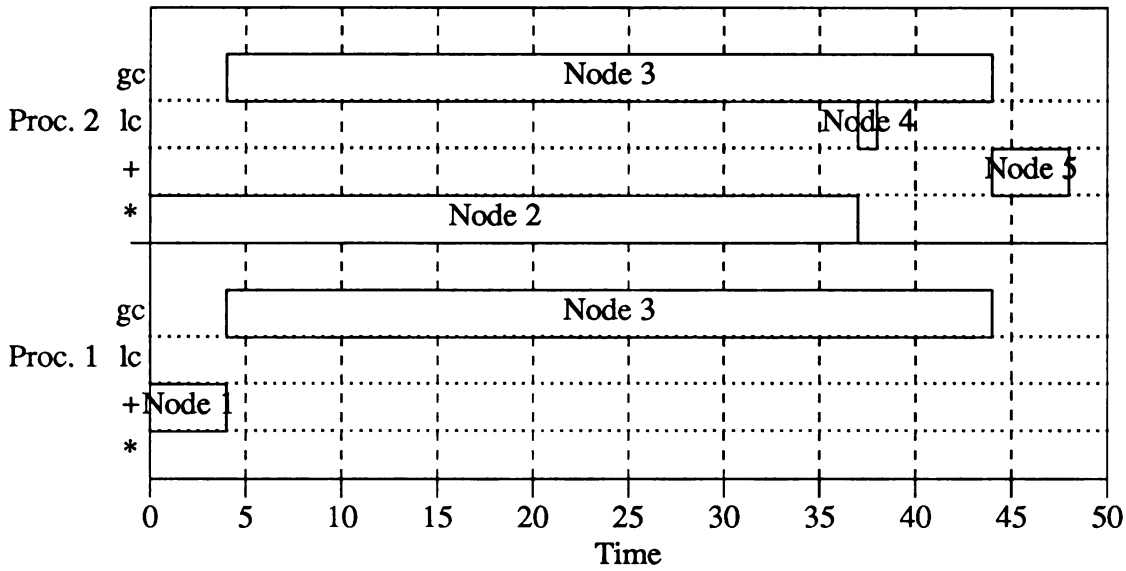

Figure 14. Gantt Chart for Example Graph

From the Gantt chart, the various α's can be calculated. In this example, the longest operation type is chosen to be included when operations overlap. Since global communication is the longest operation, α_{gc} must equal one, due to the fact that the longest operation is included for any overlap. (If the graph contained several global communications which overlapped, then α_{gc} could be less than one to account for this parallelism.) Looking next at multiplication, node 2 overlaps node 3 for $\frac{33}{37}$ of node 2's execution. Thus $\alpha_{*}=\frac{4}{37}$. Moving to addition, note that node 2 completely overlaps node 1, while nothing overlaps the execution of node 5. It follows that $\alpha_{+}=\frac{1}{2}$. Finally, for local communication, node 4 is completely overlapped by node 3, and thus $\alpha_{lc}=0$. The execution time for this partitioning of the graph is then:

$$\tau=1\times1\times40+\frac{4}{37}\times1\times37+\frac{1}{2}\times2\times4+0\times1\times1=48$$

This agrees with the execution time shown in the Gantt chart. Note that for this

example, the execution time is the longest path through the augmented graph. This is not generally the case, since the number of functional units and processors will limit the exploitation of the graph's parallelism. Obviously, if a Gantt chart is available for a partitioning of a graph, there is no need to apply the model to estimate execution time. The remainder of this chapter develops formulae for the various α's, which allow the execution times of graph partitionings to be estimated without full knowledge of the graph's execution schedule.

The remainder of this chapter contains four sections. The first section models the execution of binary merge graphs for uniprocessor execution and both random and DFS partitioning. The second section does the same for the FFT graphs. The third section presents models for the execution of random graphs. Finally, the chapter concludes with a section of general comments on the modeling of the results.

5.3. Modeling Execution Times for Binary Merge Graphs

Among the graph topologies tested, the development of execution time models is simplest for the binary merge graphs. These graphs are very regular in structure and are also very simple.

5.3.1. Uniprocessor Execution

The model for uniprocessor execution of the binary merge graphs follows from completely deterministic information. This model thus produces exact execution times. Several facts about the binary merge graphs are used in developing the model. First, the graphs are described as having 2^n inputs. Second, a graph has a total of 2^n-1 operation nodes, of which 2^{n-1} perform multiplication and $2^{n-1}-1$ perform addition. Finally, a graph has 2^n-2 local communication arcs, which will become local communication nodes during the augmentation of the graph.

The graph shown in Figure 15 is an example of an augmented binary merge graph with 8 (2^3) inputs. It has 4 multiplication nodes, 3 addition nodes, and 6 local

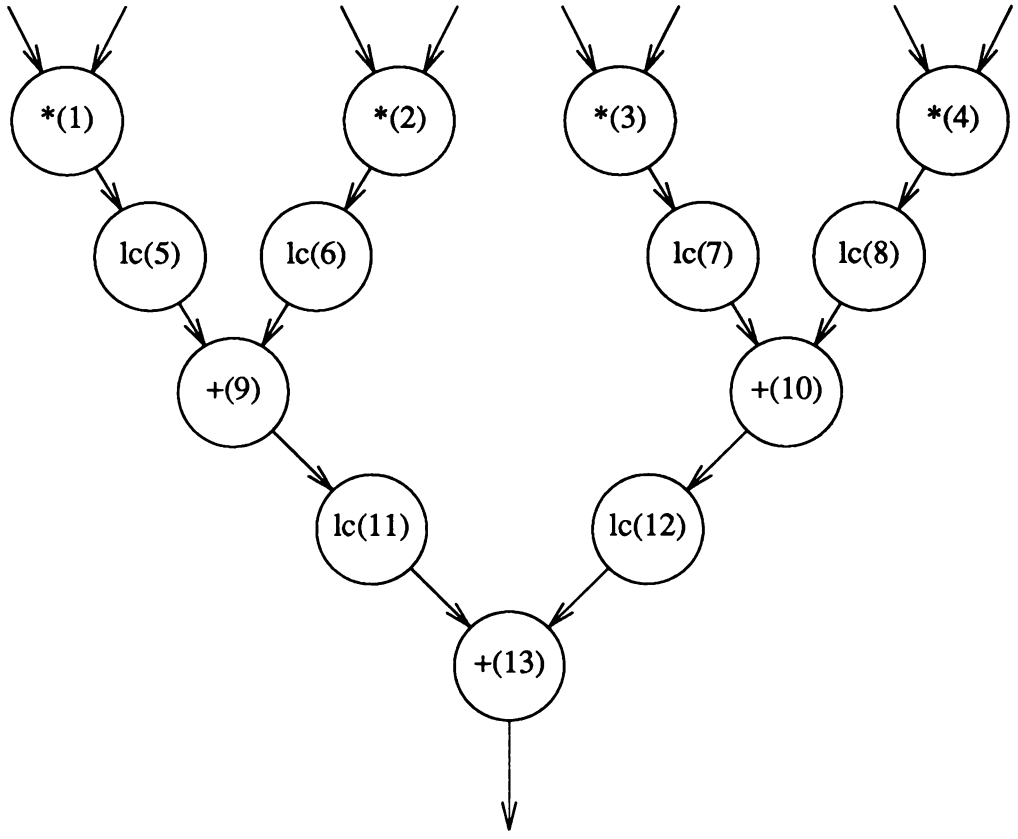

Figure 15. Example Binary Merge Graph

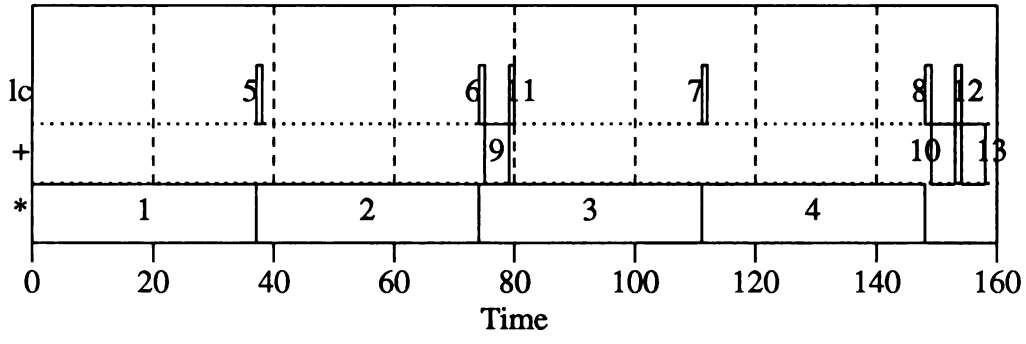

Figure 16. Gantt Chart for Single Functional Units

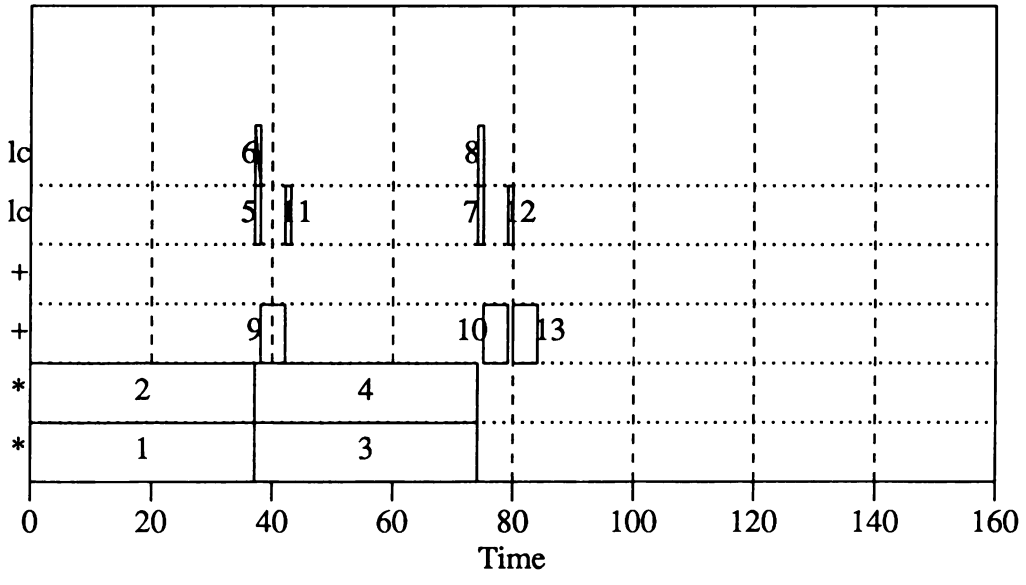

Figure 17. Gantt Chart for Double Functional Units

Table 12. Model for Binary Merge Graphs on Uniprocessors

Number of Nodes	n_F	Functional Unit Speed	Actual Execution Times (Simulated)	Predicted Execution Times (Modeled)
127	1	Slow	2398	2398
127	1	Fast	396	396
127	3	Slow	844	844
127	3	Fast	144	144
255	1	Slow	4771	4771
255	1	Fast	782	782
255	3	Slow	1626	1626
255	3	Fast	272	272
511	1	Slow	9512	9512
511	1	Fast	1552	1552
511	3	Slow	3222	3222
511	3	Fast	532	532
1023	1	Slow	18989	18989
1023	1	Fast	3090	3090
1023	3	Slow	6372	6372
1023	3	Fast	1044	1044
2047	1	Slow	37938	37938
2047	1	Fast	6164	6164
2047	3	Slow	12704	12704
2047	3	Fast	2072	2072
4095	1	Slow	75831	75831
4095	1	Fast	12310	12310
4095	3	Slow	25326	25326
4095	3	Fast	4120	4120

communication nodes. Figure 16 shows a Gantt chart illustrating the execution of the graph on a single processor with a single functional unit of each type (i.e., $n_F=1$), given that the critical path algorithm is used to schedule computations. Figure 17 shows a Gantt chart illustrating the execution of the graph on a processor with two functional units of each type. Notice that the execution time for the graph is determined by the time taken to complete all of the graph's multiplications, followed by the time needed to compute one path of additions.

The behavior of the example can be generalized to model uniprocessor execution of a binary merge tree of any size (subject only to the constraint that the number of inputs can be expressed as an integral power of two) on a single processor with any number of functional units. Specifically, since there are no global communications in the graph, α_{gc} is 0. The overlap of multiplications is determined by the number of functional units, giving $\alpha_* = \frac{\left\lceil \frac{2^{n-1}}{n_F} \right\rceil}{2^{n-1}}$. The number of additions and local communications on one path from a multiplication to an exit node is $n-1$, so $\alpha_+ = \frac{n-1}{2^{n-1}-1}$, and $\alpha_{lc} = \frac{n-1}{2^n-2}$. This gives the final equation for uniprocessor execution time of binary merge graphs as

$$\tau = \left\lceil \frac{2^{n-1}}{n_F} \right\rceil \times t_* + (n-1) \times t_+ + (n-1) \times t_{lc}.$$

Table 12 presents a comparison between the results predicted by the model and the test results for graphs with 128 to 4096 inputs. Both one and three functional units of each type were used. Two sets of execution times were used. The table shows that the experimental results exactly match the results predicted by the model.

5.3.2. Random Partitioning

The simulated execution times for randomly partitioned binary merge graphs show little variance as the number and speed of functional units is varied. The change in uniprocessor execution time for the same graphs when these parameters are varied indi-

cates that the graphs contain a great deal of potential parallelism. Therefore, the execution of random partitionings of the graphs must be dominated by global communication costs. Thus, the model for random partitioning will include only global communication operations (i.e., among the α's, only α_{gc} is non-zero).

The simulation results vary as the number of processors change. Indeed, the nature of random partitioning is such that the number of global communication arcs, n_{gc}, is a function of the number of processors, n_P. To see this, note that the probability of a node being assigned to a particular processor is $\frac{1}{n_P}$. Once the head or tail of an arc has been assigned to a processor, there is a $\frac{1}{n_P}$ chance of the arc's other node being assigned to the same processor, i.e., of the arc becoming a local communication. Thus, the expected value of n_{gc} is $(1 - \frac{1}{n_P})\times n_A$, where n_A is the total number of arcs in the graph.

Two things make the derivation of a precise value for α_{gc} more difficult than the derivation of n_A. First, the amount of overlap that occurs in any set of parallel operations that includes dependency constraints is hard to determine, due to the many possible execution sequences for the operations. Second, the multiprocessor architecture model used in developing the DFS algorithm does not currently include a scheduling mechanism for global communication. The simulator schedules global communication to communication processors in basically first-come, first-serve order. The order in which communications arrive is influenced by the arbitrary indices assigned to clusters and nodes by the simulator. Augmenting the architecture model to include some scheduling mechanism would ease the problem of deriving α_{gc}, but the general problem of estimating overlap is still an area for further research.

However, even with this problem, upper and lower bounds for α_{gc} can be developed. An obvious upper bound occurs when none of the global communications can overlap, i.e., $\alpha_{gc}=1$. This might be the case in a graph containing many nodes that must be executed serially. To develop a lower bound, note that each global

communication uses the communication processors of two processors in the multiprocessor architecture. Thus, at most $\frac{n_P}{2}$ global communications can proceed in parallel at any given time. It follows that $\alpha_{gc}=\frac{1}{\frac{n_P}{2}}$ is an upper bound on the amount of overlap. Using these bounds and the estimate of n_{gc} from above, and recalling that binary merge graphs contain 2^n-2 arcs, upper and lower bounds on expected execution time, τ_l and τ_u are given by the following:

$$\tau_l=\frac{(2^n-2)\times(1-\frac{1}{n_P})\times t_{gc}}{\frac{n_P}{2}}, \text{ and}$$

$$\tau_u=(2^n-2)\times(1-\frac{1}{n_P})\times t_{gc}.$$

Table 13. Model for Random Partitioning of Binary Merge Graphs

Number of Nodes	Number of Arcs	Number of Processors	Average of Actual Execution Times (Simulated)	Lower Bound on Execution Times (Modeled)	Upper Bound on Execution Times (Modeled)	Percent Error of Lower Bound	Percent Error of Upper Bound
127	126	4	2307	1890	3780	18.1	63.8
127	126	16	1241	591	4725	52.4	280.7
255	254	4	4428	3810	7620	14.0	72.1
255	254	16	1896	1191	9525	37.2	402.4
511	510	4	8529	7650	15300	10.3	79.4
511	510	16	3283	2391	19125	27.2	482.5
1023	1022	4	16620	15330	30660	7.8	84.5
1023	1022	16	5988	4791	38325	20.0	540.0
2047	2046	4	32831	30690	61380	6.5	87.0
2047	2046	16	11561	9591	76725	17.0	563.7
4095	4094	4	63319	61410	122820	3.0	94.0
4095	4094	16	22071	19191	153525	13.0	595.6

Table 13 compares the modeled results with the results obtained from the simulations, showing the percent error of the bounds relative to the simulation results. (For

simulated execution time τ, the percent error of the lower bound, for example, is calculated as the absolute value of $\frac{\tau_l - \tau}{\tau} \times 100\%$.) Since little variance resulted from changing the number and speed of functional units, each set of four simulation results are shown as a single average in the table. The results are not derived from deterministic information, so an exact match between the model and the simulation results does not occur. From the table, it appears that the lower bound is a better approximation than the upper bound. Notice that the error for τ_l generally decreases as the number of arcs in the graph increases. This is reasonable, as having more arcs will, in general, improve the untilization of the available communication processors. Also note that the results are poorer with more processors for a given graph size. This follows from the fact that more communication processors are available, while the number of arcs remains constant. While the modeled results are a reasonable approximation of the simulation results, improvement needs to be made in the estimation of the overlap among global communication operations.

5.3.3. DFS Partitioning

The partitioning of binary merge graphs resulting from the application of the DFS algorithm is highly regular. This regularity allows the model to predict exact execution times. Figure 18 shows the DFS partitioning for execution on two processors of the binary merge graph with eight inputs. Figure 19 shows the DFS partitioning of the same graph for execution on a four processor architecture. The binary merge graphs again have 2^{n-1} multiplication nodes and $2^{n-1}-1$ addition nodes. Now, however, the number of global and local communication arcs is determined by the number of clusters into which the graph is partitioned. Specifically, for 2^m clusters, there are 2^m-1 global communication arcs and 2^n-2^m-1 local communication arcs. Notice, each cluster includes a subtree of the entire binary merge graph and that the computation in each cluster can proceed in parallel with the other clusters, except near the root of the graph.

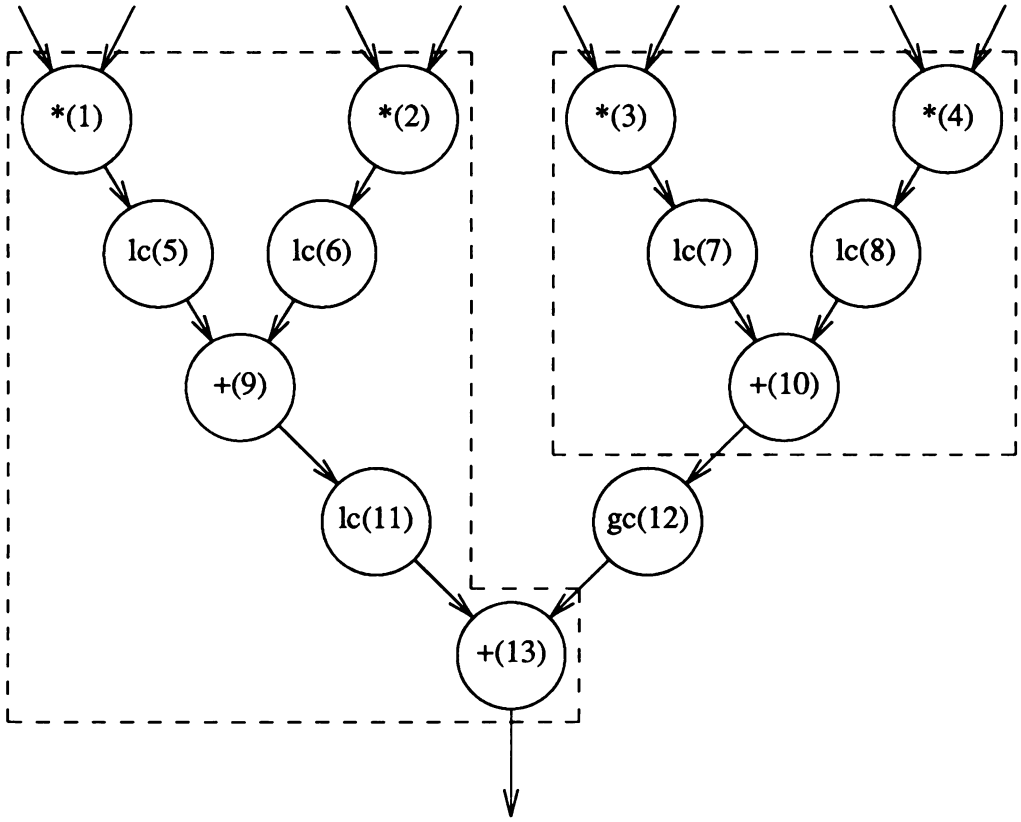

Figure 18. DFS Two Processor Partitioning of Binary Merge Graph

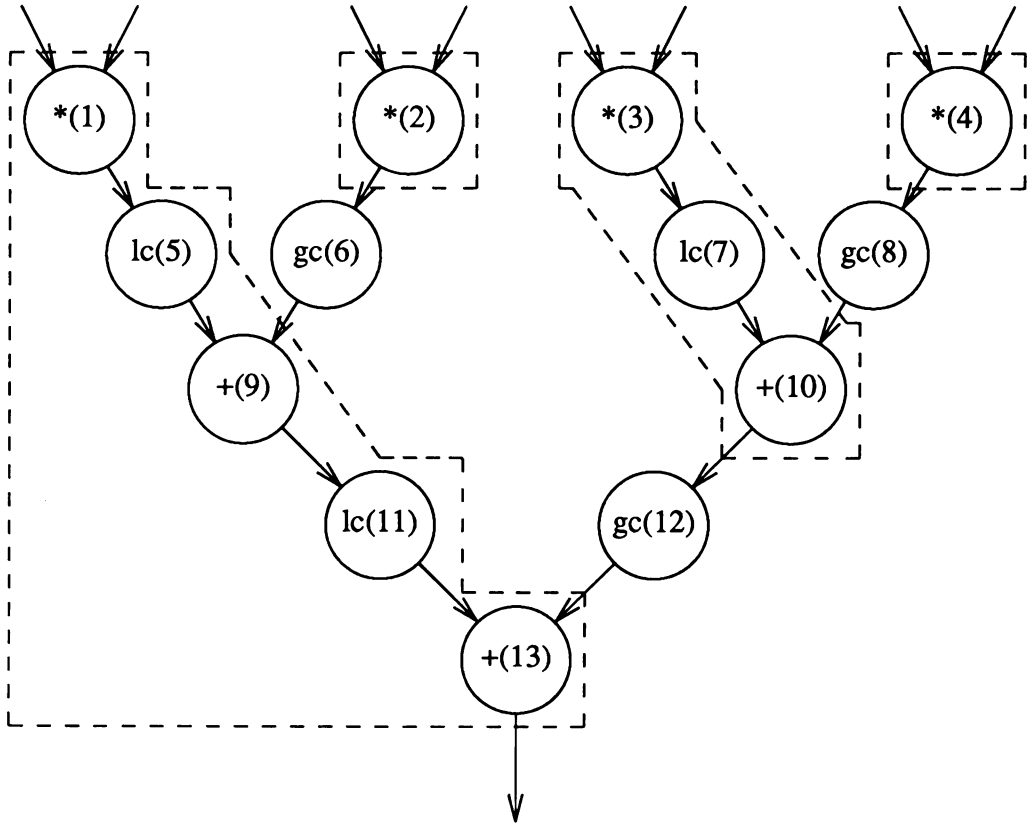

Figure 19. DFS Four Processor Partitioning of Binary Merge Graph

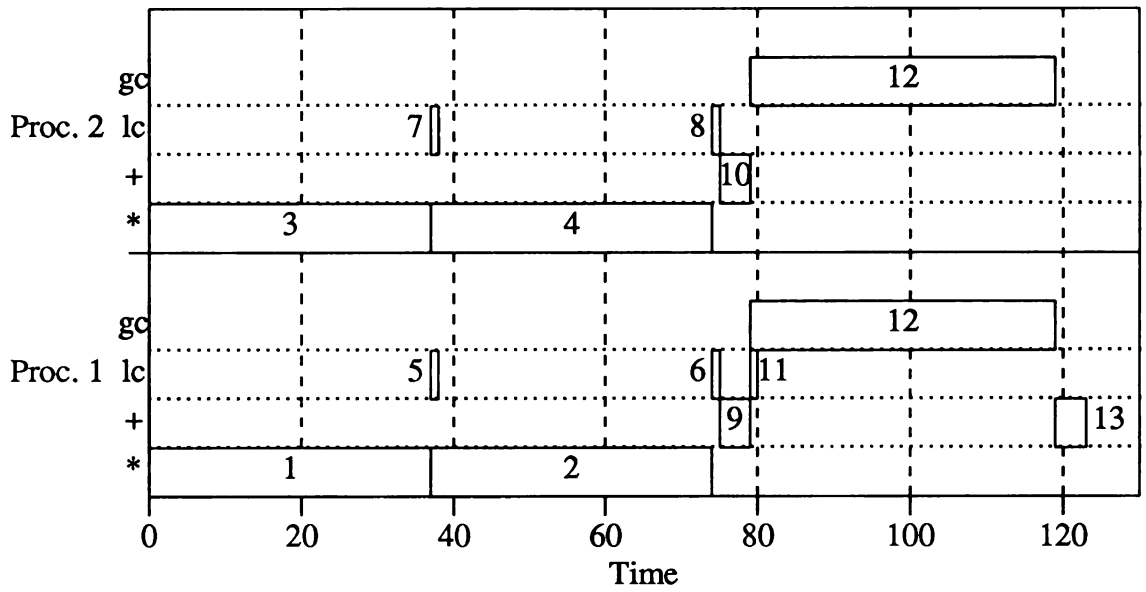

Figure 20. Gantt Chart for Two Processors

Figure 20 shows a Gantt chart for the execution of the partitioned 8-input binary merge graph on an architecture with two processors, with each processor having one functional unit of each type. Figure 21 shows the Gantt chart for the execution of the 8-input graph partitioned for an architecture with four processors, each having one functional unit of each type. Notice that, once again, the execution time is determined by the time required to complete all of the multiplications followed by the time needed to traverse a path from a multiplication node to the output node of the graph.

The results in the examples generalize to give the following model: There are m global communication arcs in a path from a multiplication node to the output, so $\alpha_{gc}=\frac{m}{2^m-1}$; extending the results for uniprocessor execution to include multiple processors gives, $\alpha_*=\frac{\left\lceil \frac{2^{n-1}}{n_F 2^m} \right\rceil}{2^{n-1}}$; there are $n-1$ additions on a path from multiply to an output arc, so $\alpha_+=\frac{n-1}{2^{n-1}-1}$. Finally, there are $n-1-m$ local communication arcs on a path from

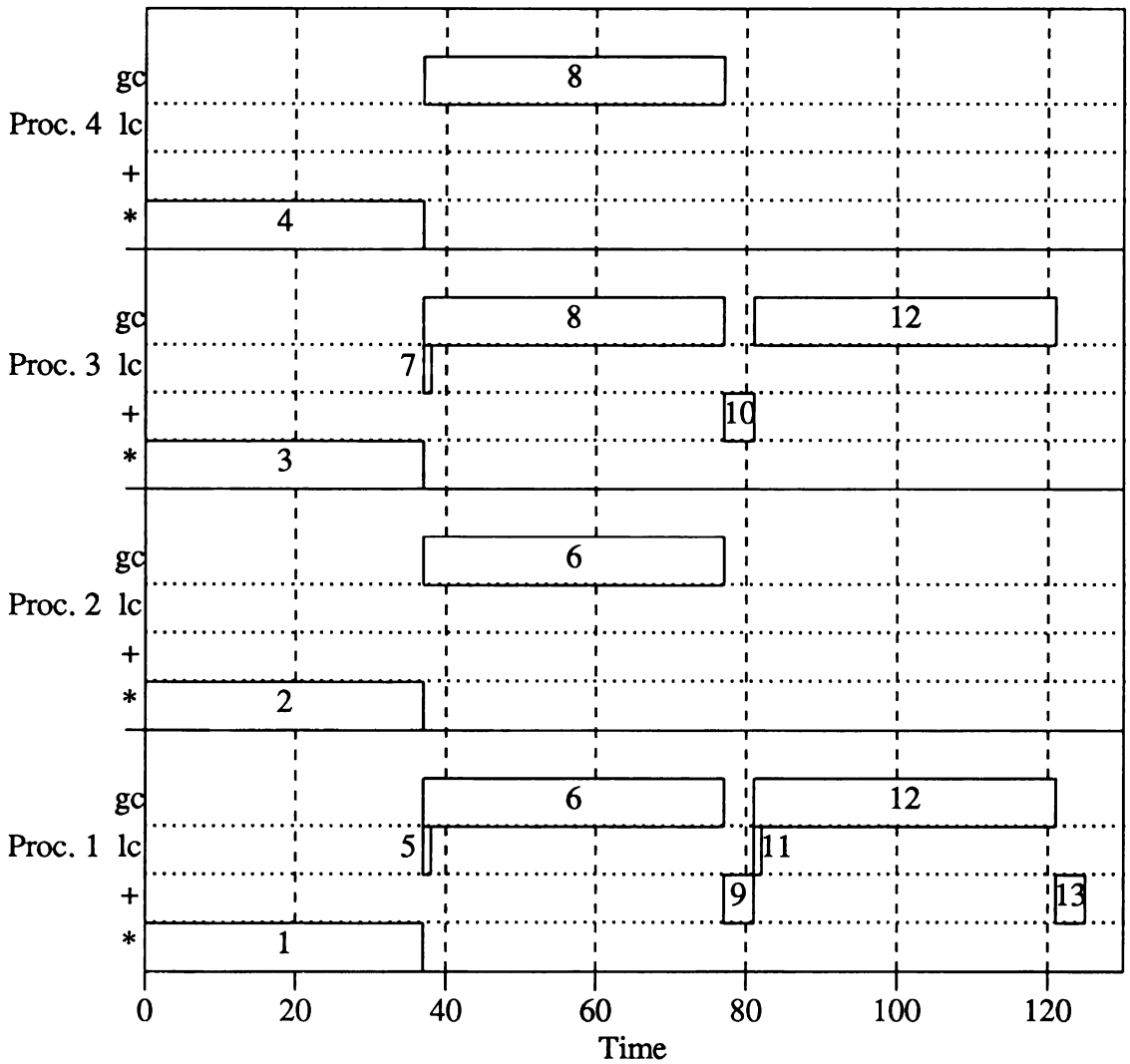

Figure 21. Gantt Chart for Four Processors

multiply to an output arc, so $\alpha_{ic} = \frac{n-1-m}{2^n - 2^m - 1}$.

The above parameters model the execution of a binary merge graph that has been partitioned into 2^m clusters by the DFS algorithm. In some instances, the DFS algorithm chooses m such that 2^m is less than n_P, the number of processors available in the architecture. Thus to completely model the execution time for binary merge graphs partitioned by the DFS, it is necessary to model the DFS algorithm's choice of m.

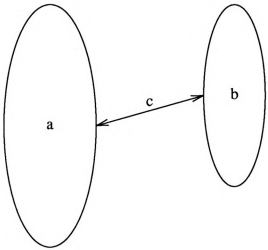

Figure 22. Illustration of DFS Merging Heuristic

The choice of m can be modeled by examining the behavior of the heuristic that controls the merging of two clusters within the DFS. Figure 22 illustrates this heuristic. In the figure, two clusters, a and b, are candidates for merging. The communication between clusters a and b is shown by c in Figure 22. Let the execution time of a on a single processor be denoted by t_a. Similarly, t_b is the time to execute cluster b. Finally, denote the time needed for the intercluster communication by t_c. Note that for binary merge trees there is alway one global communication arc between two clusters that may

be merged. Thus, $t_c = t_{gc}$. The DFS heuristic states that two clusters will be merged if the time needed to execute both clusters on a single processor (denoted $t_{a\&b}$) is less than $\max(t_a, t_b) + t_{gc}$.

Without loss of generality, assume that $t_a \geq t_b$. Then the DFS heuristic merges clusters a and b when $t_a + t_{gc} > t_{a\&b}$. When $t_* \gg t_+$ and $t_* \gg t_{lc}$, which is the case for the simulations, t_a can be approximated as $\left\lceil \frac{2^{n-1}}{n_F 2^m} \right\rceil$, with 2^m being the current number of clusters. (The division by 2^m follows from the fact that multiplications are evenly distributed among the clusters for DFS partitioning.) Similarly, $t_{a\&b}$ can be approximated by $\left\lceil \frac{2^n}{n_F 2^m} \right\rceil$ since there are twice as many multiplications in the merged cluster.

Putting the above results together gives the following model for the execution time of binary merge graphs partitioned for multiprocessor execution using the DFS.

Select the maximum m in the range $1 \leq 2^m \leq n_P$ such that $\dfrac{\left\lceil \frac{2^{n-1}}{n_F 2^m} \right\rceil t_* + t_{gc}}{\left\lceil \frac{2^n}{n_F 2^m} \right\rceil t_*} < 1.$

Then the execution time is given by

$$\tau = m \times t_{gc} + \left\lceil \frac{2^{n-1}}{n_F 2^m} \right\rceil \times t_* + (n-1) \times t_+ + (n-m-1) \times t_{lc}.$$

Table 14 presents a comparison between the results predicted by the model, which shows that the experimental results exactly match the results predicted by the model. Notice also that when the number of processors is set to one, $m=0$, and the model reduces to that developed for uniprocessor execution in the last section.

Table 14. Model for DFS Partitioning of Binary Merge Graphs

Number of Nodes	n_P	n_F	Functional Unit Speed	Actual Execution Times (Simulated)	Predicted Number of Clusters (2^m)	Predicted Execution Times (Modeled)
127	4	1	Slow	700	4	700
127	4	1	Fast	186	4	186
127	4	3	Slow	330	4	330
127	4	3	Fast	117	2	117
127	16	1	Slow	334	16	334
127	16	1	Fast	177	8	177
127	16	3	Slow	258	8	258
127	16	3	Fast	117	1	117
255	4	1	Slow	1297	4	1297
255	4	1	Fast	284	4	284
255	4	3	Slow	520	4	520
255	4	3	Fast	158	4	158
255	16	1	Slow	487	16	487
255	16	1	Fast	218	16	218
255	16	3	Slow	302	16	302
255	16	3	Fast	158	4	158
511	4	1	Slow	2486	4	2486
511	4	1	Fast	478	4	478
511	4	3	Slow	932	4	932
511	4	3	Fast	226	4	226
511	16	1	Slow	788	16	788
511	16	1	Fast	268	16	268
511	16	3	Slow	418	16	418
511	16	3	Fast	199	8	199
1023	4	1	Slow	4859	4	4859
1023	4	1	Fast	864	4	864
1023	4	3	Slow	1714	4	1714
1023	4	3	Fast	354	4	354
1023	16	1	Slow	1385	16	1385
1023	16	1	Fast	366	16	366
1023	16	3	Slow	608	16	608
1023	16	3	Fast	240	16	240

Table 14. (cont'd.)

Number of Nodes	n_P	n_F	Functional Unit Speed	Actual Execution Times (Simulated)	Predicted Number of Clusters (2^m)	Predicted Execution Times (Modeled)
2047	4	1	Slow	9600	4	9600
2047	4	1	Fast	1634	4	1634
2047	4	3	Slow	3310	4	3310
2047	4	3	Fast	614	4	614
2047	16	1	Slow	2574	16	2574
2047	16	1	Fast	560	16	560
2047	16	3	Slow	1020	16	1020
2047	16	3	Fast	308	16	308
4095	4	1	Slow	19077	4	19077
4095	4	1	Fast	3172	4	3172
4095	4	3	Slow	6460	4	6460
4095	4	3	Fast	1126	4	1126
4095	16	1	Slow	4947	16	4947
4095	16	1	Fast	946	16	946
4095	16	3	Slow	1802	16	1802
4095	16	3	Fast	436	16	436

5.4. Modeling Execution Times for FFT Graphs

Modeling execution times for the FFT graphs is more difficult than developing models for binary merge graphs. Like the binary merge graphs, the FFT graphs are very regular in structure. However, the structure of the FFT graphs is complex compared with the simple structure of the binary merge graphs.

5.4.1. Uniprocessor Execution

Although multiprocessor execution of the FFT graphs is difficult to model, uniprocessor execution is simple enough to allow an exact model to be developed. Several facts about the FFT graphs are used in developing this model. First, the FFT's will be described as having 2^n inputs. Such graphs consist of $(n+1)\times 2^n$ multiply nodes and $2n\times 2^n$ local communication arcs, which will become local communication nodes during the augmentation of the graph.

Figure 23 is an example of an augmented FFT graph with 4 inputs. It has 12 multiply nodes and 16 local communication nodes. Figure 24 shows a Gantt chart illustrating the execution of the FFT graph on a uniprocessor system with a single function unit, given that the critical path algorithm is used to schedule computations. The execution shown is very simple, with all local communication overlapped by multiplication, and execution time determined by the time required to complete the multiplications.

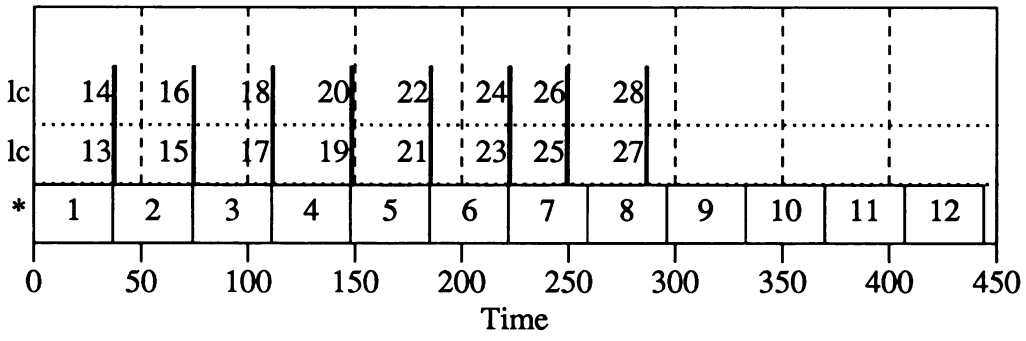

Figure 24. Gantt Chart for Uniprocessor Execution

Table 15. Model for FFT Graphs on Uniprocessors

Number of Nodes	n_F	Functional Unit Speed	Actual Execution Times (Simulated)	Predicted Execution Times (Modeled)
1024	1	Slow	37888	37888
1024	1	Fast	6144	6144
1024	3	Slow	12654	12654
1024	3	Fast	2052	2052
2304	1	Slow	85248	85248
2304	1	Fast	13824	13824
2304	3	Slow	28416	28416
2304	3	Fast	4608	4608
5120	1	Slow	189440	189440
5120	1	Fast	30720	30720
5120	3	Slow	63159	63159
5120	3	Fast	10242	10242

The behavior illustrated in Figure 24 can be generalized to develop a model for uniprocessor execution of the FFT graphs. Here multiplication is the obvious candidate for inclusion when it overlaps other operations. Since the graph contains no global communication nodes, $\alpha_{gc}=0$. Also, since all local communication is overlapped by multiplication, $\alpha_{lc}=0$. Finally, the overlap of multiplications is determined by the number of multiplications and the number of functional units, i.e., $\alpha_* = \frac{\left\lceil \frac{(n+1)\times 2^n}{n_F} \right\rceil}{(n+1)\times 2^n}$. Combining these results gives the final equation:

$$\tau = \left\lceil \frac{(n+1)\times 2^n}{n_F} \right\rceil \times t_* .$$

A comparison of the modeled results and the simulation results is presented in Table 15. Note that the modeled results exactly match the simulation results.

5.4.2. Random Partitioning

As was found for the binary merge graphs, the execution times for the random partionings of the FFT graphs is dominated by communication time. Using a similar derivation to that used for the binary merge graphs, the following bounds on execution time result:

$$\tau_l = \frac{2n\times 2^n \times (1-\frac{1}{n_P})\times t_{gc}}{\frac{n_P}{2}}, \text{ and}$$

$$\tau_u = 2n\times 2^n \times (1-\frac{1}{n_P})\times t_{gc} .$$

Table 16 compares the modeled results with the results obtained from the simulations. Percent error for the upper and lower bounds were calculated as in Table 13. As for the binary merge graphs, each simulation entry in the table is an average of four partitionings, each with a different combination of number of functional units and functional unit speed. These results are similar in nature to the results for binary merge graphs.

Table 16. Model for Random Partitioning of FFT Graphs

Number of Nodes	Number of Arcs	Number of Processors	Average of Actual Execution Times (Simulated)	Lower Bound on Execution Times (Modeled)	Upper Bound on Execution Times (Modeled)	Percent Error of Lower Bound	Percent Error of Upper Bound
1024	1792	4	28353	26880	53760	5.2	89.6
1024	1792	16	10286	8400	67200	18.3	553.3
2304	4096	4	64453	61440	122880	4.7	90.7
2304	4096	16	22083	19200	153600	13.1	595.6
5120	9216	4	14096	138240	276480	1.9	96.1
5120	9216	16	47103	43200	345600	8.3	633.6

Notice that the error for the lower bound is smaller than for the binary merge graphs. This is reasonable, as the FFT graphs contain more parallelism at all levels and are thus more likely to keep all of the communication processors busy.

5.4.3. DFS Partitioning

It is very difficult to model DFS partitioning of the FFT graphs due to the complexity of the FFT topology. The initial partitioning phase of the DFS places each node of the graph into a separate cluster. In the simplest case, the global partitioning phase then alternately groups nodes vertically and horizontally on alternate iterations. This grouping is such that each cluster is an FFT of smaller size than the original graph. Finally, the auxiliary partitioning phase of the algorithm may further merge clusters, assigning multiple "sub-FFT's" to each partition.

The auxiliary partitioning phase begins to behave poorly after performing several merges. Normally, auxiliary partitioning merges the smallest cluster with the cluster to which it is most closely connected. This is acceptable for several iterations of the auxiliary partitioning algorithm. However, a point may be reached where there are several clusters that are equally connected with all of the smallest clusters. Since no heuristic is

provided for selecting among the clusters in these circumstances, details of the algorithm's implementation force a selection. Unfortunately, with this selection the same cluster is always merged with the smallest clusters. The final result is a partitioning in which one cluster is very much larger than all of the other clusters, i.e., the load is not balanced among the processors. This explains why random partitioning does well compared with DFS partitioning for the processing architectures with only four processors. For the FFT graphs used in the simulations, the problem with the auxiliary partitioning algorithm comes into play for those architectures. A reasonable fix for this problem would be to select among the alternative clusters in a manner that balances the computational load among the partitions, at least for the FFT graphs. Testing the influence of this change is a topic for further research.

The operation of the global partitioning phase of the DFS is such that there are two ways the merging of clusters can proceed. If the number of inputs is such that $n+1$ is even (recall that there are 2^n inputs to the graph), the partitioning proceeds in a regular fashion. Figure 25 illustrates this for a graph with 8 inputs. Notice that each partition is an FFT with 2 inputs. If the number of inputs is such that $n+1$ is odd, the resulting partitioning is not as regular. Figure 26 illustrates the DFS partitioning of a graph with 4 inputs. Notice that a portion of the graph does not form a smaller FFT. This irregularity is inherited by subsequent iterations of the global partitioning algorithm, resulting in less regular partitions, which can take longer to execute. Notice that the DFS partitioning for the FFT with 2304 nodes for execution on an architecture with 16 processors and 3 slow functional units of each type results in significantly less base speedup than DFS partitioning of either the 1024 or 5120 node graphs. Notice that $n+1=9$, i.e., $n+1$ is odd, for this graph. The same problem does not occur with partitioning the same graph for other architectures because the point at which the global partitioning phase halts is also influenced by the architecture.

Notice that for the case when $n+1$ is even, all of the global communication involves

arcs between one row of nodes and another. Some of the global communication will overlap multiplication in the upper row and some will overlap multiplication in the lower row. This situation presents the same modeling difficulties as were found for random partitioning. Specifically, it is difficult to estimate the overlap of global communication with itself and with multiplication, both due to the difficulty of the problem in general and to the lack of an established mechanism for scheduling the communication. As such, it is not currently possible to estimate the execution time of FFT graphs that have been partitioned by the DFS algorithm.

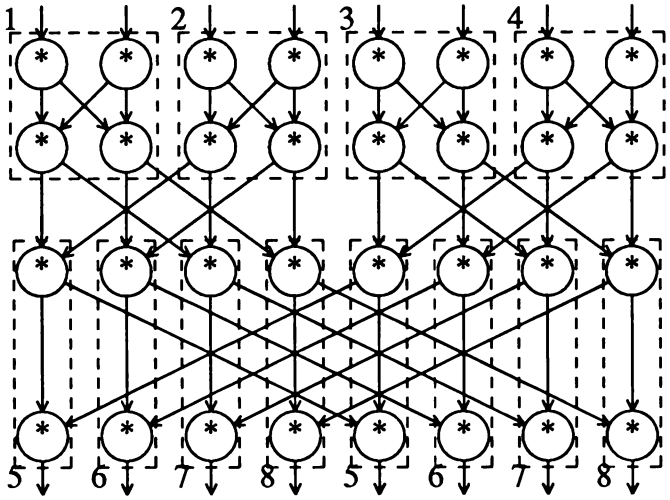

Figure 25. DFS Partitioning of 8-Input FFT Graph

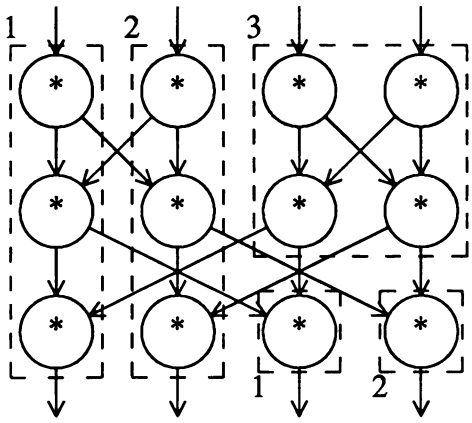

Figure 26. DFS Partitioning of 4-Input FFT Graph

While modeling the execution time of FFT graphs partitioned by the DFS algorithm is difficult, DFS partitionings of the graphs can be created without actually executing the DFS algorithm. First, the FFT graph must be numbered as illustrated for an 8-input FFT in Figure 27. Then each node is labeled with a 4-tuple, (k,l,p,q) such that, for a node numbered c, $c=p\,2^{2k}+q+(2k+1)2^n$. In this labeling, k and l define the vertical position of a node in the graph, while p and q define its the node's horizontal position. All nodes in the graph are labeled for the following values of (k,l,p,q):

$0 \le k < \frac{n+1}{2}$ (when $n+1$ is even);

$0 \le l < 2$;

$0 \le p < \frac{2^n}{2^{2k}}$; and

$0 \le q < 2^{2k}$.

This labeling easily identifies all of the 2-input FFT's that make up the full graph. Specifically, fixing k and varying l over its range while fixing q and varying p from some r to $r+1$ (r even) gives a single, 2-input FFT graph. In the 8-input FFT illustrated in Figure 27, for example, selecting nodes such that $k=1$, $l=0,1$, $p=0,1$, and $q=3$ results in the 2-input FFT made up of nodes 19, 23, 27, and 31.

To obtain a DFS partitioning, rules are used to limit the range of 4-tuples that are included in a given cluster. The simplest case occurs if $n+1$ is a power of 2. In this case, clusters are labeled (i,j), with i giving the vertical placement of the cluster and j its horizontal placement. Then for clusters h nodes in height and w nodes in width, with both h and w even, i and j are such that:

$0 \le i < \frac{n+1}{h}$; and

$0 \le j < \frac{2^n}{w}$.

Then for a given cluster (i,j), nodes labeled with (k,l,p,q) are included subject to the following rules:

$ih \le 2k+l < (j+1)h$;

if $w=1$, then $j \le p\,2^{2k}+q < (j+1)$;

if $1 < w \le \frac{2^n}{2^{2k}}$, then $jw \le p\,2^{2k} < (j+1)w$ and $0 \le q < 2^{2k}$; and

if $w > \frac{2^n}{2^{2k}}$, then $0 \le p < \frac{2^n}{2^{2k}}$, and $q = a + r\,2^{2(k-1)}$, where a is the bit reversal of the binary representation of j, and $0 \le r < \left\lceil \frac{\frac{w}{2^n}}{p\,2^{2k}} \right\rceil$.

Applying the above rules to the 128-input FFT graph, with $h=8$ and $w=8$ results in the same partitioning that resulted from the application of the DFS algorithm to the same graph and the multiprocessor architecture with 16 processors.

The rules required to describe all possible cases for DFS partitioning of FFT graph are rather complex, as illustrated by the above example. However, once the rules are developed (as in the above example), the DFS partitioning of FFT graphs can be obtained in time proportional to the number of nodes in the graph, i.e., much more quickly than they can be obtained by using the DFS algorithm. Since the labeling system given above models important aspects of the FFT graphs, the system also shows promise for exploring other partitioning strategies for these graphs.

5.5. Modeling Execution Times for Random Graphs

Of the three types of graphs used in the simulations, the random graphs are the most difficult to understand in terms of modeling execution behavior. Their structure is neither regular nor simple.

5.5.1. Uniprocessor Execution

The simulation results show that the uniprocessor execution times are completely dominated by the nodes performing division in the graph. Thus, all other computation and communication is overlapped by division. It should be noted that there are approximately the same number of nodes performing each operation in the graph. The domination of the division operation is probably due to the relatively large amount of time required to perform the operation.

This behavior leads directly to a model in which $\alpha_/$ is the only non-zero α. Specifically, $\alpha_/ = \frac{\left\lceil \frac{n_/}{n_F} \right\rceil}{n_/}$, where $n_/$ is the number of nodes in the graph that perform division. $n_/$ must be measured separately for each graph. So, the uniprocessor execution time of the random graphs used in the simulation is given by:

$$\tau = \left\lceil \frac{n_/}{n_F} \right\rceil \times t_/.$$

Tables 17a, 17b, 17c, and 17d compare the model results with the simulation results. The model matches the simulation exactly.

Table 17a. Model for First Set of Random Graphs on Uniprocessors

Number of Nodes	n_F	Functional Unit Speed	Number of Divisions	Actual Execution Times (Simulated)	Predicted Execution Times (Modeled)
500	1	Slow	123	18942	18942
500	1	Fast	123	1845	1845
500	3	Slow	123	6314	6314
500	3	Fast	123	615	615
1000	1	Slow	245	37730	37730
1000	1	Fast	245	3675	3675
1000	3	Slow	245	12628	12628
1000	3	Fast	245	1230	1230
2000	1	Slow	470	72380	72380
2000	1	Fast	470	7050	7050
2000	3	Slow	470	24178	24178
2000	3	Fast	470	2355	2355
5000	1	Slow	1243	191422	191422
5000	1	Fast	1243	18645	18645
5000	3	Slow	1243	63910	63910
5000	3	Fast	1243	6225	6225

Table 17b. Model for Second Set of Random Graphs on Uniprocessors

Number of Nodes	n_F	Functional Unit Speed	Number of Divisions	Actual Execution Times (Simulated)	Predicted Execution Times (Modeled)
500	1	Slow	140	21560	21560
500	1	Fast	140	2100	2100
500	3	Slow	140	7238	7238
500	3	Fast	140	705	705
1000	1	Slow	257	39578	39578
1000	1	Fast	257	3855	3855
1000	3	Slow	257	13244	13244
1000	3	Fast	257	1290	1290
2000	1	Slow	494	76076	76076
2000	1	Fast	494	7410	7410
2000	3	Slow	494	25410	25410
2000	3	Fast	494	2475	2475
5000	1	Slow	1262	194348	194348
5000	1	Fast	1262	18930	18930
5000	3	Slow	1262	64834	64834
5000	3	Fast	1262	6315	6315

Table 17c. Model for Third Set of Random Graphs on Uniprocessors

Number of Nodes	n_F	Functional Unit Speed	Number of Divisions	Actual Execution Times (Simulated)	Predicted Execution Times (Modeled)
500	1	Slow	128	19712	19712
500	1	Fast	128	1920	1920
500	3	Slow	128	6622	6622
500	3	Fast	128	645	645
1000	1	Slow	261	40194	40194
1000	1	Fast	261	3915	3915
1000	3	Slow	261	13398	13398
1000	3	Fast	261	1305	1305
2000	1	Slow	523	80542	80542
2000	1	Fast	523	7845	7845
2000	3	Slow	523	26950	26950
2000	3	Fast	523	2625	2625
5000	1	Slow	1246	191884	191884
5000	1	Fast	1246	18690	18690
5000	3	Slow	1246	64064	64064
5000	3	Fast	1246	6240	6240

Table 17d. Model for Fourth Set of Random Graphs on Uniprocessors

Number of Nodes	n_F	Functional Unit Speed	Number of Divisions	Actual Execution Times (Simulated)	Predicted Execution Times (Modeled)
500	1	Slow	128	19712	19712
500	1	Fast	128	1920	1920
500	3	Slow	128	6622	6622
500	3	Fast	128	645	645
1000	1	Slow	252	38808	38808
1000	1	Fast	252	3780	3780
1000	3	Slow	252	12936	12936
1000	3	Fast	252	1260	1260
2000	1	Slow	505	77770	77770
2000	1	Fast	505	7575	7575
2000	3	Slow	505	26026	26026
2000	3	Fast	505	2535	2535
5000	1	Slow	1280	197120	197120
5000	1	Fast	1280	19200	19200
5000	3	Slow	1280	65758	65758
5000	3	Fast	1280	6405	6405

5.5.2. Random Partitioning

As was the case for both the binary merge and FFT graphs, the execution of random partitionings of the random graphs is dominated by communication costs. Following a similar derivation to those used above gives the following lower and upper bounds on expected execution time:

$$\tau_l = \frac{n_A \times (1 - \frac{1}{n_P}) \times t_{gc}}{\frac{n_P}{2}}, \text{ and}$$

$$\tau_u = n_A \times (1 - \frac{1}{n_P}) \times t_{gc},$$

where n_A is the number of arcs in the random graph.

Tables 18a, 18b, 18c, 18d compare the modeled results to the simulation results, with percent error for the bounds calculated as in Table 13. As with the binary merge and FFT graphs, each entry is an average of four values. The results are similar to those for the other two graph types, in that the lower bound becomes more accurate as the number of arcs in the graph increases.

Table 18a. Model for Random Partitioning of First Set of Random Graphs

Number of Nodes	Number of Arcs	Number of Processors	Average of Actual Execution Times (Simulated)	Lower Bound on Execution Times (Modeled)	Upper Bound on Execution Times (Modeled)	Percent Error of Lower Bound	Percent Error of Upper Bound
500	429	4	7454	6435	12870	13.7	72.7
500	429	16	3224	2011	16088	37.6	399.0
1000	883	4	14507	13245	26490	8.7	82.6
1000	883	16	6191	4139	33113	33.1	434.9
2000	1714	4	26831	25710	51420	4.2	91.6
2000	1714	16	12550	8034	64275	36.0	412.2
5000	4342	4	67545	65130	130260	3.6	92.8
5000	4342	16	23257	20353	162825	12.5	600.1

Table 18b. Model for Random Partitioning of Second Set of Random Graphs

Number of Nodes	Number of Arcs	Number of Processors	Average of Actual Execution Times (Simulated)	Lower Bound on Execution Times (Modeled)	Upper Bound on Execution Times (Modeled)	Percent Error of Lower Bound	Percent Error of Upper Bound
500	436	4	7459	6540	13080	12.3	75.4
500	436	16	2956	2044	16350	30.9	953.1
1000	884	4	14774	13260	26520	10.2	79.5
1000	884	16	6125	4144	33150	32.3	441.2
2000	1759	4	27722	26385	52770	4.8	90.4
2000	1759	16	10772	8245	65963	23.5	512.4
5000	4312	4	74142	64680	129360	12.8	74.5
5000	4312	16	23575	20213	161700	14.3	585.9

Table 18c. Model for Random Partitioning of Third Set of Random Graphs

Number of Nodes	Number of Arcs	Number of Processors	Average of Actual Execution Times (Simulated)	Lower Bound on Execution Times (Modeled)	Upper Bound on Execution Times (Modeled)	Percent Error of Lower Bound	Percent Error of Upper Bound
500	444	4	7384	6660	13320	9.8	80.4
500	444	16	3913	2081	16650	46.8	325.5
1000	844	4	14498	12660	25320	12.7	74.6
1000	844	16	5876	3956	31650	32.7	438.6
2000	1732	4	27466	25980	51960	5.4	89.2
2000	1732	16	12788	8119	64950	36.5	407.9
5000	4362	4	66173	65430	130860	1.1	97.8
5000	4362	16	24286	20447	163575	15.8	573.5

Table 18d. Model for Random Partitioning of Fourth Set of Random Graphs

Number of Nodes	Number of Arcs	Number of Processors	Average of Actual Execution Times (Simulated)	Lower Bound on Execution Times (Modeled)	Upper Bound on Execution Times (Modeled)	Percent Error of Lower Bound	Percent Error of Upper Bound
500	442	4	7710	6630	13260	14.0	72.0
500	442	16	3300	2072	16575	37.2	402.3
1000	909	4	15007	13635	27270	9.1	81.7
1000	909	16	7048	4261	34088	39.5	383.7
2000	1805	4	29761	27075	54150	9.0	81.9
2000	1805	16	10631	8461	67688	20.4	536.7
5000	4356	4	67363	65340	130680	3.0	94.0
5000	4356	16	23569	20419	163350	13.4	593.1

5.5.3. DFS Partitioning

The execution times for DFS partitioning of the random graphs are dominated by nodes performing division. Empirically, approximately 30% of these nodes overlap their execution with other division nodes. However, the difficulty in accurately modeling the overlap of the graph's operations prevents the development of any accurate models for these execution times. Further research is required to develop models for this parallelism.

5.6. General Comments on Modeling Execution Times

The models developed in this chapter provide some important information about the problem of modeling multiprocessor execution, about improving the DFS algorithm, and about areas which require further research. Foremost is the fact that it is difficult to develop models of multiprocessor execution for specific situations, let alone for general cases, primarily because of the lack of good estimates for the overlap of operations in a parallel algorithm. It is important to note that the execution of random partitionings of all three graph types were dominated by communication. This shows the unsuitability of random partitioning for any graph with a large amount of communication. Before the models presented here can be refined, a scheduling mechanism for global communications must be developed and evaluated. The predominance of division operations in the random graphs indicates that these graphs may not accurately model irregular algorithms that occur in practice. Further research can identify actual examples of such algorithms and evaluate the performance of the DFS algorithm when applied to such examples.

Any attempt at improving the DFS algorithm must first correct the lack of load balancing caused by the auxiliary partitioning phase. It appears that the model presented for DFS partitioning of FFT graphs can be used as a framework to explore alternate partitionings that may provide better results than the application of the DFS algorithm. Further research should explore this possibility in more detail. Finally, a key point noted in the development of all the models presented in this chapter is the extreme difficulty of

estimating the overlap of operations in a parallel algorithm. The further investigation of estimation methods would be beneficial to the improvement of the DFS algorithm, and also to any other methods that might be used to partition parallel algorithms for multiprocessor execution.

Chapter 6: Summary and Conclusions

The rapid decrease of the cost of fabricating integrated circuits highlights the importance of reducing non-recurring engineering (NRE) costs, including the cost of human design time. Because of the limited market for individual application specific integrated circuits (ASIC's), their success depends in part on minimizing NRE costs. An important method for reducing these costs is the use of design methodologies, i.e., prescribed sequences of steps to be followed in translating a behavioral specification into a working device. The high complexity of VLSI circuits suggests that design methodologies suited for automation will be an important tool in exploiting the potential of ASIC's.

These facts led to the definition of five objectives for the research presented in this dissertation. First, define a design methodology suited for automation. Second, identify research issues crucial to the automation of the design methodology. Third, investigate one of these issues by developing an algorithm to automatically partition parallel algorithms for execution on multiprocessor architectures. Fourth, evaluate this algorithm via simulation. Finally, model these simulation results to improve the algorithm, reduce the computational cost of the simulation process, and to increase understanding of the basic problem of partitioning parallel algorithms.

Current design methodologies are limited in either the types of designs to which they apply or in their suitability for automation. Especially lacking are methods for translating behavioral specifications to structural representations. The complexity of this problem makes it difficult to perform this translation in a single step. It is also expensive to translate structural descriptions to geometrical descriptions. Thus, it is desirable to estimate the performance of a structural design without translating that design to a geometrical description. In the algorithmic design methodology (ADM) defined in this dissertation, a designer specifies desired circuit behavior as an algorithm in a familiar, high-level language. This algorithm is translated to a structural model of a VLSI circuit architecture. To deal with the complexity of this transformation it is carried out in

several steps. First, the algorithm is parallelized to produce a scalar data flow graph. Second, the scalar data flow graph is partitioned to balance the computation load among the partitions while keeping interpartition communications low. Finally, the partitioned scalar data flow graph is embedded in the VLSI circuit architecture model. The performance of the architecture is estimated by estimating the area needed for each processor in the architecture and for routing interprocessor connections.

The automation of the ADM requires the resolution of several issues. First, methods for automatically parallelizing sequential algorithms must be investigated. Second, techniques must be developed for automatically and efficiently partitioning parallel algorithms to improve the execution time of those algorithms on multiprocessor architectures. Third, partitioned parallel algorithms must be embedded into the VLSI circuit architecture model in a manner that produces designs comparable to those produced by human designers. Finally, the processing speed and chip area of an integrated circuit must be estimated from its VLSI circuit architecture representation.

The research presented in this dissertation concentrates on the problem of automatically partitioning parallel algorithms for execution on multiprocessor systems. This investigation lead to the development of the data flow scheduling (DFS) algorithm. The DFS algorithm deals with algorithms expressed as acyclic scalar data flow graphs, in which operations are at a low level of granularity. This allows the DFS algorithm to effectively utilize all of the available parallelism, subject to the limits of the processing architecture. The models used for parallel algorithms and multiprocessor architectures are sufficiently general to be applicable to a range of actual systems, while not impeding the operation of the DFS algorithm. The DFS algorithm uses a divide-and-conquer approach to reduce the time required to partition scalar data flow graphs.

The DFS algorithm was evaluated via simulation. In these simulations, several types and sizes of scalar data flow graphs were partitioned for execution on various multiprocessor systems. The execution time of algorithms partitioned by the DFS algorithm

were compared with those obtained when the graphs were randomly partitioned and when the graphs were executed on a single processor. The results show that the DFS algorithm significantly reduces execution time over uniprocessor execution of the same algorithm. The DFS algorithm also produces lower execution times than random partitioning in most, but not all, of the simulations. The simulation results show that random partitioning is incapable of effectively utilizing powerful processors.

The modeling of the simulation results uncovered several properties of the DFS algorithm and the partitioning problem. First, it is extremely difficult to develop accurate models for multiprocessor execution, primarily due to the difficulty of generating good estimates of the overlap of computations and interprocessor communication. Refinement of both the multiprocessor architecture model and the understanding of the algorithm being partitioned will make modeling of the DFS algorithm easier. Second, the execution of the randomly partitioned graphs was dominated by interprocessor communication, making this strategy unsuitable for algorithms with large amounts of communication. Finally, improvements were suggested for both the multiprocessor architecture model (e.g., the definition of a protocol for scheduling interprocessor communication) and the DFS algorithm (e.g., the lack of load balancing during auxiliary partitioning).

The investigation of the DFS algorithm has highlighted shortcomings in its behavior when mapping algorithms to architectures with a restricted number of processors. The ability of the DFS algorithm to deal with specific algorithms and actual multiprocessing systems remains to be examined. Modeling the simulations for graphs with the FFT topology resulted in a method for describing the graphs that holds promise for exploring partitioning strategies outside the framework of the DFS algorithm. The problem of estimating parallelism in algorithms is still a very difficult task and any results in this area would greatly improve partitioning algorithms. Of the four issues crucial to automating the ADM, only the one concerning partitioning parallel algorithms has been addressed, leaving the others open for further study.

LIST OF REFERENCES

LIST OF REFERENCES

[Adam74] Adam, Thomas L., Chandy, K. M., and Dickson, J. R. "A Comparison of List Schedules for Parallel Processing Systems." *Comm. ACM*, Vol. 17, No. 12, Dec. 1974. pp. 685-690.

[Aho86] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts. 1986.

[Alel82] Aleliunas, Romas and Rosenberg, Arnold L. "On Embedding Rectangular Grids in Square Grids." *IEEE Trans. on Computers*, Vol. C-31, No. 9, Sep. 1982. pp. 907-913.

[Andr88] Andrews, Warren "Silicon Compilers Still Struggling Toward Widespread Acceptance." *Computer Design*, Feb. 15, 1988. pp. 37-43.

[Arvi80] Arvind, Kathail, V., and Pingali, K. *A Data Flow Architecture with Tagged Tokens*. Technical Memo 174, Lab. for Computer Science, MIT, Sept. 1980.

[Babb85] Babb, Robert G., II "Software Engineering for Parallel Processing." *Proc. 1985 Parallel Processing Executive Seminar*, 1985. pp. 48-57.

[Chen86a] Chen, Marina C. "Placement and Interconnection of Systolic Processing Elements: A New LU Decomposition Algorithm." *Proc. 1986 IEEE Int. Conf. on Computer Design: VLSI in Computers*, 1986. pp. 275-281.

[Chen86b] Chen, Marina C. "Synthesizing VLSI Architectures: Dynamic Programming Solver." *Proc. 1986 Int. Conf. on Parallel Processing*, 1986. pp. 776-784.

[Chou82] Chou, T. C. K. and Abraham, J. A. "Load Balancing in Distributed Systems." *IEEE Trans. on Software Engineering*, Vol. SE-8, No. 4, July 1982. pp. 401-412.

[Coff76] Coffman, E. G., Jr., Ed., *Computer and Job-Shop Scheduling Theory*. John Wiley & Sons, New York, New York. 1976.

[Dris88] Driscoll, Michael A., Prins, Philip R., Fisher, P. David, and Ni, Lionel M. *Efficient Scheduling of Data Flow Graphs for Multiprocessor Architectures*. Technical Report No. MSU-ENGR-88-008, Michigan State University, East Lansing, Michigan. 1988.

[Efe82] Efe, Kemal "Heuristic Models of Task Assignment Scheduling in Distributed Systems." *IEEE Computer*, Vol. 15, No. 6, June 1982. pp. 50-56.

[Elli84] Ellis, J. A. *Embedding Graphs in Lines, Trees and Grids*. Ph.D. Dissertation, Dept. of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, 1984.

[Faro87a] Faroughi, Nikrouz and Shanblatt, Michael A. "An Improved Systematic Method for Constructing Systolic Arrays from Algorithms." *Proc. 1987 ACM-IEEE Design Automation Conference*, 1987. pp. 26-31.

[Faro87b] Faroughi, Nikrouz and Shanblatt, Michael A. "Systematic Generation and Enumeration of Systolic Arrays from Algorithms." *Proc. 1987 Int. Conf. on Parallel Processing*, 1987. pp. 844-847.

[Fern75] Fernandez, E. B. and Lang, T. "Computation of Lower Bounds for Multiprocessor Schedules." *IBM Journal of Research and Development*, Vol. 19, Sept. 1975. pp. 435-444.

[Gajs83] Gajski, D. D. "Guest Editor's Introduction: New VLSI Tools." *IEEE Computer*, Vol 16, No. 12, Dec. 1983. pp. 11-14.

[Gonz77] Gonzalez, Mario J., Jr. "Deterministic Processor Scheduling." *ACM Computing Surveys*, Vol. 9, No. 3, Sept. 1977. pp. 173-204.

[Girk88] Girkar, Milind and Polychronopoulus, Constantine "Partitioning Programs for Parallel Execution." *Proc. 1988 Int. Conf. on Supercomputing*, 1988. pp. 216-229.

[Gran87] Granski, M., Koren, I., and Silberman, G. M. "The Effect of Operation Scheduling on the Performance of a Data Flow Computer." *IEEE Trans. on Computers*, Vol. C-36, No. 9, Sept. 1987. pp. 1019-1029.

[Ho83] Ho, Lawrence Y. and Irani, Keki B. "An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment" *Proc. 1983 Int. Conf. on Parallel Processing*, 1983. pp. 338-340.

[Hu61] Hu, T. C. "Parallel Sequencing and Assembly Line Problems." *Operations Research*, Vol. 9, Nov.-Dec. 1961. pp. 841-848.

[Joha79] Johannsen, Dave "Bristle Blocks: A Silicon Compiler." *Proc. 1979 ACM-IEEE Design Automation Conference*, 1979. pp. 310-313.

[Kohl75] Kohler, Walter H. "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems." *IEEE Trans. on Computers*, Vol. C-24, no. 12, Dec. 1975. pp. 1235-1238.

[Kore83] Koren, Israel and Silberman, Gabriel M. "A Direct Mapping of Algorithms onto VLSI Processor Arrays Based on the Data Flow Approach." *Proc. 1983 Int. Conf. on Parallel Processing*, 1983. pp. 335-337

[Kung82] Kung, H. T. "Why Systolic Architectures." *IEEE Computer*, Vol. 15, No. 1, Jan. 1982. pp. 37-46.

[Leis80] Leiserson, Charles E. "Area-Efficient Graph Layouts (for VLSI)." *Proc. 1980 IEEE Conf. on the Foundations of Computer Science*, 1980. pp. 270-281.

[Mend87] Mendelson, Bilha and Silberman, Gabriel M. "An Improved Mapping of Data Flow Programs on a VLSI Array of Processors." *Proc. 1987 Int. Conf. on Parallel Processing*, 1987. pp. 871-873.

[Mira84] Miranker, W. L. and Winkler, A. "Spacetime Representation of Computational Structures." *Computing*, No. 32, 1984. pp. 93-114.

[Mold83a] Moldovan, D. I. "On the Design of Algorithms for VLSI Systolic Arrays." *Proc. of the IEEE*, Vol. 71, No. 1, Jan. 1983. pp. 113-120.

[Mold83b] Moldovan, D. I. and Varma, A. "Design of Algorithmically-Specialized VLSI Devices." *Proc. 1983 IEEE Int. Conf. on Computer Design: VLSI in Computers*, 1983. pp. 88-91.

[Park79] Parker, A., Thomas, D., Siewiorek, D., Barbacci, M., Hafer, L., Leive, G., and Kim, J. "The CMU Design Automation System: An Example of Automated Data Path Design." *Proc. 1979 ACM-IEEE Design Automation Conference*, 1979. pp. 73-80.

[Quin84] Quinton, Patrice "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations." *Proc. 11th IEEE Symposium on Computer Architecture*, 1984. pp. 208-214.

[Ravi87] Ravi, T. M., Ercegovac, T. Long, and Muntz, R. R. "Static Allocation for a Data Flow Multiprocessor System." *Proc. Second Int. Conf. on Supercomputing, Vol III*, 1987. pp. 169-178.

[Seth76] Sethi, Ravi "Scheduling Graphs on Two Processors." *SIAM Journal on Computing*, Vol. 5, No. 1, Mar. 1976. pp. 73-82.

[Scot85] Scott, Walter S., Mayo, Robert N., Hamachi, Gordon, and Ousterhout, John K., Eds., *1986 VLSI Tools: Still More Works by the Original Artists.*, Report No. UCB/CSD 86/272, University of California, Berkeley, California. 1986.

[Simo86] Simonson, Charles *Layout Problems on Trees.* Ph.D. Dissertation, Dept. of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, 1986.

[Snyd82a] Snyder, Lawrence "Introduction to the Configurable, Highly Parallel Computer." *IEEE Computer*, Vol. 15, No. 1, Jan. 1982. pp. 47-56.

[Snyd84] Snyder, Lawrence "Parallel Programming and the Poker Programming Environment." *IEEE Computer*, Vol. 17, No. 7, Jul. 1984. pp. 27-36.

[Thom79] Thompson, C. D. "Area-Time Complexity for VLSI." *Proc. 11th Annual ACM Symposium on the Theory of Computing*, 1979. pp. 81-88.

[Thom80] Thompson, C. D. *A Complexity Theory for VLSI.* Ph.D. Dissertation, Computer Science Dept., Carnegie-Mellon Univeristy, Pittsburgh, Pennsylvania, 1980.

[Ullm84] Ullman, Jeffrey D. *Computational Aspects of VLSI*. Computer Science Press, Rockville, Maryland. 1984.

[Vali81] Valiant, Leslie G. ''Universality Considerations in VLSI Circuits.'' *IEEE Trans. on Computers*, Vol. C-30, No. 2, Feb. 1981. pp. 135-140.